

dotnetpro

by DWX



DIE BESTEN KI-TOOLS

GitHub Copilot, Claude
Code, Anysphere Cursor,
Lovable, Firebase & Bolt
geschickt einsetzen.
Ab Seite 6

m:con Congress Center, Rosengarten, Mannheim

DWX

Die Konferenz für AI, Cloud, Web und
.NET Development

29. JUNI BIS 2. JULI 2026

MEHR ZU
BEIDEN EVENTS
AUF DEN SEITEN
140 UND 143

INFINITE AI CONFERENCE

Die Konferenz für KI-unterstütztes Development

30. JUNI BIS 1. JULI 2026

INFINITE AI CONFERENCE

Die Konferenz für KI-unterstütztes Development

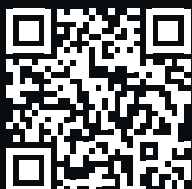
30. JUNI BIS
1. JULI 2026

m:con Congress Center
Rosengarten | Mannheim

Jetzt
Early Bird
sichern!

**KI im gesamten Software-Lifecycle
– von der Idee bis zum Produkt.**

Lerne von führenden Expert:innen die richtigen Tools, Strategien und Best Practices – für schnelleren Code, bessere Architektur und nachhaltige Systeme. Für Entwickler:innen und Engineers, die heute schon morgen bauen wollen.



www.developer-world.de/infinite-ai | [#iaic](https://twitter.com/iaic)



Präzision zählt

Ein unscheinbarer LEGO-Stein ist eine erstaunliche Analogie zu zukunftsicherer Softwareentwicklung.

Kürzlich stieß ich auf einen Artikel, der erklärte, warum ein LEGO-Stein, der vor Jahrzehnten hergestellt wurde, auch heute noch perfekt auf einen passt, der erst kürzlich produziert wurde. Im vorliegenden Fall liegen volle 68 Jahre zwischen ihren „Geburtstagen“. Zudem stammen sie aus unterschiedlichen Fabriken, wurden mit verschiedenen Maschinen gefertigt und auf verschiedenen Kontinenten produziert. Und trotzdem greifen sie ineinander, als wären sie zur selben Zeit am selben Ort auf demselben Fließband entstanden.

Der Grund: LEGO-Steine werden seit jeher mit Toleranzen von nur zehn Mikrometern gefertigt. Schon eine Abweichung von 0,02 Millimetern reicht, dass ein Stein nicht mehr ins System passt. Entscheidend sind drei Faktoren: das richtige Material, geringste Formtoleranzen und ein normierter Produktionsprozess. Sie bilden die Voraussetzung für die präzisen Bausteine.

In der Softwareentwicklung verhält es sich mit den Abweichungen nicht viel anders. Architekturen, die über Jahre ohne klare Strukturen gewachsen sind, reagieren in der Handhabung wie ein Material mit unvorhersehbaren Eigenschaften: Ist das Fundament verzogen, summiert sich jede kleine Ungenauigkeit.

„Präzision, eng definierte Toleranzen und stetige Qualitätstests schaffen zukunftsichere Systeme.“

Auch technische Schulden wirken ähnlich. Jede einzelne Abweichung scheint für sich genommen winzig klein – ein schlecht benanntes Interface hier, eine Klasse mit zu vielen Aufgaben dort. Doch am Ende addieren sich all diese Abweichungen zu einem Schuldenberg, der unüberwindbar wird.

Und dann der Prozess. „Was man nicht messen kann, kann man nicht kontrollieren.“ Dieser von Tom DeMarco publik gemachte Grundsatz lässt sich auch als Plädoyer für Unit-Tests verstehen. Denn Unit-Tests zeigen, ob ein System gegenüber Änderungen stabil bleibt – sie messen also die Funktionalität.

Nicht jedes System braucht dieselbe Architektur. Aber jedes System braucht eine bewusste Entscheidung darüber, welche Toleranzen es sich leisten kann.

Eine konstante Qualität von LEGO-Steinen seit 1958. Eine Codebasis, die auch nach zehn Jahren noch wartbar ist. Beides ist kein Zufall. Und beides bleibt – auch im Zeitalter von KI – vor allem eines: Ingenieurskunst.

Ich wünsche Ihnen eine schöne Zeit mit der neuen dotnetpro!

Herzlichst

Fernando Schneider
Chefredakteur dotnetpro



Uwe Vigerschow

verrät, warum die Reduzierung von Komplexität Softwareprojekte nachhaltig handhabbarer macht (S. 98)



Christian Wenz

analysiert die neueste OWASP Top Ten und erläutert daran wichtige Handlungsoptionen für mehr Sicherheit (S. 116)



INHALT

Focus

Mit der Lizenz zum Coden

GitHub Copilot Coding Agent effektiv nutzen

Claude, mach mal mit

Agentic Coding mit Claude Code

Mit dem KI-Agenten Cursor produktiv arbeiten

Die KI-IDE Cursor in der Praxis

Computer, bau mir eine Webseite

AI-Website-Builder auf dem Prüfstand

.NET Stack

NETirol: Magischer DI-Container

Das NuGet-Paket

Microsoft.Extensions.DependencyInjection

Next-Level Debugging

Von Logging zu Observability: OpenTelemetry und Co.

Effizienter Brückenschlag

.NET, Python und KI – Grundlagen und Ausblick

Data Access: Objekte flicken

JSON Patch für System.Text.Json

Azure Policies

Regelwerke für die Microsoft-Cloud

HTTP-APIs mit SQL Server 2025

Moderne Daten- und Applikationsintegration mit .NET 10

Vektoren nativ speichern und für RAG-Szenarien nutzen

Künstliche Intelligenz und SQL Server 2025

Keine Produktion ohne Automation

Legacy-Transformation, Teil 2

Artificial Intelligence

KI-Agenten mit Daten füttern

Mehr als nur Prompts

6

12

18

24

Bild: Monterey



Die besten KI-Tools für Developer: Copilot, Claude Code, Cursor, Lovable und Co. in der Praxis optimal nutzen.

6

30

32

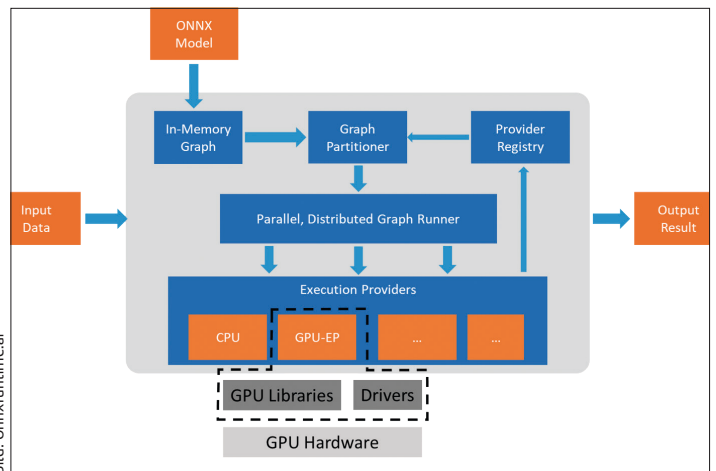
40

48

56

66

Bild: Onnxruntime.ai



Im Team stärker: Wie sich Python-KI-Modelle und ML-Frameworks effizient in C#-Projekte integrieren lassen.

40

73

78

Experten in dieser Ausgabe



Christian Kiefer zeigt, wie er in SQL Server 2025 Vektoren nativ speichert und für RAG-Szenarien nutzt.

73



Christian Jacob schließt sein smartes Legacy-Transformationsprojekt zum Thema SQL Change Tracking ab.

78

88

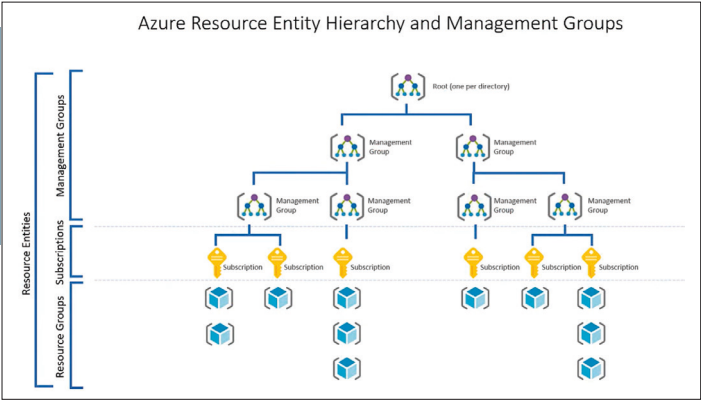
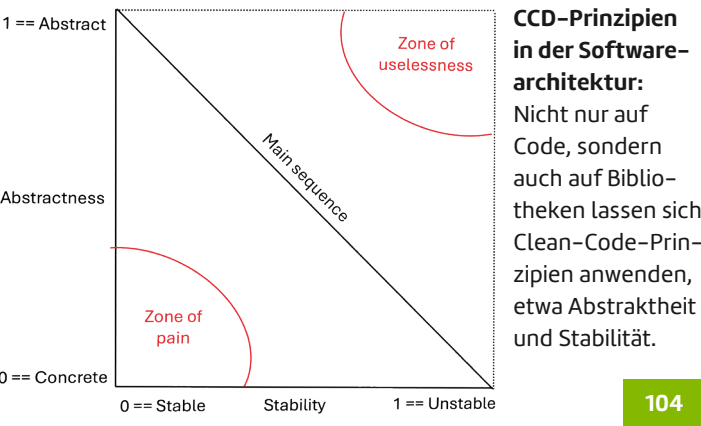


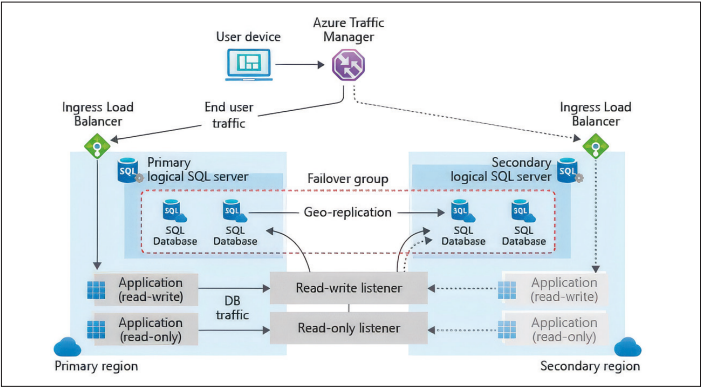
Bild: Microsoft

Regeln für die Microsoft-Cloud: Azure Policy setzt Richtlinien automatisiert um und erkennt Verstöße frühzeitig.

56



104



Wer zukunftssichere Cloud-Lösungen mit Azure bauen will, muss in Systemen denken, nicht in einzelnen Services.

110

LUST AUF MEHR?

Dann auf zur DEVELOPER WORLD

Cross Concerns

Dauids Deep Dive: Endlich gescheitert!
Wie Softwareentwicklung nicht funktioniert 96

Weniger ist so viel mehr!
Was Komplexität ausmacht und wie wir sie wirksam reduzieren 98

Prinzipien für Architektur
Clean Code und Architektur, Teil 5 104

What else?
Strategien für zukunftssichere Cloud-Lösungen
Systemarchitekturen in Azure 110

Auffrischkurs Web-Security
Sicherheitsrisiken für Webanwendungen:
OWASP Top Ten 2025 116

Ein starkes Team
Rust und Wasm 128

People & Soft Skills

Time-out: Wer entscheidet hier eigentlich?
Bauchgefühl versus Rationalität 136

Knowledge

Wissen trifft Community
Vorschau auf die DWX 2026 140

KI gehört in deinen Workflow
Vorschau auf die IAIC 2026 143

Editorial 3

Inhalt 4

Arbeitsmarkt 138

Lernstoff 144

Impressum / Vorschau / Inserenten 146



GITHUB COPILOT CODING AGENT EFFEKTIV NUTZEN

Mit der Lizenz zum Coden

Wie der GitHub Copilot Coding Agent für euch teilautonom Entwicklungsarbeit in der Cloud erledigt – von den ersten Schritten bis zu komplexen Aufgaben.

Microsoft nennt als zentrales Element der eigenen KI-Strategie fast alles „Copilot“, und so gibt es selbst als Teil von GitHub mittlerweile mindestens drei Copiloten:

- **GitHub Copilot**, der als Chat-Modus oder Agent-Modus in IDEs wie Visual Studio Code integriert ist,
- **GitHub Copilot CLI**, der im Terminal ausgeführt wird,
- **GitHub Copilot Coding Agent**, der als Dienst in der Cloud definiert ist und einige Besonderheiten mit sich bringt.

Letzterer ist unser Kernthema. Was ihn auszeichnet, ist seine Konzeption als „Teamkollege in der Cloud“, der Aufgaben unabhängig in seiner eigenen Laufzeitumgebung durchführt. Alle GitHub-Copilot-Implementierungen teilen sich einen zunehmenden Teil der darunterliegenden Architektur, sodass eine im Februar 2026 vorgestellte Neuerung ist, vom Coding Agent zu Visual Studio Code oder zu GitHub Copilot CLI wechseln zu können. Hierbei muss allerdings der Benutzer

noch selbst dafür Sorge tragen, dass die Umgebungen miteinander kompatibel sind.

Agenten einrichten

Bevor wir Copilot Coding Agent verwenden können, sollten wir uns auf Benutzerprofilebene die Einstellungen unter *Copilot* ansehen. Dort finden wir eine Reihe von Optionen, um etwa dem Copiloten zu erlauben, im Internet Suchen durchzuführen. Der Coding Agent als solcher ist grundsätzlich immer aktiviert, allerdings müssen wir den Zugriff entweder auf einzelne Repositories oder auf alle Repositories erlauben.

Auf Repository-Ebene kontrollieren wir zunächst die Netzwerkzugriffsrechte für den Coding Agent. Wir können die Firewall einschalten und für bestimmte Zieladressen den Zugriff explizit erlauben. Eine vorkonfigurierte „Allow“-Liste wichtiger Zieladressen steht zur Verfügung. Dies ist relevant, um etwa das Risiko der Datenexfiltration zu minimieren.

Aufgaben zuweisen: Viele Wege zum Pull Request

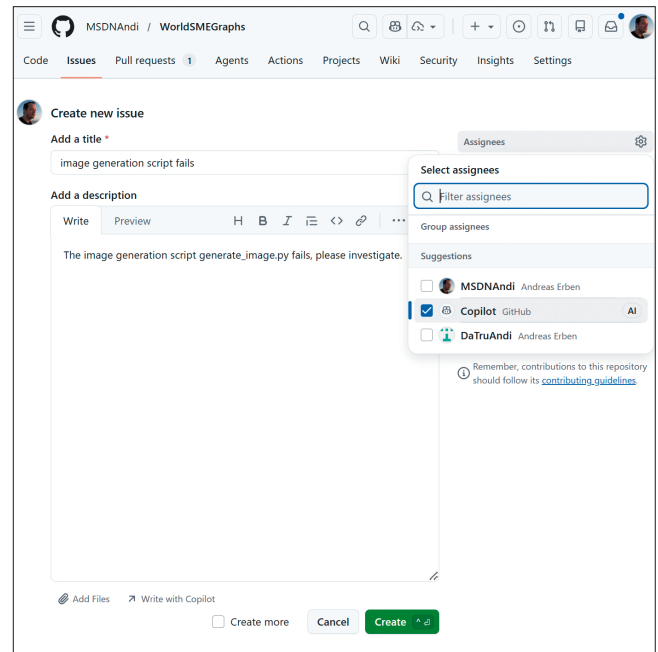
GitHub hat über die Monate zahlreiche Einstiegspunkte geschaffen. Der klassische Weg führt über ein GitHub-Issue: Erstelle ein Issue und weise Copilot als Assignee zu oder erwähne `@copilot` in einem Kommentar (siehe Bild 1). In einem bestehenden Pull-Request-Kommentar forderst du per `@copilot` Änderungen an – der Agent nimmt seine Arbeit wieder auf. Das Agents-Panel ist auf jeder GitHub-Seite erreichbar (auch direkt unter github.com/copilot/agents) und dient gleichzeitig als „Mission Control“ mit Real-Time Steering: Du kannst den Agenten während einer laufenden Session mit Klarstellungen in die richtige Richtung schubsen. In Visual Studio Code delegierst du Aufgaben über die GitHub-Pull-Request-Extension.

Die Liste der Integrationen wächst ständig. Erwähnst du etwa in Slack `@GitHub` in einem Thread, übernimmt der Agent den Konversationskontext und meldet sich mit dem fertigen PR. In Microsoft Teams funktioniert das analog über die GitHub-App, und du kannst über Azure Boards und Linear-Integration Tasks für den Agenten initiieren.

Nach der Zuweisung klonst der Agent dein Repository in eine frische VM, analysiert den Code, nimmt Änderungen vor und führt gegebenenfalls Tests und Linter aus. Er erstellt während der Bearbeitung einen Draft-Pull-Request auf einem Branch mit dem Präfix `copilot/` – ein direkter Push auf `main` ist konstruktionsbedingt ausgeschlossen. Über *View session* im PR kannst du nachvollziehen, wie der Agent zu seinen Entscheidungen kam (siehe Bild 2). Hier kannst du jetzt auch die Session per Hand-off in Copilot CLI oder Visual Studio Code weiterbearbeiten.

Den Agenten briefen: Instructions und AGENTS.md

GitHub bietet ein hierarchisches System von Konfigurationsdateien. Die wichtigste ist `.github/copilot-instructions.md` –



Issue dem Coding Agent zuweisen (Bild 1)

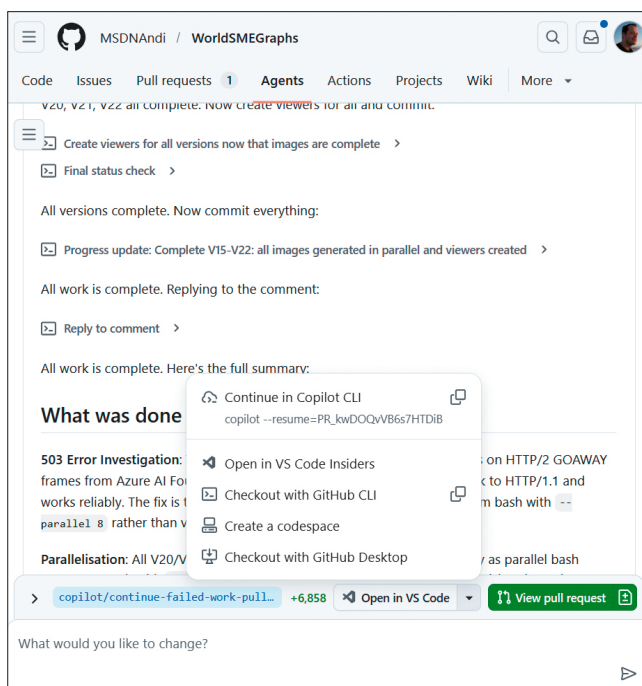
sie gilt Repository-weit und enthält die grundlegenden Spielregeln: Frameworks, Ordnerstruktur, Coding-Konventionen, Teststrategie. Zusätzlich kannst du unter `.github/instructions/` dateispezifische Anweisungen mit `applyTo`-Glob-Patterns ablegen – beispielsweise andere Regeln für Controller-Klassen als für Unit-Tests.

Daneben gibt es die `AGENTS.md` – ein offener Standard, den mehrere KI-Agenten verstehen (neben Copilot auch Claude Code, Codex und andere). Die Datei kann im Root oder in Unterverzeichnissen liegen. Listing 1 zeigt ein kompaktes Beispiel.

AGENTS.md versus copilot-instructions.md

Die `copilot-instructions.md` wird bei jeder Copilot-Interaktion berücksichtigt – Chat, Agent Mode und Coding Agent. Die `AGENTS.md` ist ein offener, agentenübergreifender Standard. Nutze einfach beide: `copilot-instructions.md` für Copilot-spezifische Anweisungen, `AGENTS.md` als gemeinsame Wissensbasis für alle Agenten. Konfigurierte Agenten lassen sich bei der Task-Erstellung direkt zuweisen, oder der Coding Agent kann diese während der Bearbeitung dynamisch hinzuziehen. Als Tipp empfehle ich neuen Benutzern, den Coding Agent selbst auch weitere Agentendefinitionen schreiben zu lassen oder ihn die `copilot-instructions.md` überarbeiten zu lassen, um hier schnell Erfahrungen zu sammeln.

In der Praxis gibt es mehrere Ansätze, Agents zu strukturieren. Manche gehen von einem rollenbasierten Agenten-Konzept aus, in dem jeder Agent Spezialkenntnisse simuliert: vom Designer über den 10x-Entwickler und den Tester bis hin zum simulierten Wissenschaftler in einem Spezialgebiet. Andere Ansätze fokussieren auf die Komplexität von Unteraufgaben und sehen günstigere Agenten für einfachere Aufgaben vor, oder sie kapseln bestimmte Funktionen aus Sicherheitsgründen für einen bestimmten Agenten. Ein wichtiger ►



Session zur Nachverfolgung mit Möglichkeit des Hand-offs (Bild 2)

● Listing 1: Beispiel einer stark simplifizierten copilot-instructions.md

```
# Projekt: WebShop API
## Architektur
- .NET 9 Minimal API mit Clean Architecture
- src/Api/ - Endpoints und Middleware
- src/Domain/ - Entities und Value Objects
- src/Infrastructure/ - EF Core, externe Services
## Konventionen
- Verwende Records fuer DTOs
- Jeder Endpoint braucht einen Integration Test
- Fehlerbehandlung ueber ProblemDetails (RFC 9457)
## Tests ausfuehren
- dotnet test tests/Unit/
- dotnet test tests/Integration/
```

Teil der Agentendefinition ist, dass jeder Agent die Tools definieren kann, welche erlaubt sind, und dass er sogar zusätzliche MCP-Server definieren darf. Agenten können vom Agents Panel explizit aufgerufen werden, oder sie werden implizit hinzugezogen, wenn das LLM des Coding Agent folgt, dass ein bestimmter Agent eine Aufgabe am ehesten lösen dürfte. Eine der wichtigsten Aufgaben von Agents ist es allgemein, bei der Aufgaben-Dekomposition zu helfen und ein eigenes, frisches Kontext-Window zu haben.

Agent-Panel: Agenten im Überblick

Das Agent-Panel, auch als Mission Control bezeichnet, ist der Ort, an dem wir sowohl neue Aufgaben (Tasks) starten können als auch laufende und bisherige Agent Runs im Überblick sehen und schnell zu den relevanten Einstellungen des Coding Agent navigieren können (siehe Bild 3). Dies beinhaltet auch die Repository-Einstellungen (*Configure*) oder die Copilot Setup Steps (*Customize environment*) und die Möglichkeit, schnell zu den Detail-Logs zu einem Agenten oder zum assoziierten Pull Request zu navigieren. In der Praxis wird man vor allem bei kleineren Aufgaben oft nicht den „Umweg“ nehmen, zuerst ein Issue anzulegen und dieses dem Copiloten zuzuweisen, sondern direkt formulieren, was der Agent machen soll.

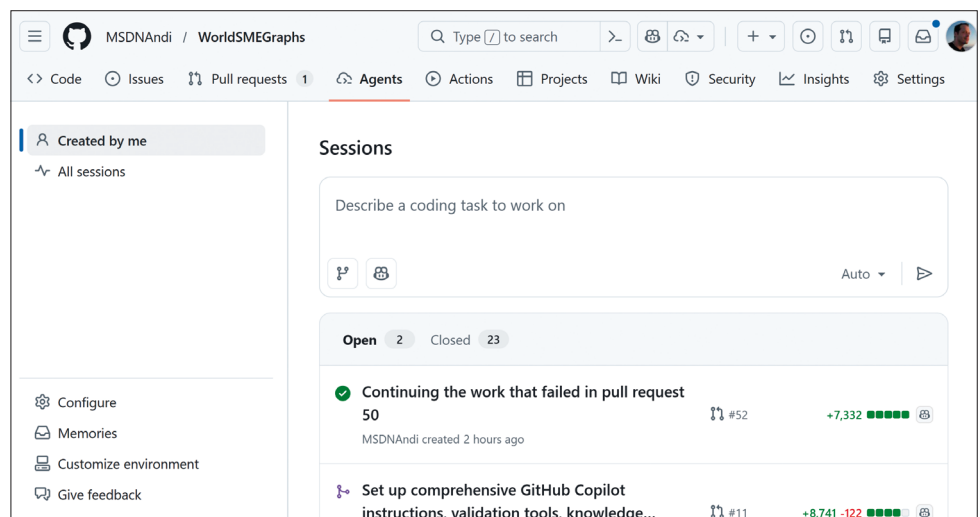
Die KI-Modell-Auswahl

Wo du eine Aufgabe starten kannst, lässt GitHub dir die Wahl, welches KI-Modell dieses bearbeiten soll (siehe Bild 4). Du siehst dort auch, wie viele Premium-Requests dafür verwendet werden. Ein nicht sonderlich gut dokumentiertes Feature ist, dass du den Coding Agent aber auch darum bitten kannst, einem Sub-Agent ein anderes Modell zuzuweisen. Neben den Agenten, die du selbst definierst, kann der Coding Agent dies flexibel selbst über das interne *task*-Tool erledigen. Ich selbst habe mir über Copilot CLI die Liste der

verfügbaren Modelle angesehen und gefunden, dass es deutlich mehr gibt als nur die, welche sich im Coding Agent UI auswählen lassen. Momentan werden die Kosten interessanterweise nur davon definiert, mit welchem Model der Agent Task gestartet wurde. Wenn der Coding Agent also einen Task mit dem Opus Model ausführen lässt, wir aber mit der „Auto“-Modell-Auswahl starten, so gelten dennoch nur die Kosten für die „Auto“-Einstellung. Beispielsweise sind gpt-4.1 oder gemini-3-pro-preview momentan nicht über die jeweiligen Drop-downs erreichbar, können aber während der Ausführung dennoch zugewiesen werden. Einige Modelle, die in Copilot CLI verfügbar sind, werden wiederum derzeit nicht in GitHub Copilot Coding Agent unterstützt, wie gpt-5.2 oder claude-opus-4.6-fast.

Die Standardumgebung GitHub Coding Agent

Der Coding Agent arbeitet in einer sogenannten „ephemeren“ Virtual Machine auf GitHub Actions. Jede Session bekommt eine frische VM – nach Abschluss wird sie verworfen.



Mission Control – das Agent-Panel (Bild 3)

Konkret: Auf einem Standard-Runner für private Repos stehen nach aktueller Dokumentation 2 CPU-Kerne, 8 GB RAM und 14 GB SSD zur Verfügung. Für Public Repos verdoppelt GitHub auf 4 Kerne und 16 GB RAM. Das Session-Zeitlimit

ist hart: maximal 59 Minuten. Kurz vor Ablauf erhält der Agent Warnungen. Zusätzlich gibt es ein zurzeit nicht genau dokumentiertes internes Command-Time-out von etwa fünf bis sechs Minuten für einzelne Befehle. Nach derartigen Fehlern wird ein „retry“ versucht. Dieses Time-out ist nicht konfigurierbar. Bezüglich der virtuellen Hardware in der Laufzeitumgebung habe ich in meinen eigenen Tests teils deutlich erhöhte Grenzen gefunden, vor allem was den Festplattenspeicher betrifft. GitHub (Microsoft) wird innerhalb der offiziell publizierten Parameter die Laufzeitumgebung durchaus anpassen. Selbst zwischen zwei Sessions für den gleichen Pull Request kann beispielsweise eine andere Prozessorgeneration gefunden werden (zum Beispiel AMD EPYC 7763 und AMD EPYC 9V74).

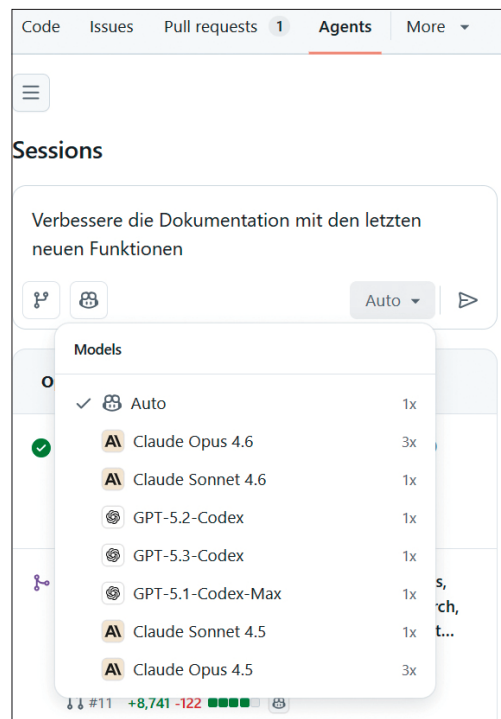
Welche Betriebssysteme unterstützt der Agent? Ausschließlich Ubuntu x64 Linux und Windows auf x64 und Arm – kein macOS. Hierzu gibt es vordefinierte Runner Images, die Versionen werden auch als „Build“ bezeichnet. Wichtig: Die integrierte konfigurierbare Firewall ist derzeit nicht kompatibel mit Windows. Momentan werden die x64-basierten Builds von GitHub und die Arm-basierten Builds von Partnern gepflegt, maßgeblich von Arm Limited selbst.

Für Windows-Builds empfiehlt GitHub Self-Hosted Runner oder „Larger Runners“ mit Azure Private Networking, wo du eigene Netzwerkkontrolle hast. Und die Programmiersprachen? Grundsätzlich sind alle möglich, die auf dem Runner installierbar sind. Die Standard-Images bringen Node.js, Python, .NET, Java, Go, Ruby und weitere mit (vollständige Liste im Repository *actions/runner-images*). GitHub empfiehlt aber, bestimmte Versionsabhängigkeiten explizit in den Setup-Steps zu pinnen.

Die Firewall: Schutz vor Data Exfiltration

Standardmäßig ist eine Firewall aktiv, die nur Zugriffe auf eine kuratierte Allowlist erlaubt: Paketquellen (NuGet, NPM, PyPI), Betriebssystem-Repos und Playwright-Hosts. Der Schutz zielt auf Data Exfiltration: Sollte der Agent durch manipulierte Eingaben zu unerwünschtem Verhalten verleitet werden, verhindert die Firewall das Übertragen von Code an externe Server.

Wichtige Einschränkungen: Die Firewall greift nur für Prozesse, die der Agent über sein Bash-Tool startet – nicht für MCP-Server und nicht für Prozesse aus den Setup-Steps. Sie funktioniert nur auf GitHub-Hosted Ubuntu-Runnern – nicht auf Windows und nicht auf Self-Hosted Runnern. Und GitHub weist explizit darauf hin, dass raffinierte Angriffe die Firewall



Modell-Auswahl bei der Task-Erstellung (Bild 4)

umgehen können – sie ist eine Schutzschicht, keine vollständige Sicherheitslösung.

Darüber hinaus stehen uns Funktionen von GitHub Advanced Security zur Verfügung, wie die GitHub-Code-Security-Analyse mit CodeQL und das Secret Scanning nach Berechtigungstokens. Diese Prüfungen laufen auf Public Repositories ohne weitere Kosten. Für Private Repositories und für erweiterte Prüfungen wie die Dependency-Prüfung gegen die GitHub Advisory Database oder eine tiefere Prüfung auf unsichere Secret-Verarbeitung kann eine zusätzliche Lizenz erworben werden.

MCP-Server und Agent Skills

Das Model Context Protocol (MCP) erweitert den Agent um Werkzeuge. Der Standard: GitHub-MCP (Issues, PRs, Code-Suche) und Playwright-MCP (E2E-Tests) sind immer dabei.

Weitere Server – etwa Azure-MCP für Bicep – konfigurierst du über Repository-Settings. Wichtig: Remote-MCP-Server müssen in der Firewall-Allowlist stehen. MCP-Server, die zusätzliche Pakete brauchen, erfordern die Installation über die *copilot-setup-steps.yml*. Aktuell unterstützt der Coding Agent keine Remote-MCP-Server mit OAuth-Authentifizierung. Für Organisationen gibt es die MCP-Allowlist: In der Einstellung *Registry only* sind nur zentral registrierte Server zugelassen.

Seit Januar 2026 sind Agent Skills GA (Generally Available) – ein offener Standard, der in Coding Agent, Copilot CLI und VS Code Copilot Agent Mode funktioniert. Skills sind Ordner mit einer *SKILL.md*-Datei, die per YAML-Frontmatter (Name und Beschreibung) definiert, wann der Skill aktiviert werden soll. Copilot liest nur die Metadaten – erst bei einem passenden Prompt wird der vollständige Skill-Body geladen. Du legst sie unter *.github/skills/* ab und baust damit spezialisierte Skills: einen Docs-Agent Skill für Dokumentation, einen Test-Agent Skill für QA, einen Security-Agent Skill für Sicherheitsanalyse. Der Unterschied zu Custom Instructions: Instructions sind allgemeine Richtlinien, die immer geladen werden. Skills sind aufgabenbezogene Fähigkeiten mit Skripten und Ressourcen, die nur bei Bedarf aktiviert werden.

Die Umgebung durch Setup-Steps anpassen: Optional, aber dringend empfohlen

Die Datei *.github/workflows/copilot-setup-steps.yml* ist technisch optional – auf GitHub-Hosted Runnern. Ohne sie versucht der Agent, per Trial and Error herauszufinden, welche Tools und Dependencies er braucht, sofern sie nicht vorhanden sind. GitHub selbst schreibt dazu in der Dokumentation: „Copilot can discover and install these dependencies itself via a process of trial and error – but this can be slow and unreliable, ►

given the non-deterministic nature of large language models.“ Bei privaten Packages scheitert das selbstverständlich.

Für Self-Hosted Runner dagegen ist die Datei unverzichtbar: Dort definierst du den *runs-on*-Wert, der den Agent auf deinen Runner lenkt. Außerdem brauchen MCP-Server, die zusätzliche Dependencies benötigen (wie *uv* oder *pipx*), die Setup-Steps zur Installation. MCP-Secrets müssen mit dem Prefix *COPILOT_MCP_* in der Copilot-Umgebung konfiguriert werden. Da der Coding Agent den GitHub- und den Playwright-MCP-Server immer aktiviert hat, müssen je nach dem Basis-Image deines Runners zum Beispiel Node.js (NPM) und Playwright selbst zur Verfügung stehen.

Die minimale Konfiguration braucht nur einen Workflow mit einem Job namens *copilot-setup-steps* und einen *runs-on*-Wert. Der Check-out ist optional – das macht der Agent automatisch. In der Praxis willst du aber mindestens sicherstellen, dass für dich relevante SDKs installiert und Dependencies restored sind.

Agenten möglichst lange laufen lassen

Während der Agent in der Regel dafür gedacht ist, genau umrissene Aufgaben zu bearbeiten, bezahlen wir momentan die gleiche Anzahl Premium Requests unabhängig davon, wie lange dieser jeweils bis zu seinem derzeitigen 59-Minuten-Maximum läuft. Darum können wir Mehrwert generieren, wenn wir den Agenten dazu ermutigen, möglichst lange zu arbeiten. Wir erreichen dies, indem wir beispielsweise eine Datei mit einem eigenen „Backlog“ im Markdown-Format pflegen und den Agenten bitten, diese schrittweise abzuarbeiten. Die Anweisungen dafür hinterlegen wir in den Copilot-Instructions. Dies lässt sich beliebig ausweiten in ein virtuelles Issue Tracking, Backlog und so weiter. Eine solche Datei legt der Copilot auch gerne für uns an und verlinkt sie in den Instructions selbst, wenn wir dies anfragen. Wir können auch über MCPs oder andere Mechanismen eine eigene Aufgabenliste einbinden. Es empfiehlt sich, in den Instructions dann beispielsweise die Anweisungen zu hinterlegen, neue Aufgaben nur bis zur 50. Minute zu beginnen und nach jeder erledigten Teilaufgabe alles mit *Report progress* zu sichern, da wir nur maximal 59 Minuten zur Verfügung haben, bevor die Laufzeitumgebung automatisch beendet wird.

Copilot Code Review: Der automatische Reviewer

Eng mit dem Coding Agent verwoben ist Copilot Code Review – ein separater Agent, der Pull Requests automatisch reviewt. Seit September 2025 gibt es dafür eine eigenständige Repository-Regel, unabhängig von *Require a pull request before merging*. Du aktivierst sie unter *Settings | Rules | Rule-sets* und kannst wählen, ob Copilot auch bei neuen Pushes und Draft-PRs reviewt. Besonders für Coding-Agent-PRs ist die Option *Review draft pull requests* sinnvoll – so werden Probleme früh erkannt.

Ein wichtiges Detail: Copilot hinterlässt immer nur „Comment“-Reviews – nie „Approve“ oder „Request Changes“. Es blockiert also nie das Mergen und zählt nicht als erforderliche Freigabe. Die Reviews kombinieren LLM-Erkenntnisse mit deterministischen Tools wie ESLint und CodeQL. Und besonders

● Tabelle 1: Kosten für Copilot Coding Agent

Plan	Preis	Inkludierte Premium Requests
Free	keine Monatsgebühr	50
Pro	10 US-Dollar pro Monat	300
Pro+	39 US-Dollar pro Monat	1500
Business	19 US-Dollar pro User und Monat	300
Enterprise	39 US-Dollar pro User und Monat	1000
Überschreitung	0,04 US-Dollar pro Request	
Coding Agent Session	1 Premium Request × Model-Multiplikator	
Steering	1 zusätzlicher Premium Request	
Code Review	1 Premium Request	
Auto-Model-Selection	10 Prozent Rabatt	

elegant: Aus einem Review-Kommentar heraus kannst du per *Implement suggestion* den Coding Agent die Korrektur direkt umsetzen lassen – ein nahtloser Kreislauf von Review zu Fix.

Empfehlung: Aktivieren! Die automatischen Reviews kosten je einen Premium Request, liefern aber frühes Feedback und ergänzen menschliche Reviews. Sie ersetzen diese jedoch nicht – Copilot kann keine Pull Requests freigeben.

Larger Runner versus Self-Hosted Runner

Wenn die Standard-Runner nicht ausreichen, gibt es zwei Wege: Larger Runners (von GitHub verwaltet) und Self-Hosted Runners (von dir betrieben). Die Wahl hängt vom Use Case ab.

- **Larger Runners** bieten von 4 bis 96 vCPUs, bis zu 384 GB RAM und 2 TB SSD – von GitHub verwaltet, ohne Infrastruktur-Aufwand: Du änderst einfach den *runs-on*-Wert. Die integrierte Firewall funktioniert weiterhin (auf Ubuntu). Über Azure Private Networking erreichst du auch interne Ressourcen. Abgerechnet wird zunächst über die inkludierten Actions Minutes relativ zu den Kosten: 4 Kerne kosten etwas das Doppelte, 8 Kerne das Vierfache.
- **Self-Hosted Runners** (unterstützt seit Oktober 2025) gibst du volle Kontrolle: Betrieb im privaten Netzwerk, Zugriff auf interne Registries und Datenbanken, spezielle Hardware. GitHub empfiehlt ARC (Actions Runner Controller) auf Kubernetes – als Minimum kannst du eine einzelne Runner-Instanz als Self-Hosted Runner laufen lassen. Auch der klassische Self-Hosted-Runner-Client funktioniert. Entscheidend ist, dass die Runner ephemere sind – also nach jeder Session verworfen werden. ARC macht das automatisch über dynamische Pod-Provisionierung.

Drei zentrale Unterschiede zu den Larger Runners kennzeichnen die Self-Hosted Runners: Erstens ist die integrierte

Firewall auf Self-Hosted Runnern nicht verfügbar – du musst eine eigene Netzwerkkontrolle auf Infrastrukturebene implementieren. Zweitens bist du selbst für Updates und Sicherheit der Runner-Images verantwortlich. Drittens fallen keine Actions Minutes an, dafür trägst du die Infrastrukturkosten.

Als konkretes Beispiel kannst du auf deiner eigenen Hardware die eigene GPU nutzen, sofern du eine zur Verfügung stellst. Im GitHub-Repo zum Artikel [1] stelle ich euch zwei recht umfangreiche Beispielkonfigurationen von Setup-Steps und allem, was ihr auf eurer Hardware benötigt, zur Verfügung: einmal für einen einfachen Runner unter WSL und einmal für eine ARC-Konfiguration mit Kubernetes

Was kostet der Spaß?

Der Coding Agent verbraucht Premium Requests und GitHub Actions Minutes. Seit Juli 2025 kostet jede Agent-Session genau einen Premium Request multipliziert mit einem Modell-Multiplikator – unabhängig von der Komplexität. Steering-Kommentare während einer Session kosten extra. Der Modell-Multiplikator ist beispielsweise 3 für das Opus-4.6-Model von Anthropic. Premium Requests teilst du dir mit anderen Copilot-Features (Chat, Code Review, CLI). Überschreitest du dein Kontingent, kostet jeder weitere Request 0,04 US-Dollar. Ungenutzte Requests verfallen am Monatsende. Tipp: Auto-Model-Selection bringt 10 Prozent Rabatt. Dazu kommen die Actions Minutes: Für öffentliche Repositories sind Standard-Runner kostenlos, für private Repositories kosten diese Geld [2], ein „Larger Runner“ entsprechend mehr. Jeder GitHub-Account hat eine bestimmte Anzahl von inkludierten Minuten, derzeit beginnt dies bei 2000 in der Free-Version und reicht bis zu 50000 mit GitHub Enterprise Cloud. Self-Hosted Runner kosten derzeit keine Actions Minutes. Details zeigt [Tabelle 1](#).

Was funktioniert gut, was funktioniert weniger gut?

GitHub beschreibt den Sweet Spot so: „Low-to-medium complexity tasks in well-tested codebases“. Konkret bedeutet dies wie überall bei Agenten, dass eine gute Beschreibung und gute Tests zu guten Ergebnissen führen. Was aber den Coding Agent besonders auszeichnet, ist, dass er dafür da ist, dass eine klar umrissene Aufgabe komplett erledigt werden kann – obwohl wir gegensteuern (steering) können, wenn etwas danebenläuft. Dies sollte uns nicht daran hindern, ihn auch für komplexere oder sogar fachfremde Dinge zu nutzen. So habe ich den Coding Agent schon „missbraucht“, um ein komplettes Comic-Buch zu planen und zu erstellen. Im Moment ist er auf jeden Fall eine hilfreiche Unterstützung für unser Team.

Mein Ratschlag ist, in bestehenden Repositories klein anzufangen, die erste kleine Änderung, den ersten kleinen Bug-Fix zuzuweisen, die Dokumentation auffrischen zu lassen, wenn sich ein Interface geändert hat, oder beispielsweise ein neues API ins eigene Projekt zu integrieren. In besonders großen und unübersichtlichen Repositories müsst ihr dem Coding Agent oft erst einmal schrittweise helfen, die für ihn notwendigen Informationen in strukturierten Markdown-Dateien zu sammeln. Die Dokumentation für einen Coding Agent hat oft andere Schwerpunkte als die für einen menschlichen Konsu-

menten. Dies ist ein Teil des „Onboarding“, den ihr für jede Art von Coding Agent einplanen solltet.

Wenn ihr den maximalen Nutzwert erreichen möchtet, ist es parallel dazu spannend, in einem Projekt von null anzufangen und es schrittweise über den Coding Agent aufbauen zu lassen. Dies kann funktionieren, wenn die To-do-Liste sehr klar definiert und das Issue Tracking unabhängig vom GitHub-eigenen konzipiert ist. Denn der Coding Agent kann derzeit keine GitHub Issues erstellen oder bearbeiten. Die erzwungene Laufzeit von maximal 59 Minuten erlaubt zwar keinen sogenannten „Ralph Loop“, dennoch lassen sich beachtliche Ergebnisse erzielen. Im Vergleich zu anderen Coding Agents ist der GitHub Coding Agent, genauer gesagt GitHub Copilot Pro oder Pro+, übrigens momentan extrem kostengünstig. Ich habe durchaus in einem Monat schon mehr Tokens mit einer Copilot Pro Subscription „verbraucht“, als selbst eine Claude Code Max abgedeckt hätte.

Fazit

GitHub Copilot Coding Agent ist ein asynchroner Teamkollege für gut definierte Aufgaben in getesteten Codebases. Seine harten Grenzen – 59 Minuten Session, 5 bis 6 Minuten Command-Time-out, 14 GB Disk (Standard), kein macOS – muss man kennen und einplanen. Wer produktiv einsteigen will, investiert in drei Dinge: gute Instructions-Dateien, eine solide Testbasis und eine durchdachte Setup-Steps-Konfiguration. Die Kombination mit automatischem Code Review schafft einen nahtlosen Kreislauf von Issue über Pull Request bis zum Merge.

Die Richtung ist klar: Die Integration eines multimodalen Workflows mit Copilot in der IDE wie VS Code und in der Kommandozeile wie mit Copilot CLI wird durch die Architektur immer weiter vorangetrieben. Mit Agentic Memory (Public Preview seit Januar 2026) lernt der Coding Agent aus jeder Session und teilt dieses Wissen mit Code Review und dem CLI – Cross-Feature-Gedächtnis mit 28-Tage-TTL und Opt-in.

Parallel dazu entwickelt sich VS Code zur Multi-Agent-Plattform: Seit v1.109 laufen Claude, Codex und Copilot-Agents nebeneinander, mit Sub-Agents, Hand-offs und einer einheitlichen Sessions-Ansicht. Der Coding Agent ist kein isoliertes Feature mehr, sondern ein Baustein in einem wachsenden Ökosystem. Der beste Zeitpunkt zum Einstieg ist jetzt. ■

[1] *Beispielcode zum Artikel,*

<https://github.com/MSDNAndi/dotnetpro>

[2] *Actions runner pricing, <https://docs.github.com/en/billing/reference/actions-runner-pricing>*



Andreas Erben

ist seit über 30 Jahren in unserer Branche tätig. Er kümmert sich bei der daenet GmbH – ACP Digital um angewandte KI, und das schon länger, als es cool ist. Von Microsoft wurde er als MVP und Regional Director ausgezeichnet.
andreas.erben@daenet.com

AGENTIC CODING MIT CLAUDE CODE

Claude, mach mal mit

Ein Erfahrungsbericht aus der Praxis zum Einsatz von Coding-Agenten – mit Überraschungen, Stärken und einem ehrlichen Blick auf die Grenzen.

Als ich 2024 das Buch „ChatGPT in Softwareprojekten“ schrieb, war meine zentrale These klar: Große Sprachmodelle sind keine Programmierer. Sie sind Werkzeuge, die bei der Definition von Anforderungen, Qualitätssicherung und Dokumentation unterstützen können, aber beim eigentlichen Coding stoßen sie schnell an Grenzen. Zu viel Kontext geht verloren, es gibt zu wenig Verständnis für die Gesamtarchitektur und zu viele Halluzinationen bei komplexen Zusammenhängen. Damals empfahl ich, KI vor allem für das Drumherum einzusetzen: Anforderungen strukturieren, Testfälle ableiten, Dokumentation generieren. Das reine Programmieren? Besser in Menschenhand.

Zwei Jahre später muss ich diese Einschätzung differenzieren. Nicht weil Sprachmodelle plötzlich perfekte Programmierer geworden wären, sondern weil eine neue Kategorie von Werkzeugen entstanden ist, die das Modell in einen echten Entwicklungsworkflow einbetten. Coding-Agenten wie Claude Code, GitHub Copilot oder OpenAI Codex haben die Spielregeln verändert. Sie schreiben nicht einfach Code auf Zuruf, sondern verstehen Projektstrukturen, planen Änderungen über mehrere Dateien hinweg und verifizieren ihre eigenen Ergebnisse. Das ist ein qualitativer Sprung, den man erst in der täglichen Praxis wirklich begreift.

Besonders Claude Code hat sich in meinem beruflichen Alltag als erstaunlich leistungsfähiger Partner etabliert. In diesem Artikel teile ich meine Erfahrungen: Was das Tool kann, wo es überzeugt, wo es an Grenzen stößt und warum am Ende trotzdem die Fachkenntnis des Entwicklers den Unterschied macht.

Dabei hat sich die KI-Agenten-Landschaft in kurzer Zeit rasant entwickelt. Während Copilot als Inline-Autocomplete den Anfang machte und vor allem Boilerplate-Code beschleunigte, schlagen die Agenten der nächsten Generation nicht nur einzelne Zeilen vor, sondern übernehmen ganze Arbeitsabläufe. Der Unterschied ist vergleichbar mit dem Sprung von einer Rechtschreibprüfung zu einem Lektor, der den gesamten Text versteht und strukturell verbessert.

Was ist Claude Code eigentlich?

Claude Code ist kein Chatbot mit Code-Funktion und kein Autocomplete-Plug-in. Es ist ein agentisches Entwicklungswerkzeug, das primär im Terminal oder der IDE lebt und die gesamte Codebasis eines Projekts versteht. Der entscheidende Unterschied zu klassischen KI-Assistenten liegt im sogenannten

```

Claude Code v2.1.42
Opus 4.6 · Claude Max
~/Repo/temp/ionic-conference-app
[*] [*] [*] · Share Claude Code and earn €20 of extra usage · /passes

Welcome to Opus 4.6

> In unserer PWA (Angular/Ionic) existiert eine Komponente zur Video-Wiedergabe.
  Erweitere diese so, dass optional, Videos auch automatisch abgespielt werden
  können.

● Explore(Find video component in codebase)
  └─ Search(pattern: "&lt;video")

      Search(pattern: "video")
      +2 more tool uses (ctrl+o to expand)
      ctrl+b to run in background

* Channeling...

> |
  
```

Claude Code im Terminal (Bild 1)

agentischen Loop: Claude Code arbeitet in einem Kreislauf aus Kontext sammeln, Aktion ausführen und Ergebnis verifizieren.

Konkret bedeutet das: Gibt man Claude Code eine Aufgabe, etwa „Implementiere ein REST-API für die Benutzerverwaltung mit Authentifizierung“, dann liest es zunächst die vorhandene Codebasis, versteht die verwendeten Frameworks und Konventionen, plant die notwendigen Änderungen, setzt sie über mehrere Dateien hinweg um und prüft anschließend, ob die Tests durchlaufen. Das ist fundamental anders als eine einmalige Code-Generierung in einem Chat-Fenster. Es ist ein iterativer Prozess, bei dem der Agent aus seinen eigenen Zwischenergebnissen lernt und den Kurs korrigiert.

Die technische Basis bilden Anthropic's Sprachmodelle: Claude Opus für komplexe Planungs- und Architekturaufgaben, Claude Sonnet für die tägliche Entwicklungsarbeit [1].

Ein erweitertes Denkmögen, das sogenannte Extended Thinking, erlaubt es dem Modell zudem, bei besonders komplexen Problemen mehr Rechenzeit in die Analyse zu investieren, bevor eine Antwort generiert wird (vergleiche Bild 1). In der Praxis merkt man das: Schwierige Architekturaufgaben profitieren sichtbar von dieser zusätzlichen „Bedenkzeit“.

Mehr als ein Assistent: Agenten und Sub-Agenten

Was Claude Code von einigen anderen Coding-Assistenten unterscheidet, ist die Fähigkeit zur Orchestrierung. Bei komplexen Aufgaben delegiert der Haupt-Agent Teilaufgaben an sogenannte Sub-Agenten. Diese arbeiten isoliert in eigenen Kontextfenstern: Einer recherchiert die Codebasis, ein ande-

rer führt Code-Reviews durch, ein dritter implementiert parallel. Dieses Verhalten muss nicht explizit konfiguriert werden; Claude Code entscheidet selbst, wann eine Delegation sinnvoll ist. Für den Nutzer ist das transparent einsehbar: Im CLI wird angezeigt, wenn ein Sub-Agent gestartet wird und welche Aufgabe er übernimmt. Das klingt abstrakt, wird aber schnell greifbar: Stellt man Claude Code vor die Aufgabe, für ein bestehendes Modul ein Refactoring vorzunehmen, kann es eigenständig einen Sub-Agenten beauftragen, um zunächst alle Abhängigkeiten zu analysieren. Ein zweiter Sub-Agent prüft die Testabdeckung. Erst danach beginnt der Haupt-Agent mit der eigentlichen Umsetzung, informiert durch die Erkenntnisse der Vorarbeit. Was hier im Hintergrund passiert, ähnelt der Arbeitsweise eines eingespielten Entwicklerteams: Arbeitsteilung, Kommunikation, Synthese.

Ein zentrales Element ist die *CLAUDE.md*-Datei, die im Root-Verzeichnis des Git-Repos abgelegt wird. Sie fungiert als persistenter Projektkontext. Vergleichbar ist das mit einer lebenden Dokumentation, die Claude Code bei jeder Sitzung einliest. Hier lassen sich Konventionen, Architekturentscheidungen und projektspezifische Regeln hinterlegen:

```
# Projekt: Kundenportal API

## Architektur
- .NET 8 mit Minimal APIs
- MongoDB als Datenspeicher
- JWT-basierte Authentifizierung

## Konventionen
- Async/Await durchgängig verwenden
- Repository-Pattern für Datenzugriff
- Keine Business-Logik in Controllern

## Wichtige Hinweise
- Alle Endpoints benötigen Autorisierung
- Logging über Serilog
- Tests mit xUnit und FluentAssertions
```

Diese Datei sorgt dafür, dass Claude Code nicht in jeder Sitzung bei null beginnt, sondern den Projektkontext versteht und konsistente Entscheidungen trifft. In der Praxis hat sich gezeigt, dass eine gut gepflegte *CLAUDE.md* den Unterschied zwischen brauchbaren und exzellenten Ergebnissen ausmacht, ähnlich wie ein gutes Briefing für einen externen Entwickler.

CLI, IDE und mehr: Wo Claude Code zu Hause ist

Claude Code lässt sich auf verschiedenen Wegen nutzen. Der primäre Zugang ist das Terminal – eine bewusste Designentscheidung, die ich sehr schätze. Im CLI sieht man exakt, welche Dateien gelesen, welche Befehle ausgeführt und welche Änderungen vorgenommen werden. Diese Transparenz ist kein Nebeneffekt, sondern ein Kernmerkmal. Wer wissen will, was sein KI-Assistent tut, findet hier volle Einsicht – kein Black-Box-Gefühl, sondern nachvollziehbare Arbeitsschritte.

Darüber hinaus integriert sich Claude Code nahtlos in VS Code und JetBrains-IDEs. In VS Code etwa erscheinen vor-

geschlagene Änderungen als Inline-Diffs, die man einzeln annehmen oder ablehnen kann (Bild 2). Das fühlt sich an wie ein Code-Review eines erfahrenen Kollegen: Man behält die volle Kontrolle, profitiert aber von der Geschwindigkeit des Assistenten. Wer lieber außerhalb der IDE arbeitet, kann Claude Code auch als Desktop-App oder über die Weboberfläche nutzen. Die Konfiguration, etwa in Gestalt der *CLAUDE.md* oder installierter MCP-Server, wird über alle Oberflächen hinweg synchronisiert.

Besonders spannend wird es bei der Automatisierung. Über sogenannte Hooks lassen sich Aktionen an bestimmte Ereignisse koppeln: Nach jeder Dateiänderung automatisch den Linter laufen lassen, bei jedem Commit eine Sicherheitsprüfung oder Vulnerability-Scans bei neuen Dependencies anstoßen. In Kombination mit GitHub Actions kann Claude Code zudem direkt in CI/CD-Pipelines eingebunden werden. Es reagiert auf @claude-Mentions in Pull Requests, führt Code-Reviews durch oder implementiert Issues eigenständig. Solche Automatisierungen machen aus einem einzelnen Werkzeug einen integrierten Bestandteil des Entwicklungsprozesses.

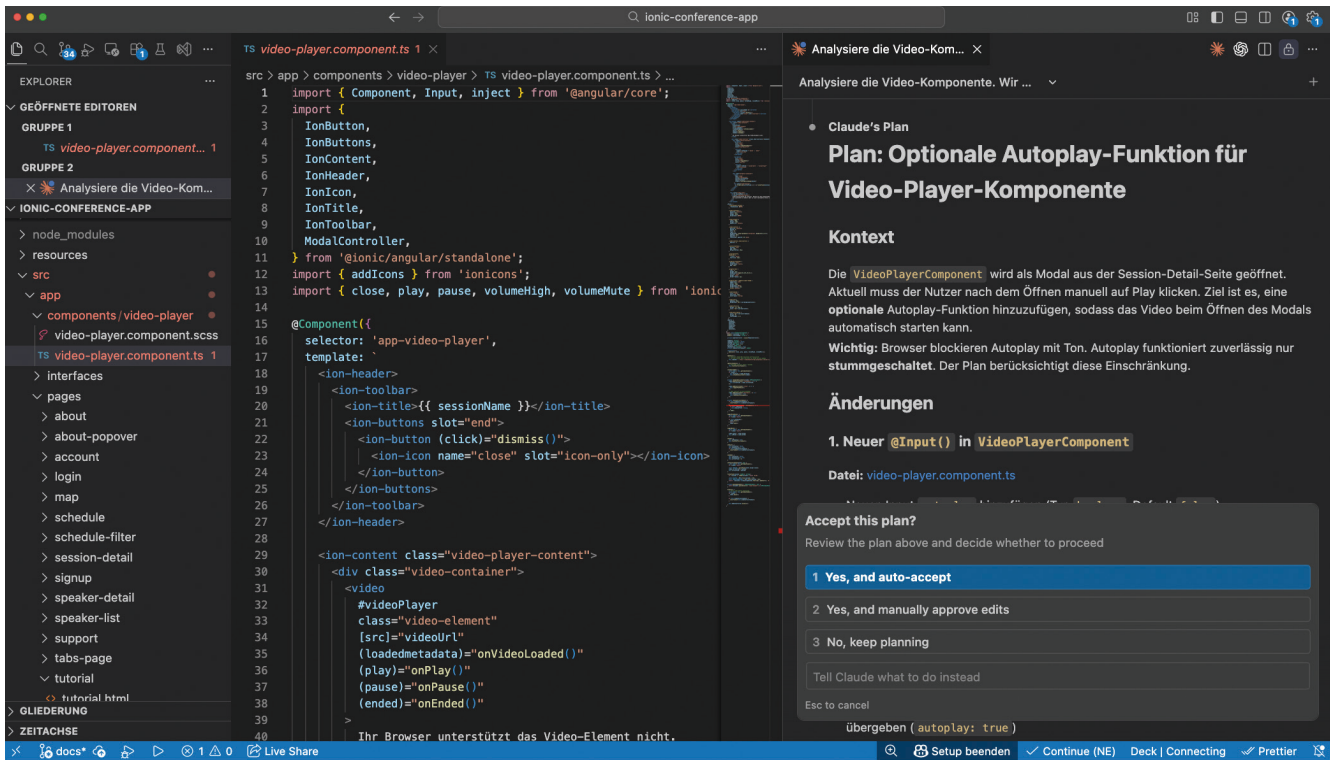
Ich durfte im direkten Vergleich auch die Research-Preview von OpenAI Codex testen und setzte es bei kleineren Web-APIs und Erweiterungen ein. Beide Tools haben ihre Berechtigung, doch im direkten Vergleich überzeugten mich bei Claude Code vor allem drei Aspekte: die CLI-Transparenz, die damit verbundenen Automatisierungsmöglichkeiten und die Kombination aus tiefgreifender Planung, Kontextgewinnung und zielgenauer Umsetzung. Während Codex bei isolierten Aufgaben schnell gute Ergebnisse liefert, spielt Claude Code seine Stärken besonders bei projektübergreifenden, mehrstufigen Aufgaben aus.

Wo ich die Stärken sehe

Nach mehreren Monaten intensiver Nutzung kristallisieren sich klare Stärken heraus. An erster Stelle steht die Planungskompetenz. Claude Code analysiert erst, bevor es handelt. Es liest die Codebasis, versteht Abhängigkeiten und erstellt einen Plan, bevor eine einzige Zeile Code geschrieben wird. In der Praxis führt das zu deutlich konsistenteren Ergebnissen als bei Tools, die sofort mit der Implementierung beginnen. Man kann diesen Plan sogar im sogenannten Plan-Modus explizit einfordern und prüfen, bevor man die Umsetzung freigibt.

Dabei beeindruckt weniger das reine Kontextfenster von 200 000 Tokens als vielmehr die Fähigkeit, den relevanten Kontext gezielt zu erkennen und zu nutzen. Claude Code liest nicht einfach wahllos Dateien ein, sondern es versteht, welche Teile der Codebasis für die aktuelle Aufgabe relevant sind, und fokussiert seine Analyse entsprechend. Noch bemerkenswerter ist die Fähigkeit, bestehende Muster und Konventionen im Code zu imitieren. Existiert im Projekt bereits eine ähnliche Komponente, etwa ein Repository für eine andere Entität, findet Claude Code diese eigenständig und nutzt sie als Vorlage.

Die neue Implementierung fügt sich dadurch nahtlos in die bestehende Architektur ein: gleiche Namenskonventionen, gleiche Fehlerbehandlung, gleiche Patterns. Das Ergebnis sind kohärente Multi-File-Änderungen über Controller, ►



Claude Code in VS Code: Vorgeschlagene Änderungen erscheinen als Inline-Diffs, die sich einzeln annehmen oder ablehnen lassen (Bild 2)

Services, Repositories und Tests hinweg. Wer diese Pattern-Erkennung noch gezielter nutzen möchte, referenziert die relevanten Vorlagen explizit im Prompt, doch oft reicht bereits die automatische Erkennung.

Das Checkpoint-System bietet ein Sicherheitsnetz: Jede Änderung wird als Snapshot gespeichert und lässt sich jederzeit zurückrollen. Das senkt die Hemmschwelle erheblich, Claude Code auch bei produktivem Code einzusetzen. In meinem Workflow nutze ich das regelmäßig: Einen experimentellen Ansatz ausprobieren, das Ergebnis bewerten und bei Bedarf in Sekunden zum Ausgangszustand zurückkehren, ohne Git-Stash-Akrobatik.

Die sogenannten Skills, wiederverwendbare Workflows als Slash-Commands, erlauben es, definierte Qualitätskriterien zu prüfen oder ganze Release-Prozesse zu orchestrieren. Ein `/review`-Skill kann beispielsweise Code-Qualität, Sicherheit und Performance-Aspekte nach definierten Regeln bewerten. In Kombination mit MCP-Servern (Model Context Protocol) lässt sich Claude Code zudem an externe Systeme anbinden: Datenbanken, Jira, Slack, Sentry oder eigene APIs [2]. Dadurch wird es zum Knotenpunkt im Entwicklungsworkflow, nicht nur zum Code-Generator. Die Unix-Philosophie der kleinen, spezialisierten Werkzeuge, die sich beliebig kombinieren lassen, ist hier konsequent umgesetzt.

Ein weiterer Pluspunkt: die Versionierung und das Teilen von Konfigurationen. Da `CLAUDE.md`, Skills und Hook-Definitionen als Dateien im Repository liegen, profitiert das gesamte Team davon. Ein neuer Kollege klonst das Repository und hat sofort Zugang zu allen Konventionen, Workflows und Automatisierungen, inklusive der Projektdokumentation, die Claude Code bei jeder Sitzung einliest. Das reduziert Onboarding-Zeit und sorgt für Konsistenz im Team.

Wo es noch hakt

Ehrlichkeit gehört zu einer fundierten Bewertung dazu, und Claude Code ist bei Weitem kein perfektes Werkzeug. Die größte Limitation ist und bleibt das Kontextfenster. Auch wenn 200 000 Tokens beeindruckend klingen: Bei sehr großen Codebasen mit Hunderten von Dateien muss man strategisch arbeiten: gezielt Dateien einbeziehen, Sub-Agenten nutzen, den Kontext durch Skills und `CLAUDE.md`-Dateien strukturieren. Wer einfach „mach mal“ sagt, wird bei komplexen Projekten enttäuscht.

Halluzinationen sind seltener als bei reinen Chat-Modellen, aber sie existieren. Besonders bei Randfällen, unklaren Anforderungen oder wenig verbreiteten Bibliotheken kann Claude Code falsche Annahmen treffen. Die Verifizierungsphase im agentischen Loop fängt vieles ab, aber nicht alles. In einem konkreten Projekt generierte Claude Code eine Datenbankabfrage, die syntaktisch korrekt war, aber eine falsche Join-Bedingung enthielt. Ohne manuellen Review wäre das in der Produktion gelandet. Solche Fälle unterstreichen, warum automatisierte Tests und Code-Reviews weiterhin unverzichtbar sind.

Die Kostenfrage ist ebenfalls relevant. Claude Opus, das leistungsfähigste Modell, verbraucht bei komplexen Aufgaben schnell viele Tokens. Wer eine umfangreiche Refactoring-Session durchführt, sollte die Token-Kosten im Blick behalten. Für den täglichen Einsatz ist Sonnet oft die bessere Wahl, schneller und günstiger, bei den meisten Aufgaben kaum weniger leistungsfähig. Anthropic bietet verschiedene Tarife an, vom Einzelnutzer bis zum Enterprise-Plan, aber die Kosten skalieren mit dem Verbrauch.

Und schließlich: Die Abhängigkeit von einem Cloud-Anbieter. Claude Code benötigt in der Standardkonfiguration eine Verbindung zu Anthropics Servern. Verfügbarkeitsaus-

fälle, Preisänderungen oder Änderungen am API können den Workflow direkt beeinflussen. Wer in hochsensiblen Bereichen arbeitet, muss zudem prüfen, ob die Datenverarbeitung den eigenen Compliance-Anforderungen entspricht. Anthropic betont zwar, dass eingesendeter Code nicht für das Training verwendet wird, aber die rechtliche Bewertung hängt vom jeweiligen Unternehmenskontext ab.

Allerdings bietet Claude Code hier eine relevante Flexibilität: Das Tool ist nicht an Anthropic's eigene Modelle gebunden. Über die Konfiguration lassen sich auch andere LLM-Anbieter einbinden, einschließlich lokal gehosteter Open-Source-Modelle. Wer mit Claude Code beginnt und die Arbeitsweise, Skills und Hooks in seinem Projekt etabliert hat, kann für ein datenschutzsensibles Vorhaben auf ein On-Premises-Modell umsteigen, ohne den gesamten Workflow neu aufbauen zu müssen. Die investierte Konfigurationsarbeit, *CLAUDE.md*, Skills und Hooks bleiben erhalten. Das macht den Einstieg mit Anthropic's Cloud-Modellen risikoarm: Man profitiert von der Leistungsfähigkeit von Claude und kann bei Bedarf datenschutzkonform auf eine lokale Lösung wechseln.

Claude Code versus OpenAI Codex: Ein Vergleich

Beide Tools verfolgen den agentischen Ansatz, unterscheiden sich aber in der Philosophie. Claude Code setzte bereits von der ersten Version an auf den „Assistenz“-Ansatz und damit auf eine tiefe IDE-Integration und ein offenes Erweiterungsmodell über MCP-Server und Skills. OpenAI Codex bietet

eine cloudbasierte Sandbox-Umgebung, die besonders für isolierte Aufgaben gut funktioniert. Meiner Erfahrung nach eignet sich Codex gut für schnelle Prototypen und kleinere Web-APIs, während Claude Code bei projektweiten Änderungen, bestehenden Codebasen und teamorientierten Workflows seine Stärken ausspielt.

Warum Fachkenntnis unverzichtbar ist

Und damit sind wir beim vielleicht wichtigsten Punkt, einem, der sich wie ein roter Faden durch meine gesamte Arbeit mit KI-Werkzeugen zieht: Claude Code ist ein Verstärker, kein Ersatz. Es macht gute Entwickler produktiver, aber es macht niemanden zum guten Entwickler.

Das beginnt bei der Softwarearchitektur: Wer die Struktur seines eigenen Projekts nicht versteht, wer nicht weiß, welche Schichten existieren, wie die Abhängigkeiten laufen, welche Patterns verwendet werden, kann einem Coding-Agenten keine sinnvollen Aufgaben stellen. Ein Prompt wie „Entwickle Modul XYZ“ reicht schlicht nicht aus. Der Agent braucht Kontext, Referenzen und Präzision. Ein wirksamer Prompt sieht in der Praxis eher so aus: „Wir haben eine bestehende Integration für den Service *PaymentProvider* in der Klasse *PaymentIntegration.cs*. Implementiere auf dieser Basis eine analoge Integration für den Service *ShippingProvider*, die API-Dokumentation findest du unter <https://shipping.example/docs>.“ Je konkreter die Referenz auf bestehenden Code, desto besser das Ergebnis. ►

DWX ACADEMY

Hands-on Workshops
und Weiterbildung für
Software-Entwickler
und -Architekten

AI-DRIVEN SOFTWARE DEVELOPMENT MIT GPT UND CO.

- GPT-Modelle und ihre Unterschiede
- AI-Tools für Software-Entwickler
- Grundlagen des Prompt Engineering
- Verwenden der OpenAI API
- AI Search und Vektordatenbanken
- Eigene LLMs bereitstellen mit Azure OpenAI
- Entwicklung eigener Assistenten
- Anbindung Backendsysteme

2 TAGE
INHOUSE ODER
REMOTE

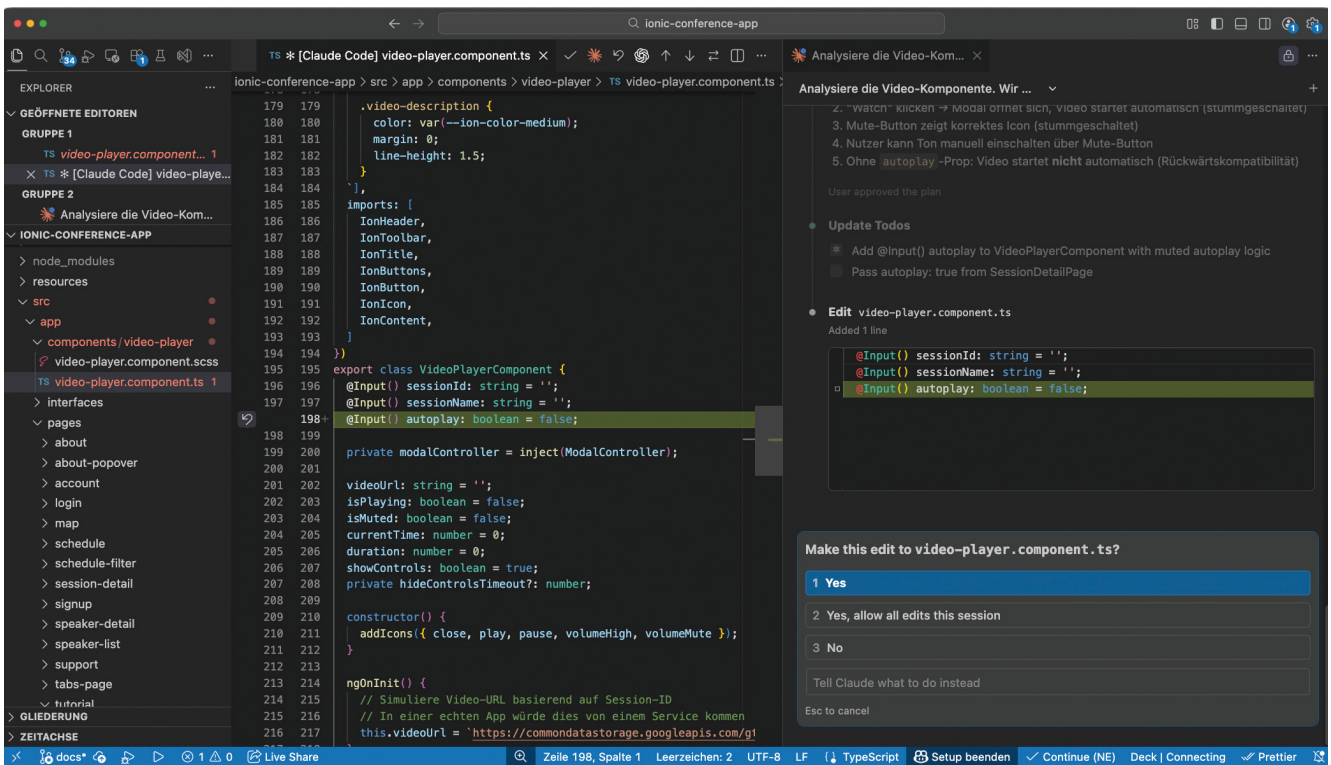


JÖRG NEUMANN



Deine Ansprechpartnerin: Susanne Herl
+49 731 88005-8835 • susanne.herl@ebnermedia.de

developer-world.de/dwx-academy



Komplexe Aufgaben werden als To-do-Listen geführt (Bild 3)

Das bedeutet: Der Entwickler muss wissen, welche Klassen, Services und Patterns bereits existieren, um sie gezielt referenzieren zu können. Wer Architekturentscheidungen nicht beurteilen kann, wird die Vorschläge von Claude Code nicht sinnvoll einordnen können. Und wer Code-Reviews nicht beherrscht, wird die generierten Implementierungen nicht zuverlässig validieren können. Das klingt selbstverständlich, wird in der Euphorie um KI-Tools aber erstaunlich oft vergessen.

Das ist exakt die These, die ich bereits in „ChatGPT in Softwareprojekten“ [3] vertreten habe, und sie gilt heute mehr denn je. Die Werkzeuge sind mächtiger geworden, aber die Grundanforderung bleibt: Anforderungsdefinition, Architekturverständnis und Qualitätssicherung sind menschliche Kompetenzen, die durch KI unterstützt, aber nicht substituiert werden können. Ein Prompt ist letztlich nichts anderes als eine Anforderungsspezifikation, und deren Qualität bestimmt die Qualität des Ergebnisses.

In der Praxis zeigt sich das besonders dann, wenn Claude Code auf Probleme stößt, die es nicht versteht. Ein erfahrener Entwickler erkennt, wann der Agent in die falsche Richtung läuft, und korrigiert den Kurs. Ein unerfahrener Nutzer akzeptiert möglicherweise eine technisch funktionierende, aber architektonisch fragwürdige Lösung, mit langfristigen Konsequenzen für Wartbarkeit und Skalierbarkeit. Das Werkzeug verstärkt also die Fähigkeiten, die bereits vorhanden sind, im Guten wie im Schlechten.

Ein konkretes Beispiel: Claude Code schlug in einem Projekt vor, eine komplexe Geschäftslogik direkt in den API-Controller zu verlagern – technisch funktional, aber ein klarer Verstoß gegen das Single-Responsibility-Prinzip. Wer das Prinzip kennt, erkennt das sofort und weist den Agenten an, die Logik in einen dedizierten Service auszulagern. Wer es

nicht kennt, hat plötzlich einen monolithischen Controller, der bei der nächsten Erweiterung zum Problem wird. Genau hier liegt die Trennlinie zwischen produktiver Nutzung und blindem Vertrauen.

Die Überraschung: Design für Nichtdesigner

Ein Aspekt, den ich nicht erwartet hatte, verdient eine eigene Erwähnung: Claude Code kann ansprechende Benutzeroberflächen generieren. Für Entwickler, die sich, Hand aufs Herz, mit UI/UX-Design schwertun, ist das ein echter Gamechanger.

In einem konkreten Projekt benötigte ich einen funktionalen Prototyp für ein internes Dashboard. Normalerweise hätte ich ein rudimentäres Bootstrap-Layout zusammengebaut und darauf gehofft, dass es „irgendwie“ professionell wirkt. Stattdessen ließ ich Claude Code eine Angular-Komponente mit Tailwind CSS generieren, und das Ergebnis war überraschend durchdacht: konsistente Abstände, sinnvolle Farbgebung, responsive Breakpoints und eine Informationshierarchie, die tatsächlich funktionierte.

Besonders beeindruckend war dabei, dass Claude Code nicht einfach ein generisches Template ausspuckte, sondern den Kontext der Anwendung berücksichtigte. Farben, die zum Corporate Design passten. Navigationselemente, die im Hinblick auf die Informationsarchitektur des Projekts sinnvoll waren. Responsive Ansätze, die für die Zielgruppe, in diesem Fall Außendienstler mit Tablets im Außendienst, optimiert waren. All das auf Basis einer präzisen Beschreibung in der *CLAUDE.md* und einem knappen Prompt.

Das ersetzt keinen professionellen Designer. Aber für Prototypen, interne Tools, MVPs oder Entwickler-Dashboards ist die Qualität mehr als ausreichend. Gerade in kleinen Teams ohne dedizierte Designer schließt das eine Lücke, die bisher

oft mit „Gut genug“-Lösungen gefüllt wurde. Und ganz ehrlich: Manch ein Ergebnis war besser als das, was ich selbst in deutlich mehr Zeit zusammengebaut hätte.

Fazit: Ein Tool, das den Werkzeugkasten verändert

Claude Code ist das erste KI-gestützte Entwicklungswerkzeug, das ich ohne Vorbehalte für den produktiven Einsatz empfehlen kann, mit einer wichtigen Einschränkung: Es richtet sich an professionelle Entwickler, die wissen, was sie tun. Es ist kein Low-Code-Tool für Nichtprogrammierer, sondern ein Produktivitätsverstärker für erfahrene Teams.

Die Kombination aus tiefem Codebasis-Verständnis, „agentischem“ Arbeiten, transparentem CLI und flexibler Erweiterbarkeit über Skills, Hooks und MCP-Server hebt Claude Code von reinen Autocomplete-Lösungen deutlich ab. Wer bereit ist, sich auf das Tooling einzulassen, und seine Projekte mit *CLAUDE.md* und passenden Konfigurationen vorbereitet [4], wird eine spürbare Produktivitätssteigerung erleben. Die Lernkurve ist überschaubar, der Return on Investment schnell spürbar.

Gleichzeitig bleibt der fundamentale Grundsatz bestehen: KI-Werkzeuge sind genau das – Werkzeuge. Sie erweitern unsere Möglichkeiten, ersetzen aber nicht unser Urteilsvermögen. Die Entwicklung geht weiter: Agent-Teams, größere Kontextfenster, tiefere IDE-Integration – die nächsten Monate werden weitere Fortschritte bringen. Bereits jetzt experimentiert Anthropic mit Agent-Teams, bei denen mehrere Claude-Code-Instanzen parallel an verschiedenen Teilauf-

gaben arbeiten und sich über gemeinsame Task-Listen koordinieren (Bild 3). Das Potenzial ist enorm.

Wer jetzt anfängt, sich mit Agentic Coding vertraut zu machen, investiert in eine Fähigkeit, die in der modernen Softwareentwicklung schnell zum Standard werden dürfte. Mein Rat: Klein anfangen, mit einer gut definierten *CLAUDE.md* und überschaubaren Aufgaben. Den Agenten beobachten, seine Arbeitsweise verstehen, Vertrauen aufbauen, und dann schrittweise mehr und mehr delegieren. So wird aus einem neuen Tool ein verlässlicher Entwicklungspartner. ■

[1] Claude-Code-Doku, <https://code.claude.com/docs>

[2] Model Context Protocol, <https://modelcontextprotocol.io>

[3] Patrick Schnell, ChatGPT in Softwareprojekten,

www.schnell.digital/de/book-chatgpt-in-softwareprojekten

[4] Claude Code Best Practices, <https://docs.anthropic.com/en/docs/claude-code/best-practices>



Patrick Schnell

ist Geschäftsführer von schnell.digital, das auf Entwicklung und Architektur von Digitalisierungsprojekten und Beratung spezialisiert ist. Dabei nutzt er unterschiedlichste Technologien wie .NET, Angular, TypeScript und Node.js.

<https://schnell.digital>

DWX ACADEMY

Hands-on Workshops
und Weiterbildung für
Software-Entwickler
und -Architekten

EINFÜHRUNG IN AZURE OPEN-AI – VON DATENINTEGRATION BIS ZUR ENTWICKLUNG EIGENER KI-AGENTEN

- Einführung in Azure OpenAI & Prompt Engineering
- Verwendung der Azure OpenAI Service Assistant-API
- Arbeiten mit RAG und Vektor-Datenbanken
- Einsatz von KI in der Cloud und on-premises
- Entwicklung und Testing von KI-Agenten
- Programmierung und Integration von KI-Agenten mit C# und Semantic Kernel
- Praktische Anwendungen in Echtzeitszenarien

1 TAG
REMOTE



GREGOR BISWANGER



Deine Ansprechpartnerin: **Susanne Herl**
+49 731 88005-8835 • susanne.herl@ebnermedia.de

developer-world.de/dwx-academy

DIE KI-IDE CURSOR IN DER PRAXIS

Mit dem KI-Agenten Cursor produktiv arbeiten

Man nehme VS Code und mische KI dazu. Heraus kommt eine App, die Entwicklern das Arbeiten in beliebigen Entwicklungsprojekten erleichtert.

Mit dem VS-Code-Fork Cursor [1] von Anysphere [2] erhalten Entwickler eine KI-IDE für alle Phasen der Softwareerstellung. Cursor generiert nicht nur Quellcode, sondern auch komplette Projektstrukturen inklusive Boilerplate und Architektur. Ein integrierter KI-Agent analysiert ganze Repositories, versteht Abhängigkeiten und schlägt konsistente Änderungen über Dateigrenzen hinweg vor. Vorgeschlagene Anpassungen lassen sich als Diff-Vorschau prüfen und direkt übernehmen. So entsteht effizientes Pair Programming, bei dem Entwickler und KI-Agent flexibel zwischen den Rollen Pilot und Navigator wechseln.

Klare und präzise Prompts dienen als Leitfaden für den KI-Workflow

Ein Prompt entspricht einer Anweisung in natürlicher Sprache an eine KI, eine bestimmte Aufgabe zu erledigen oder ein gewünschtes Resultat zu liefern. Je besser die KI die zu erfüllende Aufgabe versteht, desto genauer entspricht das Ergebnis den Erwartungen des Entwicklers. Da Arbeitsergebnisse in der Softwareentwicklung exakt die Anforderungen widerspiegeln müssen, sollten Prompts niemals vage, sondern so präzise wie möglich formuliert sein. Ein guter Prompt sollte mindestens die Ausgangssituation und die zu erledigende Aufgabe (Absicht) beschreiben, zudem den Kontext (alle Informationen zum Bearbeiten des Prompts) passend erweitern und das verlangte Ziel (den gewünschten Output) formulieren.

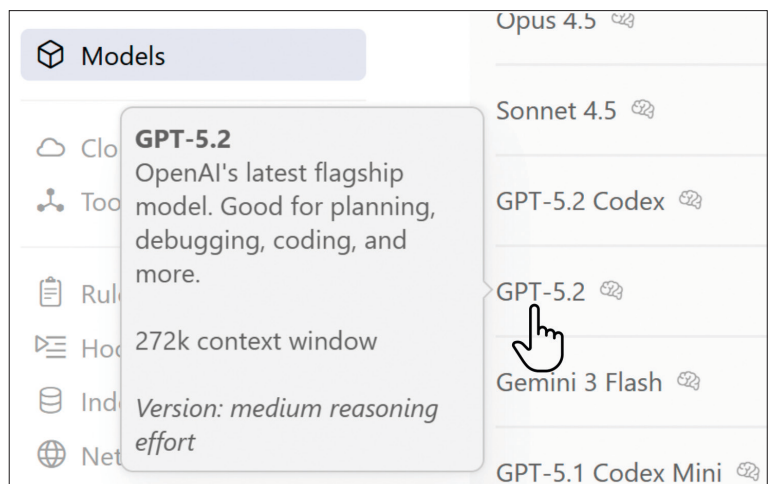
Ein Beispiel: Der Prompt „Erzeuge eine neue Blazor Web-App, die eine To-do-Liste verwaltet“ macht zwar Vorgaben zur Aufgabe und zum gewünschten Ergebnis – allerdings viel zu unkonkret. Während gängige KI-Modelle oft einfach mit Standardannahmen die Verarbeitung durchführen, bietet Cursor (insbesondere bei Einsatz des Composers) die Möglichkeit, den Workflow agentisch anzugehen. Cursor erstellt zunächst einen Plan oder bittet den Entwickler bei Unklarheiten um Präzisierung – im Beispiel etwa zum Hosting-Modell (Server-side versus WebAssembly) oder zur Datenspeicherung (SQLite, Entity Framework et cetera).

Der Entwickler muss den KI-Prozess aktiv steuern, insbesondere wenn Standardvorschläge nicht zum Projektkontext passen. Für präzise Ergebnisse benötigt Cursor konkrete Vorgaben zu Scaffolding,

Datenmodell, Persistenz und UI-Design. Ein Highlight von Cursor ist die visuelle Unterstützung: Ausgehend von Screenshots (Wireframes/Mockups) setzt der KI-Agent UX-Designs direkt in Code um. Dabei berücksichtigt das Tool konsequent externe Faktoren wie das Ökosystem, Architekturstandards, aber auch individuelle Vorgaben zum Quellcode. In einem VS-Code-Workspace verwendet Cursor dessen Codebase als zusätzlichen Input. Trotz hoher Automatisierung bleibt letztendlich der Entwickler die entscheidende Instanz für die Produktqualität.

Um die Konsistenz der Arbeitsergebnisse im Projekt über Sitzungen hinweg zu gewährleisten, implementiert Cursor ein eigenes Regelsystem. Regeln liefern für die Prompt-Ebene einen dauerhaften, wiederverwendbaren (persistenten) Kontext, den Cursor an den Anfang eines leeren Modell-Kontexts einfügt. Regeln wirken wie eine Art Gedächtnis – in der neuen Version kennt Cursor vier verschiedene Arten von Regeln:

- **Projektregeln:** Sie gelten für die gesamte Codebasis, sind im Verzeichnis `.cursor/rules/` des Projekts ([Listing 1](#)) gespeichert und unterliegen der Versionierung. Ihre Ablage erfolgt als Markdown- (`.md`) oder als `.mdc`-Dateien (mit Frontmatter und Glob Patterns).
- **Benutzerregeln:** Gelten global in der Cursor-Umgebung des Benutzers und werden vom Agenten (KI-Chat) verwendet.



Über **File | Preferences | Cursor-Settings** erreicht man die Einstellungen von Cursor; dort kann man einem KI-Agenten verschiedene LLMs zuweisen ([Bild 1](#))

- **Teamregeln:** Stehen in der Team- und Enterprise-Edition von Cursor zur Verfügung und werden im dortigen Dashboard verwaltet – sie gelten teamweit und für jedes Projekt des Teams.
- **Agent-Regeln:** Dabei handelt es sich um einfache Mark-down-Anweisungen, die der Entwickler (als Alternative zu den Regeln im Ordner `.cursor/rules`) in einer oder mehreren `AGENTS.md`-Dateien speichert. Cursor unterstützt eine Schachtelung mehrerer `AGENTS.md`-Dateien in den Unterverzeichnissen eines Projekts. Dies erlaubt eine feinere Steuerung des Agenten, da die Regeln auf den jeweiligen Einsatzbereich der Codebasis (Global, Frontend (Components), Backend et cetera) ausgerichtet sind.

Regel-Dateien dienen der Ablage einzuhaltender Standards (Architektur, Tools, Code et cetera), vorgegebener Arbeitsabläufe der Entwicklung oder Domain-spezifischen Wissens (Fachbegriffe, Prozesse und bewährter Vorgehensweisen des Anwendungsgebiets) – am besten eignen sich nach Einsatzzweck getrennte, separate Textdateien. Im `.cursor/rules`-Ord-

● Listing 1: Regeln für ein Blazor- und SQLite-Projekt

```
# Blazor & SQLite Project Rules (.NET 10)
## Tech Stack & Architecture
- **Framework:** .NET 10 Blazor (Präferenz:
  Interaktives WebAssembly oder Auto-Mode).
- **Database:** SQLite mit Entity Framework Core
  (EF Core).
- **Architecture:** Striktes "Clean Architecture"-
  Muster.
  - `Domain`: Entitäten & Interfaces.
...
## Coding Standards (C# & Razor)
- **Modern C#:** Nutze File-scoped Namespaces,
  Primary Constructors (wo sinnvoll) und striktes
  Nullable Reference Tracking.
- **Naming:** Interfaces starten mit `I`. Services
  enden auf `Service`.
- **Async:** Konsequenz `Task` / `ValueTask`.
  Niemals `Result` oder `Wait()`.
- **Razor Components:** Logik gehört in separate
  `Partial Classes` (.razor.cs). @code-Blöcke im
  .razor-File nur für minimale UI-Logik.
...
## Data & SQLite Specifics
...
## UI & Styling
...
## Cursor Workflow
- **Pre-Flight Check:** Vor Code-Generierung prüfen:
  Ist die Entität im DbContext? Ist eine Migration
  nötig?
...
```

● Du möchtest gerne mehr erfahren?

Das freut uns, denn ab Erscheinen des vorliegenden Artikels veröffentlichen wir auf developer-world.de/dwx-insights weitere Beiträge, die den Artikel fortsetzen und ausschließlich online zur Verfügung stehen. Derzeit geplant sind folgende Themen:

- Interaktive Planung und integrierte AI-Code-Reviews
- Agentenzentriertes und paralleles Arbeiten
- Sicherheit, Offline-Betrieb und Recovery
- Cursor als KI-Turbo für Fluent UI Blazor

ner sollten für deren Bezeichnung sprechende Namen wie `architecture.md`, `api-guidelines.md` oder `testing-standards.md` gewählt werden. Die Definition von Benutzer- und Projekt-Regeln erfolgt in den Cursor-Einstellungen im Bereich `Rules...` Dort lassen sich auch Regeln aus einem GitHub-Repository in das Projekt importieren.

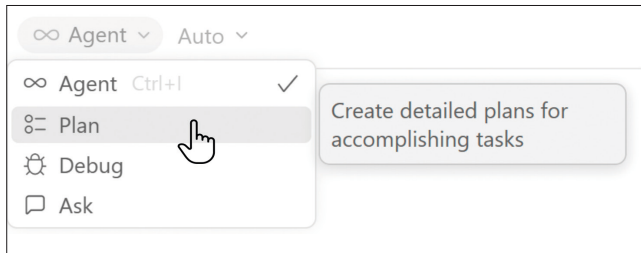
LLMs und spezielle Cursor-Features prägen die Qualität der Ergebnisse

AnySphere hat Cursor nicht auf ein einziges KI-Modell ausgerichtet, sondern unterstützt flexibel mehrere leistungsfähige LLMs (Large Language Models). Ein Alleinstellungsmerkmal von Cursor ist die intelligente Kombination dieser Modelle: Während im Editor spezialisierte, hocheffiziente Modelle im Hintergrund ein Autocomplete über das Tab-Feature (spezialisiertes Cursor-Modell für Autovervollständigung) steuern, kommen für komplexe Aufgaben im KI-Chat das Agentic-Coding-Modell Composer oder die High-End-Modelle wie GPT-5.2, Sonnet 4.5 oder Gemini 3 Flash zum Einsatz. Der Entwickler kann je nach Aufgabe im Projekt – etwa komplexe Architekturplanung oder schnelles Refactoring – das am besten geeignete Modell selbst wählen (Bild 1).

Bei der Auswahl des LLM fließen verschiedene Kriterien wie die Qualität der Logik (konsistente, schlüssige und fehlerfreie Antworten), die Beherrschung von Programmiersprachen und die Fähigkeit zur Kontextverarbeitung ein. In der Regel führt Cursor diese Modelle über eine gesicherte Cloud-Verbindung aus. Für Unternehmen mit strengen Compliance-Anforderungen bietet Cursor den Privacy Mode: In diesem Modus werden Daten des Projekts nicht für das Training der Modelle gespeichert oder verwendet. Alternativ können lokale Modelle über Tools wie Ollama [5] oder LM Studio [6] eingebunden oder eigene API-Schlüssel für eine präzise Kostenkontrolle genutzt werden.

Eine weitere Stärke von Cursor über die reine Intelligenz des LLM hinaus liegt in zwei realisierten Schlüsseltechnologien, die echte Gamechanger für die Entwicklung darstellen:

- **Advanced Codebase Indexing:** Cursor erstellt eigene performante Indexstrukturen für das gesamte Projekt in einer spezialisierten Vektordatenbank. Diese ermöglichen Cursor semantische Suchen und ein Kontextverständnis über komplette Codebasen hinweg. Die KI erkennt bei jeder ►



Über die Auswahlleiste unten im Fensterausschnitt „Agent“ legt der Entwickler dessen Arbeitsweise fest (Bild 2)

Anfrage vorhandene Abhängigkeiten – etwa welche Interfaces in C# definiert sind oder welche Architekturmuster (zum Beispiel Repository-Patterns) im Projekt konsistent angewendet werden sollen.

- **Echtzeit-Linter-Integration:** Cursor liest Compiler-Fehler und Linter-Warnungen direkt aus. Erkennt der KI-Agent einen Fehler (beispielsweise eine fehlende `using`-Direktive oder einen Typkonflikt), bietet Cursor proaktive Fixes an. Dieser Self-Healing-Code-Ansatz minimiert massiv die Iterationszyklen zwischen Codierung und Debugging. Die direkte Integration des Linters von VS Code in Cursor liefert einen wesentlichen Vorteil gegenüber einem LLM, das lediglich Text produziert, ohne zu verstehen, wie das Projekt kompiliert oder gelintet wird.

Gezielt den Agenten-Modus zur Erledigung von Aufgaben einsetzen

Mit der Tastenkombination `[Cmd]/[Strg]+[I]` startet Cursor den Agenten-Modus: Die KI-IDE erwartet eine Prompt-Eingabe, analysiert im geöffneten Projekt den markierten Code und beschafft sich Kontextinfos, Erklärungen oder Vorschläge zur Optimierung – direkt im Editor, ohne Unterbrechung im Workflow. Diese Funktion bietet schnelle Rückmeldungen und Erläuterungen zum aktuellen Projekt, ohne das Arbeitsumfeld zu verlassen. Cursor nutzt dabei standardmäßig Composer, ein eigenes, auf Code-Verständnis spezialisiertes KI-Modell. Composer agiert als aktiver Agent: Er erklärt nicht nur, sondern unterstützt bei Entscheidungen, nutzt Tools und generiert Lösungen – ideal für schnelles Debugging, Lernen oder Refactoring.

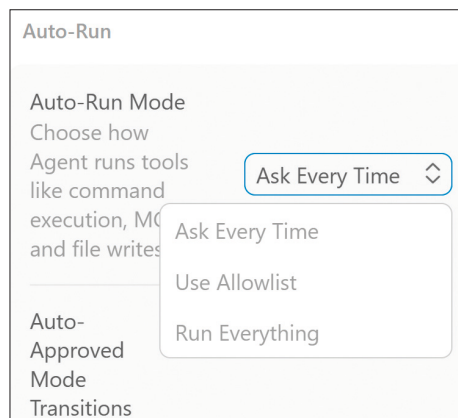
Ein KI-Agent in Cursor arbeitet autonom: Er öffnet im Hintergrund Dateien, analysiert die Codebasis, schlägt komplexe Änderungen über mehrere Dateien hinweg vor oder führt Befehle im integrierten Terminal aus. Dabei hat der KI-Agent Zugriff auf alle über VS Code [3] verfügbaren Tools (Editor, Dateistruktur, Debugging, CLI). Cursor ist in der Lage, komplexe Aufgaben in Teilschritte zu zerlegen und diese sequenziell abzuarbeiten. Bei eventuellen Unklarheiten erzeugt der KI-Agent Rückfragen

beim Entwickler. Während der Arbeiten bewertet Cursor den Fortschritt stetig selbst und korrigiert seine Ausführung, falls Fehler auftreten. Dies verdeutlicht die Mächtigkeit der Arbeitsweise des KI-Agenten. Um den Grad der Autonomie zu steuern, bietet Cursor verschiedene Modi an, welche die Arbeitsweise des Agenten definieren (vergleiche Bild 2):

- **Agent-Modus (Act):** Dieser Modus ist die leistungsfähigste Einstellung für komplexe Features wie Refactorings oder Multi-File-Änderungen. Dem Agenten stehen alle Tools zur Verfügung: Er erkundet autonom die Codebasis, schreibt Code in mehrere Dateien gleichzeitig und führt Tests oder Build-Befehle im Terminal aus, um das Ergebnis zu verifizieren. Im Bedarfsfall verlangt der Agent-Modus eine Rückmeldung vom Entwickler.
- **Plan-Modus:** Dieser Modus zielt auf Aufgaben ab, die eine sorgfältige Vorbereitung erfordern. Vor der eigentlichen Ausführung erstellt der Agent zunächst einen detaillierten Plan und bittet den Entwickler um Feedback, bevor er in den Schreibmodus wechselt. Der Modus eignet sich hervorragend zur Planung tiefgreifender struktureller Umbauten oder riskanter Änderungen.
- **Debug-Modus:** Die Fehleranalyse stellt eine Kernkompetenz des Agenten dar. Er instrumentiert Logs in den Quellcode, analysiert erhaltene Terminal-Logs oder Runtime-Fehler (Terminal Diagnostics) und schlägt proaktiv Korrekturen vor, indem er Hypothesen zur Fehlerursache aufstellt und diese direkt im Code testet. Er kann alle über VS Code zugänglichen Tools [4] nutzen, setzt aber primär die Debugging-Werkzeuge ein.
- **Ask-Modus:** Er ist auf das Lernen, Erklären und Analysieren ausgerichtet. Die KI agiert hier primär informativ und nimmt keine automatischen Änderungen zum Beispiel in Dateien vor. Dieser Modus ist ideal, um bestehenden Code sicher zu verstehen oder Architekturfragen zu klären – er nutzt nur die in VS Code integrierten Lese- und Search-Tools.

Qualität der KI-Antworten durch optimierten Kontext verbessern

Der Kontext bildet den mentalen Arbeitsraum des KI-Agenten – gewissermaßen seinen aktuellen Informationspuffer (Arbeitsgedächtnis). Durch ihn versteht die KI, um welchen Sachverhalt es im aktuellen Arbeitsschritt geht. Im Kontext bündelt der Agent alle relevanten Informationen, um sinnvoll zu reagieren, Entscheidungen zu treffen oder komplexe, mehrschrittige Aufgaben auszuführen. Dazu gehören die Inhalte des aktuellen und bisherigen Dialogs, alle aufgabenrelevanten Daten sowie das Wissen über verfügbare Tools in der Ausführungsumgebung. Aus diesem Kontext leitet der Agent die Absicht des Nutzers (User Intent) ab, versucht Fehler zu vermeiden und schlägt maßgeschneiderte Lösungen vor.



Im Fensterausschnitt „Auto-Run“ macht Cursor dem Entwickler mehrere Einstellungen für Automatismen verfügbar (Bild 3)

● Arbeiten mit mehreren KI-Agenten in einem Projekt

Um komplexe Aufgaben effizienter zu bewältigen, kann man mehrere KI-Agenten in einem Projekt einsetzen. Dieses Vorgehen ermöglicht Aufgabenteilung, Parallelisierung und Wissensspezialisierung.

- **Aufgabenteilung:** Jedem Agenten kann man eine spezifische Rolle zuweisen, zum Beispiel für Code-Review, Testgenerierung und Dokumentation.
- **Parallelisierung:** Mehrere Agenten können gleichzeitig an unterschiedlichen Teilen des Projekts arbeiten.
- **Wissensspezialisierung:** Agenten lassen sich mit unterschiedlichen Kontexten oder Prompts ausstatten, um beispielsweise Aufgaben für Frontend, Backend oder DevOps zu übernehmen.

Cursor verfügt über spezialisierte Features, um diesen Kontext im VS-Code-Workspace (Arbeitsumgebung) präzise zu strukturieren. Ein intelligentes Kontext-Management (Rules, Referenzen, semantische Suche) verhindert, dass der KI-Agent funktionierenden Code fälschlicherweise überschreibt, Architekturvorgaben ignoriert oder Abhängigkeiten falsch interpretiert. Beim Kontext unterscheidet Cursor intern zwischen zwei grundlegenden Säulen:

- **Intent-Kontext (Absicht):** Fokussiert auf das Erkennen der Absicht des Benutzers durch einen eingegebenen Prompt oder seine Interaktion mit dem UI von Cursor – dabei stellt sich der KI-Agent die Frage: Was will der Benutzer erreichen?
- **State-Kontext (Zustand):** Fokussiert auf den aktuellen Zustand des Projekts (Code-Snippets, Fehlermeldungen, Projektstruktur, Umgebung) – der KI-Agent stellt sich die beiden Fragen „Wo befinden wir uns in der Projektarbeit?“ und „Was ist bisher erledigt?“.

Erst die Symbiose aus Absicht und Zustand macht Cursor zu einem intelligenten, lernenden KI-Agenten, der nicht nur Code „dumm“ vervollständigt, sondern die Absichten des Benutzers versteht und den aktuellen Projektzustand berücksichtigt. Diese Symbiose verhindert Missverständnisse oder Halluzinationen des Modells und spezifiziert einen exakten Startpunkt für die nächsten Arbeitsschritte.

Um diesen Prozess gezielt zu steuern und schnell spezifische KI-Funktionen oder Aktionen aufzurufen, steht in Cursor das @-Symbol zur Verfügung. Dieses präzisiert explizit den Kontext und übernimmt beliebige vom Entwickler übergebene Entitäten (Files, Docs, Web) direkt in den Prompt-Flow. Anysphere verfolgt mit Cursor inzwischen einen Context-First-Ansatz; der bisherige Begriff der @Commands wurde durch @Mentions (Context References, Universal Context Provider) ersetzt. Damit stehen in der aktuellsten Version verschiedene @Mentions zur Verfügung:

- **@Code:** Durchsucht anhand der übergebenen Abfrage das gesamte Repository via Embedding-Suche nach relevanten Informationen.

- **@Docs:** Damit referenziert Cursor eine offizielle, von Anysphere ausgewählte Dokumentation bekannter Frameworks oder Bibliotheken (zum Beispiel ASP.NET, Blazor, React), um sicherzustellen, dass der KI-Agent mit aktuellem Wissen anstatt veralteten Trainingsdaten arbeitet. Weitere bisher nicht referenzierte Dokumentationen kann der Entwickler zusätzlich hinzufügen und so Cursor zugänglich machen.
- **@Files & Folders:** Bindet gezielt übergebene Dateien oder ganze Verzeichnisse ein, um den Fokus auf relevante Codestellen zu lenken. Per Drag and Drop kann man Dateien direkt dem Agenten als Kontext übergeben.
- **@Past Chats:** Zeigt verschiedene bereits durchgeführte Chats/Konversationen an und übernimmt deren Arbeitsergebnisse in den Kontext des Agenten. Damit versteht die KI bereits durchgeführte Arbeitsschritte und ermöglicht dem Entwickler, weitere anstehende Aufgaben mit vorhandenen Arbeitsergebnissen fortzusetzen.

Terminal-Befehle mit Auto-Run Mode oder per CLI interaktiv ausführen

Der Auto-Run Mode (in frühen Versionen auch als YOLO Mode bekannt) ist ein Produktivitäts-Feature, um Befehle/Tools mit minimaler Bestätigung durch den Entwickler in der Regel über ein Terminal auszuführen. Da ständige Rückfragen den Workflow unterbrechen können, folgt dieser Modus dem Prinzip, Änderungen schnell und direkt umzusetzen. Besonders nützlich ist dies bei umfangreichen Aufgaben wie Bulk-Refactorings, automatisierten Tests oder großflächigen Formatierungen, bei denen der Entwickler darauf vertraut, dass die vorgeschlagenen Befehle korrekt sind.

Die zugehörige Konfiguration befindet sich in den Cursor-Settings bei den Agenten-Einstellungen im Bereich *Auto-Run* (Bild 3). Seitens Anysphere gab es für die dortigen Einstellungen schon unterschiedliche Vorgaben – die aktuellste Version von Cursor bietet hier jetzt drei neue Stufen an:

- **Ask Every Time (niedriges Risiko):** Cursor fragt immer nach, bevor Änderungen ausgeführt werden. Der Entwickler muss jede Aktion manuell bestätigen – Cursor schreibt niemals automatisch in Dateien. Damit erhält man maximale Sicherheit, minimiert allerdings die Automatisierung. Diese Einstellung eignet sich für sicherheitskritische Projekte oder neue Codebasen, die Cursor noch nicht kennt. Als Entwickler behält man die volle Kontrolle.
- **Use Allowlist (mittleres Risiko):** Nach dieser Auswahl bietet Cursor unterhalb der Auswahlliste *Auto-Run Mode* Eingabefelder für drei Allowlists an: *Command*, *MCP* und *Fetch Domain*. Diese Allowlists wirken als Sicherheitsfilter – alles, was dort in den Eingabefeldern steht, wird vom Agenten ohne explizite Zustimmung (Klick auf *Run*) ausgeführt. In den Allowlists stehen quasi alle erlaubten Befehle/Domains – jeder als eigenständige Zeile. Verbote zur Ausführung bestimmter Befehle legt man in eine Rules-Datei ab.
- **Run Everything (hohes Risiko):** Alle generierten Befehle werden ohne weitere Rückfrage sofort ausgeführt. Cursor schreibt direkt ohne Vorschau oder Bestätigung in Dateien. Dies bietet zwar maximale Geschwindigkeit, erfordert ►

jedoch absolute Aufmerksamkeit. Cursor handelt zwar autonom, Änderungen können jedoch weiterhin rückgängig gemacht werden. Diese Einstellung ist ideal für Prototyping, schnelle Iterationen, Boilerplate-Generierung, kleine Projekte oder einfache Workflows.

Für hocheffizientes Arbeiten besitzt Cursor zudem zwei Command Line Interfaces (CLIs), die man auch außerhalb der KI-IDE einsetzen kann. Damit lässt sich direkt im Terminal die Funktionalität von Cursor nutzen:

- **Standard-CLI (cursor-Befehl):** Analog zum bekannten `code`-Befehl in VS Code ermöglicht der `cursor`-Befehl einen schnellen Workflow. Er öffnet Dateien oder ganze Projekte blitzschnell im Cursor-Editor, führt Diff-Vergleiche durch oder installiert VS-Code-Erweiterungen per Skript – ideal für ein automatisiertes Setup von Arbeitsumgebungen.
- **Cursor-CLI (agent-Befehl):** Auch Cursor Agent & Remote-Integration genannt. Der zusätzlich zu installierende `agent`-Befehl ermöglicht den Einsatz der KI-Power von Cursor in speziellen Umgebungen. Dies umfasst das Arbeiten auf Remote-Servern, in der Cloud und in Containern oder die Integration von Cursor-Workflows in externe Werkzeuge über ein Terminal. So können KI-gestützte Fehleranalysen und Debugging-Prozesse dort stattfinden, wo der Code tatsächlich ausgeführt wird.

• Cursor besitzt ein spezielles KI-Coding-Modell

Ergänzend zu den allgemeinen Sprachmodellen (LLMs) stellt AnySphere ein eigenes, speziell für Coding optimiertes LLM bereit.

Dieses Coding-Modell, Composer genannt, ist ein Mixture-of-Experts-Modell (MoE), das mittels Reinforcement Learning (RL) und Werkzeugen wie Codebase-weiter semantischer Suche gezielt darauf trainiert wurde, anspruchsvolle Aufgaben in umfangreichen Codebasen zu lösen.

Innerhalb der agentenbasierten Arbeitsumgebung fungiert Composer als zentrales Modell für Coding-Aufgaben. Es arbeitet deutlich schneller als herkömmliche LLMs und ermöglicht nahezu verzögerungsfreie Iterationen bei ähnlicher Intelligenz wie vergleichbare Modelle.

Das Modell arbeitet kontextbewusst über ganze Projekte hinweg, indem es semantische Such- und Kontextinformationen nutzt, und unterstützt agentenbasierte Workflows, bei denen mehrere KI-Agenten parallel unabhängig voneinander an Aufgaben arbeiten können.

Composer bildet die Grundlage für ein agentisch fokussiertes Entwicklererlebnis, das über klassische Code-Vervollständigung hinausgeht und mehrstufige Coding-, Refactoring- und Review-Workflows unterstützt.

MCP-Verzeichnis

Durchsuchen Sie empfohlene MCP-Server für Cursor. Verwenden Sie Suche und Filter, um Integrationen schnell zu finden.

Q Git	Filter	Zurücksetzen
Suche: Git x		3 von 149 angezeigt
Name	Installieren ^	Beschreibung
 Defang	+ Add to Cursor	Deploy apps to AWS, GCP, and DigitalOcean from Docker Compose.
 GitHub	+ Add to Cursor	Version control and collaborative development platform.
 GitLab	+ Add to Cursor	DevSecOps platform for code, CI/CD, and security.

Ihr MCP-Server fehlt? [Erstellen Sie ein Issue](#), um ihn anzufordern.

Die Online-Dokumentation von Cursor enthält ein MCP-Verzeichnis zur schnellen Integration eines gewünschten MCP-Servers in die KI-IDE (Bild 4)

Innovative Konzepte von Cursor ziel- und projektorientiert nutzen

Aktuell besitzen KI-Agenten extrem kurze Innovationszyklen – ihre Möglichkeiten entwickeln sich fast täglich weiter. AnySphere unterstützt Entwickler dabei, neue Funktionen, Modelle und UI-Änderungen von Cursor vorab zu testen. Diese Optionen werden direkt in den Cursor-Settings verwaltet. Um Zugriff auf experimentelle Versionen zu erhalten, bietet Cursor über die dortige Sidebar den dedizierten Bereich `>>Beta` an. Anhand der Auswahlliste `Update Channel` lässt sich steuern, welcher Release-Zweig genutzt wird. Während die Einstellung `Default` für maximale Stabilität in produktiven Umgebungen sorgt, bieten `Early Access` oder `Nightly` einen frühen Zugriff auf neue Features, gehen jedoch mit einem höheren Fehlerrisiko einher.

Eine entscheidende Einstellung für die Erweiterung der KI-Fähigkeiten eines Agenten wie Cursor stellt das Model Context Protocol (MCP) dar. MCP verwandelt Cursor in einen handlungsfähigen Agenten, der mit externen Tools und Daten interagiert. Hierfür erfolgt die Konfiguration bei den Cursor-Settings im Bereich `Tools & MCP`. Dort definiert eine `mcp.json`-Datei, welche externen Tools der KI-Agent ansteuern kann. Eine einfache Installation ermöglicht das MCP-Verzeichnis der Online-Dokumentation über die dortigen Schaltflächen `+ Add to Cursor` (Bild 4). Zusätzlich schützen im Bereich `Agents` zwei zentrale Sicherheitsfeatures den MCP-Workflow:

- **MCP Allowlist:** Diese Sicherheitssteuerung legt fest, welche konkreten MCP-Server oder Funktionen Cursor aufrufen darf (ein Whitelisting-Ansatz).
- **MCP Tools Protection:** Als zusätzlicher Schutzwall stellt diese Einstellung sicher, dass Cursor den Zugriff auf MCP-Tools nicht unbemerkt im Hintergrund durchführt, sondern kritische Aktionen vom Benutzer autorisiert werden müssen. Wirkt wie eine Firewall zwischen KI-Agent und externen Diensten, indem jeder Tool-Aufruf eine explizite Genehmigung erfordert.

Von der KI-Unterstützung zur autonomen Agenten-Plattform

Cursor hat im vergangenen Jahr die Transformation von einem KI-gestützten Editor zu einer agentenzentrierten Entwicklungsplattform vollzogen und damit die Softwareerstellung grundlegend verändert. Was als intelligenter Assistent begann, wurde von Anysphere zu einer nativen KI-IDE weiterentwickelt, die heute komplexe Aufgabenpakete autonom steuert. Das Spektrum reicht von der strategischen Architekturplanung und der Einbindung von UI-Design-Systemen über die koordinierte Implementierung in multiplen Dateien bis hin zur automatisierten Testabdeckung und der Orchestrierung von Deployments.

Durch die Kombination moderner Sprachmodelle mit einer tiefen Indizierung der Codebasis ermöglicht Cursor ein fehlerresistentes und hochpräzises Arbeiten im Sinne von Vibe (nicht Vibe) Coding, was einem kontrollierten, schrittweisen Workflow entspricht. Ein zentraler Fortschritt ist die Multi-Agenten-Architektur: Spezialisierte KI-Agenten agieren als virtuelle Teammitglieder in isolierten Shadow Workspaces. So kann man Lösungen validieren, bevor sie in das Projekt einfließen, und erhält wartbaren Code für den Produktiveinsatz. Ergänzend erschließt das Cursor-CLI den Einsatz der Agent-Features in externen Werkzeugen, was die Qualität von deren Arbeitsergebnissen verbessert.

Dank der vollen Kompatibilität zum VS-Code-Ökosystem bleibt Cursor hochgradig flexibel, während Anysphere kontinu-

ierlich innovative KI-Features wie das Kontext-Management oder das Model Context Protocol (MCP) integriert. Parallel dazu entfernt Anysphere nicht mehr benötigte manuelle Eingriffe aus Cursor, sobald die KI-IDE diese selbstständig durchführt. Aufgrund dieser evolutionären, agentenorientierten Vorgehensweise sollten Unternehmen Cursor nicht nur als Werkzeug, sondern als strategische Komponente in ihr eigenes Innovationsmanagement einbeziehen. Dann lässt sich Cursor auch als Katalysator für neue eigene innovative Produkte nutzen. ■

[1] Cursor, <https://cursor.com>

[2] Anysphere, <https://anysphere.inc>

[3] VS Code, <https://code.visualstudio.com/>

[4] Marketplace von VS Code, <https://marketplace.visualstudio.com/vscode>

[5] Ollama, <https://ollama.com>

[6] LM Studio, <https://lmstudio.ai>



Frank Simon

unterstützt Projekte in den Arbeitsgebieten Analyse, Spezifikation, Entwicklung, Programmierung, Test und Debugging in On-Prem-, Cloud-, Mobile- und Web-Umgebungen inklusive deren Systemmanagement.

dotnetpro@gmx.eu

DWX ACADEMY

Hands-on Workshops
und Weiterbildung für
Software-Entwickler
und -Architekten

3 TAGE
INHOUSE ODER
REMOTE

EINFÜHRUNG IN SQL

- Daten abfragen
- Daten löschen
- Daten einfügen
- Optimierungen
- Daten modifizieren
- Transaktionssteuerung



THORSTEN KANSY



Deine Ansprechpartnerin: Susanne Herl
+49 731 88005-8835 • susanne.herl@ebnermedia.de

developer-world.de/dwx-academy

AI-WEBSITE-BUILDER AUF DEM PRÜFSTAND

Computer, bau mir eine Webseite

Eine Webseite zu erstellen dauert mit AI-Tools nur noch wenige Minuten. Aber taugt das Resultat auch? Wir fühlen drei AI-Webseite-Builder-Tools auf den Zahn.

Wer als Entwickler schon mal eine Idee für ein Projekt hatte, kennt das Problem: Bevor man überhaupt anfangen kann, die eigentliche Funktionalität zu implementieren, muss erst einmal das Grundgerüst stehen. Projekt anlegen, Abhängigkeiten installieren, Build-System konfigurieren, Deployment-Umgebung konfigurieren – das kostet Zeit und Nerven. Hier versprechen die neuen AI-Website-Builder Abhilfe: Man beschreibt in natürlicher Sprache, was man bauen möchte, und erhält in wenigen Minuten eine funktionsfähige Applikation, die sich mit wenigen Klicks mit anderen teilen lässt. Klingt zu schön, um wahr zu sein? Das wollen wir herausfinden.

In diesem Artikel werfen wir einen detaillierten Blick auf drei der prominentesten Vertreter dieser Gattung: Firebase Studio von Google, Lovable und Bolt.new von StackBlitz. Als Praxisbeispiel dient eine Quiz-Applikation zum Thema Critical Thinking, bei der die Spieler logische Fehlschlüsse in Statements identifizieren müssen.

Kritisches Denken als Fallstudie

Um die drei Plattformen fair zu vergleichen, bauen wir mit jeder von ihnen eine identische Quiz-Applikation. Die Grundlage bilden 24 gängige logische Fehlschlüsse, wie sie zum Beispiel unter [1] zusammengefasst sind. Der Spieler bekommt Aussagen präsentiert, in denen sich jeweils ein logischer Fehlschluss versteckt, und muss aus den zufällig ausgewählten Antwortoptionen die korrekte Fallacy identifizieren.

Die Fehlschlüsse werden in drei Schwierigkeitsgrade eingeteilt, und auch das Quiz gibt es in drei Schwierigkeitsgraden. Im leichten Modus muss der Spieler drei Fehlschlüsse der einfachen Kategorie aufdecken. Dabei kann er unter drei möglichen Antworten auswählen. Bei mittlerer Schwierigkeit gibt es sechs Fragen aus den Kategorien leicht und mittel zu beantworten, und es stehen auch jeweils sechs mögliche Antworten zur Wahl. Wer es richtig wissen will, kann die schwere Stufe spielen: Hier gilt es neun Logikfehler aller Kategorien zu finden, wobei auch jeweils neun Antworten zur Auswahl stehen.

Das Spiel soll ohne weitere Anmeldung spielbar sein. Wenn der Nutzer es möchte, kann man sich registrieren und einen Account anlegen; in diesem Fall werden die eigenen Fortschritte gespeichert.

Aus diesen Anforderungen generieren wir uns einen Prompt, mit dem wir die Website-Builder füttern wollen. Damit die Aufgabenstellung vergleichbar bleibt, nutzen wir

denselben Prompt für alle Systeme. Wie immer, wenn man etwas mit KI tun möchte, aber nicht weiß, wie man es anfangen soll, fragt man die KI. Also ist auch der Prompt in [Listing 1](#) anhand der obigen Beschreibung von der KI erstellt worden, im vorliegenden Fall hat das Claude Opus 4.5 übernommen.

Die Testkriterien

Für den Vergleich bewerten wir jede Plattform anhand von drei Kernbereichen:

- **Leistungsumfang** umfasst die Kosten und Abrechnungsmodelle sowie die unterstützten Programmiersprachen und Frameworks. Hier zeigt sich, was man für sein Geld bekommt und wie flexibel man bei der Technologiewahl ist.
- **Entwicklerfreundlichkeit** (Developer Experience) betrachtet den Zugang zum generierten Code, die Git-Integration und die Backend-Anbindung. Entscheidend ist, wie gut sich die Tools in bestehende Entwickler-Workflows integrieren und wie viel Kontrolle man über das Ergebnis hat.
- **Betrieb und Wartung** deckt Deployment-Optionen und Monitoring-Möglichkeiten ab. Hier geht es darum, wie einfach die fertige Anwendung in Produktion gebracht und überwacht werden kann.

Firebase Studio: AI Builder powered by Google

Firebase Studio wurde im April 2025 auf der Google Cloud Next vorgestellt und ist Googles Antwort auf den AI-Builder-Trend. Es handelt sich um eine cloudbasierte Entwicklungsumgebung, die auf dem bekannten Project IDX aufbaut und Gemini als AI-Assistent integriert. Der besondere Fokus liegt auf der nahtlosen Integration mit dem Firebase-Ökosystem [2]. Um den Dienst nutzen zu können, muss man sich mit einem Google-Account anmelden.

Firebase Studio verfolgt einen interessanten Ansatz bei den Preisen: Die Entwicklungsumgebung selbst ist im Kern kostenlos, Kosten entstehen primär durch die genutzten Backend-Dienste. Jeder Nutzer erhält drei Workspaces ohne Kosten. Wer mehr benötigt, kann dem Google Developer Program beitreten: Standard (kostenlos) umfasst 10 Workspaces, Premium (14,99 US-Dollar/Monat) enthält 30 Workspaces plus erhöhte Gemini-Kontingente und 10 US-Dollar Cloud-Credits, Premium Plus (45 US-Dollar/Monat) bietet zusätzlich 45 US-Dollar Cloud-Credits und einen Certification-Voucher. Der Haken: Sobald man Firebase App Hosting nutzt

oder Backend-Dienste wie Firestore über die Free-Tier-Limits hinaus beansprucht, entstehen Kosten nach dem Pay-as-you-go-Modell [3].

Bei den unterstützten Frameworks spielt Firebase Studio seine Stärken aus. Da die Plattform auf einem vollwertigen Code-Editor (basierend auf Code OSS) aufbaut, unterstützt sie praktisch jede Sprache: React, Next.js, Angular, Vue.js, Svelte, Astro im Frontend, Flutter und React Native + Expo für Mobile, sowie Node.js, Go, Java, .NET und Python im Backend. Über 60 vorgefertigte Templates stehen zur Verfügung.

Firebase Studio bietet zwei Modi für den Code-Zugriff, zwischen denen man nahtlos wechseln kann: Der App Prototyping Agent ermöglicht promptbasiertes Arbeiten ohne Coding, ideal für schnelle Prototypen. Allerdings unterstützt der App Prototyping Agent für die automatische Generierung derzeit nur Next.js. Der Code Mode bietet einen vollständigen Code-Editor mit Syntax-Highlighting, IntelliSense und Terminal. Seit dem Update mit Agent Mode kann Gemini auch im Code-Modus autonom über mehrere Dateien hinweg Änderungen vornehmen, Tests schreiben und Fehler beheben – fragt aber vor kritischen Aktionen nach Erlaubnis.

Die Git-Integration erlaubt den Import bestehender Repositories von GitHub, GitLab, Bitbucket oder lokalen Maschinen. Änderungen können direkt committet und gepusht werden. Auch ein nachträgliches Exportieren des generierten Quellcodes in ein Git-Repository ist möglich.

Die größte Stärke von Firebase Studio liegt in der Backend-Integration. Alle Firebase-Dienste sind direkt integriert: Cloud Firestore (NoSQL mit Echtzeit-Sync), Firebase Authentication (Google, E-Mail/Password, Apple, GitHub), Cloud Functions (Serverless Backend-Logik) und Firebase Storage (Datei-Uploads). Der App Prototyping Agent kann Backend-Anforderungen erkennen und automatisch Firebase Auth und Firestore vorschlagen.

Das Deployment erfolgt mit einem Klick aus der Weboberfläche. Es ist ein Google-Cloud-Billing-Account erforderlich, in dem man 300 US-Dollar kostenlose Credits für die ersten 90 Tage erhält. Es gibt keine automatischen Abbuchungen, diese erfolgen erst, wenn man ein „Pay as you go“-Konto aktiviert hat. Alternativ kann man auch eine Vorauszahlung leisten. Für die Aktivierung des Kontos ist jedoch zur Verifikationszwecken eine Kreditkarte erforderlich. Die Applikation ►

● Listing 1: Prompt für das Logical-Fallacy-Quiz

Erstelle eine Quiz-Applikation zum Thema "Logische Fehlschlüsse".

KONZEPT:

Dem Nutzer werden Aussagen gezeigt, die jeweils genau EINEN logischen Fehlschluss enthalten. Er muss aus Multiple-Choice-Optionen den korrekten Fehlschluss identifizieren.

FEHLSCHLÜSSE (Quelle: "Your Logical Fallacy Is" von School of Thought, CC BY-NC, yourlogicalfallacyis.com): Eingeteilt in drei Schwierigkeitsstufen:

LEICHT (8): Ad Hominem, Strohmännchen, Appell an Emotionen, Dammbrech-Argument (Slippery Slope), Schwarz-Weiß-Denken, Mitläufereffekt, Autoritätsargument, Tu Quoque

MITTEL (8): Beweislastumkehr, Scheinkausalität, Suggestivfrage, Kein wahrer Schotte, Anekdotische Evidenz, Appell an die Natur, Genetischer Fehlschluss, Zirkelschluss

SCHWER (8): Fehlschluss-Fehlschluss, Persönliche Ungläubigkeit, Sonderfall-Pleading, Mehrdeutigkeit, Spielerfehlschluss, Komposition/Division, Texas Sharpshooter, Goldene Mitte

SCHWIERIGKEITSGRADE:

- Leicht: 3 Fragen, 3 zufällige Antwortoptionen pro Frage (nur aus Leicht-Kategorie)

- Mittel: 6 Fragen, 6 zufällige Antwortoptionen pro Frage (aus Leicht + Mittel)
- Schwer: 9 Fragen, 9 zufällige Antwortoptionen pro Frage (aus allen Kategorien)

DATENBANKSTRUKTUR:

- Tabelle 'fallacies': id, name, description, difficulty_level
- Tabelle 'fallacy_statements': id, fallacy_id (FK), statement_text (mehrere Statements pro Fallacy möglich)
- Tabelle 'quizsessions': id, user_id (nullable), difficulty, score, total_questions, created_at
- Tabelle 'quizsession_questions': id, quizsession_id (FK), question_number, statement_id (FK), correct_fallacy_id (FK), selected_fallacy_id (FK), is_correct

FUNKTIONEN:

- Startbildschirm mit Schwierigkeitsauswahl
- Eine Aussage pro Frage mit Multiple-Choice-Antworten
- Sofortiges Feedback nach jeder Antwort (richtig/falsch) mit Erklärung
- Ergebnisübersicht am Ende mit Details zu jeder Frage
- Optionale Benutzerregistrierung für Fortschrittsverfolgung
- Dashboard für registrierte Nutzer mit Quiz-Historie und Statistiken

UI: Modernes, cleanes Design. Responsive für Mobile. Animationen für Feedback.

wird dann über Firebase App Hosting bereitgestellt. Ohne weitere Einstellungen wird die Webseite unter einer automatisch generierten Subdomain veröffentlicht. Die Veröffentlichung selbst dauert nur ein paar Minuten, der Fortschritt lässt sich über die Firebase Console [4] verfolgen. Ein umfassendes Monitoring ist direkt verfügbar: Firebase Analytics, Crashlytics, Performance Monitoring und Cloud Logging. Die Nutzung und das Verhalten der App können direkt aus Firebase Studio oder der Firebase Console überwacht werden.

Firestore Studio und das Fallacy-Quiz

Jetzt kennen wir die Plattform und wollen nun natürlich wissen, wie es mit der Umsetzung der geforderten Aufgabe gelaufen ist. Der Prompt wird direkt mit einigen Ideen zur Umsetzung quittiert, die man noch bearbeiten kann. Unter anderem schlägt Firebase oft sofort einen möglichen Einsatz von KI innerhalb der fertigen Anwendung vor – so auch hier: Die KI soll uns erklären, warum eine Antwort richtig oder falsch ist. Warum nicht. Dazu braucht es einen Gemini API Key, den der Website-Builder uns netterweise direkt selbst erzeugt.

Dann geht die KI ans Werk, baut aber direkt im ersten Entwurf ein paar Kompilierungsfehler in der Frageseite und in der Auswertungsseite ein – immerhin erkennt die KI das von selbst und korrigiert diese auch sogleich, trotzdem wirkt der Ablauf etwas unbeholfen. Im Anschluss steht zwar die rudimentäre Quiz-Seite; die Anforderungen, eine Authentifizierung einzubauen und die Daten in einer Datenbank zu lagern, werden aber erst mal geflissentlich ignoriert.

Also bitten wir die KI, die Daten über die Fallacies in einer Datenbank zu speichern. Darüber muss die KI lange nachdenken und tut am Ende – nichts. Renitent fragen wir noch mal nach und bekommen jetzt zur Antwort, dass die Daten in eine Datenbank übertragen werden können. Dazu wird ein Firestore für uns hergerichtet, nachdem man im Chat auf „Start“ klickt. Dann erstellt die KI eine `/admin`-Seite in der Anwendung, die einen Button enthält, mit dem wir die Daten hochladen können. Das funktioniert leider wegen eines Berechtigungsfehlers nicht auf Anhieb. Darauf hingewiesen korrigiert die KI den Fehler, anschließend funktioniert es dann mit dem Upload. Danach lassen wir die Admin-Seite, die leider keine weitere Funktion erfüllt, wieder verschwinden.

Bleibt noch die Frage nach der Authentifizierung – diese wird auf Anhieb richtig implementiert. Damit hatte ich nach der etwas holperigen Erfahrung mit der Datenbank gar nicht gerechnet. Unmittelbar kann man ein Konto anlegen – eine separate Bestätigung per E-Mail wird nicht gefordert. Der Login ist problemlos. Die registrierten Benutzer und die Daten im Firestore lassen sich über die Firebase Console verwalten [4].

Für das Deployment muss man erst ein Verrechnungskonto anlegen; das ist ein bisschen aufwendig, wenn man noch nicht im Google-Ökosystem unterwegs ist. Dass man dazu eine Kreditkarte braucht, macht es nicht einfacher. Um an Telemetriedaten zu der veröffentlichten App zu gelangen, muss man erst noch Google Analytics über die Firebase Console aktivieren.

Das Endergebnis haut einen optisch zwar nicht um (vergleiche Bild 1), aber die Frage und alle Antwortmöglichkeiten sowie die Erklärung passen vollständig auf den Bildschirm.

Die Erklärung ist leider auf Englisch, und die KI hat auch zuerst auf Englisch geantwortet, obwohl der Prompt auf Deutsch war. Erst nach einem weiteren deutschen Prompt hat die KI die Sprache gewechselt.

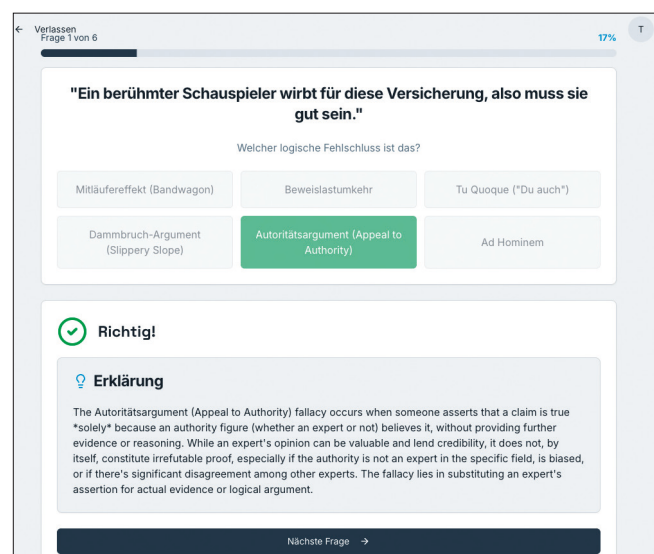
Die Tatsache, dass die KI bei der ersten Erstellung schon Fehler eingebaut hat, wirkt etwas holperig. Auch die Ladezeiten in der Vorschau sind teils schon so lang, dass man denkt, etwas sei nicht in Ordnung. Im Endeffekt hat aber alles funktioniert, und alle gewünschten Funktionen wurden umgesetzt.

Lovable: Der Shooting-Star aus Schweden

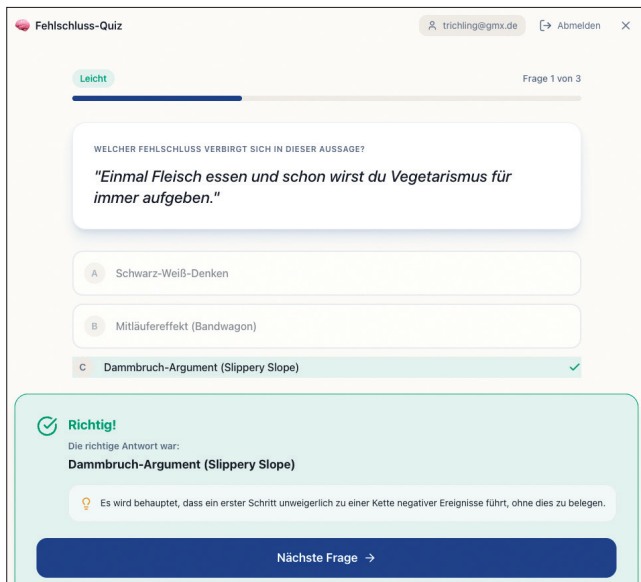
Laut Forbes ist Lovable das am schnellsten wachsende Startup der Geschichte, das die Schallgrenze von 100 Millionen US-Dollar durch zahlende Kunden in nur acht Monaten erreicht hat. Dabei richtet sich Lovable bewusst auch an Nutzer ohne technischen Hintergrund in klassischer Softwareentwicklung. Um den Dienst zu nutzen, ist ein Account erforderlich, den man entweder per E-Mail anlegen kann, oder man greift auf einen bestehenden Google- oder GitHub-Account zurück. Lovable arbeitet mit einem Credit-System. Jede Interaktion mit der AI verbraucht Credits, wobei komplexere Anfragen mehr Credits kosten als einfache. Der Free-Tier bietet 5 Credits täglich (maximal 30/Monat) für öffentliche Projekte. Pro (25 US-Dollar/Monat) umfasst 100 Credits monatlich plus 5 tägliche Bonus-Credits, private Projekte, Custom Domains und GitHub-Sync. Business (50 US-Dollar/Monat) bietet 200 Credits mit SSO und Zugriffskontrollen. Die monatlichen Credits rollen über, die täglichen Bonus-Credits hingegen verfallen [5].

Bei den Frameworks konzentriert sich Lovable auf einen einzigen Stack: React mit Tailwind CSS und Vite. Die AI ist auf diesen Stack optimiert und liefert konsistent gute Ergebnisse. Wer Vue, Angular, Svelte oder gar Blazor einsetzen möchte, schaut in die Röhre – es gibt keine Möglichkeit, den Tech-Stack zu ändern.

Für den Code-Zugriff bietet Lovable drei Wege: Der Chat Mode ist die primäre Arbeitsweise – man beschreibt Features, und Lovable setzt sie um. Der Code Mode ermöglicht direk-



Das Fallacy-Quiz Marke Firebase (Bild 1)



Das Fallacy-Quiz à la Lovable (Bild 2)

ten Zugriff auf den generierten Code zum Ansehen und Bearbeiten. Visual Edits erlauben Anpassungen an Farben und Texten, indem man direkt in der Preview auf das zu ändernde Element klickt und die gewünschte Änderung beschreibt.

Die Git-Integration ist auf GitHub fokussiert. Jedes Projekt kann mit einem Repository synchronisiert werden, bei Änderungen werden automatisch Commits erstellt. GitLab oder Bitbucket werden ebenso wenig unterstützt wie die Option, bereits bestehenden Code zu importieren. Bei der Backend-Integration setzt Lovable primär auf seine proprietäre Cloud-Lösung. Diese basiert auf der Plattform Supabase. Lovable lässt sich dabei aber nicht in die Karten schauen, will heißen: In einem existierenden Supabase-Account sieht man die Lovable-Projekte nicht – Vendor Lock-in inklusive.

Für das Deployment hostet Lovable Projekte auf *lovable.app*-Subdomains. Custom Domains sind ab dem Pro-Plan verfügbar. Wer mehr Kontrolle benötigt, kann den Code über GitHub exportieren. Das Monitoring beschränkt sich auf eingebaute Project Analytics mit grundlegenden Metriken. Für tiefergehendes Monitoring ist die Integration externer Tools erforderlich.

Wie schlägt sich Lovable bei logischen Fehlschlüssen?

Auch Lovable muss natürlich beweisen, was es tatsächlich kann. Nach dem initialen Prompt hat Lovable eine funktionsfähige Webseite mit unserem Quiz umgesetzt. Von den 5 Credits der freien Version sind noch 1,4 übrig – das lässt schon ahnen, dass wir nicht ganz fertig werden. Die Fallacies kommen aus einer statischen Liste in einer Datei *fallacies.ts*. Die Sache mit der Datenbank hat also nicht auf Anhieb funktioniert. Das ist Lovable immerhin auch selbst aufgefallen: „Für die Datenbank Features kannst du Lovable Cloud verbinden.“ Das sei auch „free to start“. Na, das hört sich doch gut an. Gesagt, getan. Nachdem die Cloud verbunden ist, wird direkt als nächste Anfrage „Datenbanken + Auth einrichten“ angeboten. Weiter so! Im Grunde hat das auch gut geklappt, man sieht die geforderten Tabellen, und auch eine Registrierungs- beziehungsweise Login-Seite gibt es. Standardmäßig

erfordert Lovable eine E-Mail-Verifikation für die Erstellung eines Accounts, mit dem Hinweis darauf, dass man diese aus Gründen der besseren Testbarkeit deaktivieren kann. Leider ist das auch nötig, denn die angekündigte E-Mail kommt nicht an. Erst ohne Verifikation gelingt es, einen Account zu erstellen, mit dem man sich dann auch einloggen kann.

Es sieht zunächst so aus, als seien damit bereits alle Anforderungen erfüllt, aber die Datenbanktabellen sind noch leer. Auch das hat Lovable selbst bemerkt und schlägt „Quiz mit Datenbank verbinden“ vor. Dummerweise sind in diesem Moment die 5 kostenlosen Credits aufgebraucht. Da Geduld nicht unbedingt die Stärke des Entwicklers ist, zücken wir die Kreditkarte und werfen 25 US-Dollar ein. Damit wir nicht aus Versehen weitere Kosten produzieren, wird das Abo gleich wieder gekündigt. Die Kündigung sieht man bei Lovable nicht gerne, und so tut sich urplötzlich die Option auf, für nur 4 US-Dollar im Monat eine abgespeckte Pro-Version zu behalten. Hier kann man immerhin noch eine eigene Domain verwenden und den Lovable-Badge deaktivieren. Der Tarif steht so nicht in der Dokumentation – gut, dass wir tiefer eingestiegen sind. Mit frischen Credits ausgerüstet gelingt es, die Daten aus der *fallacies.ts* in die Datenbank zu übertragen. Vor Änderungen an der Datenstruktur fragt Lovable artig nach unserer Zustimmung. Die Fragen werden jetzt zwar aus der Datenbank geladen, aber es werden keine Quiz-Sessions angelegt. Postwendend folgt also die Aufforderung, diese beiden Tabellen beim Anlegen einer neuen Spielrunde zu füllen. Dieser Bitte kommt Lovable zwar nach, aber in der Tabelle *quizsession_questions* werden die Spalten *is_correct* und *selected_fallacy* nicht gefüllt. Muss man denn hier alles selbst machen? Auf erneutes Bitten hin wird der Code zum Füllen der fehlenden Felder ergänzt – warum nicht gleich so? Nachdem alle Anforderungen nun wirklich umgesetzt sind, bleiben von den 100 Credits der Pro-Version noch 90,7 übrig. Alles in allem hätte man sich also noch zwei Tage gedulden müssen, um mit der kostenlosen Version zum Ziel zu kommen.

Das Veröffentlichen der Seite erfordert nur einen Klick auf *Publish*. Die Subdomain von *lovable.app* darf man selbst auswählen, dabei kommt dann ein weniger kryptischer URL rum als bei Firebase. Im Zuge der Veröffentlichung wird die Anwendung noch auf Sicherheitslücken untersucht. Dabei fällt der KI auf, das bestimmte Tabellen anonymen Zugriff erlauben. Mit diesen Fehlern geht der Prozess nicht weiter. Gut aufgepasst und auf Aufforderung hin abgestellt. Mit einem Business-Abo kann man den Zugriff auf die veröffentlichte Version einschränken, mit dem Pro-Plan bleibt nur eine öffentliche Variante übrig. Über das Webinterface von Lovable kann man sich Daten über die Besucher der Webseite wie zum Beispiel deren Herkunft und Verweildauer anzeigen lassen. Ein paar Performance-Metriken, Hinweise zur Barrierefreiheit und zur Suchmaschinenoptimierung bekommt man ebenfalls. Nicht üppig, aber für den Anfang kann man damit sicher schon eine Menge rausholen.

Der gesamte Entwicklungsprozess läuft mit Lovable sehr flüssig. Die Performance ist gut, und die Ergebnisse sind es im Wesentlichen auch. Zwar sind nicht alle Anforderungen im ersten Durchlauf umgesetzt worden, aber Lovable führt ►

den Nutzer sehr gut durch den Prozess, indem der nächste Prompt oft schon vorgeschlagen wird – unter dem Strich hätte die drei Nachprompts aber nicht sein müssen. Im Hinblick auf die Optik ist das Ergebnis gut, aber bei mehr als drei Antwortmöglichkeiten passen nicht mehr alle Antworten auf den Bildschirm (vergleiche [Bild 2](#)). Hier besteht in Sachen UX also noch etwas Nachholbedarf.

Schön ist die Security-Überprüfung vor der Veröffentlichung, die einem zumindest das Gefühl von Sicherheit verschafft. Hier selbst mitzudenken wird aber sicherlich nicht schaden. Leider lässt sich in der kostenlosen Version keine Idee bis zu einem sinnvollen Ergebnis führen, also heißt es bei Lovable, schon mal die Kreditkarte bereitzuhalten.

Bolt.new: Die Entwickler-IDE im Browser

Bolt.new stammt von StackBlitz und bringt eine technische Innovation mit: Die WebContainer-Technologie ermöglicht es, vollständige Node.js-Umgebungen direkt im Browser auszuführen. Das bedeutet: Keine lokale Installation, kein Setup – man öffnet den Browser und kann sofort loslegen.

Bolt.new verwendet ein Token-basiertes Abrechnungsmo-
dell. Tokens werden bei jeder AI-Interaktion verbraucht. Der Free-Tier bietet 150 000 Tokens täglich und 1 Million monatlich. Pro (25 US-Dollar/Monat) umfasst 10 Millionen Tokens und Custom Domains – je mehr Geld man einwirft, desto mehr Tokens bekommt man [6]. Die Menge an Tokens klingt erst mal überwältigend hoch, aber der Token-Verbrauch kann schnell eskalieren – so wurde der Demo-Prompt mit den 150 000 freien Tokens nicht erfüllt, und von den 10 Millionen Pro-Tokens haben die wenigen Nachprompts bereits 800 000 verbraucht.

Bei den Frameworks unterstützt Bolt.new alle JavaScript-Frameworks, die auf StackBlitz laufen: React, Vue, Next.js, Astro, Svelte, SvelteKit, Remix und Angular im Frontend, React Native mit Expo für Mobile, sowie Node.js und Express im Backend. Die Einschränkung: Ausschließlich JavaScript/TypeScript. Sprachen wie PHP, Python oder C# (.NET/Blazor) werden nicht unterstützt. Wenn man keine weiteren Anweisungen gibt, erstellt Bolt die Webseite auf Basis von React mit Vite und Tailwind. Auf Nachfrage ist die KI sogar bereit, die Applikation auf einen anderen Tech-Stack umzubauen – das klappt aber nicht auf Anhieb und kostete für die Demo-Applikation gut 1 Million Tokens. Es ist also empfehlenswert, sich darüber vorher Gedanken zu machen und direkt im ersten Prompt die Wahl des Tools mit einzubeziehen.

Bolt.new ist im Kern eine browserbasierte IDE mit AI-Unterstützung. Der Code ist jederzeit sichtbar und editierbar. Ein vollständiges Terminal im Browser und Live-Preview mit Hot Reload runden das Entwicklererlebnis ab. Damit fügt sich Bolt gut in die Landschaft der bestehenden Tools ein. Die Git-Integration unterstützt GitHub-Export. Der gesamte generierte Code kann in ein Repository übertragen werden – es gibt keinen Vendor Lock-in. Man kann auch direkt mit einem GitHub-Repository starten, damit steht Bolt auf einer Ebene mit Firebase. Auch ein Download der Sourcen als ZIP-Archiv ist möglich. Bei der Backend-Integration bietet Bolt.new mit Bolt Cloud eine integrierte Backend-Lösung, die auf Supabase basiert. Was schön ist: Man kann sich einen unabhängigen

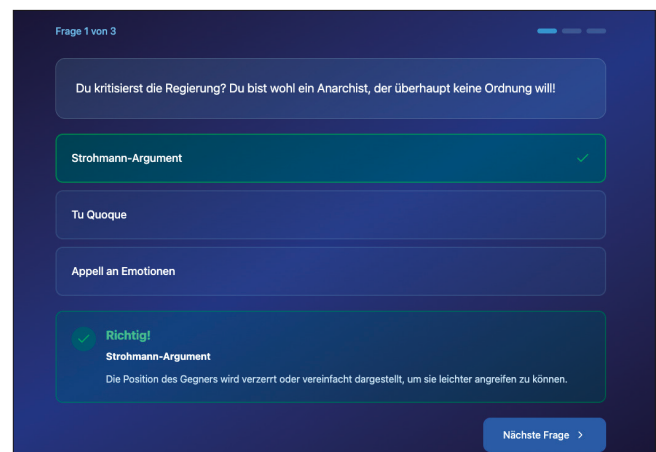
Supabase-Account anlegen und das Backend für die Webseite dorthin umziehen. Damit wird man in puncto Backend unabhängig von Bolt.

Für das Deployment bietet Bolt per One-Click die Möglichkeit, die Webseite auf einer *bolt.host*-Subdomain bereitzustellen. Eigene Domains anzubinden ist ab dem Pro-Abo möglich. Auch hier gefällt es, dass man nicht auf den eingebauten Mechanismus angewiesen ist. Ähnlich wie beim Backend kann man auch für das Deployment einen unabhängigen Netlify-Account verbinden. Das Monitoring, das Bolt mitbringt, ist im Vergleich zu den anderen Anbietern sehr rudimentär, hier wird man sehr bald zu einer entsprechenden eigenen Lösung greifen müssen.

Bolt und die logischen Fehlschlüsse

Selbstredend muss sich auch Bolt in der Erstellung des Logical-Fallacy-Quizspiels beweisen. Und hierbei überrascht es klar im Vergleich zur Konkurrenz: Zunächst kann man sich ab der Pro-Version das zugrunde liegende LLM aussuchen. Hier hat man die Wahl aus den Anthropic-Modellen. Im Free-Tier bekommt man Sonnet 4.5, das gute Ergebnisse liefert. Gleich zu Anfang macht Bolt sich einen Plan, der erfreulicherweise mit der Erstellung der Datenbank beginnt, danach mit dem Authentication-Setup weitergeht, an das sich die Erstellung der Seiten anschließt. Das hört sich mal nach einem guten Plan an – voller Vorfreude erwarten wir das Ergebnis.

Leider enttäuscht das Resultat, denn die Free Credits reichen für die Anforderung nicht aus. Es gibt nicht mal einen Zwischenstand. Also muss man auch hier direkt die Kreditkarte zücken. Hat man die 25 US-Dollar für die nächsten 10 Millionen Tokens nachgeworfen, nimmt Bolt die Arbeit wieder auf und liefert. Und zwar so richtig. Anders als die beiden Konkurrenten funktioniert die Website auf Anhieb, mitsamt aller geforderter Funktionalität. Die Datenbank ist erstellt, angebunden und wird komplett richtig gefüllt. Es ist möglich, sich einen Account zu erstellen und sich einzuloggen, und für die Ergebnisse hat Bolt als einziges der Tools sogar schon eine Seite mit persönlichen Statistiken des Nutzers eingefügt. Zwei kleine Wermutstropfen gibt es. Erstens fehlt der Logout-Button, was sich leicht beheben lässt. Zweitens kann man als



So stellt sich Bolt ein Logical-Fallacy-Quiz vor ([Bild 3](#))

● Weitere AI-gestützte Website-Builder

- **v0 (Vercel):** UI-Komponenten-Generierung für React/Next.js. Keine Backend-Generierung. Ab 20 US-Dollar/Monat [7].
- **Replit Agent:** Cloud-IDE mit AI-Agent. Über 30 Sprachen. Ab 15 US-Dollar/Monat [8].
- **Cursor:** AI-gestützter Code-Editor für professionelle Entwickler. Ab 20 US-Dollar/Monat [9].
- **Wix:** Pionier seit 2016. Fokus auf klassische Websites und E-Commerce [10].
- **Relume:** Sitemaps und Wireframes aus Prompts. Export nach Figma/Webflow [11].
- **Squarespace Blueprint AI:** Website-Builder mit AI. Ideal für Marketing-Seiten [12].

anonymer Nutzer kein Quiz starten, auch das behebt die KI auf einen einfachen Hinweis hin.

Das eingebaute Backend von Bolt funktioniert, aber wir versuchen direkt mal, unsere Fesseln abzustreifen und unser Backend auf ein separates Supabase-Projekt umzuziehen. Bolt führt durch diesen Prozess, der mit wenigen Klicks problemlos klappt. Die bereits bestehenden Daten und Nutzer werden übertragen. Sogar die eigene Backend-Funktionalität funktioniert weiterhin, man kann also quasi über die Bolt-Webseite in den Supabase-Account schauen. Dafür muss man Bolt freilich großzügige Rechte einräumen. Das läuft in Supabase auf Ebene der Organisation, weshalb man aufpassen sollte, was man dort sonst noch so unterbringen möchte.

Beim Veröffentlichen der Seite führt Bolt wie auch Lovable einen Security-Check durch, dem auch auffällt, dass einige unserer Tabellen für anonyme Schreibzugriffe zugänglich sind. Auch hier wiegt man uns also in Sicherheit, selbst prüfen ist aber auch hier sicher besser als blindes Vertrauen. Darüber hinaus ist die Webseite binnen weniger Sekunden auf der `bolt.host`-Subdomain live.

Bild 3 zeigt das Ergebnis von Bolt – optisch macht es nicht das meiste her, aber die Usability ist in Ordnung, auch wenn wie bei Lovable nicht unbedingt alle Antwortmöglichkeiten auf eine Seite passen. Die Tatsache, dass direkt noch eine Seite für Statistiken mitgeliefert wird, tröstet darüber hinweg.

Fazit

Wir haben die drei großen Player im Bereich der AI-unterstützten Website-Builder in Augenschein genommen. Wie im Kasten **Weitere AI-gestützte Website-Builder** zu lesen ist, gibt es noch eine Menge weitere Tools, die in eine ähnliche Richtung gehen, aber oft eher bestimmte Nischen bedienen oder keine vollwertigen Webapplikationen produzieren.

Das Feld der Favoriten zeigt sich in den wesentlichen Merkmalen recht ähnlich. Kein Tool sticht so klar hervor, dass man es uneingeschränkt empfehlen könnte. Wer sich im Google-Universum zu Hause fühlt, für den ist sicherlich Firebase eine gute Wahl. Hierfür spricht auch dessen äußerst großzügiger Free-Tier, der es ermöglicht, eine Idee auch in Gänze

zu realisieren. Leider ist die Performance eher lahm, was schon ein bisschen nervig ist. Lovable macht optisch am meisten her, auch das UI der Webseite selbst ist sehr aufgeräumt. Leider spürt man hier auch am stärksten den Vendor-Lock-in, denn es gibt keinen Ausweg aus dem proprietären Backend. Die Free Credits reichen für einen ersten Entwurf, man gelangt aber schnell an die Grenzen. Ein kleiner Überraschungserfolg war Bolt. Die Ergebnisse waren hier klar am besten: Bereits im ersten Anlauf wurden alle geforderten Funktionalitäten umgesetzt, mit zwei kleinen Nachprompts auch in vollem Umfang funktional. Dafür schlägt das Tool auf der Kostenseite heftig ein – die Tokens rauschen nur so durch.

Abgesehen von Firebase, das vom Tooling her nicht den besten Eindruck gemacht hat, sollte man also davon ausgehen, dass man immer Geld nachwerfen muss. Generell eignen sich alle Tools gut, um eine Idee schnell umzusetzen – Klickdummies gehören damit der Vergangenheit an. Auch nichttechnische User können schnell zu brauchbaren Ergebnissen kommen. Als klassisch geschulter Entwickler bleibt bei mir jedoch eine gesunde Skepsis, wenn es darum geht, ein ernsthaftes Projekt mit solchen Tools umzusetzen. Zu groß ist die Gefahr subtiler Sicherheitslücken, das Umsetzen ernsthafter Backend-Logik bleibt ein Problem, die Flexibilität bei der Umsetzung der Deployments hat Grenzen, und Integrationen in eigene Backendsysteme sind problematisch.

Unter dem Strich sind diese Website-Builder Tools – und „a fool with a tool is still a fool“. Wenn man sich der Einsatzzwecke aber bewusst ist, kann man Prototypen sehr schnell umsetzen – auch wenn man sie vielleicht später noch mal „in echt“ programmieren muss. Den Plan dazu sollte man zumindest schon mal im Hinterkopf haben. Zum Glück gibt es ja auch für diesen Zweck eigene KI-Helferlein. ■

[1] School of Thought, <https://yourlogicalfallacyis.com>

[2] Firebase Studio, <https://firebase.google.com/docs/studio>

[3] Preise, Kontingente und Limits für Firebase Studio, <https://firebase.google.com/docs/studio/pricing?hl=de>

[4] Firebase Console, <https://console.firebase.google.com>

[5] Lovable, Plans and credits, <https://docs.lovable.dev/introduction/plans-and-credits>

[6] Bolt.new, Pricing, <https://bolt.new/pricing>

[7] v0 (Vercel), <https://v0.app/>

[8] Replit, <https://replit.com/~>

[9] Cursor, <https://cursor.com>

[10] Wix, <https://wix.com>

[11] Relume, <https://relume.io>

[12] Squarespace Blueprint AI, <https://squarespace.com>



Tobias Richling

arbeitet als Softwareentwickler und betreut die B2B-Plattformen der apetito AG. Er sucht dabei stets nach Möglichkeiten zur Verbesserung von Funktionalität und Performance. Sie erreichen ihn unter tobias.richling@apetito.de oder [@trichling](https://twitter.com/trichling).

MICROSOFT.EXTENSIONS.DEPENDENCYINJECTION

Magischer DI-Container



Spannende Einblicke in Microsofts zentralen DI-Container.

Im Paket `Microsoft.Extensions.DependencyInjection` steckt das offizielle Framework für Dependency Injection (DI). Es stellt einen zentralen Container bereit, in dem Entwickler Klassen oder Services registrieren können, damit sie später automatisch von .NET instanziiert und verwaltet werden können. Man registriert Klassen oder Services, und .NET erstellt die Objekte, wenn sie gebraucht werden. Als Entwickler muss man sich nicht um Konstruktion, Lebensdauer oder Abhängigkeiten kümmern.

Microsoft hat von Anfang an sehr deutlich gemacht, dass es sich dabei um einen einfachen DI-Container mit Fokus auf Performance und universelle Verwendung handelt. Deswegen bietet er keine erweiterten Funktionen wie etwa Interception. Im Grunde spricht Microsoft sogar von einem unveränderlichen Container, der einmal erzeugt wird und dessen Funktionen und Registrierungen danach unverändert bleiben. Deshalb arbeitet er bei der Auflösung von Instanzen besonders schnell. Der Container durchläuft dafür zwei Phasen: zunächst Setup sowie Konfiguration und danach das einmalige Zusammenbauen der Anleitungen.

.NET

Microsoft.Extensions.DependencyInjection exact match by: aspnet
dotnetframework Microsoft
.NET 5.0 .NET Core 2.0 .NET Standard 2.0 .NET Framework 4.6.1
↓ 6.423.383.472 total downloads last updated a month ago Latest version: 11.0.0-preview.1.26104.118
Default implementation of dependency injection for Microsoft.Extensions.DependencyInjection.

.NET

Microsoft.Extensions.DependencyInjection.Abstractions by: aspnet
dotnetframework Microsoft
.NET 5.0 .NET Core 2.0 .NET Standard 2.0 .NET Framework 4.6.1
↓ 10.260.063.225 total downloads last updated a month ago Latest version: 11.0.0-preview.1.26104.118
Abstractions for dependency injection. Commonly Used Types: Microsoft.Extensions.DependencyInjection.IServiceCollection

Das NuGet-Paket `Microsoft.Extensions.DependencyInjection` (Bild 1)

Konfiguration

Die Konfiguration erfolgt durch das Instanzieren der Klasse `ServiceCollection`. Die meisten Entwickler kommen mit dem Interface `IServiceCollection` in Berührung, weil Microsoft strikt zwischen Implementierung und Abstraktion trennt. Das NuGet-Paket `Microsoft.Extensions.DependencyInjection` enthält die Implementierung des „Motors“ (Bild 1), das Schwes-terpaket `Microsoft.Extensions.DependencyInjection.Abstractions` liefert die Interfaces und die Abstraktionsschicht.

Instanzieren wir eine `ServiceCollection`, werden wir mit den Methoden `AddTransient`, `AddScoped` und `AddSingleton` kon-

frontiert (Bild 2). Die Methoden nehmen entweder ein oder zwei generische Argumente entgegen. Löst man über den Container einen konkreten Typ wie `MyClass` auf, so genügt ein Argument. Will man jedoch auch abstrahieren (etwa über ein Interface), so benötigt man beide Argumente.

Bei genauerem Hinsehen erkennt man, dass `AddTransient`, `AddScoped` und `AddSingleton` Erweiterungsverfahren sind, deren „Wurzel“ immer der letzte Aufruf ist. Es wird ein `ServiceDescriptor` erzeugt, der Kenntnis davon hat, welcher Typ auf welchen Typ gemappt wird, und die Lebenszeit der Registrierung kennt:

- **Transient** ist die einfachste Registrierungsform. Eine Instanz kann vom Container erzeugt werden und wird direkt zurückgegeben. Der Container kümmert sich nicht um den Lifecycle der Instanz, er vergisst quasi sofort, dass er soeben

```
ServiceCollection services = new ();
services.AddTransient<MyClass>();
services.AddScoped<MyClass>();
services.AddSingleton<MyClass>();

services.AddTransient<MyClass, MyClass>();
services.AddScoped<MyClass, MyClass>();
services.AddSingleton<MyClass, MyClass>();

ServiceDescriptor serviceDescriptor = new ServiceDescriptor(typeof(MyClass),
    typeof(MyClass), ServiceLifetime.Singleton);
services.Add(serviceDescriptor);

11 usages
public class MyClass
{
}
```

Alle Formen der Registrierung (Bild 2)

```
ServiceCollection services = new ();
services.AddSingleton<MyClass>();
ServiceProvider provider = services.BuildServiceProvider(new ServiceProviderOptions()
{
    ValidateOnBuild = true,
    ValidateScopes = true
});
```

Der ServiceProvider (Bild 3)

eine Instanz erzeugt hat. Der Aufrufer kann mit der Instanz anschließend machen, was er möchte, und muss sich auch selbst um die Instanz kümmern.

- **Singleton** arbeitet genau umgekehrt: Der Container merkt sich die erzeugte Instanz sein gesamtes „Leben“ lang (meist applikationsweit), und jeder, der erneut einen sol-

Entwicklung (*Environment=DEVELOPMENT*), in der Produktionsumgebung wird es aus Performancegründen deaktiviert. Der ServiceProvider baut sich anschließend für jeden Typ eine *CallSite* auf, die beschreibt, wie er zu diesem Typ kommt.

ServiceProviderEngine

Die ServiceProviderEngine ist der eigentliche Motor, der einen ganzen Abhängigkeitsbaum zusammenbauen kann. Und weil die Anforderungen dafür vielfältig sind, gibt es von dieser Engine nicht nur eine Implementierung, sondern mehrere (Bild 4). Häufig wird die *IL.Emit.ServiceProviderEngine* genutzt, welche IL-Code erzeugt und somit pfeilschnell zu Instanzen kommt, ohne dass dafür brem-

Varianten der
ServiceProvider-
Engine (Bild 4)

internal abstract class ServiceProviderEngine	Implementations of 'ServiceProviderEngine'
<pre>{ IL code public abstract Func<S }</pre>	- ExpressionsServiceProviderEngine (in Microsoft.Extensions.DependencyInjection.ServiceLookup) - IL.Emit.ServiceProviderEngine (in Microsoft.Extensions.DependencyInjection.ServiceLookup) - RuntimeServiceProviderEngine (in Microsoft.Extensions.DependencyInjection.ServiceLookup) - DynamicServiceProviderEngine (in Microsoft.Extensions.DependencyInjection.ServiceLookup)

chen Typ haben möchte, erhält dieselbe Instanz. Implementiert die Instanz auch *IDisposable*, so ruft der Container am Laufzeitende auch die *Dispose*-Funktion auf.

- **Scoped.** Solche Container können einen Scope erzeugen, und innerhalb dieses Scopes wird immer dieselbe Instanz zurückgegeben, allerdings bekommt jeder Scope seine eigene Instanz. Auch hier führt der Container die *Dispose*-Funktion aus, allerdings bereits zum Laufzeitende des Scopes, der in der Regel ein definiertes Ende hat, während die Applikation weiterläuft. In ASP.NET Core etwa gibt es einen Scope für jeden Request, der entgegengenommen wird, und bei Windows Forms oder WPF einen für jedes Fenster.

Die Auflösungsphase

Hat man einen Container konfiguriert, so kann man den ServiceProvider verwenden, der wiederum *IServiceProvider* implementiert (Bild 3). Dem ServiceProvider kann man mitgeben, dass der Container prüfen soll, ob er selbst korrekt konfiguriert wurde und ob die Scopes korrekt zusammenspielen. Das bedeutet beispielsweise, dass in einem Singleton kein Scoped-Typ verwendet wird, da dies nicht möglich ist. In ASP.NET Core tut man das zum Beispiel während der

sende Reflection-Aufrufe erforderlich sind. Nachdem Microsoft mehrere Umgebungen unterstützt, entscheidet man zur Laufzeit, welche Variante die passende ist. Das geschieht über die *RuntimeFeatures* (Bild 5).

Wie schon gesagt, ist IL-Code Emit die schnellste Variante, da der IL-Code ohne weitere Arbeit hochperformant läuft. Voraussetzung dafür ist selbstverständlich, dass es möglich ist, den Code dynamisch zu kompilieren – was zum Beispiel bei MAUI oder WebAssembly nicht klappt. Im schlimmsten Fall kann man auf die *RuntimeServiceProviderEngine* zurückgreifen, die es mit starker Reflection immer noch möglich macht, dass der Container seine Dienste tut. Für Developer ist das am Ende kein Thema, denn Microsoft tut das, was auf der Plattform am schnellsten läuft.

Fazit

Der zentrale DI-Container ist in der .NET-Welt nicht mehr wegzudenken. Unter der Haube ist die ein oder andere magische Funktion zu finden, die dazu führt, dass alles so reibungslos abläuft, wie wir das erwarten. ■

```
_ = RuntimeFeature.IsDynamicCodeSupported;
_ = RuntimeFeature.IsDynamicCodeCompiled;
```

Die *RuntimeFeatures* entscheiden, welcher Motor genutzt wird (Bild 5)



Christian Giesswein

studierte Wirtschaftsinformatik in Wien und entwickelt von klein auf Software mit .NET und C#. In Tirol hat er das Unternehmen Giesswein Software-Solutions gegründet, das sich auf Individualsoftware und Consulting spezialisiert hat. christian@software.tirol

VON LOGGING ZU OBSERVABILITY: OPENTELEMETRY UND CO.

Next-Level Debugging

Erweiterte Fehlerbehandlung und Logging-Strategien in .NET.

Debugging und Laufzeitanalyse in .NET-Anwendungen bedeutet heute weit mehr, als Breakpoints zu setzen und Exceptions im Debugger zu untersuchen: Moderne Anwendungen bestehen aus Web-APIs, Background Services und verteilten Komponenten, in denen Fehler oft nur sporadisch auftreten und sich lokal nicht reproduzieren lassen. Klassisches Logging liefert dabei häufig zu wenig Kontext, um Ursachen zuverlässig zu analysieren. Konzepte wie Structured Logging, verteiltes Tracing und Observability setzen genau hier an: Sie machen Zusammenhänge sichtbar, zeigen den Weg eines Requests über Systemgrenzen hinweg und ermöglichen es, Fehler nicht nur zu erkennen, sondern auch gezielt einzugrenzen.

Über viele Jahre hinweg war Debugging in .NET eine weitgehend kontrollierbare Disziplin: Anwendungen liefen als klassische Desktop- oder Serverprozesse, der Code war lokal verfügbar, Programmlogik ließ sich mit Breakpoints, Step-Through-Debugging und Watch-Fenstern in Visual Studio und Co. zuverlässig untersuchen. Fehler traten meist deterministisch auf, ließen sich reproduzieren und konnten direkt im heimischen Entwicklungswerkzeug analysiert werden.

Mit der zunehmenden Verlagerung hin zu Webanwendungen, cloudbasierten Architekturen, asynchronen Programmiermodellen und verteilten Systemen hat sich dieses Bild

Grenzen, da sich Produktionsprobleme lokal oft nicht nachstellen lassen, Breakpoints in verteilten Systemen kaum praktikabel sind und isolierte Log-Einträge selten den notwendigen Kontext liefern.

Um stabile Systeme zu entwickeln, reicht es daher nicht mehr aus, einzelne Codepfade zu debuggen: Notwendig ist ein ganzheitlicher Blick auf Laufzeitverhalten, Abhängigkeiten und Systemzustände, der Ursachen sichtbar macht, bevor sie sich in schwer reproduzierbaren Fehlern oder instabilem Betriebsverhalten äußern.

Nicht alles ist ein Log: Logging versus Tracing versus Metrics versus Auditing

Logging ist seit jeher das zentrale Mittel, um Informationen über den Zustand einer Anwendung festzuhalten. Log-Einträge dokumentieren Ereignisse zu einem bestimmten Zeitpunkt, etwa das Auftreten einer Exception, den Abschluss eines Verarbeitungsschritts oder das Ergebnis einer Validierung. Diese Informationen sind wertvoll, um bekannte Fehler nachzuvollziehen oder um Hinweise auf ungewöhnliches Verhalten zu erhalten. Abgesehen davon, dass viele Entwickler immer noch Exceptions zur Steuerung des logischen Programmablaufs missbrauchen, stößt in modernen Anwendungen klassisches Logging [2] schnell an Grenzen: Log-Einträge stehen oft isoliert nebeneinander, zeigen keinen direkten Bezug zueinander und liefern nur begrenzten Kontext darüber, wie ein bestimmter Zustand entstanden ist. Besonders in verteilten Systemen ist es schwierig, aus einzelnen Logzeilen den vollständigen Ablauf eines Requests oder die Ursache eines Fehlers zu rekonstruieren.

Verteiltes Tracing [3] setzt genau dort an, wo Logging an seine Grenzen stößt: Statt einzelne Ereignisse isoliert zu betrachten, wird der vollständige Zyklus eines Requests oder einer Ausführung über mehrere Komponenten hinweg verfolgt. Jeder Verarbeitungsschritt wird als „Operation“ erfasst und einem übergeordneten Trace zugeordnet. Dadurch entsteht ein zeitlich und logisch zusammenhängendes Bild, das zeigt, welche Services beteiligt waren, wie lange einzelne Schritte gedauert haben und an welcher Stelle Fehler oder Verzögerungen aufgetreten sind. Tracing beantwortet damit Fragen, die mit Logging allein kaum lösbar sind, etwa warum ein Request oder eine Methode langsam war oder in welchem Service eine Exception ursprünglich ausgelöst wurde.

Während Logs und Traces konkrete Einzelereignisse und Abläufe abbilden, liefern Metriken [4] eine aggregierte Sicht auf das Systemverhalten: Sie fassen Messwerte wie Antwort-

```
try
{
    await orderService.ProcessOrder(order);
}
catch (InvalidOperationException exception)
{
    logger.LogError(exception,
        "Order processing failed for OrderId {OrderId}", order.Id);

    throw; // rethrowing the exception
}
```

Try-Catch-Block in C#: Ein Fehler beim Verarbeiten einer Bestellung wird geloggt und anschließend erneut geworfen (rethrow) (Bild 1)

grundlegend verändert: Moderne .NET-Anwendungen bestehen aus mehreren Services, nutzen externe APIs, Message Queues und Hintergrundprozesse und werden häufig dynamisch skaliert oder in Containern betrieben. Fehler entstehen dadurch nicht mehr an einer klar definierten Stelle, sondern ergeben sich immer mehr aus dem Zusammenspiel vieler Komponenten, zeitlicher Abhängigkeiten und nichtdeterministischer Umgebungen, Abhängigkeiten oder Laufzeitbedingungen. Klassisches Debugging [1] stößt hier schnell an

```
using (IDisposable scope = logger.BeginScope("Processing order {OrderId}",
order.Id))
{
    logger.LogInformation("Validating order");
    await Validate(order);

    logger.LogInformation("Persisting order");
    await Save(order);
}
```

Logging mit BeginScope zur Nachverfolgung von Validierung und Speicherung einer Bestellung (Bild 2)

zeiten, Fehlerraten oder Ressourcenauslastung über Zeiträume hinweg zusammen und machen Trends sowie Abweichungen sichtbar.

Metriken eignen sich besonders gut, um den allgemeinen Gesundheitszustand einer Anwendung beziehungsweise eines Systems zu überwachen oder um Schwellenwerte und Alarmer zu definieren, die anschließend zum Beispiel auf Dashboards angezeigt werden können. Sie erklären jedoch nicht, warum ein bestimmter Wert erreicht wurde, sondern zeigen lediglich, dass sich das Systemverhalten verändert hat. Für eine Ursachenanalyse müssen Metriken daher immer mit Logs oder Traces kombiniert werden.

Auditing verfolgt ein anderes Ziel als klassisches Logging oder Tracing: Im Vordergrund steht nicht die technische Fehleranalyse, sondern die revisionssichere Nachvollziehbarkeit von fachlichen oder sicherheitsrelevanten Aktionen. Audit-Logs dokumentieren beispielsweise, wer eine bestimmte Änderung vorgenommen hat, wann sie erfolgt ist und welche Daten betroffen waren. Diese Informationen sind häufig rechtlich oder organisatorisch relevant und unterliegen besonderen Anforderungen an Vollständigkeit, Integrität und Aufbewahrung, weshalb sie nicht in klassischen Logging-Systemen, sondern in der Regel getrennt und abhängig von der individuellen Applikationsimplementierung in Datenbanken persistiert werden. Für Debugging-Zwecke sind Audit-Daten nur eingeschränkt geeignet, sie spielen jedoch eine zentrale Rolle, wenn es um Compliance, Sicherheit oder forensische Analysen geht.

In modernen Anwendungen haben Logging, Tracing, Metrics und Auditing jeweils klar getrennte Aufgaben und ergänzen sich gegenseitig. Erst ihr Zusammenspiel ermöglicht es, komplexe Systeme stabil zu betreiben und Ursachen zuverlässig zu analysieren.

Logging-Grundlagen: Exceptions, Logging, Levels, Scopes und Correlation IDs

Sauberes Logging beginnt mit einer klaren Trennung zwischen Fehlerbehandlung und Protokollierung: Exceptions [5] dienen einzig und allein dazu, fehlerhafte Zustände zu behandeln, und sollen nicht als Logik- oder reines Logging-Instrument missverstanden werden. Während eine Exception beschreibt, dass etwas wirklich schiefgelaufen ist (inklusive teurem Stack

Trace [6]), liefert das Logging den notwendigen Kontext, um später zu verstehen, warum es dazu kam und unter welchen Bedingungen der Fehler überhaupt aufgetreten ist.

In .NET sollte daher nicht jede Exception automatisch geloggt werden, sondern nur dort, wo zusätzlicher fachlicher oder technischer Kontext sinnvoll ergänzt werden kann.

Die Wahl des passenden Log-Levels [2] spielt dabei eine zentrale Rolle. *Trace* und *Debug* eignen sich für detaillierte Entwicklungsinformationen, *Information* beschreibt den normalen Ablauf

einer Anwendung, *Warning* weist auf unerwartete, aber beherrschbare Situationen hin, und *Error* kennzeichnet fehlgeschlagene Vorgänge. *Critical* bleibt schwerwiegenden Fehlern vorbehalten, bei denen ein stabiler Weiterbetrieb nicht mehr möglich ist.

Eine saubere Trennung der Log-Levels ist Voraussetzung dafür, Logs später wie in Bild 1 gezeigt gezielt filtern und auswerten zu können.

```
string correlationId = httpContext.TraceIdentifier;

logger.LogInformation("CorrelationId: {CorrelationId}", correlationId);
```

Protokollierung der CorrelationId (aus TraceIdentifier) zur anfrageübergreifenden Nachverfolgung von Logs (Bild 3)

Ein weiterer wichtiger Baustein sind Logging-Scopes [7] (Bild 2): Sie ermöglichen es, mehrere Log-Einträge logisch zusammenzufassen und mit gemeinsamem Kontext anzureichern, etwa einer Order-ID oder einer Benutzerkennung. Gerade bei der Verarbeitung mehrerer Schritte innerhalb eines Requests erhöhen Scopes die Lesbarkeit der Logs erheblich – ohne dazu jeden Log-Eintrag manuell erweitern zu müssen.

In verteilten Systemen reicht lokaler Kontext jedoch nicht aus: Hier kommen Correlation IDs (auch Trace IDs oder Activity IDs genannt) [8] ins Spiel, mit denen sich zusammengehörige Log-Einträge über Service-, Prozess-, und Instanzgrenzen hinweg verknüpfen lassen (Bild 3). Häufig wird dafür eine Request-ID [9] aus einem HTTP-Header übernommen oder neu erzeugt und konsequent in Logs, Scopes und weitergeleiteten Requests verwendet. Nur so lässt sich später der vollständige Weg eines Requests nachvollziehen.

Durch das bewusste Zusammenspiel von Exceptions, sinnvoll eingesetzten Log-Levels, Scopes und Correlation IDs ►

```
private readonly ILogger<OrdersController> _logger;
public OrdersController(ILogger<OrdersController> logger)
{
    _logger = logger;
}
```

Dependency Injection von `ILogger<OrdersController>` zur zentralen Nutzung von Logging im Controller (Bild 4)

entstehen Logs, die nicht nur Informationen enthalten, sondern echten Kontext liefern. Sie bilden damit die Grundlage für weiterführende Konzepte wie verteiltes Tracing und Observability, ohne die modernes Debugging in .NET-Anwendungen kaum noch möglich ist.

Logging in .NET: ILogger, Serilog und Full-Structured Logging

ASP.NET Core bringt mit Microsoft.Extensions.Logging [10] eine konsistente Logging-Abstraktion mit, die in praktisch jedem Hosting-Szenario – ganz gleich ob Konsolenanwendung,

Im Gegensatz zu klassischen String-Logs, die primär für menschliche Leser gedacht sind, bilden strukturierte Logs eine belastbare Datengrundlage für Suche, Analyse und Debug-

```
[HttpPost]
public IActionResult Create([FromBody] CreateOrderRequestModel request)
{
    _logger.LogInformation(
        "Create order request received. CorrelationId {CorrelationId}, CustomerId {CustomerId}",
        HttpContext.TraceIdentifier, request.CustomerId);

    return Ok();
}
```

Logging eines eingehenden POST-Requests mit CorrelationId und CustomerId im Controller zur besseren Nachvollziehbarkeit von API-Aufrufen (Bild 5)

```
// string interpolation
_logger.LogInformation(
    $"Create order request received. CorrelationId {HttpContext.TraceIdentifier}, CustomerId {request.CustomerId}");

// structured logging
_logger.LogInformation(
    "Create order request received. CorrelationId {CorrelationId}, CustomerId {CustomerId}",
    HttpContext.TraceIdentifier, request.CustomerId);
```

Vergleich von String-Interpolation und strukturiertem Logging: Strukturierte Platzhalter ermöglichen besser durchsuchbare Log-Daten (Bild 6)

WinForms oder WPF, ASP.NET, ASP.NET Core – funktioniert und sich über Provider erweitern lässt. In Controllern, Minimal APIs und Services wird ein *ILogger<T>* per Dependency Injection [11] injiziert, sodass Logs kontextbezogen (Kategorie = Typname) erzeugt werden können, ohne sich früh auf eine konkrete Logging-Bibliothek festlegen zu müssen (Bild 4).

Gleichzeitig „schenkt“ ASP.NET Core bereits nützlichen Kontext: Die Correlation ID wird in Form des *HttpContext.TraceIdentifier* automatisch generiert und kann direkt in Logs verwendet werden. Auch Informationen wie die aktuelle Route, HTTP-Methode oder User-Identität lassen sich über Log-Scopes leicht hinzufügen, ohne jeden Log-Eintrag manuell erweitern zu müssen (Bild 5).

Was in Bild 6 zu sehen ist, ist des Weiteren nicht nur Logging, sondern Full-Structured Logging [12]: Log-Einträge werden nicht als fertig formatierte Strings erzeugt, sondern als strukturierte Ereignisse, die aus einer Nachricht und klar definierten Properties bestehen. Werte wie *CustomerId*, *OrderId* oder *CorrelationId* werden dabei nicht in Text interpoliert, sondern als eigenständige Felder erfasst und gespeichert. Dadurch bleiben sie maschinenlesbar und können in zentralen Log-Systemen gezielt gefiltert, aggregiert und miteinander korreliert werden.

```
[18:36:09 INF] Now listening on: https://localhost:5001
[18:36:09 INF] Application started. Press Ctrl+C to shut down.
[18:36:09 INF] Hosting environment: Development
[18:36:09 INF] Content root path: C:\source\benabt\dotnetpro\aspnetcore-serilog
```

Konsolen-Startup-Logs der ASP.NET-Core-Anwendung mit Serilog (Bild 7)

ging in produktiven Umgebungen. ASP.NET Core unterstützt diesen Ansatz bereits über Message Templates im ILogger. Sein volles Potenzial entfaltet er jedoch erst dann, wenn die

```
using Serilog;

WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Host.UseSerilog((ctx, lc) => lc
    // Min Level = Information, damit wir die Logs auch in der Console sehen
    .MinimumLevel.Information());

// Ausgabe in der Console
    .WriteTo.Console());
);

WebApplication app = builder.Build();

app.MapGet("/", () => "Hallo dotnetpro mit Serilog!");

// => dotnet run
app.Run();
```

Grundkonfiguration von Serilog in ASP.NET Core mit Mindest-Loglevel-Information und Konsolenausgabe (Bild 8)

Ausgabe selbst strukturiert erfolgt – typischerweise als JSON – und zusätzlicher Kontext automatisch angereichert wird, wie es etwa Serilog als dedizierte Logging-Pipeline ermöglicht.

Serilog als strukturierte Logging-Pipeline

Serilog [13] – der mit Abstand am meisten verbreitete Logging-Provider für .NET – baut auf den strukturierten Message Templates von ILogger auf, geht jedoch einen entscheidenden Schritt weiter, indem es Logs konsequent als strukturierte Events behandelt: Statt Log-Einträge lediglich zu formatieren, reichert Serilog sie automatisch mit Kontextinformationen an, etwa zur Anwendung, Umgebung oder zum aktuellen Request, und gibt sie standardmäßig in maschinenlesbaren Formaten wie JSON aus (Bild 7).

Dadurch eignen sich Serilog-Logs nicht nur zum Lesen, sondern sie lassen sich effizient in zentrale Log-Systeme einspeisen, durchsuchen und korrelieren. In ASP.NET Core wird Serilog häufig als Host-Logger integriert, sodass sowohl eigene Log-Einträge als auch Framework-Logs einheitlich und strukturiert erfasst werden. Die Konfiguration erfolgt dabei entweder über ein Fluent-API, das es ermöglicht, Log-Levels, Ausgabeziele (Sinks) und Anreicherungen flexibel zu definieren, oder – der empfohlene Weg – über die `appsettings.json`, was insbesondere in produktiven Umgebungen von Vorteil ist, da sich die Logging-Konfiguration ohne Codeänderungen anpassen lässt (Bild 8).

Beim Aufruf von `https://localhost:5001/` werden die Logs von Serilog automatisch mit Kontextinformationen angereichert und in der Konsole ausgegeben, wie in Bild 9 gezeigt.

Eine zentrale Rolle bei Serilog spielen die sogenannten Sinks [14]: Serilog alleine ist die Basis des Logging-Providers, die Sinks definieren jedoch, wohin Log-Events geschrieben werden; etwa in die Konsole, in Dateien, in Datenbanken oder in zentrale Log-Systeme wie Azure Log Analytics oder Seq.

Sinks sind nicht nur daher einfache Ausgabekanäle wie bei anderen Implementierungen, sondern übernehmen häufig zusätzliche Aufgaben wie Pufferung, asynchrones Schreiben oder die strukturierte Aufbereitung der Logdaten. Dadurch lässt sich das Logging flexibel an unterschiedliche Anforderungen und Umgebungen anpassen, ohne Code zu verändern. Gerade in produktiven Systemen macht diese Trennung von Log-Erzeugung und Log-Verarbeitung Serilog besonders mächtig und zu einem zentralen Baustein für skalierbares Debugging und Monitoring.

Von Logging zu Observability: OpenTelemetry und Distributed Tracing

Logging ist unverzichtbar, reicht jedoch nicht aus, um moderne .NET-Systeme zu verstehen: Einzelne Log-Einträge liefern Ereignisse, aber kaum Kontext über Ursachen oder Zusammenhänge, insbesondere in verteilten Architekturen. Observability [15] setzt hier an und macht Systeme anhand ihrer Laufzeitdaten nachvollziehbar.

Observability stützt sich auf drei Säulen: Logs, Metrics und Traces: Logs halten Ereignisse fest, Metriken zeigen aggregierte Zustände und Traces machen den vollständigen Ablauf – zum Beispiel eines Web-Requests – über Systemgrenzen hinweg sichtbar. Distributed Tracing ist damit der Schlüssel zur Ursachenanalyse und häufig der Einstieg in Observability. ►

```
[18:37:38 INF] Request starting HTTP/2 GET https://localhost:5001/
[18:37:38 INF] Executing endpoint 'HTTP: GET /'
[18:37:38 INF] Executed endpoint 'HTTP: GET /'
[18:37:38 INF] Request finished HTTP/2 GET https://localhost:5001/ - 13.4837ms
```

Log-Formatierung durch Serilog beim Aufruf eines Endpunkts (Bild 9)

```
using System.Diagnostics;
using OpenTelemetry.Metrics;
using OpenTelemetry.Resources;
using OpenTelemetry.Trace;

WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

string serviceName = "aspnetcore-opentelemetry";
string serviceVersion = "1.0.0";

builder.Services.AddOpenTelemetry()
    .ConfigureResource(resource => resource
        // Service Name und Version als Identifizierung für die Telemetrie-Daten.
        .AddService(serviceName, serviceVersion)
        // Zusätzliche Attribute, z.B. die Umgebung (Development, Production, etc.).
        .AddAttributes(new Dictionary<string, object>
        {
            ["deployment.environment"] = builder.Environment.EnvironmentName
        })
    // Registrierung von Tracing-Quellen.
    .WithTracing(tracing => tracing
        // Aktivieren von ASP.NET Core Instrumentation, um
        // automatisch Telemetrie-Daten für *eingehende* HTTP-Anfragen zu erfassen.
        .AddAspNetCoreInstrumentation()
        // Aktivieren von HTTP Client Instrumentation, um
        // automatisch Telemetrie-Daten für *ausgehende* HTTP-Anfragen zu erfassen.
        .AddHttpClientInstrumentation()
        // Ausgabe der gesammelten Tracing-Daten in der Konsole.
        .AddConsoleExporter()
    );

WebApplication app = builder.Build();

// Beispiel-Endpoint, um die automatische Erfassung von Telemetrie-Daten sowie
// die Anreicherung von Spans mit benutzerdefinierten Tags zu demonstrieren.
app.MapGet("/", () =>
{
    // Den automatisch erzeugten Server-Span der ASP.NET Core Instrumentation anreichern.
    Activity.Current?.SetTag("order.id", Guid.NewGuid().ToString());

    return "Hallo dotnetpro mit OpenTelemetry!";
});

app.Run();
```

OpenTelemetry-Setup in ASP.NET Core mit Service-Metadaten, automatischer HTTP-Instrumentierung und Konsolen-Exporter für Tracing (Bild 10)


```

info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\source\benabt\dotnetpro\aspnetcore-opentelemetry
Activity.TraceId:      cb735a9cbee781f685e79f3968071db6
Activity.SpanId:       eb7ed63744157205
Activity.TraceFlags:    Recorded
Activity.DisplayName:   GET /
Activity.Kind:          Server
Activity.StartTime:     2026-02-07T18:09:43.7560461Z
Activity.Duration:      00:00:00.1208556
Activity.Tags:
  server.address: localhost
  server.port: 5001
  http.request.method: GET
  url.scheme: https
  url.path: /
  network.protocol.version: 2
  user_agent.original: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
  (KHTML, like Gecko) Chrome/144.0.0.0 Safari/537.36
  order.id: 77c4c7e1-3045-4fde-945a-fe50b748bcba
  http.route: /
  http.response.status_code: 200
Instrumentation scope (ActivitySource):
  Name: Microsoft.AspNetCore
Resource associated with Activity:
  deployment.environment: Development
  service.name: aspnetcore-opentelemetry
  service.namespace: 1.0.0
  service.instance.id: ab6ba52f-e175-4c54-9fcd-bcc3bdcd89e6
  telemetry.sdk.name: opentelemetry
  telemetry.sdk.language: dotnet
  telemetry.sdk.version: 1.15.0

```

Konsolenausgabe eines OpenTelemetry-Traces mit HTTP-Request-Metadaten, Tags und Service-Informationen (Bild 11)

Der Framework- und System-übergreifende Standard für verteiltes Tracing ist OpenTelemetry [16]: OpenTelemetry ist ein offener, herstellernerutraler Standard zur Erfassung von Traces, Metriken und Logs. Es ist kein Monitoring-Backend, sondern definiert eine einheitliche Telemetrie-„Sprache“, mit der Daten einmal im Code instrumentiert und flexibel an unterschiedliche Zielsysteme exportiert werden können. Im Kern basiert OpenTelemetry auf einem Tracing-Modell aus Traces und Spans: Ein Trace beschreibt einen Request, ein Span einen einzelnen Verarbeitungsschritt wie einen Endpunkt-Aufruf, einen HTTP-Request oder eine Datenbankabfrage. Diese Struktur macht Abläufe, Laufzeiten und Fehlerquellen transparent.

Durch die enge Integration in .NET und ASP.NET Core lässt sich OpenTelemetry ähnlich unkompliziert wie Serilog mit wenigen Zeilen Code aktivieren: Nach der Registrierung sammelt das Framework – abhängig von der individuellen Konfiguration – automatisch Telemetriedaten für eingehende HTTP-Anfragen, ausgehende HTTP-Requests, Datenbankzugriffe und weitere typische Infrastrukturkomponenten. Die erfassten Traces können anschließend über sogenannte Exporter – funktional vergleichbar mit Serilog-Sinks – entweder direkt in der Konsole ausgegeben oder an zentrale Tracing-Systeme wie Jaeger, Zipkin oder Azure Monitor übertragen werden (Bild 10). Dadurch ist es möglich, Telemetriedaten

lokal zu inspizieren oder sie in produktionsnahen Umgebungen zentral auszuwerten, ohne den Anwendungscode grundlegend anzupassen.

Beim Starten der Anwendung und dem Aufruf von `https://localhost:5001/` erzeugt die ASP.NET-Core-Instrumentation automatisch einen Server-Span für die eingehende HTTP-Anfrage. Dieser enthält neben technischen Metadaten wie HTTP-Methode, Route und Statuscode auch benutzerdefinierte Tags, die während der Request-Verarbeitung ergänzt wurden. Die Konsolenausgabe zeigt dabei nicht nur den einzelnen Span, sondern auch die zugehörigen Ressourceninformationen wie Service-Name, Umgebung und SDK-Version, die später eine eindeutige Zuordnung in zentralen Tracing-Systemen ermöglichen (Bild 11).

Spans gezielt anreichern: Kontext statt nur Technik

Die automatische Instrumentierung von OpenTelemetry liefert bereits wertvolle technische Informationen, etwa zu HTTP-Anfragen, Antwortzeiten oder Statuscodes. Für eine effektive Fehleranalyse in realen Anwendungen reicht dieser technische Kontext jedoch oft nicht aus. In der realen Welt interessieren weniger einzelne Requests als vielmehr die fachlichen

Zusammenhänge dahinter: Welche Bestellung wurde verarbeitet? Für welchen Kunden trat der Fehler auf? In welchem Verarbeitungsschritt kam es zu einer Ausnahme? Genau hier kommt die gezielte Anreicherung von Spans ins Spiel.

OpenTelemetry erlaubt es, bestehende Spans [17] mit zusätzlichen Attributen zu versehen oder eigene Spans für fachlich relevante Operationen zu erzeugen. Diese Informationen werden dem aktuellen Trace hinzugefügt und stehen später in Tracing-Tools zur gezielten Filterung und Analyse zur Verfügung. Wichtig ist dabei, bewusst zwischen technischem und fachlichem Kontext zu unterscheiden und nur solche Daten zu

```

app.MapPost("/orders", async (Order order) =>
{
    Activity.Current?.SetTag("order.id", order.Id);
    Activity.Current?.SetTag("order.amount", order.Amount);
    Activity.Current?.SetTag("order.source", "web-api");

    // Fachliche Verarbeitung
    await ProcessOrder(order);

    return Results.Ok();
});

```

Anreicherung des aktuellen OpenTelemetry-Spans mit fachlichen Tags (Bild 12)

```

async Task ProcessOrder(Order order)
{
    // Erstellen einer neuen ActivitySource für
    // die Auftragsverarbeitung, um fachliche Spans zu erzeugen.
    ActivitySource OrderProcessingActivitySource = new("OrderProcessing");

    // Erstellen eines neuen Spans für die Validierung des Auftrags.
    using Activity? activity = OrderProcessingActivitySource
        .StartActivity("ValidateOrder");

    // Den Span um relevante Informationen
    // anreichern, z.B. die Auftrags-ID.
    activity?.SetTag("order.id", order.Id);

    // Simulieren von Validierungsschritten, z.B.
    // durch eine Datenbankabfrage oder Regeln.
    await Validate(order);

    // Erstellen eines neuen Spans für die Zahlungsausführung.
    using Activity? paymentActivity = OrderProcessingActivitySource
        .StartActivity("ProcessPayment");

    // Simulieren der Zahlungsausführung, z.B.
    // durch einen Aufruf an einen Zahlungsdienstleister.
    await ProcessPayment(order);
}

```

Manuelles Erzeugen fachlicher OpenTelemetry-Spans (ValidateOrder, ProcessPayment) mit ActivitySource zur detaillierten Auftragsverfolgung (Bild 13)

erfassen, die für Diagnosezwecke tatsächlich relevant sind – denn zu viele irrelevante Informationen können die Analyse erschweren, die Performance beeinträchtigen und tatsächlich auch negative Auswirkungen auf die Kosten haben, wenn sie in zentralen Tracing-Systemen gespeichert werden.

Im einfachsten Fall kann der automatisch erzeugte Server-Span während der Request-Verarbeitung wie in Bild 12 gezeigt um zusätzliche Attribute ergänzt werden. Solche Tags ermöglichen es später, Traces gezielt nach bestimmten Geschäftsobjekten oder Verarbeitungspfaden zu durchsuchen. In komplexeren Szenarien ist es jedoch sinnvoll, eigene Spans zu definieren, um einzelne Verarbeitungsschritte klar voneinander abzugrenzen. Dazu wird eine eigene *ActivitySource* [18] verwendet, die als Quelle für benutzerdefinierte Spans dient (Bild 13).

Durch diese Struktur entsteht ein hierarchischer Trace, der den fachlichen Ablauf klar widerspiegelt und es ermöglicht, einzelne Schritte unabhängig voneinander zu analysieren (Bild 14). Gerade bei Performance-Problemen oder sporadischen Fehlern wird so schnell sichtbar, welcher Teil der Verarbeitung betroffen ist.

Entscheidend ist dabei nicht die Anzahl der Spans, sondern deren Aussagekraft: Gut platzierte, fachlich aussagekräftige Spans liefern einen deutlich größeren Mehrwert als eine Vielzahl rein technischer Telemetrie-Daten.

Spans sollten gezielt und mit Bedacht angereichert werden, da nicht jede Information für Tracing-Zwecke geeignet ist: Sensible oder personenbezogene Daten wie Zugangstoken, Passwörter oder vollständige Inhalte haben in Spans nichts verloren, da Telemetriedaten häufig zentral gespeichert und ausgewertet werden.

Genauso ungeeignet sind große oder unstrukturierte Datenmengen wie vollständige Request- oder Response-Bodies,

da sie die Qualität der Diagnose nicht erhöhen, aber Speicherbedarf und Kosten enorm steigern können. Attribute sollten immer einen klaren Zweck erfüllen und zur Analyse von Abläufen, Fehlern oder Performance beitragen, statt wahllos technische Details zu sammeln. Spans beschreiben zeitlich begrenzte Operationen und ersetzen weder strukturierte Logs noch Metriken, sondern entfalten ihren Nutzen erst durch eine klare Abgrenzung und das bewusste Zusammenspiel mit anderen Observability-Disziplinen.

Metriken: Trends erkennen und als Frühwarnsystem nutzen

Während Logs und Traces dabei helfen, konkrete Fehlerfälle oder einzelne Requests nachzuvollziehen, verfolgen Metriken [19] einen anderen Ansatz. Sie beschreiben das Laufzeitverhalten einer Anwendung aggregiert über die Zeit und machen Entwicklungen sichtbar, die in einzelnen Log-Einträgen verborgen bleiben würden. Steigende Antwortzeiten, zunehmende Fehlerraten oder wiederkehrende Lastspitzen lassen sich so frühzeitig erkennen und einordnen. Gerade für Dashboards, Alarmierung und langfristige Beobachtung sind Metriken daher unverzichtbar, da sie stabiler und besser vergleichbar sind als einzelne Ereignisse.

Damit Metriken über Tools und Teams hinweg konsistent bleiben, folgen sie in OpenTelemetry klaren Konventionen ►

```

Activity.TraceId:      c7ea1e4fe19e77698226403eb3e53865
Activity.SpanId:      6f7cfce497e9a122
Activity.TraceFlags:   Recorded
Activity.DisplayName:  POST /orders
Activity.Kind:         Server
Activity.StartTime:    2026-02-07T18:28:38.9318734Z
Activity.Duration:     00:00:00.1047954
Activity.Tags:
  server.address: localhost
  server.port: 5001
  http.request.method: POST
  url.scheme: https
  url.path: /orders
  network.protocol.version: 1.1
  user_agent.original: PostmanRuntime/7.51.1
  order.id: ed94d902-5a67-478a-b7c6-ac965e849c0e
  order.amount: 1
  order.source: web-api
  http.route: /orders
  http.response.status_code: 200
Instrumentation scope (ActivitySource):
  Name: Microsoft.AspNetCore
Resource associated with Activity:
  deployment.environment: Development
  service.name: aspnetcore-opentelemetry
  service.namespace: 1.0.0
  service.instance.id: 8ccefcf4-6a69-40d7-90f5-b2647204d5f9
  telemetry.sdk.name: opentelemetry
  telemetry.sdk.language: dotnet
  telemetry.sdk.version: 1.15.0

```

OpenTelemetry-Trace eines POST /orders-Requests mit fachlichen Tags und HTTP-Metadaten (Bild 14)


```

WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

// Eigener Meter für Custom-Metriken, i.d.R. Domänen-spezifisch,
// z.B. "OrderMetrics" für Bestellungen-bezogene Metriken.
Meter meter = new("OrderMetrics", "1.0.0");

// Zähler für erstellte Bestellungen
Counter<long> ordersCreated = meter.CreateCounter<long>(
    "orders.created",
    description: "Number of created orders");

// OpenTelemetry mit Metriken konfigurieren
builder.Services.AddOpenTelemetry()
    .ConfigureResource(resource => resource
        // Service Name und Version als Identifizierung für die Telemetrie-Daten.
        .AddService(serviceName, serviceVersion)
        // Zusätzliche Attribute, z.B. die Umgebung (Development, Production, etc.).
        .AddAttributes(new Dictionary<string, object>
        {
            ["deployment.environment"] = builder.Environment.EnvironmentName
        })
        // Registrierung von Tracing-Quellen.
        .WithTracing(tracing => tracing
            // .. Tracing aus vorherigem Beispiel
        )
        // Registrierung von Metriken.
        .WithMetrics(metrics =>
        {
            // Aktivieren von ASP.NET Core Instrumentation, um automatisch
            // Metriken für *eingehende* HTTP-Anfragen zu erfassen.
            metrics.AddAspNetCoreInstrumentation();

            // Aktivieren von HTTP Client Instrumentation, um automatisch
            // Metriken für *ausgehende* HTTP-Anfragen zu erfassen.
            metrics.AddHttpClientInstrumentation();

            // Eigene Metriken registrieren
            metrics.AddMeter("OrderMetrics");

            // Ausgabe der Metriken in der Konsole
            metrics.AddConsoleExporter((exporterOptions, metricReaderOptions) =>
            {
                // Periodische Ausgabe der Metriken alle 1000 Millisekunden (1 Sekunde).
                metricReaderOptions
                    .PeriodicExportingMetricReaderOptions
                    .ExportIntervalMilliseconds = 1000;
            });
        });
    );

```

OpenTelemetry-Metriken in ASP.NET Core mit eigenem Meter (OrderMetrics), Counter orders.created und periodischer Konsolen-Ausgabe (Bild 15)

[20] – inklusive Regeln für Namen, Attribute und insbesondere Einheiten: Instrument-Units sollen sich an UCUM (Unified Code for Units of Measure, ein standardisiertes System zur eindeutigen, maschinenlesbaren Darstellung von Maßeinheiten) orientieren, Zähler sind in der Regel einheitslos („1“) und Zeitdauern werden standardisiert in Sekunden (s) angegeben.

Das Beispiel in Bild 15 zeigt, wie OpenTelemetry-Metriken in einer ASP.NET-Core-Anwendung aktiviert und um eine einfache fachliche Kennzahl ergänzt werden. Neben den automatisch erfassten HTTP-Metriken wird ein eigener Counter für die Anzahl erstellter Bestellungen definiert. Für den Einstieg genügt ein Console-Exporter, um die erfassten Messwerte sichtbar zu machen; in produktiven Umgebungen werden Metriken üblicherweise an zentrale Backends wie Prometheus oder Azure Monitor exportiert.

In diesem Beispiel wird ein eigener Counter für die Anzahl erstellter Bestellungen definiert und in der Order-Verarbeitung inkrementiert. Die ASP.NET-Core-Instrumentation liefert gleichzeitig automatisch Metriken zu HTTP-Anfragen, Antwortzeiten und Fehlerraten. Sämtliche erfassten Metri-

ken werden über den Console-Exporter ausgegeben, was insbesondere für erste Experimente oder lokale Entwicklung hilfreich ist (Bild 16).

Nachdem Metriken erfasst wurden, übernimmt der Exporter die Aufgabe, sie an ein geeignetes Zielsystem weiterzuleiten. Dabei hält OpenTelemetry die Erfassung bewusst von der Ausgabe getrennt: Messwerte werden zunächst im Speicher gesammelt, über definierte Zeitintervalle aggregiert und anschließend exportiert. Für lokale Entwicklung oder erste Tests eignet sich der Console-Exporter, um Struktur und Inhalt der Metriken schnell sichtbar zu machen (Bild 17).

Für den Einsatz in ASP.NET-Core-Anwendungen haben sich insbesondere der Prometheus-Exporter [21] sowie die Integration mit Azure Monitor [22] etabliert. Beide Ansätze kommen ohne zusätzlichen Agenten aus und ermöglichen den Export der Metriken entweder über einen HTTP-Endpoint im Prometheus-Format oder über das OpenTelemetry Protocol (OTLP) an einen OpenTelemetry Collector oder ein direkt angebundenes Backend. Das Beispiel in Bild 18 erweitert die bestehende OpenTelemetry-Konfiguration um einen Prometheus-Exporter und stellt zusätzlich einen Scrape-Endpoint bereit.

Metriken funktionieren am besten, wenn sie wenige, stabile Signale liefern: Counter für Werte, die nur steigen (zum Beispiel orders.created), Histogramme für Verteilungen (zum Beispiel Latenzen oder Beträge) und Gauges/Up-Down-Counter für Zustände, die steigen und fallen (zum Beispiel Queue-Länge). Entscheidend ist Disziplin bei Labels (Tags): Alles mit hoher „Cardinality“ – etwa order.id, User-IDs, vollständige URLs mit IDs oder Exception-Messages – erzeugt zu

viele Zeitreihen und macht Dashboards teuer und unübersichtlich; solche Details gehören in Traces oder Logs. Sinnvoll sind wenige, kontrollierte Labels wie route, method, status_code, dependency oder grobe Kategorien wie result=ok|error.

Außerdem sollten Einheiten konsistent angegeben werden (OpenTelemetry-Konventionen/UCUM), damit Backends korrekt aggregieren und vergleichen können, und Metriken sollten einen klaren Zweck haben: Was nicht in einem Dash-

```

app.MapPost("/orders", async (Order order) =>
{
    // Business-Metriken erfassen
    ordersCreated.Add(1);

    // Fachliche Verarbeitung
    await ProcessOrder(order);

    return Results.Ok();
});

```

Erfassung einer Business-Metrik (ordersCreated) im Endpoint POST /orders vor der fachlichen Verarbeitung (Bild 16)

```
Metric Name: orders.created, Description: Number of created orders, Metric Type: LongSum
Instrumentation scope (Meter):
  Name: OrderMetrics
  Version: 1.0.0
(2026-02-08T11:19:42.6397216Z, 2026-02-08T11:20:21.6292677Z]
Value: 1
```

Konsolenausgabe der OpenTelemetry-Metrik orders.created mit aktuellem Zählerstand aus dem OrderMetrics-Meter (Bild 17)

board oder Alarm landet, wird in der Praxis selten gepflegt und sollte eher weggelassen werden.

Fazit: Logging, Tracing und Metriken als Teamplayer

Logging, Tracing und Metriken verfolgen jeweils unterschiedliche Ziele, entfalten ihren eigentlichen Nutzen jedoch erst im Zusammenspiel. Logs liefern detaillierte Informationen zu einzelnen Ereignissen, Traces machen Abläufe und Abhängigkeiten über Systemgrenzen hinweg nachvollziehbar, und Metriken zeigen frühzeitig, wie sich das System insgesamt verhält. OpenTelemetry verbindet diese Disziplinen über einen gemeinsamen Standard und ermöglicht es, Telemetriedaten konsistent zu erfassen und unabhängig vom Zielsystem weiterzuverarbeiten.

Entscheidend für moderne, stabile .NET-Anwendungen ist dabei nicht die bloße Aktivierung von Telemetrie, sondern ein bewusster und strukturierter Einsatz: klare Konventionen, fachlich sinnvolle Anreicherung und eine saubere Trennung der Verantwortlichkeiten. So entsteht Laufzeittransparenz, die über klassisches Debugging hinausgeht und hilft, Probleme nicht nur schneller zu erkennen, sondern sie nachhaltig zu verstehen und zu beheben. ■

- [1] Debug .NET Framework source, <https://learn.microsoft.com/en-us/visualstudio/debugger/how-to-debug-dotnet-framework-source>
- [2] Logging in C# and .NET, <https://learn.microsoft.com/en-us/dotnet/core/extensions/logging/overview>
- [3] .NET distributed tracing, <https://learn.microsoft.com/en-us/dotnet/core/diagnostics/distributed-tracing>
- [4] .NET metrics, <https://learn.microsoft.com/en-us/dotnet/core/diagnostics/metrics>

```
using OpenTelemetry.Metrics;

builder.Services.AddOpenTelemetry()
    // ... AddOpenTelemetry Konfiguration wie im vorherigen Beispiel
    .WithMetrics(metrics =>
    {
        // ... Metrics Konfiguration wie im vorherigen Beispiel

        // Export als Prometheus-Scrape-Endpoint (statt Console-Exporter)
        metrics.AddPrometheusExporter();
    });

WebApplication app = builder.Build();

// Prometheus-Endpoint (Standard: /metrics)
app.MapPrometheusScrapingEndpoint();

app.Run();
```

Konfiguration des Prometheus-Exporters in OpenTelemetry mit Bereitstellung der Metriken über den Endpoint /metrics (Bild 18)

- [5] Best practices for exceptions, <https://learn.microsoft.com/en-us/dotnet/standard/exceptions/best-practices-for-exceptions>
- [6] Sustainable Code – Condition vs Exception, <https://github.com/BenjaminAbt/SustainableCode/tree/main/csharp/condition-vs-exception>
- [7] Log scopes, <https://learn.microsoft.com/en-us/dotnet/core/extensions/logging/overview#log-scopes>
- [8] Activity IDs, <https://learn.microsoft.com/en-us/dotnet/core/diagnostics/distributed-tracing-concepts#activity-ids>
- [9] HttpContext.TraceIdentifier Property, <https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.http.httpcontext.traceidentifier?view=aspnetcore-10.0>
- [10] NuGet-Paket Microsoft.Extensions.Logging, <https://nuget.org/packages/microsoft.extensions.logging/>
- [11] .NET dependency injection, <https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection/>
- [12] Structured Data, <https://github.com/serilog/serilog/wiki/Structured-Data>
- [13] Serilog, <https://serilog.net>
- [14] Provided Sinks, <https://github.com/serilog/serilog/wiki/Provided-Sinks>
- [15] .NET observability with OpenTelemetry, <https://learn.microsoft.com/en-us/dotnet/core/diagnostics/observability-with-otel>
- [16] OpenTelemetry, <https://opentelemetry.io>
- [17] Spans, <https://opentelemetry.io/docs/concepts/signals/traces/#spans>
- [18] Setting up an ActivitySource, <https://opentelemetry.io/docs/languages/dotnet/instrumentation/#setting-up-an-activitysource>
- [19] OpenTelemetry Metrics, <https://opentelemetry.io/docs/specs/otel/metrics/>
- [20] Instrument units, <https://opentelemetry.io/docs/specs/semconv/general/metrics/#instrument-units>
- [21] Prometheus, <https://prometheus.io>
- [22] OpenTelemetry on Azure, <https://learn.microsoft.com/en-us/azure/azure-monitor/app/opentelemetry>



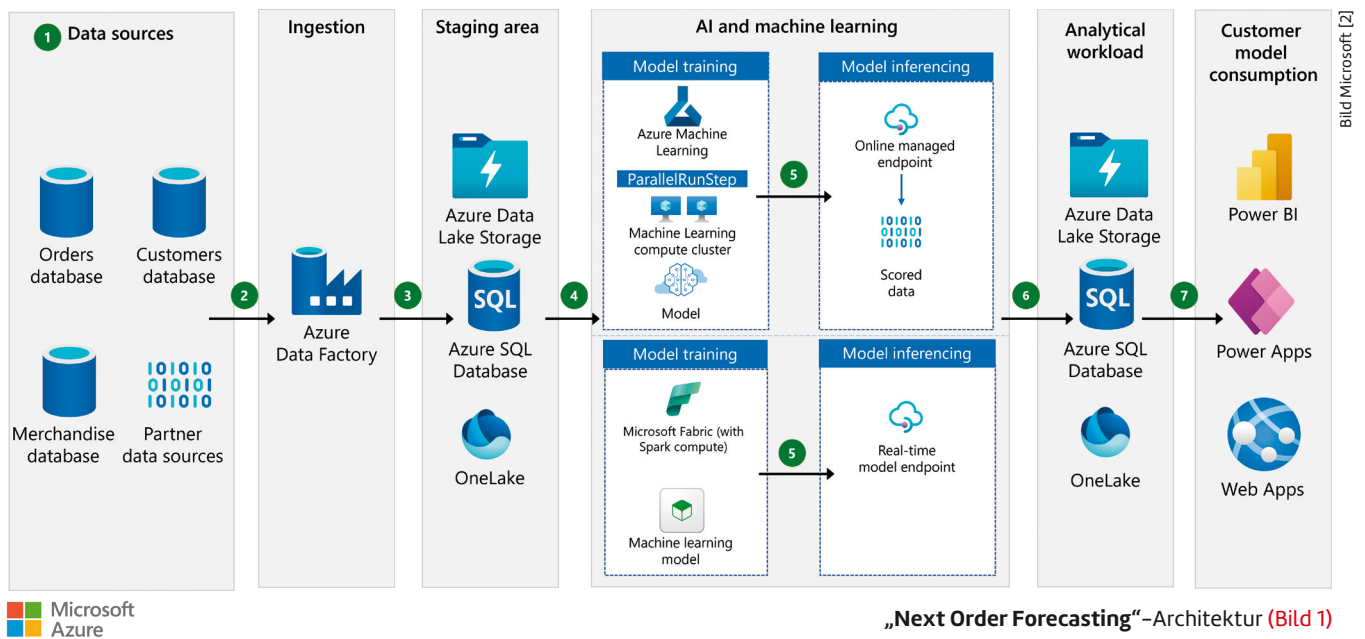
Benjamin Abt

arbeitet als Principal Software Engineer bei der MediaLesson GmbH in Pforzheim, betreibt das C#-Forum myCSharp.de und tritt als Trainer sowie Sprecher auf Konferenzen und User Groups auf. Seit 2015 ist er Microsoft MVP für .NET.
mail@benjamin-abt.com

.NET, PYTHON UND KI – GRUNDLAGEN UND AUSBLICK

Effizienter Brückenschlag

Künstliche Intelligenz hält längst Einzug in produktive .NET-Anwendungen. Doch wie lassen sich Python-Modelle und ML-Frameworks effizient in C#-Projekte integrieren?



Die rasante Entwicklung der Künstlichen Intelligenz (KI) und des Machine Learning (ML) stellt .NET-Entwickler vor neue Herausforderungen und Chancen. Oft entstehen ML-Modelle in der Welt der Data Scientists mit Python als bevorzugter Sprache, konkret beispielsweise mit populären Frameworks wie Scikit-Learn, TensorFlow oder PyTorch. Gleichzeitig werden Unternehmensanwendungen jedoch häufig in .NET (C#) entwickelt und betrieben [1]. Hier entsteht ein Gap zwischen den Werkzeugen und Technologien beider Welten.

Dieses Spannungsfeld macht deutlich, warum KI-Funktionalität in .NET-Anwendungen integriert werden muss, um die Stärken beider Ökosysteme zu vereinen. Herkömmliche Ansätze zur Überbrückung dieser Lücke sind umständlich. Eine Möglichkeit besteht darin, dass der Data Scientist dem .NET-Team lediglich die trainierten Modellparameter übergibt, woraufhin der Entwickler das Modell manuell in C# nachimplementiert. Dies führt jedoch zu doppeltem Aufwand, erhöhter Fehleranfälligkeit und längeren Release-Zyklen.

Motivation und Use Cases

In .NET-basierten Unternehmenslösungen kommen künstliche Intelligenz und maschinelles Lernen heute in vielen Bereichen zum Einsatz: von der Verarbeitung natürlicher Sprache über Bild- und Videoanalyse bis hin zur Entscheidungsunterstützung und Anomalieerkennung. Klassische BI- und

ERP-Systeme erhalten so prädiktive Fähigkeiten. Ein Beispiel ist die Bedarfs- und Absatzprognose: Händler nutzen historisches Verkaufs- und Kundenverhalten, um die zukünftige Bestellmenge eines Kunden vorherzusagen. Die „Next Order Forecasting“-Architektur (Bild 1) bei einem Warenhändler etwa legt dar, wie die prognostizierte Nachbestellung eines Kunden genutzt wird, um Produktempfehlungen zu generieren und optimale Bestellmengen vorzuschlagen [2]. Dadurch lassen sich Über- oder Unterproduktion vermeiden und Umsatzpotenziale besser realisieren.

Ein weiteres typisches Einsatzszenario sind Empfehlungssysteme. Anhand der Kauf- oder Bewertungsdaten kann ein System beispielsweise voraussagen, welche Produkte ein Kunde mag, und gezielt solche Angebote oder Empfehlungen ausspielen. Konkret sagt das Modell etwa vorher, welche Bewertung ein Nutzer einem Artikel geben wird, und empfiehlt alle Artikel, deren prognostizierte Bewertung über einem Schwellenwert liegt. So erhalten Kunden personalisierte Produktempfehlungen anstelle von allgemeinen Auswahllisten.

Auch im Dokumenten- und Datenverarbeitungsbereich gewinnt KI an Bedeutung. Wo früher Formulare und Rechnungen manuell gescannt und erfasst wurden, können moderne KI-Dienste strukturierte Informationen automatisch extrahieren. So analysiert der Azure-Dienst AI Document Intelligence eingescannte Dokumente mit ML, erkennt Text und Daten, un-

ter anderem Felddaten aus Belegen, Rechnungen und dergleichen, und wandelt diese in strukturierte Datensätze um. Ein typischer Anwendungsfall ist die automatische Rechnungserfassung: Ein System liest per OCR das Rechnungsdatum, Beträge und Empfängerdaten aus und befüllt ERP-Felder automatisch. Auch das Klassifizieren und Auswerten eingehender E-Mails oder Support-Tickets lässt sich per KI optimieren. Typische Textklassifikationsszenarien sind Spam-Erkennung oder Sentiment-Analyse (Stimmungserkennung) in Kundenkommentaren. So kann ein CRM-System automatisch erkennen, ob ein Kundenfeedback positiv, neutral oder negativ ist, und Supportanfragen lassen sich nach Dringlichkeit sortieren. Mithilfe des Playgrounds des KI-Dienstes Foundry von Microsoft kann man dies sehr leichtgewichtig erproben (Bild 2).

Im Bereich Computer Vision erweitern neuronale Netze (Deep Learning) .NET-Anwendungen um visuelle Intelligenz: etwa für Objekt- und Bilderkennung. In der Praxis könnte eine Produktionsüberwachungs-App Kamerabilder nach Produktfehlern oder Sensorbilder nach Abweichungen durchforsten. In der Logistik hilft Bilderkennung etwa bei der automatischen Wareneingangskontrolle (Barcode- und Mustererkennung). Selbst Gesichtserkennung oder Analyse der CCTV-Kamerabilder in einer Security-Anwendung sind denkbar. Gleichzeitig eignet sich KI für Anomalieerkennung: Bei sensorgestützten Produktionsdaten oder Transaktionsströmen erkennt ein ML-Modell automatisch ungewöhnliche Muster (Maschinenfehler, Wartungsbedarf, Betrugsversuch) und warnt das System. Sofern die betrieblichen Monitoring-Daten historisch vorliegen, können über eine ML-basierte Zeitreihenanalyse plötzliche Ausreißer („Anomalien“) identifiziert werden.

.NET-Entwicklern stehen für die Realisierung dieser Use Cases umfassende Frameworks zur Verfügung. Für klassische ML-Aufgaben bietet ML.NET eine umfangreiche Pipe-

Du möchtest gerne mehr erfahren?

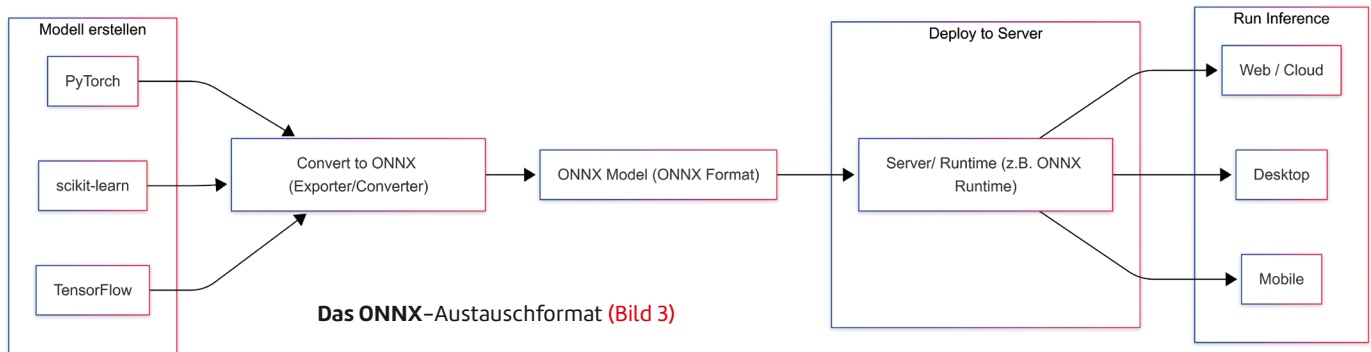
Das freut uns, denn ab Erscheinen des vorliegenden Artikels veröffentlichen wir auf developer-world.de/dwx-insights weitere Beiträge, die den Artikel fortsetzen und ausschließlich online zur Verfügung stehen. Derzeit geplant sind folgende Themen:

- Python in .NET – Integration mit Python.NET
- Maschinelles Lernen in .NET – ML.NET, Scikit-Learn und ONNX
- Deep Learning in .NET – TensorFlow.NET und TorchSharp
- Beispiele und Ausblick – KI-Lösungen mit .NET und Python

line-Bibliothek für Regression, Klassifikation, Clustering und vieles mehr [3]. ML.NET unterstützt unter anderem die Integration vortrainierter TensorFlow- und ONNX-Modelle. Insbesondere kann man in Python oder R trainierte Modelle ins offene ONNX-Format exportieren und nahtlos in ML.NET-Anwendungen einsetzen. Für Deep Learning gibt es C#-Bindings etablierter Bibliotheken: TensorFlow.NET ist ein .NET-Standard-Wrapper um TensorFlow, der es ermöglicht, neuronale Netze direkt in C# zu definieren und zu trainieren. TorchSharp [4] ist das .NET-Pendant zu PyTorch. Damit können Entwickler leistungsfähige KI-Modelle erstellen, zum Beispiel Bildklassifizierung mit Convolutional Neural Networks (CNNs), ohne die .NET-Plattform zu verlassen. ML.NET integriert in der aktuellen Version beispielsweise bereits textbasierte Deep-Learning-Modelle über TorchSharp.

Insgesamt erweitern KI-Technologien .NET-Anwendungen um Lern- und Vorhersagefähigkeiten. Sie heben traditionelle Business-Apps dadurch auf ein neues Level, indem ►

Sprachanalyse (Bestimmung der Stimmung) mithilfe von Microsoft Foundry (Bild 2)



sie manuelle Regelwerke oder statische Algorithmen durch datengesteuerte Modelle ersetzen. Die Motivation ist klar: KI-Modelle erschließen weitere Potenziale in Abläufen und Kundenprozessen, die mit klassischer Programmierung schwer abzubilden wären, und steigern damit Effizienz und Wertschöpfung in .NET-basierten Business-Umgebungen.

Python-Integration mit Python.NET

Eine erste Option, um Python und .NET zusammenzubringen, ist die direkte Integration von Python-Code in eine .NET-Anwendung. Hier kommt Python.NET [5] ins Spiel, eine .NET-Library, die Interoperabilität zwischen dem .NET-Laufzeitsystem und dem Python-Interpreter bietet. Mit Python.NET lassen sich Python-Funktionen und -Bibliotheken direkt aus C# heraus nutzen. Das bedeutet, .NET-Entwickler können beispielsweise eine in Python entwickelte Scikit-Learn-Funktion oder NumPy-Routine unverändert in ihre C#-Anwendung einbinden und ausführen. Python.NET öffnet die Tür zum riesigen Python-Ökosystem direkt innerhalb von .NET-Applikationen – inklusive zahlreicher ML-Bibliotheken, für die es in .NET vielleicht keine direkten Entsprechungen gibt.

Wie funktioniert Python.NET? Technisch gesehen bindet Python.NET den nativen CPython-Interpreter ein. Über das Python.Runtime-API kann C# den Python-Interpreter initialisieren und anschließend Python-Module laden, Objekte erzeugen und Funktionen ausführen. Datentypen werden bei Bedarf automatisch konvertiert, etwa einfache Zahlentypen, Listen oder Arrays. Sogar der Aufruf von C#-Code aus Python heraus ist möglich, was eine voll flexible Integration beider Sprachen erlaubt. Praktisch heißt das: Anstatt das Rad in .NET neu zu erfinden, kann man etablierte Python-Packages (von Datenanalyse bis Machine Learning) wiederverwenden.

Allerdings muss man die Einschränkungen im Blick behalten. Python-Code innerhalb einer .NET-Anwendung bedeutet, dass zur Laufzeit ein Python-Interpreter mitläuft. Häufige Wechsel zwischen C# und Python können die Performance beeinträchtigen. Python selbst ist bekanntlich langsamer als C#, und durch die Interop-Schicht wird es tendenziell noch langsamer. Für performanzkritische Teile einer Anwendung oder sehr viele Aufrufe pro Sekunde ist dieser Ansatz also mit Vorsicht zu genießen. Außerdem setzt Python.NET voraus, dass auf dem Zielsystem eine kompatible Python-Version installiert ist. Dies erhöht den Aufwand beim Deployment der Anwendung.

Trotz dieser Punkte ist Python.NET äußerst nützlich, um schnell erste Prototypen zu erstellen oder spezialisierte ML-Funktionen verfügbar zu machen, die es in dieser Form in .NET (noch) nicht gibt.

Modell-Interoperabilität mit ONNX

Während Python.NET Python-Code zur Laufzeit ausführt, gibt es einen alternativen Ansatz, der besonders für fertige ML-Modelle attraktiv ist: ONNX (Open Neural Network Exchange) [6]. ONNX ist ein offenes Standardformat für KI-Modelle, das von Microsoft und Facebook initiiert wurde, um die Interoperabilität zwischen ML-Frameworks zu ermöglichen. Die Idee dahinter: Ein in einem bestimmten Framework, zum Beispiel PyTorch oder Scikit-Learn, trainiertes Modell wird in das ONNX-Format konvertiert und kann anschließend in einer anderen Umgebung genutzt werden, zum Beispiel in .NET mit ML.NET oder ONNX Runtime (Bild 3). ONNX definiert hierzu ein standardisiertes Format für Modellgraphen, Operatoren und Gewichte.

Das Format wird von vielen gängigen ML-Frameworks unterstützt: Man kann heute aus TensorFlow, PyTorch, Scikit-Learn, Keras und vielen weiteren Frameworks Modelle als ONNX exportieren. Für Scikit-Learn gibt es beispielsweise das Hilfspaket skl2onnx, mit dem sich trainierte Modelle mit wenigen Zeilen konvertieren lassen. Auf .NET-Seite benötigt man dann eine Laufzeit, um das ONNX-Modell auszuführen. Hier kommt die ONNX Runtime ins Spiel [7]. Das ist eine von Microsoft als Open Source bereitgestellte Laufzeitbibliothek, die ONNX-Modelle auf verschiedenster Hardware performant ausführen kann, vergleiche den Kasten **ONNX Runtime**. Es existieren APIs für mehrere Sprachen, darunter C# (über das NuGet-Paket Microsoft.ML.OnnxRuntime) ebenso wie Python, Java, JavaScript und weitere.

In .NET-Anwendungen gibt es zwei Hauptwege, ONNX-Modelle zu nutzen:

- **Direkt mit OnnxRuntime:** Man lädt das ONNX-Modell in ein *InferenceSession*-Objekt und übergibt Eingabedaten für eine Vorhersage. Dieses Low-Level-API eignet sich gut, um ONNX-Modelle in beliebigen .NET-Projekten (zum Beispiel Konsolenanwendung, ASP.NET Web API et cetera) einzubetten.
- **Über ML.NET:** ML.NET bietet einen *OnnxTransformer*, mit dem ONNX-Modelle in eine ML.NET-Pipeline eingebunden werden können. So kann man zum Beispiel ein trainier-

tes ONNX-Modell in einer größeren Pipeline nutzen oder es via Prediction Engine wie ein natives ML.NET-Modell konsumieren. Intern verwendet ML.NET dabei ebenfalls die ONNX Runtime für die Ausführung.

Welchen Weg man wählt, hängt vom Anwendungsfall ab. Für einfache Inferenz ohne weitere Datenvorverarbeitung genügt oft die direkte Nutzung der ONNX Runtime. Soll das ONNX-Modell in einen komplexeren ML-Prozess im .NET-Kontext eingebettet werden, also mit zusätzlichen Vorverarbeitungsschritten, Postprocessing et cetera, ist ML.NET mit OnnxTransformer praktisch.

Zwischenfazit: ONNX erleichtert den plattformübergreifenden Austausch von ML-Modellen erheblich. Für .NET-Teams bedeutet dies, dass sie Modelle, die mit Python-Tools entwickelt wurden, direkt nutzen können, ohne sie neu implementieren zu müssen. Das ONNX-Format in Kombination mit der ONNX Runtime ermöglicht echtzeitfähige Vorhersagen in .NET-Anwendungen, selbst wenn das Training in einem ganz anderen Ökosystem stattfand. Wichtig ist eine saubere Konvertierung und Abstimmung der Datenvorverarbeitung, damit die Ergebnisse stimmen. Unterm Strich reduzieren ONNX und ONNX Runtime den Bruch zwischen Data Science und .NET-Entwicklung und erlauben eine effizientere Zusammenarbeit.

ML.NET – maschinelles Lernen nativ in .NET

Sowohl Python.NET als auch ONNX verfolgen den Ansatz, Python und dessen ML-Bibliotheken in die .NET-Welt hereinzuholen – sei es durch direkte Code-Ausführung oder durch Austausch von Modellen. Ein anderer Weg besteht darin, Machine Learning direkt mit nativen .NET-Technologien zu betreiben. Hierfür hat Microsoft ML.NET erstellt, ein platt-

formübergreifendes Open-Source-ML-Framework für das .NET-Ökosystem. Mit ML.NET können Entwickler ML-Pipelines vollständig in C# oder F# erstellen und ausführen, ohne auf externe Python-Abhängigkeiten angewiesen zu sein.

Was bietet ML.NET? Als End-to-End-Framework deckt ML.NET viele klassische ML-Aufgaben ab: von Regression und Klassifikation über Clustering bis hin zu Bildklassifikation oder Anomalieerkennung. Entwickler können damit zum Beispiel Prognosemodelle, Empfehlungssysteme oder Textklassifizierer bauen. Der Vorteil: Dies geschieht in einer dem Entwickler vertrauten .NET-Umgebung mit den bekannten Sprachen, Tools und Libraries. Dank der tiefen Integration in das .NET-Ökosystem lässt sich ML.NET in jede ASP.NET-, Desktop- oder Mobil-Anwendung einbetten, und auch das Deployment gestaltet sich einfach, da kein separater Python-Interpreter oder -Server benötigt wird. Alles läuft innerhalb der .NET-Laufzeitumgebung.

ML.NET ist entwicklerfreundlich gestaltet, insbesondere für .NET-Entwickler, die neu in ML sind. Es verwendet Konzepte und Syntax, die an .NET angelehnt sind. So erinnert der *MLContext* [8] an den bekannten *DbContext* aus Entity Framework; er dient als Ausgangspunkt für alle ML-Operationen. Daten werden typischerweise in benutzerdefinierten Klassen mit Attributen (*[ColumnName]*, *[LoadColumn]* et cetera) modelliert, sodass stark typisierte Pipelines entstehen, die gut in C#-Code integrierbar sind. Zudem bietet ML.NET Hilfsmittel wie den Model Builder [9] – einen GUI-Wizard in Visual Studio – und ein CLI, mit denen man auch ohne tiefes ML-Wissen erste Modelle erstellen kann. ML.NET bietet sogar AutoML-Fähigkeiten [10], die automatisch optimale Algorithmen und Hyperparameter suchen können.

Weitere Vorteile sind Produktionsreife und Performance. ML.NET wurde ursprünglich über Jahre hinweg intern ►

ONNX Runtime

Die ONNX Runtime arbeitet mit verschiedenen Hardware-Beschleunigungsbibliotheken über ihr erweiterbares EP-Framework (Execution Providers Framework) [13], um die ONNX-Modelle optimal auf der Hardwareplattform auszuführen. Diese Schnittstelle ermöglicht es Anwendungsentwicklern, ihre ONNX-Modelle in verschiedenen Cloud- und Edge-Umgebungen bereitzustellen und die Ausführung durch die Nutzung der Rechenleistungen der Plattform zu optimieren. Die EP-Bibliotheken, die in der Ausführungsumgebung vorinstalliert sind, verarbeiten und führen den ONNX-Subgraphen auf der Hardware aus. Diese Architektur abstrahiert die Details der hardware-spezifischen Bibliotheken, die für die Optimierung der Ausführung tiefer neuronaler Netzwerke über Hardwareplattformen wie CPU, GPU, FPGA oder spezialisierte NPUs unerlässlich sind (Bild 6).

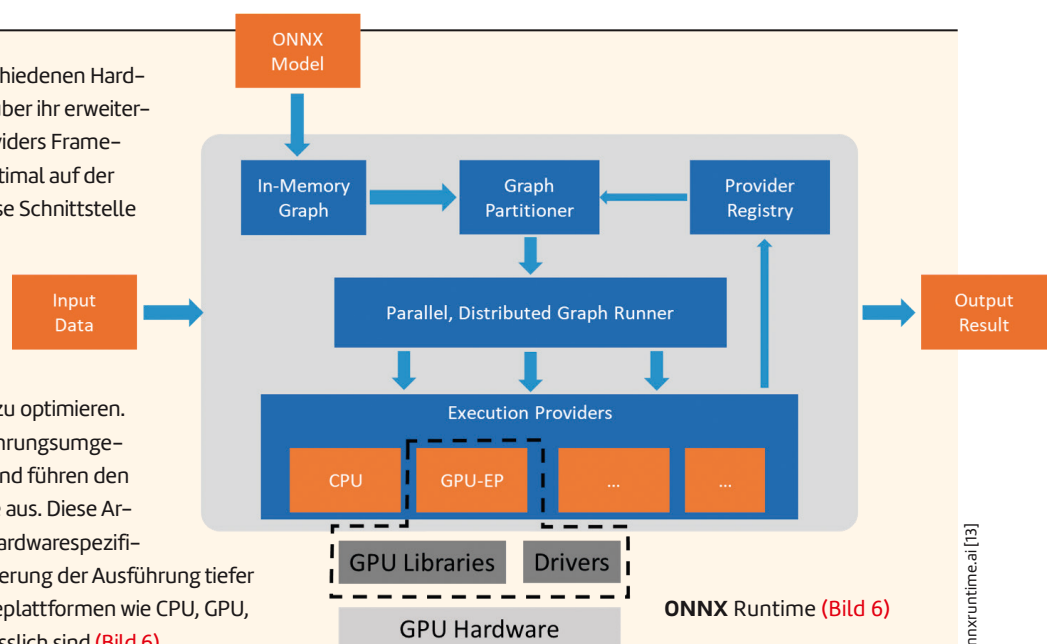


Bild: Onnxruntime.ai [13]

Listing 1: Nutzung von ML.NET in .NET-Anwendungen

```
using Microsoft.ML;
using Microsoft.ML.Data;
using System.Collections.Generic;

// 1. Datenklassen definieren
public class SentimentData
{
    public string Text { get; set; }
    public bool Label { get; set; }
}

public class SentimentPrediction
{
    [ColumnName("PredictedLabel")]
    public bool PredictedLabel { get; set; }
    public float Score { get; set; }
}

// 2. ML.NET Kontext erstellen
MLContext mlContext = new MLContext();

// 3. Trainingsdaten bereitstellen (Beispieldaten)
var trainingData = new List<SentimentData>
{
    new SentimentData { Text =
        "Ich liebe dieses Produkt.", Label = true },
    new SentimentData { Text = "Dieses Produkt ist
        schrecklich.", Label = false }
};

IDataView trainData =
    mlContext.Data.LoadFromEnumerable(trainingData);

// 4. Pipeline definieren: Text-Vektorisierung +
//    binäre Klassifikation
var pipeline = mlContext.Transforms.Text.FeaturizeText(
    "Features", nameof(SentimentData.Text))
    .Append(mlContext.BinaryClassification.Trainers.
        SdcaLogisticRegression(labelColumnName:
            nameof(SentimentData.Label),
            featureColumnName: "Features"));

// 5. Modell trainieren
ITransformer model = pipeline.Fit(trainData);

// 6. Vorhersagen mit dem Modell durchführen
var predEngine = mlContext.Model.CreatePredictionEngine<
    SentimentData, SentimentPrediction>(model);
var sample = new SentimentData { Text =
    "Ich mag dieses Produkt überhaupt nicht." };
var result = predEngine.Predict(sample);
Console.WriteLine($"Eingabe: {sample.Text}\n
    Vorhergesagte Sentiment-Klasse: {(
        result.PredictedLabel ? "Positiv" : "Negativ")}");
```

bei Microsoft entwickelt, zum Beispiel für Dienste wie Windows, Bing und PowerPoint Design Ideas, und später als Open Source freigegeben. Es ist auf Performance optimiert, nutzt .NET-Core-Features und wird stetig verbessert. Durch die Ausführung innerhalb des .NET-Prozesses entfallen die Kommunikations-Overheads, die bei externen ML-Services entstehen würden. Vorhersagen lassen sich mit ML.NET also sehr schnell und mit niedriger Latenz durchführen, was zum Beispiel in Web-APIs ein großer Pluspunkt ist.

Natürlich hat ML.NET auch Grenzen. Es unterstützt klassische ML-Aufgaben hervorragend, aber wenn es um die allerneuesten Entwicklungen in Deep Learning geht, ist das Python-Ökosystem nach wie vor breiter aufgestellt. Dennoch: Für viele typische Anwendungsfälle, beispielsweise Textklassifikation, Forecasting, Recommendations, Anomalieerkennung oder einfache Bildanalysen, bietet ML.NET eine komplette Infrastruktur, ohne dass man die .NET-Welt verlassen muss. Zudem ist ML.NET erweiterbar – es erlaubt das Einbinden von TensorFlow-Modellen und ONNX-Modellen als Teil der Pipeline. So schlägt ML.NET gewissermaßen auch eine Brücke zu Python-Frameworks, falls nötig.

In diesem Abschnitt wollen wir ein Beispiel skizzieren. Es handelt sich um eine Sentiment-Analysis zur Textklassifizierung, also die Identifikation von positiven und negativen Stimmungen. Wir wollen die Trainingsdaten direkt in C# de-

finieren, ein Modell trainieren und eine Prognose durchführen (Listing 1). Schritt für Schritt passiert hier Folgendes:

- **Datenstruktur:** Wir definieren zwei Klassen *SentimentData* (Eingabedaten mit Text und Label) und *SentimentPrediction* (Ausgabedaten mit dem vorhergesagten Label). Die Attributierung *[ColumnName("PredictedLabel")]* stellt sicher, dass ML.NET das Ausgabefeld dem entsprechenden Ergebnis des Modells zuordnet.
- **MLContext:** Wir instanzieren einen *MLContext*, der den ML.NET-Kontext kapselt (Random-Seed, Logging et cetera).
- **Daten laden:** Hier erstellen wir eine kleine Liste von Trainingsdaten direkt im Code und laden sie mit *LoadFromEnumerable* in ein *IDataView*. In realen Szenarien würde man Daten aus Dateien oder Datenbanken laden.
- **Pipeline:** Wir definieren eine ML-Pipeline. Zunächst wandeln wir den Text mit *FeaturizeText* in numerische Merkmale (Features) um – dies nutzt im Hintergrund ein Wörterbuch und berechnet zum Beispiel TF-IDF-Werte (Term Frequency-Inverse Document Frequency (TF-IDF) ist ein statistisches Verfahren, das die Relevanz eines Wortes für ein Dokument innerhalb einer Sammlung misst). Anschließend hängen wir einen Trainer für binäre Klassifikation an (*SdcaLogisticRegression*), der die Feature-Spalte auf das Label (Sentiment positiv/negativ) mappt.

- **Training:** Mit *Fit* durchläuft ML.NET die Pipeline mit den Trainingsdaten und trainiert das Modell. Das Ergebnis ist ein *ITransformer*-Objekt, das das trainierte Modell (samt Vorverarbeitung) enthält.
- **Inferenz:** Über eine *PredictionEngine* (Wrapper für Einzelvorhersagen) verwenden wir das Modell, um für einen neuen Beispieltext eine Prognose zu berechnen. Das Ergebnis (*result.PredictedLabel*) gibt *true* (positiv) oder *false* (negativ) zurück, was wir entsprechend ausgeben.

Dieses Beispiel zeigt die Einfachheit, mit der man in C# ein ML-Modell trainieren und nutzen kann. Alles – vom Einlesen der Daten bis zur Ausgabe des Ergebnisses – geschieht innerhalb der .NET-Welt, und das in nur wenigen Zeilen Code. Natürlich ist dies ein sehr einfaches Beispiel; ML.NET erlaubt auch deutlich komplexere Pipelines.

Externe ML-Modelle in C#-Applikationen einbetten

Ein häufiger Anwendungsfall in der Praxis ist der Einsatz von extern trainierten Modellen in einer .NET-App. Wir haben mit ONNX bereits einen wichtigen Mechanismus kennengelernt, wie dies gelingen kann: Das Data-Science-Team kann in seiner bevorzugten Umgebung (Python, R und andere) ein Modell erstellen und trainieren, dann exportiert es das Modell in ein austauschbares Format – idealerweise ONNX – und übergibt es dem .NET-Team. Dieses bindet das Modell in die Anwendung ein und nutzt es für Vorhersagen. Der Vorteil liegt auf der Hand: Jeder nutzt die Tools, die er am besten beherrscht, und dennoch landet das finale Modell ohne Neucodierung in der Produktionsanwendung.

Neben ONNX gibt es noch weitere Formate und Tools, um externe Modelle einzubetten. Ein Beispiel ist das TensorFlow-SavedModel-Format [11]. ML.NET bietet die Möglichkeit, TensorFlow-Modelle direkt zu laden und auszuführen.

Ebenso existieren Community-getriebene Bibliotheken wie TensorFlow.NET [12] oder SciSharp-Wrapper, mit denen

nuell als Datei ins Projekt zu kopieren, sondern aus einem zentralen Model-Repository zu laden, zum Beispiel über einen Webservice oder eine Datenbank. Das gewährleistet, dass Updates des Modells zentral ausgerollt werden können, ohne den Application-Code zu ändern.

Zusammenfassend betrachtet besteht der Workflow beim Einbetten externer Modelle meist aus den folgenden Schritten (Bild 4):

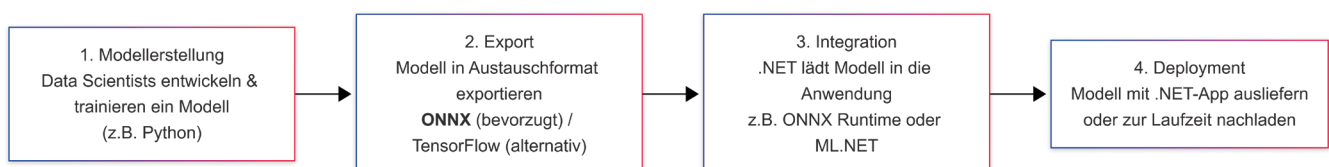
- **Modellerstellung:** Data Scientists entwickeln und trainieren ein Modell in Python oder einer anderen Umgebung.
- **Export:** Das Modell wird in ein kompatibles Austauschformat exportiert, vorzugsweise ONNX, alternativ TensorFlow oder Ähnliche.
- **Integration:** .NET-Entwickler laden das Modell in der Anwendung, etwa mittels ONNX Runtime oder ML.NET.
- **Deployment:** Das Modell wird gemeinsam mit der .NET-Anwendung bereitgestellt oder von dieser zur Laufzeit nachgeladen.

Dieser Ablauf stellt sicher, dass jeder in „seinem“ Tool bleibt und dennoch das Endergebnis – eine KI-gestützte Funktion – im .NET-System läuft.

Kombination von Python-Training und .NET-Deployment

Aus den bisherigen Abschnitten wird deutlich, dass sich Python und .NET keineswegs ausschließen, sondern vielmehr wunderbar ergänzen können. Viele Organisationen fahren erfolgreich zweigleisig: Im Training-Schritt nutzen sie die Flexibilität und den Reichtum der Python-ML-Landschaft, um Modelle zu erstellen. Im Deployment-Schritt – wenn es also darum geht, das Modell in ein fertiges Produkt oder einen Dienst zu integrieren – setzen sie auf die Robustheit, Skalierbarkeit und das Ökosystem von .NET.

Dazu ein klassisches Beispiel: Stellen wir uns ein Team vor, das mit Python und Scikit-Learn einen Machine-Learning-Algorithmus für die Betrugserkennung entwickelt hat. Die-



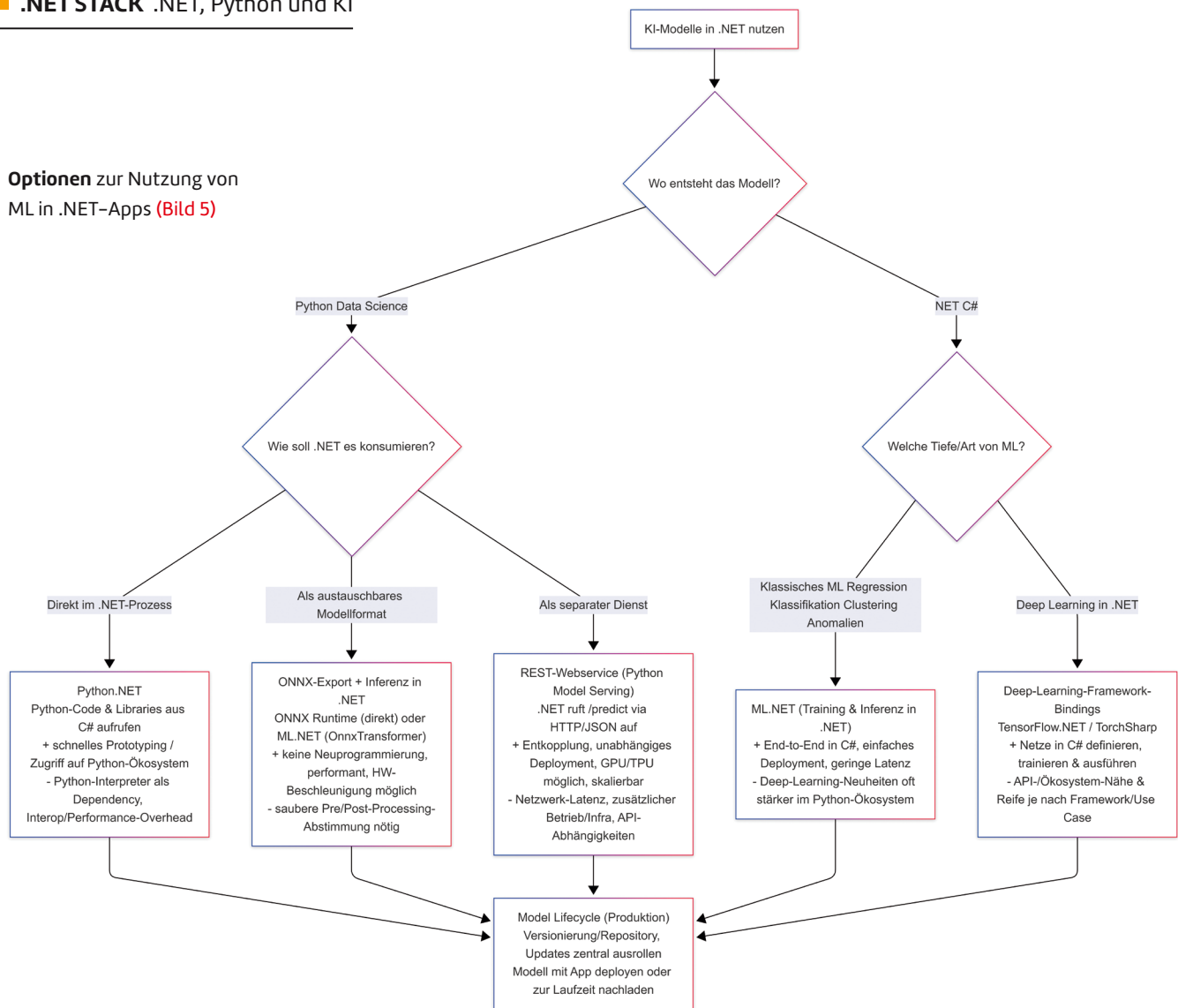
Workflow zur Modelleinbettung (Bild 4)

man zum Beispiel Keras-Modelle in .NET nutzen kann. In der Regel hat sich ONNX jedoch als De-facto-Standard etabliert, da es Framework-unabhängig ist und von vielen großen Unternehmen unterstützt wird. Auch komplexe Deep-Learning-Modelle lassen sich in ONNX darstellen und dann in C# laden, dann oft mithilfe der ONNX Runtime für eine Hardware-Beschleunigung.

Für das Einbetten externer Modelle sollte man außerdem an die Versionierung und Wartbarkeit denken. In produktiven Umgebungen ist es empfehlenswert, Modelle nicht ma-

ses Team kann das finale trainierte Modell mit ONNX exportieren. Das .NET-Team nimmt nun dieses ONNX-Modell und bindet es in eine bestehende C#-basierte Finanzanwendung ein, vielleicht als Teil eines ASP.NET Web API, das Transaktionsdaten entgegennimmt und in Echtzeit eine Bewertung „auffällig/unauffällig“ zurückgibt. Durch die Nutzung der ONNX Runtime im API erzielt man geringe Latenzen, und die .NET-Anwendung kann wie gewohnt skaliert, überwacht und gewartet werden. Sollte das Data-Science-Team das Modell verbessern, wird einfach eine neue ONNX- ►

Optionen zur Nutzung von ML in .NET-Apps (Bild 5)



Version bereitgestellt; die .NET-Anwendung muss dazu nicht umgeschrieben werden.

Ein anderes Kombinationsszenario ist das aktive Prototyping in Python und die nachfolgende Portierung nach .NET. Manchmal entwickeln Data Scientists Prototypen und testen Ideen schnell in Python, weil dort viele Beispiele und Libraries verfügbar sind. Sobald sich ein Ansatz bewährt hat, kann ein .NET-Entwickler entscheiden, ob er das Ergebnis eins zu eins nutzt (via ONNX oder Python.NET) oder ob er eine eigene Implementierung in .NET anstrebt. Letzteres kann Sinn ergeben, wenn langfristig keine Python-Abhängigkeit gewünscht ist oder Performance und Integrationsgründe dies nahelegen. ML.NET könnte dann etwa genutzt werden, um ein ähnliches Modell mit den gleichen Daten in C# neu zu trainieren. Dieser Weg erfordert eine enge Abstimmung und gegebenenfalls Expertise auf beiden Seiten, bietet aber den Vorteil, dass man am Ende ein rein natives .NET-Produkt hat.

Aber auch den umgekehrten Fall gibt es: Mit ML.NET oder TorchSharp in .NET trainierte Modelle können in Python genutzt werden, etwa indem man sie als ONNX exportiert. Das zeigt, dass der Datenaustausch keine Einbahnstraße ist. Die Offenheit von ONNX und Co. ermöglicht es, je nach Bedarf verschiedene Technologien zu kombinieren. Für Deep-Learning-Szenarien existieren inzwischen sogar komplette .NET-

Bindings für die großen Frameworks, was die Kombination noch einfacher macht. TensorFlow.NET implementiert nahezu das vollständige TensorFlow-API in C# (inklusive hohem Level wie Keras). In ähnlicher Weise bietet TorchSharp Zugriff auf PyTorch-Backend-Funktionen aus .NET heraus. Damit können neuronale Netze direkt in .NET geschrieben, trainiert und ausgeführt werden, ohne Python zu benutzen.

ML-Modell als RESTful-Webservice

Ein gängiges Integrationsmuster besteht darin, das Python-ML-Modell als separaten RESTful-Webservice zur Verfügung zu stellen und von der .NET-Anwendung aus per HTTP aufzurufen. Hierzu lädt ein Python-Server das trainierte Modell, gespeichert etwa als PyTorch-/TensorFlow-Datei, und bietet eine HTTP-Schnittstelle an, etwa einen `/predict`-Endpoint. Eingehende JSON-Anfragen werden an das Modell übergeben (`model.predict()` beziehungsweise `model(data)`), und das Ergebnis wird als JSON-Antwort zurückgegeben. Die .NET-Anwendung konsumiert diesen Service typischerweise mit einem HTTP-Client (etwa `HttpClient`), sendet `POST`-Anfragen mit JSON-Payload an den Python-Endpunkt und verarbeitet die empfangene JSON-Antwort. Auf diese Weise kann die .NET-Applikation ML-Funktionen aus Python nutzen, ohne die Modelle selbst in C# nachprogrammieren zu müssen.

Typische Einsatzszenarien für diesen Architekturansatz sind folgende:

- **Leichtgewichtige Erweiterung von .NET-Anwendungen:** Bestehende Enterprise-Anwendungen, zum Beispiel Web-APIs oder Desktop-Software, nutzen Python-Modelle für Funktionen wie Vorhersagen, Empfehlungen oder Bildanalyse. Oft sind solche Modelle bereits in Python erstellt, weshalb eine REST-Kopplung naheliegt. Umfangreiche Änderungen an der bestehenden Applikation sind oft nicht notwendig, das heißt, die Daten werden an das ML-Modell von der .NET-Applikation gesendet und die Antwort verarbeitet. Der Anwendung ist es dabei gleichgültig, ob es sich um ein API zur Bereitstellung von „herkömmlichen“ Daten oder um einen ML-basierten Dienst handelt.
- **Leistungsintensive Modelle:** Komplexe Deep-Learning-Modelle (PyTorch, TensorFlow) mit GPU-Beschleunigung werden über den REST-Service bereitgestellt, da sie in .NET selbst schwer abbildbar sind. Dieser Ansatz ist besonders für rechenintensive Modelle geeignet.
- **Microservice-Architekturen:** In verteilten Systemen arbeiten Data-Science- und Entwicklerteams häufig getrennt. Ein Python-Microservice für ML kann unabhängig deployt werden, während das .NET-Frontend oder -Backend die Ergebnisse abrufen.

Die Vorteile dieses Vorgehens sind:

- **Skalierbarkeit:** Durch eigenständige Dienste kann der Python-Service unabhängig von der .NET-Anwendung skaliert (zum Beispiel vervielfältigt oder in Containern deployt) werden.
- **Entkopplung und Wartbarkeit:** Das Modell lässt sich getrennt versionieren und aktualisieren (zum Beispiel über CI/CD-Pipelines). Klare Schnittstellengrenzen fördern Testbarkeit und Modularität.
- **Technologie-Vielfalt:** Data Scientists können die reichhaltigen Python-Bibliotheken (NumPy, Pandas, Scikit-Learn, PyTorch et cetera) nutzen, während die .NET-Logik unverändert bleibt.
- **GPU- und spezialisierte Hardware:** Leistungsintensive ML-Berechnungen, etwa neuronale Netze, laufen im Python-Service auf geeigneter Hardware (GPU, TPU). FastAPI beziehungsweise Flask können hierfür gezielt deployt werden.
- **Unabhängige Entwicklungszyklen:** Jeder Dienst kann unabhängig entwickelt und releast werden – dies entspricht dem Microservices-Paradigma, bei dem Dienste über HTTP kommunizieren.

Wo Licht ist, ist allerdings auch Schatten. Nachteile des beschriebenen Vorgehens sind:

- **Netzwerk-Overhead und Latenz:** Jeder Aufruf erfordert HTTP-Kommunikation und JSON-Serialisierung. Dies führt zu zusätzlicher Latenz im Vergleich zu einem rein lokalen Aufruf.
- **Komplexität:** Ein zusätzlicher Service muss betrieben und gewartet werden. Das bedeutet mehr Infrastruktur (Container, Service Discovery, Load Balancer) und komplexeres Fehler-Handling über das Netzwerk.

- **Kosten:** Mehr Services bedeuten höhere Betriebskosten, zum Beispiel durch zusätzliche Server- oder Container-Ressourcen, Monitoring und Sicherheit.
- **Abhängigkeiten:** Die .NET-Anwendung ist auf Erreichbarkeit und API-Stabilität des Python-Services angewiesen. Fehler oder API-Änderungen können den Betrieb stören.

Fazit und Ausblick

Im Rahmen eines Überblicks haben wir gezeigt, wie KI-Modelle und Python-Code in .NET-Anwendungen integriert werden können. Wir haben die Motivation beleuchtet: Die unterschiedlichen Technologiewelten von Data Science und Anwendungsentwicklung verlangen nach Brückenschlägen, damit ML-Modelle ihren Weg in produktive .NET-Systeme finden. Genau diese Brücken haben wir vorgestellt, von der direkten Integration mittels Python.NET über den modellbasierten Austausch via ONNX bis hin zur nativen Umsetzung mit ML.NET. Alle hier vorgestellten Varianten der Nutzung von ML-Modellen aus einer .NET-App sind in **Bild 5** zusammengefasst. ■

- [1] *Deploy Sci-kit Learn models in .NET Core Applications*, <https://medium.com/data-science/deploy-sci-kit-learn-models-in-net-core-applications-90e24e572f64>
- [2] *Order Forecasting mit KI*, <https://learn.microsoft.com/de-de/azure/architecture/ai-ml/idea/next-order-forecasting>
- [3] *ML.NET*, <https://dotnet.microsoft.com/en-us/apps/ai/ml-dotnet>
- [4] *TorchSharp auf GitHub*, <https://github.com/dotnet/TorchSharp>
- [5] *Python.NET*, <https://pythonnet.github.io/>
- [6] *Open Neural Network Exchange*, <https://onnx.ai>
- [7] *Get started with ORT for C#*, <https://onnxruntime.ai/docs/get-started/with-csharp.html>
- [8] *MLContext-Klasse*, <https://learn.microsoft.com/de-de/dotnet/api/microsoft.ml.mlcontext?view=ml-dotnet>
- [9] *ML.NET Model Builder*, <https://dotnet.microsoft.com/en-us/apps/ai/ml-dotnet/model-builder>
- [10] *Microsoft Learn, Make predictions using an AutoML ONNX model in .NET*, <https://learn.microsoft.com/en-us/azure/machine-learning/how-to-use-automl-onnx-model-dotnet?view=azureml-api-2>
- [11] *TensorFlow, Using the SavedModel format*, www.tensorflow.org/guide/saved_model
- [12] *TensorFlow.NET*, <https://scisharp.github.io/tensorflow-net-docs/>
- [13] *ONNX Runtime Execution Providers*, <https://onnxruntime.ai/docs/execution-providers/>



Dr. Veikko Krypczyk

ist seit Jahren begeisterter Softwareentwickler, Trainer und Fachautor. Er bietet unter anderem Seminare und Workshops zu den Themen WinUI 3, .NET MAUI und Blazor an. Weitere Informationen finden Sie unter <https://larinet.com>.



DATA ACCESS

Die Datenzugriffsberatung mit DR. HOLGER SCHWICHTENBERG

JSON PATCH FÜR SYSTEM.TEXT.JSON

Objekte flicken

Microsofts JSON-Bibliothek kann nun auch JSON-Patch-Dokumente anwenden.

System.Text.Json beherrscht seit Version 10.0 den JSON-Patch-Standard nach RFC 6902 [1]. JSON Patch ist ein standardisiertes Format, um Änderungen an JSON-Dokumenten in Form einer Sequenz von Operationen auszudrücken. Jede Operation spezifiziert dabei den Typ der Änderung (*add*, *remove*, *replace*, *move*, *copy* oder *test*, siehe [Tabelle 1](#)) sowie den Pfad innerhalb des JSON-Dokuments, auf den sich die Modifikation bezieht.

Durch diese strukturierte Form der Beschreibung lassen sich inkrementelle Änderungen effizient übertragen, ohne das gesamte Dokument erneut austauschen zu müssen. Insbesondere in RESTful Web-APIs ist JSON Patch ein etabliertes Mittel, um partielle Updates über HTTP-PATCH zu implementieren.

Ein wesentliches Merkmal von JSON Patch ist die strikte Definition der Pfadangaben über JSON Pointer (RFC 6901), siehe das Beispiel in [Listing 1](#). Dadurch werden Änderungen eindeutig und positionsgenau adressiert. JSON Patch eignet sich auch für komplexe Datenmodelle mit tiefen Verschachtelungen oder Arrays.

Der Vorteil von JSON Patch zeigt sich insbesondere bei großen Objekten und eingeschränkter Bandbreite, da nur inkrementelle Änderungen übertragen werden. Bei sehr kleinen Datenstrukturen kann der Mehraufwand für das Erzeugen und Verarbeiten eines Patch-Dokuments größer sein als der Gewinn gegenüber einem vollständigen Update.

Bis zur Veröffentlichung von System.Text.Json 10.0 mussten .NET-Entwicklerinnen und -Entwickler für JSON Patch ►

● Listing 1: Beispiel für eine Objekthierarchie

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string PrivateWebsite { get; set; }
    public string CompanyWebsite { get; set; }
    public string PrivateEmail { get; set; }
    public string OfficeEmail { get; set; }
    public List<PhoneNumber> PhoneNumbers { get; set; }
    = new();
    public Address Address { get; set; }
}

public class Address
{
    public string Street { get; set; }
    public string City { get; set; }

    public string State { get; set; }
    public string ZipCode { get; set; }
}

public enum PhoneNumberType
{
    Mobile,
    Home,
    Work,
    Other
}

public class PhoneNumber
{
    public string? Number { get; set; }
    public PhoneNumberType Type { get; set; }
}
```



● **Tabelle 1: Erlaubte JSON-Patch-Operationen nach RFC 6902**

Operation	Beschreibung	Felder	Beispiel
<i>add</i>	Fügt einen Wert an einem bestimmten Pfad hinzu.	<i>path, value</i>	<code>{ "op": "add", "path": "/a/b/c", "value": "foo" }</code>
<i>remove</i>	Entfernt den Wert an einem bestimmten Pfad.	<i>path</i>	<code>{ "op": "remove", "path": "/a/b/c" }</code>
<i>replace</i>	Ersetzt den Wert an einem Pfad durch einen neuen Wert.	<i>path, value</i>	<code>{ "op": "replace", "path": "/a/b/c", "value": 42 }</code>
<i>move</i>	Verschiebt einen Wert von einem Pfad zu einem anderen.	<i>from, path</i>	<code>{ "op": "move", "from": "/a/b/c", "path": "/a/b/d" }</code>
<i>copy</i>	Kopiert einen Wert von einem Pfad zu einem anderen.	<i>from, path</i>	<code>{ "op": "copy", "from": "/a/b/c", "path": "/a/b/e" }</code>
<i>test</i>	Prüft, ob ein Wert an einem Pfad einem bestimmten Wert entspricht.	<i>path, value</i>	<code>{ "op": "test", "path": "/a/b/c", "value": "foo" }</code>

● **Listing 2: Verändern einer Objekthierarchie mit JSON Patch**

```

public void JSONPatchDemo()
{
    CUI.Demo();

    // Originalobjekt
    var person = new Person
    {
        FirstName = "Holger",
        LastName = "Schwichtenberg",
        PrivateEmail = "buero@IT-Visions.de",
        PrivateWebsite = "www.dotnet-doktor.de",
        CompanyWebsite = "www.IT-Visions.de",
        PhoneNumbers = [new PhoneNumber() { Number =
            "0201 649590-0", Type = PhoneNumberType.Work }],
        Address = new Address
        {
            Street = "Fahrenberg 40b",
            City = "Essen",
            State = "NRW"
        }
    };

    CUI.H2("Ausgabe des Objekts in seinem alten Zustand");
    CUI.Print(ITVisions.ObjectDumper.Dump(person));

    #region ----- Beispiel 1: Anwenden eines
    JSON-Patch-Dokuments auf ein Objekt
    // JSON-Patch-Dokument: Erlaubte Operationen nach
    // RFC 6902 sind: "add", "remove", "replace",
    // "move", "copy" und "test"
    string jsonPatch = "[
        { \"op\": \"test\", \"path\": \"/FirstName\",
          \"value\": \"Holger\" },
        { \"op\": \"test\", \"path\": \"/LastName\",
          \"value\": \"Schwichtenberg\" },
        { \"op\": \"replace\", \"path\": \"/FirstName\",
          \"value\": \"Dr. Holger\" },
        { \"op\": \"move\", \"from\": \"/PrivateEmail\", \"path\":
          \"path\": \"/OfficeEmail\" },
        { \"op\": \"remove\", \"path\": \"/PrivateWebsite\" },
        { \"op\": \"add\", \"path\": \"/Address/ZipCode\",
          \"value\": \"45257\" },
        { \"op\": \"add\", \"path\": \"/PhoneNumbers/-\",
          \"value\": { \"Number\": \"0201 649590-40\" } }
    ]";

    // JSON-Patch-Dokument laden
    JsonPatchDocument<Person> patchDoc = JsonSerializer.
        Deserialize<JsonPatchDocument<Person>>(jsonPatch);

    // JSON-Patch-Dokument anwenden auf das Person-
    // Objekt mit ApplyTo()
    patchDoc!.ApplyTo(person, patchError =>
        CUI.PrintError(patchError.ErrorMessage));

    CUI.H2("Ausgabe des Objekts in seinem neuen Zustand");
    CUI.Print(ITVisions.ObjectDumper.Dump(person));
    #endregion
}

```


die ältere Community-Bibliothek Newtonsoft.Json einsetzen. Nun geht es auch mit der neueren Microsoft-Bibliothek System.Text.Json (Hintergründe zum Umstieg von Newtonsoft.Json auf System.Text.Json, den Microsoft seit 2019 vollzieht, gibt es unter [2]).

Für JSON Patch mit System.Text.Json gibt es eine Erweiterung für System.Text.Json in Gestalt des neuen NuGet-Pakets Microsoft.AspNetCore.JsonPatch.SystemTextJson, das erstmalig zusammen mit der Preview 4 von .NET 10.0 erschienen ist [3].

JSONPatchDemo

Ausgabe des Objekts in seinem alten Zustand

```
{FCL10_JSON+Person}
  FirstName: "Holger"
  LastName: "Schwichtenberg"
  PrivateWebsite: "www.dotnet-doktor.de"
  CompanyWebsite: "www.IT-Visions.de"
  PrivateEmail: "buero@IT-Visions.de"
  OfficeEmail: null
  PhoneNumbers: ...
    {FCL10_JSON+PhoneNumber}
      Number: "0201 649590-0"
      Type:
        {FCL10_JSON+PhoneNumberType}
          value__: 2
          Mobile: Mobile
          Home: Home
          Work: Work
          Other: Other

  Address:
    {FCL10_JSON+Address}
      Street: "Fahrenberg 40b"
      City: "Essen"
      State: "NRW"
      ZipCode: null
```

Ausgabe des Objekts in seinem neuen Zustand

```
{FCL10_JSON+Person}
  FirstName: "Dr. Holger"
  LastName: "Schwichtenberg"
  PrivateWebsite: null
  CompanyWebsite: "www.IT-Visions.de"
  PrivateEmail: null
  OfficeEmail: "buero@IT-Visions.de"
  PhoneNumbers: ...
    {FCL10_JSON+PhoneNumber}
      Number: "0201 649590-0"
      Type:
        {FCL10_JSON+PhoneNumberType}
          value__: 2
          Mobile: Mobile
          Home: Home
          Work: Work
          Other: Other

    {FCL10_JSON+PhoneNumber}
      Number: "0201 649590-40"
      Type:
        {FCL10_JSON+PhoneNumberType}
          value__: 0
          Mobile: Mobile
          Home: Home
          Work: Work
          Other: Other

  Address:
    {FCL10_JSON+Address}
      Street: "Fahrenberg 40b"
      City: "Essen"
      State: "NRW"
      ZipCode: "45257"
```

Ausgabe von Listing 2 (Bild 1)

Ausgabe des Objekts in seinem neuen Zustand als JSON

```
{
  "FirstName": "Dr. Holger",
  "LastName": "Schwichtenberg",
  "PrivateWebsite": null,
  "CompanyWebsite": "www.IT-Visions.de",
  "PrivateEmail": null,
  "OfficeEmail": "buero@IT-Visions.de",
  "PhoneNumbers": [
    {
      "Number": "0201 649590-0",
      "Type": 2
    },
    {
      "Number": "0201 649590-40",
      "Type": 0
    }
  ],
  "Address": {
    "Street": "Fahrenberg 40b",
    "City": "Essen",
    "State": "NRW",
    "ZipCode": "45257"
  }
}
```

Ausgabe des geänderten Objekts im JSON-Format (Bild 2)

Listing 3: Testen mit JSON-Patch-Dokument

```
CUI.H2("Testen mit JSON-Patch-Dokument");
string jsonPatchTest = ""
[
  { "op": "test", "path": "/FirstName",
    "value": "Dr. Holger" },
  { "op": "test", "path": "/PrivateEmail",
    "value": null },
  { "op": "test", "path": "/PrivateWebsite",
    "value": null },
  { "op": "test", "path": "/Address/ZipCode",
    "value": "45257" }
];

// JSON-Patch-Dokument laden
JsonPatchDocument<Person>? patchTestDoc =
  JsonSerializer.Deserialize<
    JsonPatchDocument<Person>>(jsonPatchTest);

// JSON-Patch-Dokument anwenden auf das Person-
// Objekt
patchTestDoc!.ApplyTo(person, patchError =>
  CUI.PrintError(patchError.ErrorMessage));
```

```
Reines Testen eines Objekts mit JSON-Patch-Dokument (absichtlich mit drei Fehlern!)
The current value 'Dr. Holger' at path 'FirstName' is not equal to the test value 'Dr.Holger'.
The current value '' at path 'PrivateWebsite' is not equal to the test value ''.
The current value '45257' at path 'ZipCode' is not equal to the test value '45251'.
```

Fehlertexte als Ausgabe von Listing 4 (Bild 3)

Microsoft verspricht den Entwicklerinnen und Entwicklern in der neuen Implementierung „eine verbesserte Leistung und weniger Speichernutzung im Vergleich zu der existierenden Implementierung in Newtonsoft.Json“ (Zitat aus den Release Notes [4]) in Verbindung mit einem ähnlichen Design wie bei Newtonsoft.Json, einschließlich der Aktualisierung eingebetteter Objekte und Arrays.

Bisher nicht verfügbar ist JSON Patch für dynamische Objekte, weil Newtonsoft.Json dafür Reflection einsetzt. System.Text.Json soll aber auch mit dem NativeAOT-Compiler funktionieren. Zu beachten ist: Das neue JSON-Patch-Paket funktioniert bisher leider nicht in Verbindung mit dem NativeAOT-Compiler des modernen .NET, sondern nur mit dem Just-in-Time-Compiler.

● Listing 4: Testen mit JSON-Patch-Dokument und eingebauten Fehlern

```
CUI.H2("Reines Testen eines Objekts mit JSON-Patch-
Dokument (absichtlich mit drei Fehlern!);
string jsonPatchTest = ""
[
    { "op": "test", "path": "/FirstName",
      "value": "Dr.Holger" },
    { "op": "test", "path": "/PrivateEmail",
      "value": null},
    { "op": "test", "path": "/PrivateWebsite",
      "value": ""},
    { "op": "test", "path": "/Address/ZipCode",
      "value": "45251" }
];
```

● Listing 5: JSON Patch auf Mengen (Arrays) anwenden

```
List<int> Zahlen = [1, 2, 3, 4, 5];

string jsonPatchZahlen = ""
[
    { "op": "add", "path": "/2", "value": 42 },
    { "op": "remove", "path": "/0" },
    { "op": "replace", "path": "/2", "value": 99 },
    { "op": "test", "path": "/0", "value": 2 },
    { "op": "test", "path": "/1", "value": 42 },
    { "op": "test", "path": "/2", "value": 99 },
    { "op": "test", "path": "/3", "value": 4 },
    { "op": "test", "path": "/4", "value": 5 }
]
"";

JsonPatchDocument<List<int>>? patchZahlen =
    JsonSerializer.Deserialize<JsonPatchDocument<
List<int>>>(jsonPatchZahlen);
patchZahlen!.ApplyTo(Zahlen, patchError =>
    CUI.PrintError(patchError.ErrorMessage));

foreach (var z in Zahlen)
{
    Console.WriteLine(z);
}
```

JSON Patch außerhalb von ASP.NET Core

Trotz des Bestandteils „AspNetCore“ im Namen funktioniert das Zusatzpaket Microsoft.AspNetCore.JsonPatch.SystemTextJson auch außerhalb von ASP.NET Core, was Listing 2 beweisen soll.

Im NuGet-Paket Microsoft.AspNetCore.JsonPatch.SystemTextJson gibt es eine Klasse *Microsoft.AspNetCore.JsonPatch.SystemTextJson.JsonPatchDocument<T>* mit einer Methode *ApplyTo(obj)*, die eine JSON-Patch-Operation auf das übergebene Objekt anwendet, siehe Listing 2.

JSON Patch auf Mengen (Arrays) anwenden

```
2
42
99
4
5
```

Ausgabe des Beispiels in Listing 5 (Bild 4)

Vor dem Anwenden von *ApplyTo()* lädt man das JSON-Patch-Dokument via *JsonSerializer.Deserialize<JsonPatchDocument<Person>>()* in eine Instanz von *JsonPatchDocument<T>*.

ApplyTo() erwartet im zweiten Parameter vom Typ *Action<JsonPatchError>* eine Methode, die als Parameter ►

The screenshot shows a REST client in Visual Studio. The request is a PATCH to `https://localhost:7022/Person_MitTypedResult/123` with a `Content-Type: application/json-patch+json`. The request body is a JSON patch document with the following operations:

```

[
  { "op": "test", "path": "/FirstName", "value": "Holger" },
  { "op": "test", "path": "/LastName", "value": "Schwichtenberg" },
  { "op": "replace", "path": "/FirstName", "value": "Dr. Holger" },
  { "op": "move", "from": "/PrivateEmail", "path": "/OfficeEmail" },
  { "op": "remove", "path": "/PrivateWebsite" },
  { "op": "add", "path": "/Address/ZipCode", "value": "45257" },
  { "op": "add", "path": "/PhoneNumbers/-", "value": { "Number": "0201 649590-40" } }
]

```

The response is a 200 OK status with a time of 3.7 ms and a size of 344 bytes. The response body is a JSON object:

```

{
  "firstName": "Dr. Holger",
  "lastName": "Schwichtenberg",
  "privateWebsite": null,
  "companyWebsite": "www.IT-Visions.de",
  "privateEmail": null,
  "officeEmail": "buero@IT-Visions.de",
  "phoneNumbers": [
    {
      "number": "0201 649590-0",
      "type": 2
    },
    {
      "number": "0201 649590-40",
      "type": 0
    }
  ],
  "address": {
    "street": "Fahrenberg 40b",
    "city": "Essen",
    "state": "NRW",
    "zipCode": "45257"
  }
}

```

Objekte vom Typ *JsonPatchError* entgegennimmt. Als *JsonPatchError* signalisiert *ApplyTo()* fehlerhafte Operationsnamen, fehlerhafte Pfade und fehlgeschlagene Test-Operationen.

Wenn ein Fehler bei einer oder mehreren Operationen auftritt, werden die übrigen fehlerfreien Operationen dennoch verarbeitet. Strukturelle Fehler im JSON-Patch-Dokument führen schon beim Deserialisieren zum Fehler.

● Listing 6: JSON Patch mit Microsoft.AspNetCore.JsonPatch.SystemTextJson (Teil 1)

```

using System;
using System.Formats.Asn1;
using System.Net.ServerSentEvents;
using System.Runtime.CompilerServices;
using ITVisions;
using Microsoft.AspNetCore.Http.HttpResults;
using Microsoft.AspNetCore.JsonPatch.SystemTextJson;
using Microsoft.AspNetCore.Mvc;

namespace NET10_WebAPIController.Controllers;

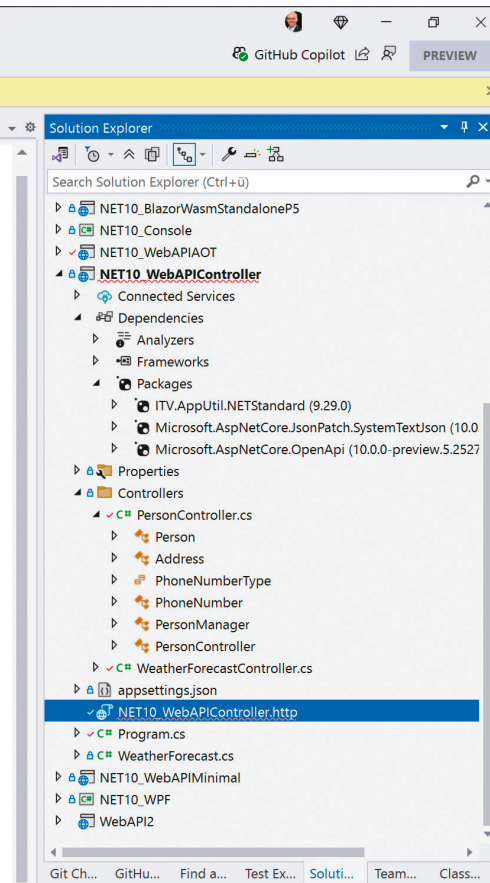
/// <summary>
/// Personendaten holen und aktualisieren mit einem
/// JSON-Patch-Dokument
/// </summary>
[ApiController]
[Route("[controller]")]
public class PersonController : ControllerBase
{
    private readonly ILogger<WeatherForecastController>
        _logger;

    public PersonController(
        ILogger<WeatherForecastController> logger)
    {
        _logger = logger;
    }

    /// <summary>
    /// Diese Web-API-Operation liefert die Daten zu
    /// einer Person anhand der ID
    /// </summary>
    /// <remarks>Es wird aktuell immer die gleiche
    /// Person geliefert ;-(</remarks>
    /// <param name="id">ID der Person</param>
    /// <returns>Person-Objekt mit Adresse</returns>

```

Aufruf eines ASP.NET-Core-basierten Web-API, das ein JSON-Patch-Dokument erwartet und ein geändertes Objekt liefert, in einer HTTP-Datei innerhalb von Visual Studio (**Bild 5**)



Die in **Listing 2** verwendeten Objekte sieht man in **Listing 1**. Das resultierende JSON-Dokument für die Person nach Anwendung des JSON-Patch-Dokuments sieht dann aus wie in **Bild 1** gezeigt.

```
[HttpGet]
[Route("/Person_ohneTypedResult/{id}")]
public async Task<Person> Get1(int id = 0)
{
    var person = await new
        PersonManager().GetPersonByIdAsync(id);
    return person;
}

/// <summary>
/// Diese Web-API-Operation liefert die Daten zu
/// einer Person anhand der ID
/// </summary>
/// <remarks>Es wird aktuell immer die gleiche
/// Person geliefert ;-(</remarks>
/// <param name="id">ID der Person</param>
/// <returns>Person-Objekt mit Adresse
/// </returns>
```

Alternativ kann man das Objekt natürlich auch als JSON ausgeben (**Bild 2**). Das ist nur in dem obigen Listing nicht erfolgt, weil nicht der Eindruck entstehen sollte, man müsse das Objekt als JSON weiterverwenden.

```
// Ausgabe des Resultats
var serializerOptions = new JsonSerializerOptions()
{
    IndentSize = 1, WriteIndented=true,
    IncludeFields = true,
};

Console.WriteLine(JsonSerializer.Serialize(
    person, serializerOptions));
```

Man kann JSON Patch rein zum Testen von Werten in einem Objekt verwenden, ohne etwas zu ändern, siehe **Listing 3**. In diesem Fall gibt es keine Fehlerausgabe. Wenn man das JSON-Patch-Dokument aber wie in **Listing 4** verändert, erhält man die in **Bild 3** gezeigten drei Fehlertexte.

Dabei ist die Fehlermeldung bei „PrivateWebsite“ unzureichend, denn sie unterscheidet nicht zwischen Leerstring und null. Im Objekt steht null, getestet wird auf Leerstring. In der Fehlermeldung steht in beiden Fällen ' '.

JSON Patch kann auch auf Mengen und auch auf primitive Typen angewendet werden. Dies zeigt das Beispiel in **Listing 5** an einem Zahlenarray. Die Ausgabe ist in **Bild 4** zu sehen.

JSON Patch in ASP.NET-Core-Web-APIs

Das Beispiel in **Listing 6** und **Listing 7** zeigt, wie Entwicklerinnen und Entwickler das neue NuGet-Paket Microsoft.AspNetCore.JsonPatch.SystemTextJson in einem ASP.NET-Core-basierten Web-API anwenden. ▶

```
[HttpGet]
[Route("/Person_MitTypedResult/{id}")]
[ProducesResponseType<IEnumerable<WeatherForecast>>(
    StatusCodes.Status200OK, Description = "Person-
    Objekt mit Adresse (aus [ProducesResponseType])")]
[ProducesResponseType<ValidationProblem>(StatusCodes.
    Status400BadRequest, Description = "Fehlerhafte
    Anfrage / ID muss größer als 0 sein!")]
[ProducesResponseType<ValidationProblem>(StatusCodes.
    Status404NotFound, Description =
    "Person nicht gefunden")]

public async Task<Results<Ok<Person>,
    ValidationProblem, NotFound<ProblemDetails>>>
    Get_MitTypedResult(int id = 0)
{
    if (id <= 0)
    {
```


Listing 6: JSON Patch mit Microsoft.AspNetCore.JsonPatch.SystemTextJson (Teil 2)

```

return TypedResults.ValidationProblem(
    new Dictionary<string, string[]> { { "id",
        new[] { "ID muss größer als 0 sein!" } } });
}

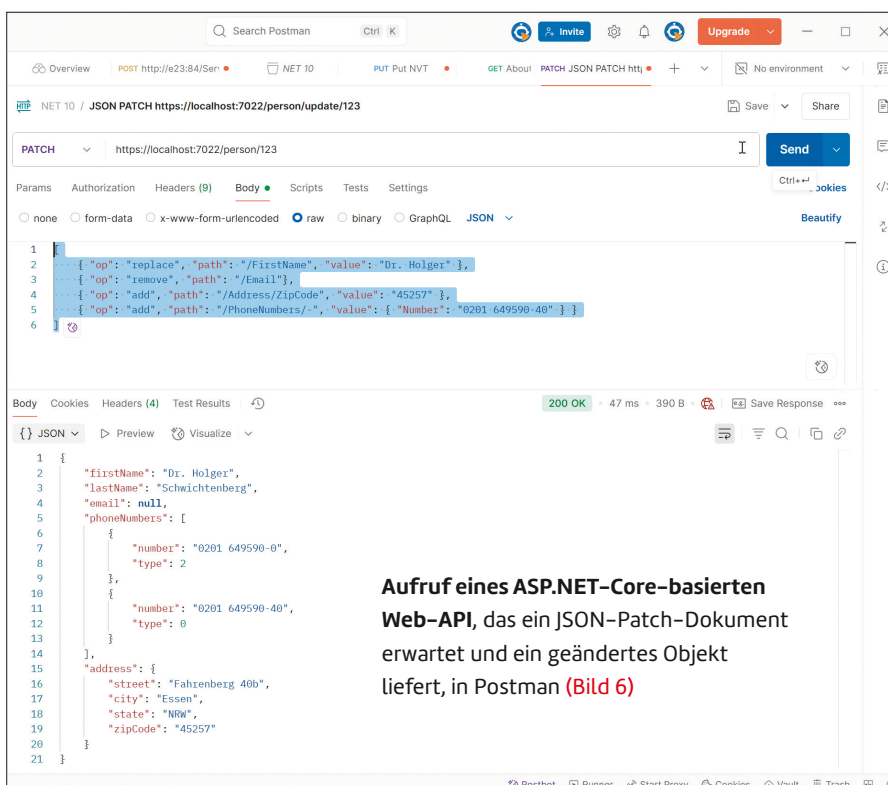
var person = await new PersonManager().
    GetPersonByIdAsync(id);
if (person == null)
{
    return TypedResults.NotFound(new ProblemDetails
    { Title = "Person not found", Detail =
        $"Person mit ID {id} nicht gefunden!" });
}
return TypedResults.Ok(person);
}

// PATCH https://localhost:7022/person/123
[HttpPatch]
[Route("{id}")]
public async Task<Results<Ok<Person>,
    ValidationProblem, NotFound<ProblemDetails>>>
    UpdatePerson(int id, JsonPatchDocument<Person>
        patchDoc = null)
{
    var person = await new PersonManager().
        GetPersonByIdAsync(id);
    if (person == null)
    {
        return TypedResults.NotFound(new ProblemDetails
            { Title = "Person not found", Detail =
                $"No person found with ID {id}" });
    }

    if (patchDoc != null)
    {
        Dictionary<string, string[]> errors =
            new Dictionary<string, string[]>();
        // JSON-Patch-Dokument anwenden auf das Person-
        // Objekt
        int errorCount = 0;
        patchDoc!.ApplyTo(person, patchError =>
        {
            errors.Add(patchError.AffectedObject.GetType().
                Name + "#" + ++errorCount, new[] {
                    patchError.ErrorMessage });
        });

        if (errors.Count > 0)
        {
            return TypedResults.ValidationProblem(errors);
        }
    }
    return TypedResults.Ok(person);
}

```



Die Datenklassen sind die gleichen wie im ersten Beispiel (siehe Listing 1): *Person*, *Address* und *PhoneNumber*. Die Web-API-Aktionsmethode besitzt einen Parameter vom Typ *JsonPatchDocument<Person>*. Das JSON-Patch-Dokument wird im Body des HTTP-Aufrufs erwartet.

Da man einen *PATCH*-Aufruf nicht in einem Webbrowser auslösen kann, verwendet man zum Testen die HTTP-Tests in Visual Studio (HTTP-Dateien) (Bild 5) oder ein externes Werkzeug wie Fiddler [5] oder Postman [6] (Bild 6). Das JSON-Patch-Dokument übergibt man via HTTP-Body. ■

- [1] Paul C. Bryan, Marc Nottingham, *JavaScript Object Notation (JSON) Patch*, <https://datatracker.ietf.org/doc/html/rfc6902>
- [2] Holger Schwichtenberg, *System.Text.Json*, <https://dotnet-lexikon.de/SystemTextJson/Lex/10722>

Listing 7: Simulierte Geschäftslogik für Listing 5

```
public class PersonManager
{
    public async Task<Person?> GetPersonByIdAsync(int id)
    {
        // Originalobjekt
        var person = new Person
        {
            FirstName = "Holger",
            LastName = "Schwichtenberg",
            PrivateEmail = "buero@IT-Visions.de",
            PrivateWebsite = "www.dotnet-doktor.de",
            CompanyWebsite = "www.IT-Visions.de",

            PhoneNumbers = [new PhoneNumber() { Number =
                "0201 649590-0", Type = PhoneNumberType.Work }],
            Address = new Address
            {
                Street = "Fahrenberg 40b",
                City = "Essen",
                State = "NRW"
            }
        };
        return await Task.FromResult(person);
    }
}
```

[3] NuGet-Paket `Microsoft.AspNetCore.JsonPatch.System-Text.Json`, www.nuget.org/packages/Microsoft.AspNetCore.JsonPatch.SystemText.Json

[4] JSON Patch with System.Text.Json, <https://github.com/dotnet/core/blob/main/release-notes/10.0/preview/preview4/aspnetcore.md#json-patch-with-systemtextjson>

[5] Fiddler, www.telerik.com/fiddler

[6] Postman, www.postman.com

Dr. Holger Schwichtenberg

gehört zu den bekanntesten .NET- und Webexperten in Deutschland. Von Microsoft ist er seit 22 Jahren als MVP ausgezeichnet. Mit dem 51-köpfigen Expertenteam bei www.IT-Visions.de bietet er Beratungen, Schulungen und Softwareentwicklung an.

www.dotnet-doktor.de

DWX ACADEMY

Hands-on Workshops
und Weiterbildung für
Software-Entwickler
und -Architekten

EINFÜHRUNG IN REQUIREMENTS ENGINEERING

- Requirement-Typen
- Rollen
- Produktvision
- Requirements-Hierarchien
- Schätzungen
- Projekt-Roadmap
- User Stories
- Tools
- Agile Entwicklung
- Lifecycle

NEU!
2 TAGE



DAVID TIELKE



Deine Ansprechpartnerin: Susanne Herl
+49 731 88005-8835 • susanne.herl@ebnermedia.de

developer-world.de/dwx-academy

REGELWERKE FÜR DIE MICROSOFT-CLOUD

Azure Policies

Motivation, Struktur und Funktion von Azure Policies.

Cloud Computing hat die Art und Weise verändert, wie Unternehmen IT-Infrastrukturen aufbauen und verwalten. Mit der zunehmenden Verlagerung von Workloads in die Cloud wächst die Notwendigkeit, zentrale Vorgaben und Richtlinien durchzusetzen. Azure Policies bieten eine Möglichkeit, die Governance, Compliance und Standardisierung in einer Microsoft-Cloud-Umgebung über Ressourcen und Subscriptions hinweg sicherzustellen. Mit wachsender Komplexität von Cloud-Umgebungen steigt die Gefahr für Inkonsistenzen und Sicherheitslücken. Besonders in großen Organisationen – aber grundsätzlich auch in allen produktiven Umsetzungen – ist es unerlässlich, dass Vorgaben wie erlaubte Protokolle, erreichbare Endpunkte, Ressourcentypen oder geografische Einschränkungen eingehalten werden.

Ohne zentrale Steuerung können einzelne Teams oder Benutzer Ressourcen erstellen, die gegen interne oder externe Compliance-Vorgaben verstoßen. Dies muss entweder direkt verhindert oder zumindest in einem lückenlosen Monitoring transparent dargestellt werden. Branchenübergreifende Compliance-Vorgaben verlangen nach klaren Nachweisen für Sicherheit und Datenschutz und sind für Organisationen verpflichtend. Azure Policies helfen, diese Anforderungen in der Cloud durchzusetzen und Verstöße frühzeitig zu erkennen. Gleichzeitig erlauben sie Unternehmen und Projekten, ihre eigenen Standards konsistent zu implementieren.

Manuelles Überwachen und Anpassen von Ressourcen lässt sich in großen Cloud-Umgebungen nicht realisieren. Policies, insbesondere zusammen mit einem codebasierten Deployment-Ansatz, schaffen die Grundlage für automatisierte Prüfungen und Korrekturen, wodurch menschliche Fehler reduziert und Skaleneffekte realisiert werden können. Mithilfe von Azure Arc als Brückenservice ist es sogar möglich, Teile von On-Premises-Ressourcen zu überwachen.

Aufbau und Funktionsweise

Eine Azure Policy ist im Kern eine Regel, die prüft, ob Ressourcen beziehungsweise deren Konfiguration bestimmten Anforderungen entsprechen. Je nach Bedarf kann die Policy dann nichtkonforme Ressourcen melden, blockieren oder sogar automatisch anpassen. Es wird im Prinzip ein Grund gesucht, warum etwas nicht passt – und dann je nach Vorgabe weiter verfahren.

Unter der Haube ist eine Policy eine JSON-Datei. Sie wird auch Definition genannt, und ihr Aufbau muss einer bestimmten Struktur folgen. Zunächst hat das Vorhandensein einer Policy keine Auswirkung im Tenant – dies geschieht erst durch die Zuweisung (*assignment*) in einem festgelegten Gültigkeitsbereich. Überprüft werden können grundsätzlich

Azure Subscriptions, Ressourcengruppen und Ressourcen selbst. Jede Policy ist aufgeteilt in folgende Abschnitte:

- Name und Beschreibung,
- Metadaten und Modus,
- Parameter und Policy-Regel.

Metadaten können zum Beispiel eine Kategorie sein oder eine Versionsnummer. Wird gegen den Azure Resource Manager (ARM) und damit die Azure Infrastruktur gearbeitet, ist der Modus in der Regel *All*. Als Variante gibt es noch *indexed*. In diesem Fall werden lediglich Ressourcentypen ausgewertet, welche auch Tags unterstützen und eine Region als Eigenschaft haben.

Policies können auf der Ebene des Data Plane bei bestimmten Ressourcenanbietern auch im *provider mode* implementiert werden. So unterstützt der Azure Kubernetes Service (AKS, Microsoft.Kubernetes.Data) Cluster und deren Pods oder Container. Weitere Optionen gibt es unter anderem noch für den KeyVault oder die Data Factory.

Parameter dienen dazu, Werte an die Policy zu übergeben und dynamischer und wiederverwendbarer zu gestalten. Sehr oft ist das zumindest der Effekt der Policy, aber auch mögliche Wertausprägungen mit Listen, Zeichenketten oder Booleans sind häufig sinnvoll für die Regeldefinitionen.

Regeln

Das Herzstück einer Policy besteht aus dem JSON-Block *policyRule*. In diesem findet sich eine If-Then-Kombination, die festlegt, welche Bedingungen (*if*) für den *compliance*-Trigger benötigt werden und was dann im weiteren Verlauf geschehen soll (*then*) – hier kommt der definierte Effekt der Policy zum Tragen. Anders formuliert: Welcher Umstand beziehungsweise welche Konfiguration der Azure-Ressourcen soll gefunden werden, und soll dann das Erstellen oder Anpassen dieser Ressource blockiert oder auditiert werden? Im Bereich der Bedingungen ist viel Flexibilität möglich – Kombinationen und Verschachtelungen aus einem logischen *UND*, einem *ODER* sowie einer Negation. In der Praxis resultiert das üblicherweise darin, dass ein Teil der Bedingung konkret auf den gewünschten Service-Typ gemappt wird und ein anderer einen spezifischen Wert überprüft.

Als einfaches Beispiel dienen soll die Policy „Allowed SQL Database collations“ [1] (Bild 1): Hier gibt es zwei Parameter für mögliche erlaubte Werte der Collations und den Policy-Effekt, der dann in der Policy-Regel verwendet wird. Für unterschiedliche Zielbereiche könnte das anders gehandhabt werden; beispielsweise würde in einer produktiven Umgebung zunächst der Standardeffekt Auditierung verwendet, um ein

```
{
  displayName: "Allowed SQL Database collations",
  description: "Will deny deploying databases with collations not listed as part of the allowed list",
  metadata: {
    category: "SQL",
    version: "1.0.0"
  },
  mode: "All",
  parameters: {
    listAllowedCollations: {
      type: "Array",
      metadata: {
        displayName: "Allowed Collation Names",
        description: "A list of approved collations, example: Latin1_General_BIN;SQL_Latin1_General_CP1_CI_AS"
      }
    },
    effect: {
      type: "String",
      metadata: {
        displayName: "Effect",
        description: "Deny, Audit or Disabled the execution of the Policy"
      },
      allowedValues: [
        "Deny",
        "Audit",
        "Disabled"
      ],
      defaultValue: "Audit"
    }
  },
  policyRule: {
    if: {
      allOf: [
        {
          field: "type",
          equals: "Microsoft.SQL/servers/databases"
        },
        {
          field: "name",
          notEquals: "master"
        },
        {
          not: {
            field: "Microsoft.SQL/servers/databases/collation",
            in: "[parameters('listAllowedCollations')]"
          }
        }
      ]
    },
    then: {
      effect: "[parameters('effect')]"
    }
  }
}
```

Die Policy „Allowed SQL Database collations“ (Bild 1)

Blockieren zu vermeiden. Im Dev-Bereich wird im nichtkonformen Beispiel das Deployment direkt abgelehnt. Die Regel prüft zum einen, ob es sich um die korrekte Ressource handelt, welche die Policy-Engine im Scan gerade betrachtet. Zum anderen geht es um den Vergleich des gelesenen Wertes der Datenbank *collation* mit einer Liste von erlaubten Werten. Für einen Policy-Alarm muss hier alles zutreffen (*allOf*): Der Ressourcentyp soll eine MSSQL-Datenbank sein, gleichzeitig aber nicht die master DB, und auch die gesetzte Collation ist nicht in der Liste der erlaubten Werte zu finden. Ist diese Regel erfüllt, meldet diese Policy die Ressource als non-compliant. Trifft das ein, wird der übergebene Effekt angewendet.

Für den Wertevergleich steht eine Vielzahl von Optionen zur Verfügung, bekannt aus den üblichen Verfahren in der Entwicklung: gleich, ungleich, ähnlich, kleiner, größer, matching – um nur einige zu nennen. Weitere Details finden Sie unter [2].

Felder und Alias

Der größte Aufwand beim Erstellen von Policies entsteht meist beim Formulieren der Bedingungen, um am Ende wirklich genau den gesuchten Umstand zu finden. Für die Umsetzung stehen Field-Expressions zur Auswahl. Diese liefern Daten über die aktuell durch die Policy-Engine gescannte Res-

source. Wie in der genannten Policy zu sehen und in der Praxis häufig verwendet, sind das zum Beispiel der Name des Service und der Ressourcentyp. Für die konkreten Werte der Ressourcen-Attribute kann auf gemappte Verweise, genannt Alias, zurückgegriffen werden. Im angegebenen Beispiel wäre das die Zeile *field: "Microsoft.*

SQL/servers/databases/collation". Dahinter verbirgt sich der API-Pfad, unter dem die Property abgerufen werden kann. Meistens sind das, wie in der Collation, einzelne Werte. Es können aber auch Arrays zum Beispiel mit IP-Adressen sein, durch die dann Regeln für Vergleiche iteriert werden können, wie etwa beim Storage Account mit *Microsoft.Storage/storageAccounts/networkAcls.ipRules[*]*. Um für die Policy-Entwicklung den richtigen Alias zu finden, gibt es verschiedene Wege beziehungsweise Abfragen: mit der Azure PowerShell (*Get-AzPolicyAlias*), dem Azure CLI (*az provider show*) oder über das Azure-REST-API. Hier ist dann ein Filter auf einen bestimmten Provider sinnvoll, beispielsweise kann für SQL folgendes CLI-Kommando verwendet werden:

```
az provider show --namespace Microsoft.SQL
--expand "resourceTypes/aliases"
--query "resourceTypes[]
.aliases[].name"
```

Eine praktische Übersicht bietet [3], und Details für andere Optionen finden Sie unter [4].

Policy-Effekte

Ergibt die Auswertung der Policy-Bedingungen, dass eine Ressource nicht den Regeln entspricht, stellt sich wie bereits erwähnt die Frage, was mit diesem Zustand weiter geschehen soll. Dann kommen die definierten Maßnahmen – genannt Effekte – zum Tragen. Die dafür aktuell verfügbaren Effekte sind:

- *addToNetworkGroup*
- *append*
- *audit*
- *auditIfNotExists*
- *deny*
- *denyAction*
- *deployIfNotExists*
- *disabled*
- *manual*
- *modify*
- *mutate*

An dieser Stelle kann nicht auf alle Effekte wirklich detailliert eingegangen werden. Von der eigenen Praxis her sind die wichtigsten beziehungsweise am meisten verwendeten ►

Effekte *audit* und *auditIfNotExists* (*AINE*) sowie *deny* und *deployIfNotExists* (*DINE*). Im Fall von *audit* kann das Erstellen beziehungsweise das Update einer Ressource auch im nicht regelkonformen Fall (non-compliant) vorgenommen werden, der nicht konforme Zustand wird jedoch gemeldet. *deny* blockiert komplett und sollte deshalb mit Bedacht eingesetzt werden, ist daher aber auch sicherer. Gerade bei bereits existierenden Services, welche angepasst werden sollen, kann das aber ungeplante Auswirkungen haben. Wichtig zu wissen ist, dass es am Ende keine Rolle spielt, an welcher Eigenschaft eine Änderung vorgenommen werden soll und welche Property nicht konform ist, da die ganze Ressource vor dem Absenden zum Azure Resource Manager (ARM) geprüft wird. Ein Beispiel: Überprüft die Policy vorhandene Tags eines Storage-Accounts, die nicht regelkonform sind, kann auch keine Änderung an Netzwerkeinstellungen vorgenommen werden. Solange die Policy aktiv ist und zum Beispiel noch ein gefordertes Tag fehlt, gibt es kein Update der Ressource.

deployIfNotExists kann sehr praktisch für das (fast) automatische Anpassen der Services sein. Stimmt die Bedingung nicht, kann eine Eigenschaft geändert oder hinzugefügt werden. Gängiges Beispiel ist hier das Erstellen eines Tags, das gewünscht ist, aber nicht direkt beim Deployment angegeben wird. Das Ganze funktioniert im Zusammenspiel mit einem integrierten ARM-Template und einer angegebenen *identity*. Das Template bestimmt an dieser Stelle, was geändert wird, die Identität benötigt im Hintergrund die nötigen Rechte (RBAC), dies zu tun. Wie genau das Aufräumen (Remediation) funktioniert, wird im weiteren Verlauf noch besprochen.

Zum besseren Verständnis der Paare *audit*-*AINE* und *deny*-*DINE* sei noch angemerkt, dass sie zwar ähnlich funktionieren, aber eine andere Zielgruppe haben: *audit* und *deny* prüfen Eigenschaften auf Ebene der Ressourcen, *AINE* oder *DINE* betrachten Properties von Child-Ressourcen oder Extensions. Auch die Such-Logik ist etwas anders: Während *deny* im Prinzip einen Grund zum Meckern sucht, wird *AINE* erst aktiv, wenn die Suche nicht erfolgreich war. Ebenso ist die JSON-

Struktur der Policies etwas unterschiedlich. Die Effekte sind in vielen Fällen in Paaren austauschbar, das heißt, *deny*-*audit*, *AINE*-*DINE* oder *append-modify* können jeweils ohne beziehungsweise mit nur geringer Regelanpassung geändert werden. Die verschiedenen Effekte erlauben es daher, unterschiedliche Governance-Strategien abzubilden – vom bloßen Überwachen bis zum rigorosen Erzwingen von Standards.

Dynamische Policies

Prinzipiell sind Azure Policies recht individuell beziehungsweise starr, bedingt durch das JSON-Format und weil sie unter anderem meist nur auf einem bestimmten Ressourcen-Provider arbeiten. Dennoch gibt es Möglichkeiten, diese flexibler und dynamisch zu gestalten, unabhängig von zusätzlichen IaC-Optionen (IaC = Infrastructure as Cloud). Hervorzuheben ist an dieser Stelle die *field*-Implementierung, die für den *if*-Bereich einer Policy elementar ist.

Azure-Policy-Funktionen erlauben es, komplexe Logik innerhalb einer Policy-Regel zu implementieren, da sie sowohl auf Eigenschaften der zu prüfenden Ressourcen als auch auf Kontextinformationen zugreifen können. Grundsätzlich stehen fast alle Funktionen aus den Templates des ARM zur Verfügung. Einige speziellere oder sicherheitsrelevante Funktionen sind jedoch in Policy-Regeln ausgeschlossen, etwa *copyIndex()*, *deployment()*, oder *managementGroup()*. Diese ausgeschlossenen Funktionen bleiben im Template-Bereich lediglich innerhalb von *deployIfNotExists*-Policies nutzbar.

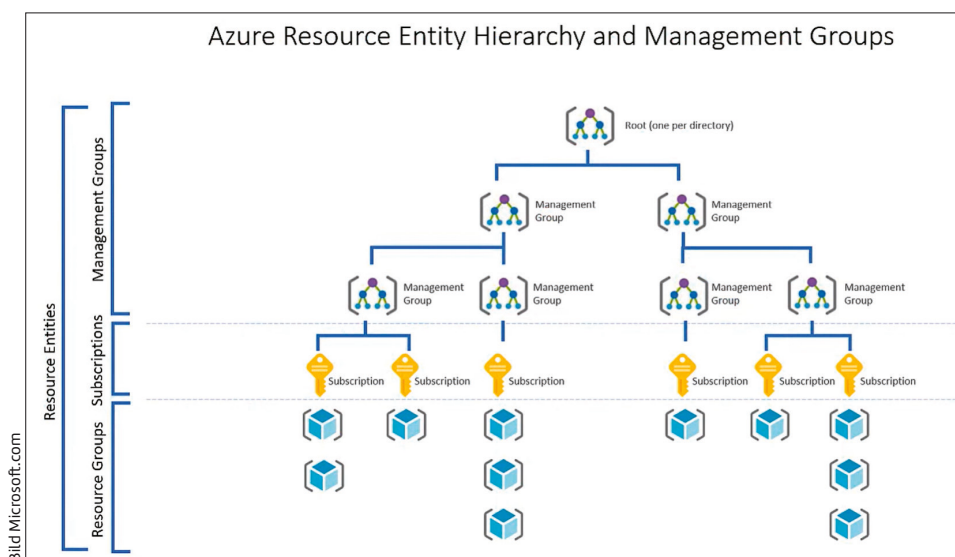
Um Informationen aus dem aktuellen Kontext der ausgeführten Policy zu erhalten, kann *resourceGroup()* oder *requestContext()* für die verwendete API-Version hilfreich sein. Wird mit einer Aufzählung gearbeitet, weil der Alias ein Array zurückgibt, kommt *current()* für den aktuellen Index zum Einsatz. *concat()* für das Verketteten von Zeichenketten findet sich beispielsweise in einigen Tag-Policies [5].

Ein kurzes praktisches Beispiel für den Einsatz liefert eine Policy zur Suche nach einem bestimmten Tag, übergeben als Parameter:

```
"if": {
  "not": {
    "field": "[concat('tags[' ,
      parameters(
        ' neededTagName '),
        ''])]",
    "equals": "[parameters(
      ' neededTagValue ')]"
  }
}
```

Häufig wird die Anzahl bestimmter Elemente eines Alias-Arrays für eine Policy-Bedingung gezählt (*count*), etwa bei Private-Link-Überprüfungen:

```
{
  "count": {
```



Azure-Tenant-Hierarchie (Bild 2)

```

    "field": "Microsoft.Network/privateEndpoints/
      privateLinkServiceConnections[*].groupIds[*]",
    "where": {
      "field": "Microsoft.Network/privateEndpoints/
        privateLinkServiceConnections[*].groupIds[*]",
      "equals": "mysqlServer"
    }
  },
  "greaterOrEquals": 1
}

```

Zuweisen einer Policy

Damit eine Policy überhaupt wirken kann, muss sie auf einen gewünschten Gültigkeitsbereich (Scope) verwiesen (assigned) werden – die reine Definition allein bewirkt zunächst gar nichts. Die Policy selbst kann auf Ebene einer Managementgruppe oder einer Subscription abgelegt werden, das Assignment kann zusätzlich noch auf einer Ressourcengruppe und einem einzelnen Service vorgenommen werden. Von großer Bedeutung ist hier außerdem die Ressourcen-Hierarchie: Die Zuweisung einer Policy lässt sich nur im gleichen Element, in der sie gespeichert ist, oder darunter vornehmen, vergleiche Bild 2.

Außerdem gilt hier das Prinzip der Vererbung von oben nach unten sowie bei Überlappungen. *cumulative most restrictive*, eine Policy, die etwa einer Managementgruppe zugewiesen wird, gilt automatisch auch für Subscription und Ressourcengruppen darunter. Schränkt eine Policy weiter oben in der Hierarchie ein, kann dies nicht durch ein Erlauben weiter unten übersteuert werden. Für diese Fälle, wenn beispielsweise eine Development-Subscription von einer Policy ausgenommen werden soll, muss dies bereits bei der Zuweisung als Exklusion im oberen Scope definiert werden. Ebenso können Ausnahmen für einzelne Ressourcen vorgenommen werden.

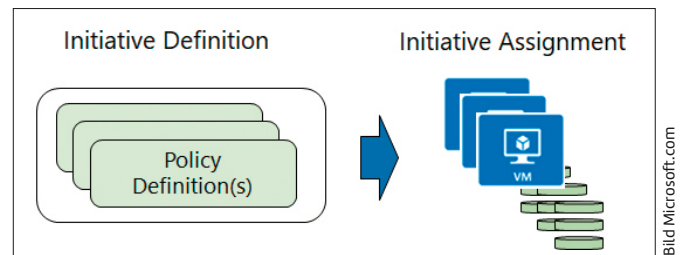
Auch wenn eine Richtlinie auf Ebene der Managementgruppe zugewiesen werden kann, werden grundsätzlich nur Ressourcen der Subscriptions oder Ressourcengruppen ausgewertet.

Initiativen

Als Möglichkeit, Policies zu gruppieren beziehungsweise zu bündeln und damit den Prozess der Verwaltung etwas zu vereinfachen, stehen Initiativen (*policy sets*) zur Verfügung. Vordefinierte oder selbst erstellte Policies werden für das Paket ausgewählt, danach muss lediglich noch die Initiative selbst zugewiesen werden, damit alle enthaltenen Regelwerke wirken (Bild 3).

Hilfreich sind oft die bereits eingebauten Initiativen. Ein Beispiel dafür ist *Audit Public Network Access*, die für eine große Sammlung an Services einen eventuell erlaubten öffentlichen Internetzugriff als nicht regelkonform protokolliert.

ASC Default ist die StandardPolicyInitiative, die in Azure Tenants automatisch aktiv ist und als Basis für den Defender for Cloud (ehemals Azure Security Center) dient. Die Vorschläge zur Optimierung der Tenant-Konfiguration, die im Security-Dashboard zu sehen sind, stammen von dieser Initiative. Die



Definition und Zuweisung einer Initiative (Bild 3)

Anwendung von Microsoft verwalteter Sets kommt insbesondere im Bereich der Regulator Compliance zum Tragen.

Soll ein öffentlicher Security-Standard komplett abgebildet werden, ist dies über eine zusätzliche kostenpflichtige Komponente (Plan) im Defender for Cloud möglich. Hier eine Auswahl der Optionen:

- ISO 27001
- NIST Cybersecurity Framework
- PCI DSS (Payment Card Industry Data Security Standard)
- HIPAA (Health Insurance Portability and Accountability Act)
- CIS Controls (Center for Internet Security)

Wie beim Dashboard von Defender for Cloud verwenden die aktivierten Standards Policies für ihre Empfehlungen.

Parameter können auf Ebene von Initiativen ebenfalls definiert und an die einzelnen Policies übergeben werden. Andernfalls müssten diese für die einzelnen Policies bereits während des Erstellens der Initiativen festgelegt werden; im Assignment-Schritt ist das nicht mehr möglich. Generell kann die Verwaltung von Initiativen etwas umständlich werden, vor allem wenn sich Parameter-Definitionen der Policies ändern (Bild 4). Sollte die Initiative größer werden, kann mit Gruppen noch etwas Übersicht geschaffen werden.

Built-in versus custom

Sollte man fertige Lösungen kaufen oder sie doch lieber selbst bauen? Das ist eine häufig gestellte Frage, auch bei Azure Policies.

Built-in Policies oder Initiativen werden von Microsoft verwaltet, mittlerweile ist hier eine beachtliche Anzahl verfügbar. Sie decken bereits eine Vielzahl gängiger Compliance-Anforderungen und Best Practices ab, beispielsweise beim Schutz sensibler Daten, bei der Einhaltung von Sicherheitsstandards oder der Ressourcenverwaltung. Sie sind im Tenant direkt verfügbar und somit sofort einsatzbereit, können schnell zugewiesen werden und erhalten auch Updates von Microsoft, um neue Anforderungen oder Plattformänderungen abzudecken. Das kann den manuellen Aufwand reduzieren.

Zu den häufig genutzten Built-in-Policies und Initiativen in Azure gehören:

- *Allowed locations (deny)* – Ressourcen sollen nur in bestimmten, genehmigten Regionen erstellt werden.
- *Inherit a tag from the subscription (modify)* – Fügt bei einer Ressource ein Tag und den zugehörigen Wert von der übergeordneten Subscription hinzu. ►

- *Azure Key Vault should disable public network access (deny)* – Verhindert den Zugriff auf die Ressource über das Internet.
- *Container Apps should only be accessible over HTTPS (audit)* – Erzwingt oder auditiert die Protokollverwendung. Gibt es wie beim Key Vault auch äquivalent für viele andere Services.
- *ASC Default-Initiative* als Standard für den Defender for Cloud.

Reference ID	Parameter name	Type	Value Type	Value(s)
Require a tag and its va...	Tag Name	String	Set value	env
Require a tag and its va...	Tag Value	String	Use Initiative Pa...	envvalue
Allowed locations_1	Allowed locations	Array	Set value	2 selected

Definition von Parametern im Bereich „Initiativen“ des Azure Portals (Bild 4)

Mittlerweile wird auch Versionierung in den Policies unterstützt, was Abhilfe bei bekannten Problemen schafft, wenn Regeln beispielsweise per Name zugewiesen wurden und sich die Definition später geändert hat. Unter anderem deshalb sahen sich etliche Projekte veranlasst, die JSON-Daten einfach in ein eigenes Repository zu kopieren und als Custom Policy anzuwenden, um die Kontrolle über Updates zu behalten. Mit Versionen lässt sich nun ein deutlich besseres Change Management fahren, zum Beispiel über ein stufenweises Ausrollen der Updates über die Umgebung. Im Git-Hub-Repository unter [6] sind die eingebauten Policies einsehbar.

Custom Policies erlauben eine angepasste Gestaltung der Regelwerke nach spezifischen Bedürfnissen und Vorgaben. Sie kommen immer dann zum Einsatz, wenn die vorhandenen Standardrichtlinien nicht ausreichen, etwa bei besonderen regulatorischen Anforderungen, unternehmensspezifischen Prozessen oder branchenspezifischen Vorgaben.

Lassen sich die Vorgaben nicht mit eingebauten Policies umsetzen, muss man selbst Hand anlegen. Dabei ist viel darstellbar. Durch die Vielzahl an Aliases zum Zugriff auf die Ressourceneigenschaften, das Einbinden zusätzlicher Funktionen und die Möglichkeiten, Bedingungen zu definieren, lässt sich durchaus komplexe individuelle Logik umsetzen. Allerdings sind dafür mehr Fachwissen und Aufwand erforderlich – nicht nur für den Aufbau, sondern auch für Wartung und Aktualisierung der Eigenbau-Policies.

Zusammengefasst bieten eingebaute Policies einen schnellen Einstieg und decken viele Standardszenarien ab, während Custom Policies maximale Flexibilität für individuelle Governance bieten – verbunden mit höherem Aufwand bei Entwicklung und Pflege.

Als ein möglicher Ansatz empfiehlt es sich, mit eingebauten Policies zu starten und diese bei Bedarf durch maßgeschneiderte Policies zu ergänzen. Nicht immer ist der Aufwand für eine individuelle Policy sehr groß. Orientiert man sich an den vorhandenen Regeln, ergibt dies oft bereits eine gute Basis, in manchen Fällen genügt es sogar, Parameter oder Effekt anzupassen, um neben *audit* dann auch *deny* zu unterstützen.

Um das Thema abzurunden, sei noch der Policy-Typ *static* erwähnt. Dahinter verbirgt sich eine Regulatory-Compliance-Definition unter Microsoft-Verwaltung mit Assessments von deren Infrastruktur.

Grenzwerte für Policies

Auch wenn im Bereich der Cloud häufig der Eindruck entstehen kann, es gäbe keine Grenzen, stimmt das bei Azure Policies genauso wenig wie bei den anderen Services. Weder in den Bestandteilen von Policies oder Initiativen noch in der Regel-Definition kann es unbegrenzt komplex werden. Beispielsweise sind nur 20 Parameter pro Policy erlaubt, andere Grenzen gibt es für die Anzahl der Definitionen, Zuweisungen oder File-Größen. In den Regeln gibt es unter anderem Limits für die Zahl an Conditions oder die Verschachtelungen von Funktionen – im üblichen Rahmen sollte das aber keine Probleme bereiten, weitere Details dazu finden Sie im Web unter [7].

Wo die Grenzen in der Praxis zu Schwierigkeiten führen, hängt vom Anwendungsfall ab. Aber vermutlich benötigen die wenigsten Fälle 2048 Funktionen pro Regel oder 1000 Policies pro Initiative – genauso wenig wie 10 000 Managementgruppen oder 50 Tags für eine Ressource.

Monitoring und Compliance

Was folgt nun, wenn eine Policy definiert und zugewiesen ist? Bevor gezeigt wird, wo die Ergebnisse der Auswertungen zu sehen sind und welche weiteren Schritte unternommen werden können, zunächst ein Blick auf die Auslöser eines Policy-Scans, denn eine Neubewertung der Ressourcen-Compliance kann durch verschiedene Ereignisse in der Umgebung ausgelöst werden. Typische Trigger sind unter anderem:

- Wenn einem Scope eine neue Policy oder Initiative zugewiesen wird, etwa eine Ressourcengruppe. Alle darin enthaltenen Services werden geprüft.
- Wenn eine Ressource erstellt oder innerhalb eines Gültigkeitsbereichs mit der Policy-Zuweisung aktualisiert wird – also *create* oder *update*.
- Wenn die Definition einer bereits zugewiesenen Policy oder Initiative geändert wird. Das heißt, das Assignment selbst muss nicht aktualisiert werden.
- Zusätzlich finden regelmäßige Überprüfungen in einem Zyklus von 24 Stunden statt.

Um nicht den 24-Stunden-Rhythmus oder ein Update einer Ressource abwarten zu müssen, kann ein Trigger auch manuell für einen bestimmten Scope ausgelöst werden. Dies ist insbesondere im Entwicklungsprozess hilfreich. Eine Option ist das folgende Azure-CLI-Snippet:

```
az policy state trigger-scan --resource-group "rg-test"
```

PowerShell oder das Azure API funktionieren ebenfalls, und für die Anwendung im DevOps-Bereich gibt es eine eigene GitHub-Action dafür.

Reihenfolge

Interessant ist auch die Reihenfolge, in der die Auswertung der Regeln erfolgt. Die Policy-Engine bewertet grundsätzlich Anfragen zum Erstellen oder Aktualisieren von Ressourcen bereits bevor diese zum Resource Manager gesendet werden. Das soll unnötigen Traffic vermeiden, wenn der Service nicht regelkonform konfiguriert ist. Dabei wird geprüft, welche Policy-Zuweisungen für die jeweilige Ressource gelten, und diese werden nacheinander abgearbeitet. Die Reihenfolge der Effekte, welche für den Ressourcen-Scope zutreffen, ist dann wie folgt:

Zunächst wird geprüft, ob die Policy deaktiviert ist, dann werden *append* und *modify* verarbeitet – eventuell können diese ein späteres *deny* verhindern –, gefolgt von *deny*, *audit*, *manual*, *auditIfNotExists* und abschließend der *denyAction*. Nach erfolgreicher Bearbeitung durch den Ressourcen-Provider wird zusätzlich geprüft, ob weitere Compliance-Prüfungen oder Aktionen durch *auditIfNotExists* oder nachträgliche Konfigurationsanpassungen durch *deployIfNotExists* notwendig sind.

Policy-Status

Navigieren Sie im Azure Portal zur Sektion *Policies*, begrüßt Sie ein Compliance-Dashbaord mit einer Ansicht des Zustands der Azure-Umgebung (Bild 5). Ein kurzer Blick informiert über Zahl und prozentualen Anteil regelkonformer Res-

ourcen sowie darüber, aus welchen Policies oder Initiativen diese stammen. Von hier aus kann bequem weiter recherchiert werden. Definitionen, Zuweisungen, Ausnahmen und der Bereich Bereinigung (Remediation) stehen direkt zur Verfügung.

Für einen detaillierteren Blick auf den Tenant kann der Abschnitt *Compliance* ausgewählt werden, wo zusätzliche Filter verfügbar sind, wie Gültigkeitsbereich, Art der Definition oder Status. Wenn erforderlich, können in dieser Sektion auch temporäre Ausnahmen für Non-Compliance-Ressourcen erstellt werden.

Der *compliance state* kann auch per Code abgefragt werden, beispielsweise mit dem Azure CLI:

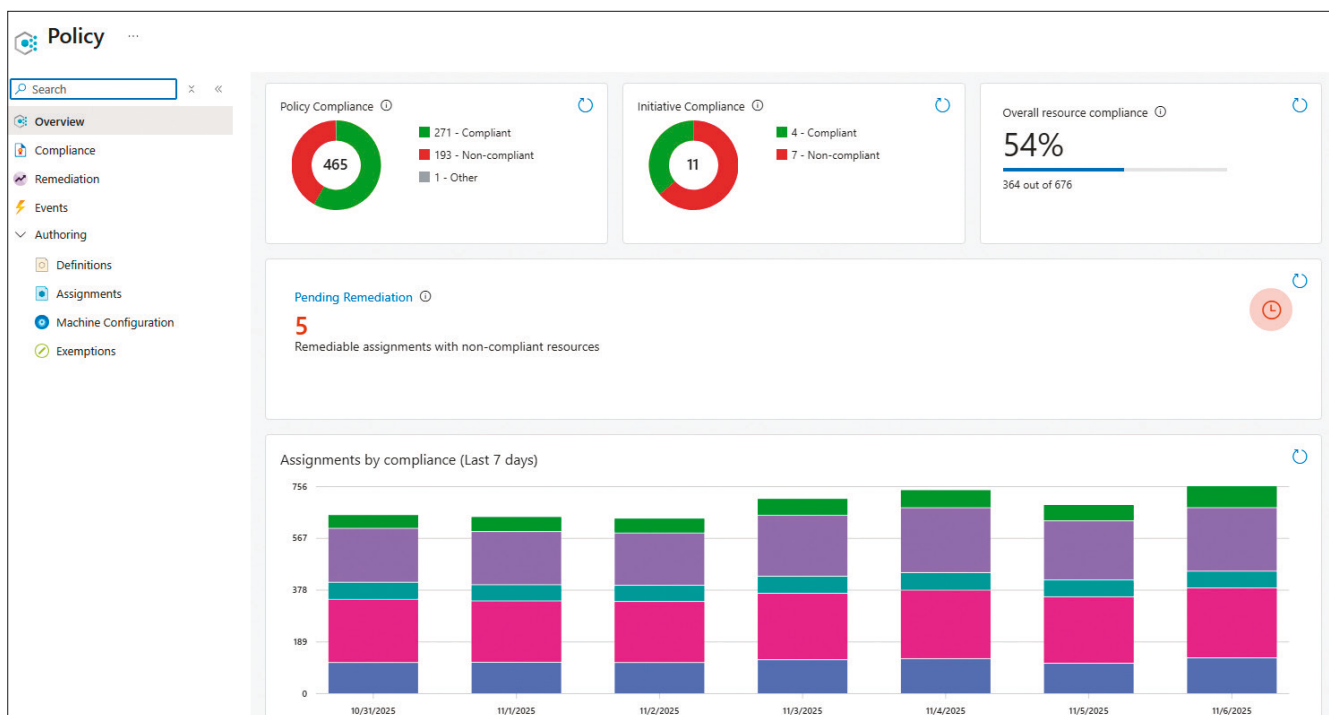
```
az policy state list
```

Das Kommando liefert eine komplette Übersicht im aktuell gewählten Subscription-Kontext. Weitere Parameter und Filteroptionen für *scopes* oder *resource types* gibt es natürlich auch. Wer die Kusto Query Language (KQL) bevorzugt, kann auch über den Resource Graph gehen [8].

Remediation

Der Policy-Status ist ausgewertet, nun geht es ums Aufräumen. Zunächst dort, wo es direkt unterstützt wird. Die Effekte *deployIfNotExists* und *modify* ermöglichen das Wiederherstellen der Compliance entweder vollautomatisch oder mit einer schnell umsetzbaren Aufgabenerstellung. Ressourcen, die lediglich auditiert wurden, müssen auf jeden Fall manuell korrigiert werden.

Eine häufige Anwendung findet dieses Prinzip beim Hinzufügen fehlender Tags oder beim Anpassen spezifischer ►



Compliance-Überblick im Azure Portal (Bild 5)

Konfigurationswerte. Der Benutzer wird also nicht komplett blockiert, wenn eine Angabe nicht stimmt, sie wird im Nachgang angepasst.

Um eine Bereinigung durchzuführen, die nicht automatisch getriggert wurde, muss ein Remediation-Task erstellt werden. Dies erfolgt entweder durch eine Konfiguration in der Policy-Zuweisung oder als zusätzlicher Schritt im weiteren Verlauf. Am Ende führt der Task einfach das Template-Deployment aus, welches in der Policy definiert wurde. Damit das erfolgreich funktioniert, braucht es die Angabe der entsprechenden Rollendefinition (beispielsweise Contributor, aber idealerweise mit so wenig Rechten wie möglich) sowie die Zuweisung dieser Rolle zu einer Identity der Policy. Diese kann, wie aus anderen Bereichen von Azure bekannt, eine konkret auf die Policy gemappte (System-assigned Identity) oder eine wiederverwendete User-assigned Identity sein.

Am in Bild 6 gezeigten Beispiel für die SQL DB transparent data encryption können Sie das Prinzip einer DINE-Policy nachvollziehen. Sie sehen die im Vergleich zu *deny* etwas andere Struktur. Hier gibt es Einträge für *roleDefinitionIds*, die bestimmen, welche Rolle für die Deployments am Ende benötigt wird. Die *existenceCondition* prüft das Vorhandensein des gewünschten Settings, und die Deployment-Sektion enthält das nötige Template für die Anpassung der Konfiguration.

Funktioniert die Bereinigung nicht automatisch, ist eine Möglichkeit zur Umsetzung direkt im Azure Portal verfügbar. Im Policy-Bereich unter *remediation* kann der Task erstellt werden und entweder alle oder ausgewählte Ressourcen bereinigen. Im Übersichts-Tab der Tasks lässt sich der Status der Durchführung verfolgen. Schlägt er fehl, liegt das in vielen Fällen an fehlenden Rechten der zugewiesenen Identity. Eine Remediation kann aber auch per Azure-PowerShell-Skript getriggert werden.

Prinzipiell ist es so, dass neu erstellte oder gänderte Ressourcen (*DINE* oder *modify*) automatisch angepasst werden können. Existierende Ressourcen werden bei Abweichungen in der Konfiguration als nicht regelkonform (non-compliant) angezeigt und müssen extra bereinigt werden. In der Praxis ist die Sachlage nicht immer so klar, besonders wenn der Automatismus nicht funktioniert. Häufig liegt das dann daran, dass zum Beispiel Policy-Logik und Update-Request auf unterschiedliche Endpunkte zielen.

```
"if": {
  "field": "type",
  "equals": "Microsoft.Sql/servers/databases"
},
"then": {
  "effect": "deployIfNotExists",
  "details": {
    "type": "Microsoft.Sql/servers/databases/transparentDataEncryption",
    "name": "current",
    "evaluationDelay": "AfterProvisioning",
    "roleDefinitionIds": [
      "/subscriptions/{subscriptionId}/providers/Microsoft.Authorization/roleDefinitions/{roleGUID}",
      "/providers/Microsoft.Authorization/roleDefinitions/{builtinroleGUID}"
    ],
    "existenceCondition": {
      "field": "Microsoft.Sql/transparentDataEncryption.status",
      "equals": "Enabled"
    },
    "deployment": {
      "properties": {
        "mode": "incremental",
        "template": {
          "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
          "contentVersion": "1.0.0.0",
          "parameters": {
            "fullDbName": {
              "type": "string"
            }
          },
          "resources": [
            {
              "name": "[concat(parameters('fullDbName'), '/current')]",
              "type": "Microsoft.Sql/servers/databases/transparentDataEncryption",
              "apiVersion": "2014-04-01",
              "properties": {
                "status": "Enabled"
              }
            }
          ]
        },
        "parameters": {
          "fullDbName": {
            "value": "[field('fullName')]"
          }
        }
      }
    }
  }
}
```

Beispiel einer DINE-Policy (Bild 6)

Implementierungsmöglichkeiten

Das Erstellen, Zuweisen und Überprüfen von Policies klappt per Klick im Portal oder per Code. Im Portal gibt es ein Template. In dieses kann die Policy-Regel inklusive der definierten Parameter eingefügt werden, dazu kommen noch die Metadaten wie Beschreibung oder Version in bestimmte Felder (Bild 7). Bei der Zuweisung ist es ähnlich.

Alternativ erlauben es Azure CLI und PowerShell, Richtlinien per Skript zu verwalten; dafür gibt es etliche Beispiele und Referenzen. Wer Infrastructure as Code (IaC) verwendet, kann Policies auch mit Tools wie Terraform oder Bicep verwalten, die als Wrapper agieren. Die Prinzipien ähneln sich: Es gibt jeweils Ressourcen für die Definition und Zuweisung. Das Kernstück bleibt der JSON-Code der Policy-Regel, die dann üblicherweise in eine Extradatei ausgelagert ist.

Azure CLI

Mit dem Azure CLI lässt sich eine Policy-Definition wie folgt erstellen und zuweisen:

```
# Policy-Definition erstellen
az policy definition create --name 'mypolicy'
```

```
--display-name 'my policy' --description '123'
--rules 'mypolicy.json' --mode All
```

```
# Policy-Definition zuweisen
az policy assignment create --name 'mypolicy'
--scope '/subscriptions/<guid>/resourceGroups/rg-test'
--policy '/subscriptions/<guid>/providers/
Microsoft.Authorization/policyDefinitions/mypolicy'
```

Um eine konsistente, sichere und wiederholbare Policy-Verwaltung umsetzen, sollten wie erwähnt allerdings Automatisierungslösungen bevorzugt werden; das Portal sollte eher für PoCs, Testing und Troubleshooting eingesetzt werden.

IaC – Terraform

Für die Policy-Bereitstellung mit Terraform auf Azure käme dann der Resource-Typ `azurerms_policy_definition` für die Regel sowie, je nach Level, ein anderer wie `azurerms_resource_group_policy_assignment` für die Zuweisung zum Einsatz (hier auf einer Ressourcengruppe). Eine Initiative kann mit dem Typ `azurerms_policy_set_definition` erstellt werden.

Speziell bei der Arbeit mit Terraform als Lösung für IaC-Ressourcen-Provisionierung und anpassenden Policies wie *DINE* ist allerdings etwas Vorsicht angebracht – hier kann sich schnell mal der Zuständigkeitsbereich überlappen, und es folgt eine Art Henne-Ei-Problem, weil die Policy etwas anpasst, was der Terraform-State nicht kennt.

Vorgehensweise und Strategien

Wie kann nun sinnvoll vorgegangen werden, wenn im Unternehmen ein Policy-Framework entwickelt werden soll? Falsch implementiert können Policies die Workloads lahmlegen oder andere ungewünschte Effekte haben. Je nach Größe der Azure-Umgebung empfehlen sich verschiedene Ansätze im Gesamtkonzept, ein besonderer Fokus liegt je-

doch immer auf Assignment-Scope und Aktivierungsstatus der Policy (Bild 8).

Bevor Policies auf hohen Ebenen wie Managementgruppen für einen breiten Gültigkeitsbereich zugewiesen werden, empfiehlt es sich, für den Entwicklungs- und Testprozess einen kleinen, konkreten Scope zu wählen, etwa eine Ressourcengruppe. In dieser werden dann genau die Servicetypen und spezifischen Konfigurationen getestet und verifiziert, die es für den aktuellen Fall braucht, und es gibt keine Überlappungen beziehungsweise Seiteneffekte von anderen Services. Diese kommen schneller vor, als man vielleicht denken mag, denn viele Services wie VMs oder Storage sind auch mit anderen Ressourcentypen verknüpft. Herrscht Sicherheit, kann der Scope ganz einfach erweitert werden.

Im Gegensatz zur Zuweisung kann beziehungsweise sollte die Definition einer Policy auf einer höheren Ebene stattfinden, da eine Zuweisung nur auf dem gleichen beziehungsweise dem darunterliegenden Level geschehen kann.

Bezogen auf den Aktivierungsgrad gibt es zum einen die Option, die Policy im Effekt einfach zunächst auf *disabled* zu stellen. So kann sie bereits implementiert werden und ist inaktiv, wird gleichzeitig aber auch immer als *compliant* angezeigt. Effektiver ist es für viele Fälle, nach der Policy-Entwicklung im Testscope zunächst einen auditiven Effekt zu verwenden und diesen nach dem Testprozess zu härten. Es wird also ein *deny* entwickelt, mit *audit* ausgerollt und (eventuell) später wieder auf *deny* gestellt.

Außerdem kann im Bereich des Assignments das Policy-Enforcement deaktiviert werden. Dies hat zur Folge, dass die Policy normal ausgewertet wird, die Compliance ersichtlich ist und gleichzeitig noch keine Änderung vorgenommen wird. Die Policy wird ohne Codeänderung in den Auditmodus versetzt, allerdings sind für *DINE* und *modify*-Effekte die Remediation-Tasks verfügbar, wenn bestimmte Ressourcen doch noch direkt bereinigt werden sollen. Mit dem Fortführen der Testphase lässt sich dann das Aktivieren der Zuweisungen schrittweise und sicher vornehmen.

Bedacht werden muss hier aber, dass es bei diesem Ansatz keine generierten Logs mit *compliance state* im Event-Grid gibt [9].

Sofern überhaupt mit Effekten gearbeitet wird, sollte es eher bei unterstützenden Elementen wie Tags und Diagnostic Settings zum Logging bleiben – die eigentlichen Workload-Provisionierungen gehören in IaC-Pipelines, beispielsweise mit Bicep oder Terraform.

Auf Effekte zur direkten Anpassung oder Provisionierung komplett zu verzichten ist vollkommen legitim, Gründe gibt es genug dafür.

Stattdessen dauerhaft mit *Audit* oder *AINE* zum Tracken der Compliance und beispielsweise einem Alerting basierend auf den Event-Grid-Logs zu ►

Policy-Definition im Azure Portal (Bild 7)



Policy-Entwicklungsprozess nach CAF (Bild 8)

arbeiten, ist ebenfalls ein vertretbarer Weg. Wer auf eigene Initiativen setzt, kann diese als Rahmen früh aufsetzen und zuweisen, um dann die Policies mit Abschluss des Testings nach und nach hinzuzufügen und damit die Verwaltung etwas zu vereinfachen, da das Assignment nicht wiederholt werden muss.

Auch bei Policy-Definitionen, Initiativen und deren Zuweisung gilt es als Best Practice, diese grundsätzlich als Code in Repositories zu verwalten und im Idealfall über Deployment-Pipelines zu provisionieren.

Stolpersteine bei der Policy-Anwendung

Bei der Arbeit mit Azure Policies gibt es leider auch ein paar Herausforderungen, nicht nur bei Entwicklung und Anwendung der Policy. Viele Schwierigkeiten treten zwar in diesem Kontext, aber in Bereichen außerhalb der Policy-Logik auf. Werden Template-Funktionen wie beispielsweise `substring()` verwendet und wirft diese Funktion einen nicht abgefangenen Fehler, schlägt die komplette Policy-Auswertung fehl, und es folgt ein `deny` des Deployments.

Viel hängt bei Policies an den verfügbaren Aliases. Die gewünschte Konfiguration mag im Azure Portal oder über ein CLI-Kommando sichtbar sein – wenn es jedoch keinen Alias dafür gibt, kann das Setting nicht von der Policy-Logik berücksichtigt werden.

Leider ist auch das Vorhandensein eines Alias keine Garantie, dass die Policy korrekt funktioniert – API-Abfragen im Hintergrund liefern leider nicht immer den korrekten Wert. Die Eigenschaft `federatedClientId` eines SQL-Servers gibt durch eine GUID direkt eine Aussage darüber, ob eine solche Identity für den Zugriff konfiguriert ist (Bild 9).

Der zugehörige Alias (`Microsoft.SQL/servers/databases/federatedClientId`), mit dem in der Policy gearbeitet werden kann, liefert aktuell einen eventuell vorhandenen Wert noch nicht mit, sondern enthält nur eine leere Zeichenkette – damit liefert die Policy eine fehlerhafte Compliance.

```
"properties": {
  "collation": "SQL_Latin1_General_CP1_CI_AS",
  "maxSizeBytes": "2147483648",
  "catalogCollation": "SQL_Latin1_General_CP1_CI_AS",
  "zoneRedundant": false,
  "readScale": "Disabled",
  "requestedBackupStorageRedundancy": "Local",
  "maintenanceConfigurationId": "/subscriptions/64296",
  "isLedgerOn": false,
  "federatedClientId": "ca878fe8-85d5-4f56-9d4e-8462d",
  "availabilityZone": "NoPreference"
}
```

CLI-Abfrage für SQL-Server-Parameter (Bild 9)

Service ist nicht gleich Service – fast immer gibt es unterschiedliche SKUs, die dann zum Teil andere Funktionen und damit auch Eigenschaften unterstützen. Ein häufiges Beispiel sind hier Konfigurationen im Bereich private VNET-Integration, die nicht in jeder SKU vorhanden sind. Wird etwa für Azure Databricks die eingebaute Policy „Azure Databricks Clusters should disable public IP“ angewendet, funktioniert

das gut für den Hybrid-Modus, blockiert aber als Seiteneffekt die Serverless-Variante komplett. Diese verfügt nicht über die Eigenschaft `enableNoPublicIp` für die Cluster, da der Service in dem Fall alles selbst verwaltet. Dafür ist dann doch wieder eine individuelle Policy erforderlich.

Wurde die Policy entwickelt und beispielsweise im Portal getestet, heißt das leider noch nicht, dass die Arbeit getan ist. Gerade bei Extension-Ressourcen wie `Audit Log settings` werden dann über das Azure-CLI oder Terraform im Hintergrund mitunter andere Endpunkte angesteuert, die damit die Policy-Auswertung umgehen können. Um beispielsweise eine bestimmte Retention für `SQL Server audit logs` zu erzwingen, reicht für Deployments aus dem Azure Portal sowie dem CLI in der Policy die Berücksichtigung des Alias `Microsoft.Sql/servers/auditingSettings/retentionDays` aus. Terraform interessiert das bei seinem Deployment für ein SQL retention setting ohne den weiteren Alias `Microsoft.Sql/servers/extendedAuditingSettings/retentionDays` herzlich wenig. Aufgrund eines anderen angesprochenen Endpunkts kann die Policy über Terraform somit noch ausgehebelt werden.

Genauso kann es passieren, dass verschiedene Deployment-Ansätze – zum Beispiel Portal und Terraform – unterschiedliche Konfigurationen deployen, wie bei der TLS-Version für einen MySQL Server. Portal und CLI erstellen ein Array für die Versionen 1.2-1.3, Terraform nutzt aktuell noch 1.0-1.2 (Bild 10).

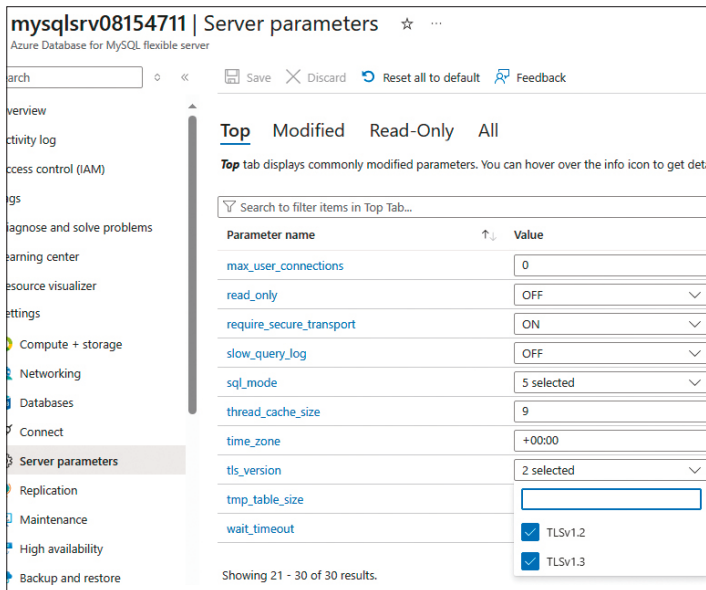
Soll hier etwa Version 1.3 über per `DINE` erzwungen werden, gibt es Probleme. Grundsätzlich gilt es bei diesem Ansatz etwas Vorsicht walten zu lassen: Je nachdem, welches Objekt in der `DINE`-Policy zum Setzen einer Eigenschaft benötigt wird, umfasst dieses auch andere Eigenschaften und überschreibt diese gegebenenfalls.

Bedacht werden sollte auch, dass einzelne Policies für einen spezifischen Service, beispielsweise einen Storage Account, vielleicht passend entwickelt wurden – dann aber zum Beispiel bei einer Azure Function, die diesen ebenfalls verwendet, Probleme machen kann.

Außerdem sei noch erwähnt, dass gerade im Entwicklungsprozess hohe Kosten entstehen können, denn schließlich müssen Ressourcen für die Tests erstellt werden. Ist hier der Prozess nicht sauber und kein entsprechender Lifecycle definiert, kann es teuer werden.

Fazit

Azure Policies bieten zahlreiche Vorteile: Sie ermöglichen eine zentrale und konsistente Steuerung von Compliance- und Governance-Anforderungen innerhalb einer Microsoft-Cloud-Umgebung. Dadurch lassen sich Richtlinien automatisiert durchsetzen und Verstöße frühzeitig erkennen. Die Ope-



TLS-Versionen bei CLI und Portal (Bild 10)

ration-Teams der Azure-Umgebungen können damit entlastet werden. Wenn das Zusammenspiel mit dem gewählten IaC-Weg der Ressourcen-Provisionierung abgestimmt ist, können Policies bestimmte Blueprint-Funktionen für neu bestellte Workload-Subscriptions übernehmen, etwa durch automatisch erstellte Logging-Konfigurationen.

Andererseits schränken sie auch ein. Eine aktive *deny*-Policy verhindert erst einmal Updates, unabhängig von den Rechten – und liefert gerne auch ungeplante Seiteneffekte in der Ressourcenkonfiguration. In Entwicklungs- und Testumgebungen kann das den Aufwand erheblich erhöhen, in der Produktion wirkt eine ungewollte Blockade verheerend.

Zudem kann ein klassischer Flaschenhals entstehen, wenn Änderungen oder Ausnahmen erforderlich sind, aber aufwendig abgestimmt werden müssen oder zu wenige Personen mit nötigen Rechten verfügbar sind. Und eines ist relativ sicher: Änderungen der Regelvorgaben oder der Ressourcen-Konfiguration werden kommen. Auch ein Policy-Setup funktioniert selten nach dem Modell „Einmal bauen, nie wieder anfassen.“

Für sicher und regelkonform aufgebaute Microsoft-Tenants führt kein Weg an Azure Policies vorbei. Über Menge, Scope und Art der Effekte in den Regelwerken kann sicher viel diskutiert werden. Ganz sicher gibt es keinen One-fits-all-Ansatz. Guidance mit Vorschlägen für verschiedene Anwendungsbereiche gibt es aber sehr wohl, unter anderem durch das Cloud Adaption Framework (CAF) von Microsoft. Wer sich mit Policies beschäftigt, wird unweigerlich auf die Website AzPolicyAdvertizer von Julian Hayward stoßen [10], ein großartiges Community-Projekt, das mittlerweile mehrere Tausend Policies gesammelt hat und definitiv einen tieferen Blick wert ist.

Wie so häufig mag es beim Einstieg vielleicht einfach klingen, doch der Teufel steckt auch bei Policies im Detail. Viele, aber nicht alle Einstellungen oder Szenarien lassen sich problemlos abbilden. Unterschiedliche Tools können verschie-

dene Werte erwarten und bereitstellen, was zu Inkonsistenzen und Problemen beim Erzwingen spezifischer Konfigurationen führen kann. Auch in verschiedenen Bereichen außerhalb der eigenen Kontrolle treten zusätzlich immer Bugs wieder auf. Die Flexibilität der diversen API-Endpunkte macht das Leben nicht immer einfacher.

Grundsätzlich ist auch im Bereich von Azure Policies ein strukturierter und eingehaltener Gesamtprozess für Implementierung, Verwaltung und Monitoring sehr wichtig. Je mehr Policies verwendet werden, desto schwieriger wird es, diese zu verwalten.

Policies sollten nicht isoliert betrachtet werden, sondern auch im Zusammenspiel mit Rechtemanagement (RBAC) und Resource Locks. Die Einführung von Versionierung seitens Microsoft ist eine praktische Neuerung, um den Aufwand und eigene Codeverwaltung zu optimieren, ohne Planungssicherheit zu verlieren. Policy-Projekte sind immer zu einem bestimmten Maß individuelle Setups. Empfehlungen gibt es, aber nicht die eine goldene Initiative, die alles abdeckt. ■

- [1] Policy Allowed SQL Database collations, www.azadvertizer.net/azpolicyadvertizer/be6d249b-259f-45ac-9a2d-74f4b42ad519.html
- [2] Conditions, <https://learn.microsoft.com/en-us/azure/governance/policy/concepts/definition-structure-policy-rule#conditions>
- [3] Azure Policy Aliases, <https://policyalias.mats.codes/>
- [4] Azure Policy definition structure aliases, <https://learn.microsoft.com/en-us/azure/governance/policy/concepts/definition-structure-alias>
- [5] Policy functions, <https://learn.microsoft.com/en-us/azure/governance/policy/concepts/definition-structure-policy-rule#policy-functions>
- [6] Azure Policies bei GitHub, <https://github.com/Azure/azure-policy/tree/master>
- [7] Policy rule limits, <https://learn.microsoft.com/en-us/azure/governance/policy/concepts/definition-structure-policy-rule#policy-rule-limits>
- [8] Azure Resource Graph sample queries for Azure Policy, <https://learn.microsoft.com/en-us/azure/governance/policy/samples/resource-graph-samples?tabs=azure-cli>
- [9] Adopt policy-driven guardrails, <https://learn.microsoft.com/en-us/azure/cloud-adoption-framework/ready/enterprise-scale/dine-guidance>
- [10] AzAdvertizer, www.azadvertizer.net



Thomas Hafermalz

ist als Berater und Trainer im Bereich Azure Infrastruktur, Architektur und Automatisierung unterwegs. Sein Wissen gibt er gerne in Blogs, Artikeln und Community-Events weiter.

www.linkedin.com/in/thomashafermalz/

MODERNE DATEN- UND APPLIKATIONSINTEGRATION MIT .NET 10

HTTP-APIs mit SQL Server 2025

Mit SQL Server 2025 auf ein eigenes .NET-10-HTTP-API zugreifen – dargestellt am Beispiel von Webhooks.

Die Integration von Datenbanken und Anwendungen hat sich in den letzten Jahren dramatisch verändert. Während früher hauptsächlich direkte Datenbankverbindungen oder komplexe Middleware-Lösungen zum Einsatz kamen, ermöglichen moderne Technologien wie REST-APIs und native HTTP-Unterstützung in Datenbanksystemen völlig neue Architekturansätze, siehe den Kasten **Webhook, REST und Co. – alles ist möglich**. Dieser Artikel zeigt praxisnah, wie du mit ASP.NET Core in .NET 10 einen Webhook-Receiver erstellen und diesen direkt aus SQL Server 2025 heraus konsumieren kannst – eine Kombination, die neue Möglichkeiten für Microservices-Architekturen, Datenintegration und hybride Cloud-Szenarien eröffnet.

SSL ist Pflicht!

SQL Server 2025 macht bei HTTP-API-Aufrufen keine halben Sachen: Ohne SSL (also HTTPS) läuft hier schlicht gar nichts. Und das ist kein Selbstzweck, sondern eine bewusst harte Sicherheitslinie. Sobald der SQL Server als Client agiert und Daten über HTTP nach außen schickt – etwa Webhooks, externe Services oder KI-Endpoints –, verlangt er eine verschlüsselte Verbindung. Unverschlüsseltes HTTP wird konsequent abgelehnt und nicht durchgeführt, selbst im vermeintlich harmlosen lokalen Netzwerk. SQL Server 2025 setzt voraus, dass das Ziel einen gültigen SSL-geschützten Endpoint bereitstellt. Das zwingt Entwickler dazu, HTTP-APIs von Anfang an sauber und sicher zu betreiben – inklusive Zertifikaten, korrekter Hostnamen und HTTPS-Konfiguration. Genau deshalb müssen PC und Visual Studio 2026 so eingerichtet sein, dass

● Nicht immer sind Fehlermeldungen hilfreich

Bis alles korrekt aufgesetzt ist, erzeugt SQL Server 2025 zeitweise wenig zutreffende oder nicht hilfreiche Fehler wie „Msg 31608, Level 16, State 29, Procedure sys.sp_invoke_external_rest_endpoint_internal, Line 1 [Batch Start Line 28] An error occurred, failed to communicate with the external rest endpoint. HRESULT: 0x80070008 (ERROR_NOT_ENOUGH_MEMORY).“ Eventuell lohnt sich angesichts dessen ein Blick in das System-Log oder SQL-Server-Log.

● Webhook, REST und Co. – alles ist möglich

Das beschriebene Vorgehen lässt sich ebenso auf andere Arten von HTTP-Aufrufen übertragen. Ob REST, Webhook oder andere Schnittstellenvarianten – das Prinzip bleibt gleich. Webhooks dienen hier lediglich als Beispiel, da sie sich besonders gut als Grundlage für vielfältige weiterführende Ideen eignen.

HTTPS von Anfang an sauber funktioniert. Ein Web-API, das lokal nur auf `http://localhost` hört oder ein Zertifikat verwendet, das ausschließlich auf `localhost` ausgestellt ist, reicht in diesem Szenario nicht mehr aus. SQL Server 2025 prüft den Endpoint strikt: Hostname, Zertifikat und Verschlüsselung müssen zusammenpassen. Läuft das API also unter dem tatsächlichen Rechnernamen, muss auch das SSL-Zertifikat genau auf diesen Namen ausgestellt sein – inklusive gültigem Subject beziehungsweise Subject Alternative Name (SAN).

Der Nebeneffekt ist durchaus positiv: Wer lokal schon echtes SSL nutzt, entdeckt Konfigurationsfehler früh und erspart sich böse Überraschungen beim Deployment. Wer dies nicht beachtet, erhält die in **Bild 1** gezeigte Fehlermeldung.

SQL Server 2025 muss dem Zertifikat vertrauen

Ein weiterer Punkt, der in der Praxis gern übersehen wird, aber entscheidend ist: Verwendet man ein Self-Signed-Zertifikat, muss der SQL Server diesem auch vertrauen. Genauer: der Account, mit dem der SQL-Server-Dienst läuft. Ein selbst signiertes Zertifikat ist technisch vollkommen ausreichend für HTTPS und TLS – es ist jedoch zunächst niemandem bekannt. Während Browser oft noch mit einer Warnung „überredet“ werden können, ist SQL Server 2025 deutlich weniger kooperativ. Erkennt er das Zertifikat nicht als vertrauenswürdig, wird der HTTP-Aufruf schlicht abgelehnt.

Das bedeutet konkret: Das Self-Signed-Zertifikat des lokalen HTTP-API muss im Zertifikatsspeicher des Systems, auf dem der SQL Server läuft, als vertrauenswürdige Stammzertifizierungsstelle hinterlegt sein. Nur dann akzeptiert SQL Server 2025 die SSL-Verbindung ohne Zertifikatsfehler. Gerade in Entwicklungs- und Testumgebungen ist das der Normalfall – aber eben nur dann, wenn man diesen Schritt be-

```
Msg 31610, Level 16, State 1, Procedure sys.sp_invoke_external_rest_endpoint_internal, Line 1 [Batch Start Line 28]
Accessing the external endpoint is only allowed via HTTPS.
```

```
Completion time: 2025-12-26T18:39:07.2596265+01:00
```

Fehler 31610 wird geworfen, wenn ein Aufruf ohne SSL versucht wird (**Bild 1**)

wusst geht. SSL allein genügt nicht, das Vertrauen gehört zwingend dazu. Erst wenn Zertifikat, Hostname und Vertrauenskette sauber zusammenspielen, verhält sich die Entwicklungsumgebung so, wie SQL Server 2025 es erwartet: strikt, leise – und sicher. Und: Der Autor war nur erfolgreich, wenn der SQL-Server-Dienst mit einem Konto (lokal oder aus der Domain) konfiguriert war. *Local System* oder *NT SERVICE\MSSQLSERVER* haben nicht funktioniert, vergleiche auch den Kasten **Nicht immer sind Fehlermeldungen hilfreich**.

Nachdem diesem wichtigen Punkt genügend Tiefe gewidmet wurde, kann es nun mit Webhooks weitergehen.

Was ist ein Webhook?

Ein Webhook ist ein benutzerdefinierter HTTP-Callback, der von einem Server an einen anderen Server gesendet wird, um über ein bestimmtes Ereignis zu informieren [1]. Im Unterschied zu herkömmlichen APIs, bei denen der Client aktiv Anfragen stellt, sendet der Server beim Eintreten des Ereignisses automatisch eine Nachricht an den URL, welcher bei der Einrichtung des Webhooks angegeben wurde. Besonders nützlich ist dies für Echtzeitanwendungen.

Der Webhook-Vertrag als stabiler Umschlag

Der erste echte Entwurfsschritt ist nicht der Endpoint selbst, sondern der Payload-Vertrag. Wir entscheiden uns in **Listing 1** für ein sogenanntes Envelope-Modell. Dieser „Umschlag“ trennt zwei Dinge sauber voneinander: Metadaten und fachliche Nutzlast. Das ist kein Luxus, sondern eine praktische Entscheidung für Wartbarkeit.

Die *EventId* existiert nicht primär für Logs oder Debugging. Ihre wichtigste Aufgabe ist Idempotenz. Sobald ein Webhook mehr als einmal ankommt – und das wird er –, brauchst du eine Möglichkeit, dies zu erkennen und zu sagen: Dieses Ereignis habe ich bereits gesehen. Eine andere Möglichkeit ist eine Kombination von zum Beispiel *StatusCode* nach Abschluss und dem Body, wofür **Listing 4** ein Beispiel zeigt.

● Listing 1: Webhook-Vertrag als record

```
public sealed record WebhookEnvelope(
    string EventId,
    string EventType,
    DateTimeOffset OccurredUtc,
    JsonObject Data,
    string? TraceId = null
);
```

EventType beschreibt nicht, wie etwas verarbeitet wird, sondern was passiert ist. Das ist ein feiner, aber wichtiger Unterschied. Ein guter Eventtyp ist eine Aussage, kein generischer Befehl. SQL-Server-bezogene Beispiele können sein: *BackupCompleted*, *SqlJobFailed*, *IndexRebuildCompleted* et cetera.

Der Zeitstempel *OccurredUtc* macht klar, dass es hier um den Zeitpunkt im sendenden System geht, nicht um den Empfang. Das ist später entscheidend, wenn du Ereignisse sortierst, vergleichst oder erneut abspielst.

Data bleibt bewusst generisch. Das erlaubt dir, den Webhook stabil zu halten, während sich die fachlichen Details weiterentwickeln.

Idempotenz (Idempotency)

Idempotenz oder Idempotency bedeutet, dass dieselbe Aktion mehrfach ausgeführt werden kann, ohne das Ergebnis zu verändern [2]. Übertragen auf Webhooks heißt das: Derselbe Event darf mehrmals beim Empfänger ankommen, aber nur einmal Wirkung entfalten. Genau das ist entscheidend, weil Webhooks praktisch immer nach dem Prinzip „at least once“ zugestellt werden. Time-outs, Netzwerkprobleme oder Retries auf Senderseite führen ganz regulär dazu, dass ein identischer Request erneut gesendet wird. Ohne Idempotenz entstehen dann doppelte Buchungen, doppelte Benachrichtigungen oder mehrfach gestartete Folgeprozesse. Für Webhooks ist Idempotenz deshalb kein Nice-to-have, sondern eine Grundvoraussetzung für korrekte Systeme: Der Empfänger muss Wiederholungen erkennen und sicher ignorieren können, meist anhand einer stabilen Event-ID oder eines zusammengesetzten ebenso stabilen Wertes, welche jeweils nur einmal verarbeitet werden. Eine Kombination aus *EventType* und *Data* ist an dieser Stelle eine valide Möglichkeit. **Listing 2** zeigt eine mögliche Implementierung für einen *Idempotency-Service*. Der konkrete Einsatz ist dann in **Listing 4** zu sehen. Dort ist ein Controller mit Action implementiert.

Aber Vorsicht: Bei verteilten ASP.NET-Core-Endpoints ist ein Memory-Cache ungeeignet, da jeder Serverknoten seinen eigenen Speicher hat und Cache-Einträge nicht automatisch geteilt werden. Das führt dazu, dass identische Requests je nach Instanz unterschiedliche Zustände sehen – Konsistenz und Idempotenz gehen dabei verloren. Die naheliegende Alternative ist ein zentraler, verteilter Cache, auf den alle ASP.NET-Core-Instanzen gemeinsam zugreifen können. Typische Lösungen sind Redis oder eine SQL-Server-basierte Speicherung, bei der Locks und Idempotency-Informationen zentral gehalten werden. Dadurch sehen alle Knoten denselben Zustand, Duplikate lassen sich zuverlässig erkennen und horizontales Skalieren bleibt möglich, ohne die Konsistenz zu opfern. ►

Listing 2: Ein IdempotencyService mit Memory-Cache als Kurzzeit-Persistenz

```
public interface IIdempotencyService
{
    /// <summary>
    /// Versucht, für den angegebenen Idempotency-Key
    /// eine Verarbeitungssperre zu erhalten.
    /// Rückgabewerte:
    /// - CompletedResult != null: Anfrage wurde bereits
    /// abgeschlossen -> diese Antwort zurückgeben
    /// - AcquiredLock = false: Anfrage wird aktuell
    /// verarbeitet -> 202 zurückgeben
    /// - AcquiredLock = true: Aufrufer darf verarbeiten
    /// und muss anschließend CompleteAsync aufrufen
    /// </summary>
    IdempotencyCheckResult CheckAndAcquire(string key,
        TimeSpan inProgressTtl);

    /// <summary>
    /// Speichert die finale Antwort für Duplikate und
    /// gibt die laufende Verarbeitungssperre frei.
    /// </summary>
    void Complete(string key, IdempotentResponse
        response, TimeSpan completedTtl);

    /// <summary>
    /// Gibt die laufende Verarbeitungssperre frei, ohne
    /// die Verarbeitung abzuschließen
    /// (z.B. bei einem fatalen Fehler).
    /// In der Regel sollte stattdessen eine kurzlebige
    /// Fehlantwort gespeichert werden.
    /// </summary>
    void Release(string key);
}

public sealed class IdempotencyService(
    IMemoryCache cache) : IIdempotencyService
{
    private readonly IMemoryCache _cache = cache;

    public IdempotencyCheckResult CheckAndAcquire(
        string eventId, TimeSpan inProgressTtl)
    {
        var doneKey = DoneKey(eventId);
        if (_cache.TryGetValue(doneKey, out
            IdempotentResponse? completed))
        {
            return IdempotencyCheckResult.Completed(
                completed!);
        }

        // Erwirbt atomar eine Verarbeitungssperre.
        // Vermeidet das Race-Condition-Problem von
        // "TryGetValue dann Set" durch die Verwendung
        // von GetOrCreate.
        var lockKey = LockKey(eventId);
        var acquired = false;

        _cache.GetOrCreate(lockKey, entry =>
        {
            acquired = true;
            entry.AbsoluteExpirationRelativeToNow =

```

HMAC (Hash-based Message Authentication Code)

HMAC ist ein kryptografisches Verfahren, bei dem ein geheimer Schlüssel und der Nachrichteninhalt zu einer Prüfsumme kombiniert werden, um Herkunft und Unveränderbarkeit einer Nachricht zu verifizieren [3]. Der Empfänger kann mit demselben Schlüssel prüfen, ob die Nachricht authentisch ist und unterwegs nicht manipuliert wurde.

Warum also verwenden Webhooks zusätzlich häufig HMAC? Der Grund liegt weniger in der Verschlüsselung als in der Nachweisbarkeit der Herkunft. Ein HMAC erlaubt dem Empfänger, zu prüfen, ob der Request tatsächlich vom erwarteten Absender stammt und ob die Payload unterwegs manipuliert wurde. Das ist besonders wichtig bei Webhooks, da sie asynchron und oft öffentlich erreichbar sind. SSL schützt die Transportstrecke, aber es sagt nichts darüber aus,

Listing 3: Middleware zur Prüfung des API-Keys

```
public class ApiKeyMiddleware(RequestDelegate next)
{
    public async Task Invoke(HttpContext context,
        WebhookDbContext dbContext)
    {
        if (!context.Request.Headers.TryGetValue(
            "x-api-key", out var extractedApiKey))
        {
            context.Response.StatusCode = 401; // Unauthorised
            await context.Response.WriteAsync(
                "API Key was not provided.");
            return;
        }
    }
}
```

```

        inProgressTtl;
        return true;
    });

    if (!acquired)
    {
        return IdempotencyCheckResult.InProgress();
    }

    return IdempotencyCheckResult.Acquired();
}

public void Complete(string key, IdempotentResponse
    response, TimeSpan completedTtl)
{
    // Finale Antwort speichern
    _cache.Set(DoneKey(key), response, completedTtl);

    // Gibt die Verarbeitungssperre frei, sodass der
    // "done"-Eintrag künftige Duplikate steuert
    _cache.Remove(LockKey(key));
}

public void Release(string key)
{
    _cache.Remove(LockKey(key));
}

private static string DoneKey(string eventId) =>
    $"idem:done:{eventId}";

private static string LockKey(string eventId) =>
    $"idem:lock:{eventId}";
}

/// <summary>
/// Die Antwort, die für Duplikate wiederholt werden
/// soll.
/// Bewusst einfach gehalten: Statuscode + JSON-Body.
/// </summary>
public sealed record IdempotentResponse(
    string EventId);

public sealed record IdempotencyCheckResult(
    bool AcquiredLock, bool IsInProgress,
    IdempotentResponse? CompletedResult)
{
    public static IdempotencyCheckResult Completed(
        IdempotentResponse response)
        => new(AcquiredLock: false, IsInProgress: false,
            CompletedResult: response);

    public static IdempotencyCheckResult InProgress()
        => new(AcquiredLock: false, IsInProgress: true,
            CompletedResult: null);

    public static IdempotencyCheckResult Acquired()
        => new(AcquiredLock: true, IsInProgress: false,
            CompletedResult: null);
}

```

wer den Request ausgelöst hat, sobald er korrekt über HTTPS angekommen ist. HMAC ergänzt SSL also um eine zweite Sicherheitsebene auf Nachrichtenebene.

Zwingend notwendig ist HMAC aus technischer Sicht nicht – dank SSL sind die Daten bereits verschlüsselt und vor Manipulation während der Übertragung geschützt. In kontrollierten Szenarien, etwa bei internen Systemen oder klar abgegrenzten Trust-Zonen, kann man bewusst darauf ver-

zichten. In der Praxis wird HMAC dennoch gern eingesetzt, weil es unabhängig vom Transportprotokoll funktioniert, Replay-Angriffe erschwert und auch dann greift, wenn Requests weitergeleitet oder zwischengespeichert werden. Kurz gesagt: SSL sichert den Kanal, HMAC sichert die Botschaft. Beides zusammen ist robust – eines von beiden kann, je nach Kontext, ausreichend sein. Aus diesem Grund wird das Thema im Folgenden ausgeklammert. ►

```

var dbApiKey =
    await dbContext.ApiKeys.FirstOrDefaultAsync(
        x => x.Key == extractedApiKey);
bool isValidApiKey = dbApiKey != null;

if (!isValidApiKey)
{
    context.Response.StatusCode = 401; // Unauthorised
    await context.Response.WriteAsync(
        "Unauthorized client.");
    return;
}
await next(context);
}

```


● Listing 4: Controller für den Webhook

```
[ApiController]
[Route("api/webhook")]
public class WebhookController(
    IWebhookProcessor webhookProcessor,
    IIdempotencyService idempotencyService) :
    ControllerBase
{
    [HttpPost("sqlserver")]
    public async Task<IActionResult>
        SqlServerWebHook(
            WebhookEnvelope webhookEnvelope,
            CancellationToken cancellationToken = default)
    {
        string eventId = $"{webhookEnvelope.EventType}:
            {webhookEnvelope.Data}".GetHashCode().ToString();

        var checkResult = idempotencyService.
            CheckAndAcquire(eventId, TimeSpan.FromSeconds(30)
                // Lock-Dauer für "in progress"
            );

        // Fall 1: Request wurde bereits erfolgreich
        // verarbeitet
        if (checkResult.CompletedResult != null)
        {
            return Ok(checkResult.CompletedResult);
        }

        // Fall 2: Request wird gerade von einem anderen
        // Request verarbeitet
        if (checkResult.IsInProgress)
        {
            return StatusCode(202,
                "Request is currently being processed");
        }

        // Fall 3: Neuer Request - wir haben den Lock
        // erworben. Auf gehts.
        if (checkResult.AcquiredLock)
        {
            try
            {
                // Deine Business-Logik hier, z.B. Daten
                // in DB speichern, externes API aufrufen,
                // etc.
                await webhookProcessor.ProcessAsync(
                    webhookEnvelope, cancellationToken);

                // Verarbeitung speichern
                var response =
                    new IdempotentResponse(eventId);
                idempotencyService.Complete(
                    eventId,
                    response,
                    TimeSpan.FromHours(24) // Wie lange soll die
                    // Antwort gespeichert werden?
                );

                return Ok(response);
            }
            catch
            {
                // Bei Fehler: Lock freigeben, damit ein Retry
                // möglich ist
                idempotencyService.Release(
                    webhookEnvelope.ToString());

                // Alternative: Fehler-Response speichern
                // var errorResponse = new IdempotentResponse(
                //     webhookEnvelope.EventId);
                // idempotencyService.Complete(webhookEnvelope.
                //     EventId, errorResponse,
                //     TimeSpan.FromMinutes(5));

                return StatusCode(500, "Processing failed");
            }
        }

        return StatusCode(
            500, "Unexpected idempotency state");
    }
}
```

API-Key

Ein API-Key ist eine einfache Möglichkeit, einen Aufrufer eindeutig zu identifizieren und ungewollte Zugriffe zumindest grob zu begrenzen. Er eignet sich gut für einfache Autorisierung, Rate-Limiting oder Monitoring und lässt sich leicht implementieren. Zwingend erforderlich ist er jedoch nicht, wenn die Kommunikation ausschließlich in abgeschlossenen Systemen stattfindet und die Gegenstelle bereits über andere Mechanismen wie Netzwerkgrenzen, Zertifikatsvertrauen oder signierte Payloads (HMAC) abgesichert ist.

In vielen Szenarien ist der API-Key daher weniger eine Sicherheitsmaßnahme als vielmehr eine zusätzliche Kontroll- und Ordnungshilfe. [Listing 3](#) enthält beispielhaft eine einfache Implementierung.

Wichtig ist zudem, dass die Middleware in *Program.cs* mittels *app.UseMiddleware<ApiKeyMiddleware>()* in der HTTP-Request-Pipeline (recht weit am Anfang) registriert werden muss. Im Code der Actions muss das nicht mehr berücksichtigt werden – nur wenn der API-Key korrekt vorliegt, kommt es überhaupt zu einer Ausführung.

Webhook-Implementierung

Die Implementierung in Listing 4 zeigt die vollständige Logik gebündelt in einem Controller, um alle relevanten Aspekte an einer Stelle nachvollziehbar zu halten – konzeptionell lässt sich derselbe Ansatz jedoch ebenso gut mit Minimal APIs umsetzen. Der Endpoint ist bewusst als *POST* ausgeführt, da Webhooks und ähnliche Callback-Szenarien immer einen Request-Body mit Ereignisdaten übertragen und dabei einen serverseitigen Zustand verändern. *GET* zum Beispiel wäre semantisch falsch, da der Aufruf (sensibler) Daten nicht über den URL, sondern ausschließlich über den Request-Body transportiert werden soll. Aber natürlich sind auch HTTP-Verben möglich.

SQL Server, der Aufrufer

In vielen Architekturen ist SQL Server 2025 nicht nur Datenhaltung, sondern aktiver Auslöser von Prozessen. Abschlüsse von Jobs, erfolgreiche Batchläufe, fehlgeschlagene Wartungen oder fachliche Statuswechsel entstehen häufig direkt dort, wo die Daten liegen. Statt diese Ereignisse periodisch „abzufragen“, ist es oft sinnvoller, sie ereignisgetrieben nach außen zu melden. Genau an dieser Stelle tritt SQL Server 2025 als Aufrufer auf: Er erkennt, dass etwas Relevantes passiert ist oder Daten benötigt werden, formt daraus einen Request und ruft das .NET-HTTP-API an. Damit wird die Datenbank vom passiven Speicher zum aktiven Teilnehmer in einer entkoppelten, modernen Integrationsarchitektur.

Credentials

Bei HTTP-API-Aufrufen werden Credentials in der Regel nicht im Body oder in der Query, sondern als HTTP-Header übertragen, damit sie unabhängig von der eigentlichen Nutzlast ausgewertet werden können. Typische Varianten sind Autorisierungs-Header wie *Authorization* (zum Beispiel mit Tokens) oder spezialisierte Header für Signaturen, Zeitstempel oder API-Keys.

Der Server prüft diese Header vor der fachlichen Verarbeitung des Requests und entscheidet bereits auf Transport- beziehungsweise Protokollebene, ob der Aufrufer legitim ist. Dadurch bleibt die eigentliche API-Logik sauber von Sicherheitsaspekten getrennt, und wiederholbare, standardisierte Sicherheitsmechanismen werden ermöglicht.

Wenn SQL Server als Aufrufer eines HTTP-API agiert, werden Credentials nicht „automatisch“ verwaltet, sondern explizit vom aufrufenden Mechanismus mitgegeben. Das bedeutet, dass SQL Server die benötigten Zugangsdaten (etwa den API-Key) erzeugt oder selbst vorhält und sie beim HTTP-Aufruf gezielt als HTTP-Header übergibt. Die Credentials werden dabei nicht als Klartext abgelegt, sondern verschlüsselt gespeichert.

Hierzu ist eine Master Key Encryption notwendig, weil sie die Wurzel des Vertrauens für alle weiteren geheimen Informationen im SQL Server bildet. Diese wiederum ver- und entschlüsselt Credentials, Zertifikate, symmetrische Schlüssel oder Secrets, die der SQL Server für externe Aufrufe benötigt. Ohne einen Schutz wie diesen müssten die sensiblen Daten entweder im Klartext gespeichert oder selbst verschlüsselt wer-

den. Die Master Key Encryption stellt damit sicher, dass Secrets verschlüsselt persistiert, automatisch nutzbar und gleichzeitig gegen unbefugten Zugriff geschützt sind – selbst wenn jemand direkten Zugriff auf die Datenbankdateien erhält.

Der Name der Credentials ist dabei der URL, für den sie benötigt werden. Dieser muss dabei nicht vollständig sein. So reicht zum Beispiel *https://LaptopK:7013* für alle URLs, die so beginnen, oder sie können detaillierter sein wie zum Beispiel *https://LaptopK:7013/api/webhook*. Hostname, Protokoll und gegebenenfalls Port müssen aber immer dabei sein, je nachdem, wie genau man sein möchte oder muss. Listing 5 zeigt, wie die Credentials für einen *x-api-key*-Header erzeugt und bei Bedarf geändert werden, wenn sich das „Secret“ ändert. Gut zu erkennen sind der Name und der Wert des Headers.

Aufruf mittels `sys.sp_invoke_external_rest_endpoint`

Den eigentlichen Aufruf erledigt dann die System Stored Procedure *sys.sp_invoke_external_rest_endpoint* [4]. Deren Parameter sind die Methode (HTTP-Verb), der URL, Credential, Payload (Body), Time-out, Retry Count und weitere.

Die Rückgabe (*@returnCode*) ist dann ein Hinweis, ob der Aufruf erfolgreich war, oder nicht. Sie ist eine 0, wenn der HTTPS-Aufruf erfolgreich durchgeführt wurde und der empfangene HTTP-Statuscode im 2xx-Bereich (Erfolg) liegt. Liegt der empfangene HTTP-Statuscode nicht im 2xx-Bereich, wird dieser HTTP-Statuscode als Rückgabewert geliefert. Kann der HTTPS-Aufruf gar nicht ausgeführt werden, wird eine Ausnahme ausgelöst.

Die eigentliche Rückgabe (*@response*), also die Antwort des aufgerufenen API, ist dann via *OUTPUT*-Parameter in Form von JSON oder alternativ als XML verfügbar. Listing 6 enthält beispielhaft einen solchen Aufruf.

Rückgabe

Die Rückgabe (*@response* in Listing 6) enthält dann strukturiert alle Informationen: vom ReturnCode über den Rückgabe-Header bis zur Rückgabe selbst. Listing 7 zeigt eine Rückgabe im JSON-Format; möglich wäre aber auch XML, ►

Listing 5: DMK und Credential für HTTP-API-Aufruf

```
CREATE MASTER KEY ENCRYPTION BY PASSWORD =
    N'$91yjYxqIWWpM';
GO
CREATE DATABASE SCOPED CREDENTIAL [https://LaptopK:
    7013/api]
    WITH IDENTITY = 'HTTPEndpointHeaders',
    SECRET = '{"x-api-key":"123"}';
GO
ALTER DATABASE SCOPED CREDENTIAL [https://LaptopK:
    7013/api]
    WITH IDENTITY = 'HTTPEndpointHeaders',
    SECRET = '{"x-api-key":"9hYMyw"}';
```

Listing 6: sys.sp_invoke_external_rest_endpoint in Aktion

```

DECLARE @EventId UNIQUEIDENTIFIER = NEWID();
DECLARE @EventType NVARCHAR(128) = 'BackupCompleted';
DECLARE @OccurredUtc DATETIMEOFFSET(0) =
    SYSUTCDATETIME();
DECLARE @TraceId VARCHAR(128) =
    CONCAT(@EventType, '|', NEWID());

-- Data-Teil als json
DECLARE @Data JSON =
(
    SELECT
        'SampleDB'          [DatabaseName],
        104857600           [BackupSizeInBytes],
        128                 [BackupDurationInSeconds],
        SYSUTCDATETIME()    [BackupCompletionTime],
        'F:\SampleDB.bak'   [BackupFilePath]
    FOR JSON PATH, WITHOUT_ARRAY_WRAPPER
);

-- Envelope als json
DECLARE @Envelope JSON =
(
    SELECT
        CONVERT(nvarchar(36), @EventId) [EventId],
        @EventType                      [EventType],
        @OccurredUtc                    [OccurredUtc],
        @TraceId                        [TraceId],
        JSON_QUERY(CAST(
            @Data AS NVARCHAR(MAX)))    [Data]
    FOR JSON PATH, WITHOUT_ARRAY_WRAPPER
);

SELECT @Envelope
DECLARE @Envelope NVARCHAR NVARCHAR(MAX) =
    CAST(@Envelope AS NVARCHAR(MAX));

-- GRANT EXECUTE ANY EXTERNAL ENDPOINT TO {someone};
DECLARE @returnCode INT;
DECLARE @response NVARCHAR(MAX);

EXEC @returnCode =
    sys.sp_invoke_external_rest_endpoint
    @url =
        N'https://LaptopK:7013/api/webhook/sqlserver',
    @credential = [https://LaptopK:7013/api],
    @method = 'POST',
    @payload = @Envelope NVARCHAR,
    @timeout = 20,
    @retry_count = 3,
    @response = @response OUTPUT;

```

wenn das HTTP-API dieses sendet. Wichtig ist, dass es eine Rückgabe gibt, auch wenn sie nichts Essenzielles enthält.

Fazit

SQL Server 2025 zeigt mit seinen Möglichkeiten zum Zugriff auf HTTP-APIs, dass eine moderne Datenbank längst mehr ist als ein passiver Datenspeicher. Ereignisse entstehen dort, wo Daten verarbeitet werden, und können gezielt und sicher nach

außen kommuniziert werden. Durch den Einsatz von HTTPS, klar definierte Credentials, saubere Schlüsselverwaltung und strukturierte JSON-Payloads lassen sich HTTP-Aufrufe kontrolliert und nachvollziehbar integrieren. Richtig eingesetzt bleibt SQL Server die bisherige verlässliche, stabile Plattform – jedoch erweitert um die Fähigkeit, externe Systeme ereignisgetrieben anzustoßen, statt sie nur periodisch abzufragen. ■

- [1] Webhook, <https://en.wikipedia.org/wiki/Webhook>
- [2] Idempotenz, <https://de.wikipedia.org/wiki/Idempotenz>
- [3] HMAC, <https://en.wikipedia.org/wiki/HMAC>
- [4] Microsoft Learn, `sp_invoke_external_rest_endpoint` (Transact-SQL), <https://learn.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sp-invoke-external-rest-endpoint-transact-sql?view=sql-server-ver17&tabs=request-headers>

Listing 7: Rückgabe im JSON-Format

```

{
  "response": {
    "status": {
      "http": {
        "code": "",
        "description": ""
      }
    },
    "headers": {}
  },
  "result": {}
}

```



Thorsten Kansy

ist seit über 25 Jahren als Softwarearchitekt, Entwickler und Trainer international tätig. Er spricht regelmäßig auf Konferenzen und schreibt Bücher und Artikel rund um die Entwicklung komplexer Anwendungen mit .NET im Microsoft-Serverumfeld.

KÜNSTLICHE INTELLIGENZ UND SQL SERVER 2025

Vektoren nativ speichern und für RAG-Szenarien nutzen

Durch die native Unterstützung von Embeddings und Ähnlichkeitssuchen wird der SQL Server RAG-fähig.

Das Thema Künstliche Intelligenz hat sich in Microsofts Entwicklungswerkzeugen in kurzer Zeit von einem Zusatzfeature zu einem festen Bestandteil moderner Softwarearchitekturen entwickelt. Während frühe Integrationen vor allem auf Code-Vervollständigung zielten, verschiebt sich der Fokus seit rund 18 Monaten deutlich hin zu agentischen Systemen, Retrieval Augmented Generation (RAG) und tief integrierten KI-Workflows – sowohl in kommerziellen Produkten als auch im Open-Source-Umfeld.

wie SDKs, die große Sprachmodelle systematisch in Anwendungen integrieren. Parallel dazu entwickelt die .NET-Community mit Projekten wie Semantic Kernel [3], Semantic Flow [4], OllamaSharp [5] und LangChain.NET [6] neue Architekturmuster für lokale und cloudbasierte LLM-Szenarien.

Mit Microsoft SQL Server 2025 [7] und Entity Framework Core 10 [8] erreichen KI-Konzepte nun auch die Datenhaltung: Vektordatentypen, Vektorsuche und REST-Anbindungen externer Modelle verbinden relationale Datenbanken direkt mit modernen KI-Workloads. Ergänzt durch .NET AI Essentials [9] entsteht eine einheitliche Abstraktion für KI-Provider und Antwortmodelle.

Der SQL Server 2025 ist damit weniger ein isolierter Datenbank-Release als ein zentraler Baustein einer durchgängigen KI-Strategie. Die Integration externer LLMs und von Vector Search bildet den Schwerpunkt der folgenden Abschnitte.

Architekturüberblick

Die Architektur zeigt einen pragmatischen Ansatz, um in einer Entwicklungsumgebung eine vektorbasierte Suche und RAG-Szenarien mit bestehenden Microsoft-Technologien umzusetzen und zu testen. Im Zentrum steht der Microsoft SQL Server 2025, der erstmals native Unterstützung für Vektordatentypen und Ähnlichkeitssuchen bietet und damit eine neue Rolle in KI-gestützten Anwendungen einnimmt.

Der SQL Server fungiert dabei nicht als KI-Modell, sondern als persistenter semantischer Speicher. Embeddings werden außerhalb der Datenbank erzeugt und als Vektoren gespeichert, die anschließend für Approximate-Nearest-Neighbor-Suchen (ANN) direkt in T-SQL genutzt werden können – kombiniert mit klassischen relationalen Abfragen.

Für die Embedding-Erzeugung kommt Ollama als lokaler Modellserver zum Einsatz, angebunden über eine REST-Schnittstelle. Die Integration erfolgt über die Systemprozedur `sp_invoke_external_rest_endpoint`, die REST-Aufrufe direkt aus T-SQL ermöglicht. SQL Server Management Studio 21 dient zur Analyse von Vektorspalten und kNN-Ergebnissen (k-Nearest Neighbor).

Diese Architektur bildet eine kompakte Grundlage für Retrieval-Augmented-Generation-Szenarien, bei denen der SQL Server 2025 semantisch relevante Daten liefert und die Textgenerierung außerhalb der Datenbank erfolgt. ►

TSQL25AGA.... - dbo.Books			
	Column Name	Data Type	Allow Nulls
🔑	ID	int	☐
	ISBN	nvarchar(13)	☐
	Author	nvarchar(50)	☐
	Title	nvarchar(250)	☐
	Description	nvarchar(MAX)	☐
	Pages	int	☐
	Price	float	☐
	Publisher	nvarchar(100)	☐
	PublicationYear	int	☐
	Category	nvarchar(50)	☐
	Keywords	nvarchar(MAX)	☐
	CreatedAt	datetime	☐
	UpdatedAt	datetime	☐
	IsActive	bit	☐
▶	Embedding	vector(768)	☒

Neu in Microsofts SQL Server Management Studio 21: die Unterstützung von Vektordatentypen (Bild 1)

Auf Basis von GitHub Copilot [1] hat Microsoft KI-Funktionalitäten schrittweise in IDEs und Entwicklungsprozesse eingebettet. Darauf aufbauend entstanden agentische Erweiterungen [2], strukturierte Prompt- und Workflow-Ansätze so-

Umsetzung im Microsoft SQL Server 2025 – Fokus auf Vektordaten

Der zentrale Baustein der Architektur ist die Fähigkeit des Microsoft SQL Server 2025, Vektoren nativ zu speichern und zu verarbeiten. Damit wird er erstmals zu einem vollwertigen Bestandteil semantischer Such- und RAG-Szenarien, ohne seine Rolle als relationale Datenbank zu verlieren.

Zur Demonstration nutzen wir eine Tabelle eines fiktiven Buchverlags mit den Spalten *ID*, *Author*, *Title* und *Description*. Für die semantische Suche ergänzen wir eine Spalte *Embedding*, in der die erzeugten Vektoren gespeichert werden. Da das Modell „nomic-embed-text“ Embeddings mit 768 Dimensionen liefert, wird die Spalte als *vector(768)* angelegt (Bild 1). Dies erfordert Microsofts SQL Server Management Studio 21, das erstmals Vektortypen unterstützt.

Ein Embedding projiziert einen Text in einen hochdimensionalen numerischen Raum, in dem semantisch ähnliche Inhalte nahe beieinanderliegen [10]. SQL Server speichert diese Vektoren dauerhaft und ermöglicht darauf aufbauend Ähnlichkeitssuchen, die sich mit klassischen relationalen Kriterien kombinieren lassen.

● Installation von Caddy

Um die HTTPS-Anforderung von SQL Server 2025 zu erfüllen, dient Caddy als Reverse Proxy. Zur Installation laden Sie die Caddy-Installationsdatei unter [15] herunter und installieren die Anwendung, zum Beispiel im Ordner *C:\Caddy*. Im gleichen Ordner legen Sie eine Datei *caddyfile* (ohne Dateiergung *.txt*) mit folgendem Inhalt an:

```
https://localhost:443 {
    reverse_proxy localhost:11434
    tls internal
}
```

Die Konfigurationsbefehle im Einzelnen erläutert:

- **https://localhost:443** – Caddy lauscht auf Port 443 und nimmt eingehende HTTPS-Anfragen entgegen.
- **reverse_proxy localhost:11434** – Alle Anfragen werden an das lokale Ollama-API weitergeleitet.
- **tls internal** – Caddy erzeugt automatisch ein internes, selbstsigniertes Zertifikat für localhost und fungiert als eigene Zertifizierungsstelle (CA). Dadurch kann SQL Server eine verschlüsselte Verbindung herstellen.

Das von Caddy erzeugte Root-Zertifikat muss noch in den Trusted-Root-Certification-Authorities-Speicher des Windows-Servers, auf dem der SQL Server läuft, importiert werden. Ohne diesen Schritt bleibt die Verbindung „untrusted“.

Gestartet wird Caddy schließlich mit folgenden Befehlen:

```
cd C:\Caddy
caddy run
```

Im nächsten Schritt wird die *Embedding*-Spalte per Stored Procedure befüllt. Dafür muss zuvor Ollama auf dem gleichen Rechner installiert werden, um die Vektoren über dessen lokales REST-API zu erzeugen.

Einrichten von Ollama

Ollama [11] ist ein lokal betreibbarer Open-Source-Modellserver, mit dem sich Large Language Models (LLMs) und Embedding-Modelle direkt auf dem eigenen Rechner ausführen lassen. Ziel ist es, KI-Modelle ohne Cloud-Abhängigkeit über ein einfaches REST-API bereitzustellen. Anwendungen – einschließlich des Microsoft SQL Server – greifen per HTTP/HTTPS auf Modelle zu, ohne sich um deren Ausführung kümmern zu müssen. Ollama eignet sich besonders für Entwicklungs-, Test- und On-Prem-Szenarien sowie für datenschutz-sensible Umgebungen.

Die Installation ist bewusst minimal gehalten. Ollama steht für Windows, macOS und Linux bereit und wird als einzelnes Setup beziehungsweise Paket installiert. Danach läuft es als lokaler Dienst und stellt standardmäßig einen Endpoint unter *http://localhost:11434* bereit; zusätzliche Konfiguration ist für einfache Szenarien nicht nötig.

Modelle werden über einen Pull-Mechanismus verwaltet. Für Text-Embeddings wird häufig das Modell „nomic-embed-text“ genutzt, das speziell für semantische Vektorrepräsentationen optimiert ist. Die Installation erfolgt mit einem einzigen Kommando:

```
ollama pull nomie-embed-text
```

Ollama lädt das Modell herunter und stellt es anschließend über das REST-API bereit. Installierte Modelle lassen sich per *ollama list* oder über *http://localhost:11434/api/tags* anzeigen. Insgesamt stehen derzeit mehr als 250 Modelle zur Verfügung [12].

Zugriff vom SQL-Server auf das Rest-API von Ollama

Mit der neuen Stored Procedure *sp_invoke_external_rest_endpoint* in Microsoft SQL Server 2025 lässt sich von T-SQL aus auf ein externes REST-API zugreifen, allerdings nur über das Protokoll HTTPS, da der Microsoft SQL Server 2025 aus Sicherheitsgründen zwingend verschlüsselte HTTPS-Verbindungen zu REST-APIs verlangt. Die Prozedur ist standardmäßig deaktiviert und muss explizit freigeschaltet werden.

Ollama auf der anderen Seite stellt sein REST-API jedoch nur über HTTP-Endpoints zur Verfügung. Dieses Problem lässt sich durch die Installation eines kleinen Proxys lösen. Es empfiehlt sich, den Proxy Caddy zu verwenden, da dieser automatisch ein selbstsigniertes Zertifikat für localhost erstellt und somit als Reverse Proxy dienen kann (siehe Kasten *Installation von Caddy*).

Automatisiertes Erstellen von Embeddings/Vektoren in der Books-Tabelle

Nachdem die Tabelle *Books* um die Spalte *Embedding vector(768)* erweitert wurde, kann der SQL Server 2025 über den

Listing 1: Stored Procedure GetOllamaEmbeddingProc

```
CREATE PROCEDURE [dbo].[GetOllamaEmbeddingProc]
(
    @Text NVARCHAR(MAX),
    @Model NVARCHAR(100) = N'nomi-embed-text',
    @EmbeddingOut NVARCHAR(MAX) OUTPUT
)
AS
BEGIN
    DECLARE @url NVARCHAR(2000) =
        N'https://localhost:443/api/embeddings';
    DECLARE @body NVARCHAR(MAX) =
        N'{"model":"' + @Model +
        '","prompt":"' +
        REPLACE(@Text, '","', '\\"') + '"}';

    DECLARE @response NVARCHAR(MAX);
    EXEC sp_invoke_external_rest_endpoint
        @url = @url,
        @method = N'POST',
        @headers = N'{
            "Content-Type":"application/json"}',
        @payload = @body,
        @response = @response OUTPUT;

    SELECT @EmbeddingOut = CAST(JSON_QUERY(
        @response, '$.result.embedding')
        AS NVARCHAR(MAX));
END;
```

Caddy-Proxy auf das Ollama-API zugreifen. Dazu wird wie in Listing 1 gezeigt zunächst eine Stored Procedure erstellt, die Ollama aufruft und für einen übergebenen Text ein Embedding zurückgibt. *GetOllamaEmbeddingProc* nimmt anschließend den Eingabetext sowie einen optionalen (Embedding-) Modellnamen entgegen, bereitet diese Daten als strukturiertes JSON-Objekt auf, erzeugt eine JSON-Payload, maskiert Sonderzeichen und sendet per *sp_invoke_external_rest_endpoint* schließlich eine HTTPS-POST-Anfrage an die lokale Ollama-Instanz.

Der Aufruf für unsere Books-Tabelle würde dann folgendermaßen aussehen:

```
EXEC [dbo].[UpdateTableWithEmbeddingsUniversal]
    @TableName = 'Books',
    @IdColumnName = 'ID',
    @ContentColumnName = 'Description';
```

Die Prozedur identifiziert alle Datensätze ohne Embedding, durchläuft sie per Cursor und ruft für jede Beschreibung *Get-*

```
SQLQuery3.sql...1-c207r (70)*
1 DECLARE @EmbeddingText NVARCHAR(MAX);
2 EXEC dbo.GetOllamaEmbeddingProc @Text = N'Dies ist nur ein Beispieltext.', @EmbeddingOut = @EmbeddingText OUTPUT;
3
4 print @EmbeddingText;
```

Messages

```
[{"-0.10467354208230972,-0.8220322728157043,-3.2093703746795654,1.7121598720550537,0.33822178840637207,0.11688684672117233,-1.6556130647659302,-0.401134997
```

Completion time: 2026-02-12T14:36:54.4757918+01:00

Aufruf von GetOllamaEmbeddingProc (Bild 2)

Die Antwort wird per *JSON_QUERY* ausgewertet und als Array mit 768 Gleitkommazahlen zurückgegeben (Bild 2). Wir setzen innerhalb der Prozedur bewusst nicht auf *JSON_ARRAY_TO_VECTOR*, da dessen Nutzung einen statisch festgelegten Vektor-Datentyp wie *VECTOR(1536)* erfordert. Durch die Rückgabe als *NVARCHAR(MAX)* bleibt die Implementierung hingegen Modell-agnostisch und kann auch für andere Embedding-Größen oder Metadaten-Extraktionen wieder verwendet werden.

Auf dieser Basis lässt sich die *Embedding*-Spalte automatisch befüllen. Eine zweite Stored Procedure (Listing 2) übernimmt dies generisch für beliebige Tabellen.

OllamaEmbeddingProc auf. Das Ergebnis wird sicher in die Zieltabelle zurückgeschrieben; *QUOTENAME* schützt dabei vor SQL-Injection. Die Lösung eignet sich gut für prototypische, inkrementelle oder kleinere bis mittlere Datenmengen.

Optional kann ein SQL Server Agent regelmäßig prüfen, ob neue Bücher hinzugekommen sind, oder ein Trigger kann nach Änderungen der Description automatisch ein neues Embedding erzeugen.

Vektor-Suche

Nachdem wir die Embedding-Spalte der Tabelle *Books* befüllt haben, fehlt nur noch ein Index auf dieser Spalte. Er ►

wird mit folgendem Befehl erstellt [13]:

```
CREATE VECTOR INDEX
    IX_Books_Embedding
ON Books(Embedding)
WITH (
    METRIC = 'COSINE',
    TYPE = 'DISKANN');
```

Damit ist die Tabelle bereit für Vektorsuchen. Für eine konkrete Anfrage – etwa eine Fragestellung, die ein Benutzer im Chatbot eingibt – erzeugen wir zunächst mit *GetOllamaEmbeddingProc* das entsprechende Embedding. Anschließend suchen wir per Approximate-Nearest-Neighbor-Verfahren (ANN) die ähnlichsten Einträge.

Da das Modell „nomic-embed-text“ speziell für die Kosinus-Ähnlichkeit (Cosine Similarity) optimiert wurde, empfiehlt es sich, dieses Distanzmaß explizit in der Funktion *VECTOR_DISTANCE* zu verwenden, um die höchste semantische Präzision zu erreichen.

Die Wahl des korrekten Distanzmaßes ist hierbei entscheidend: Während ein unpassendes Verfahren wie die euklidische Distanz zwar mathematisch „nahe“ Vektoren liefert, führt oft nur die Cosine Similarity zu semantisch präzisen Suchergebnissen.

```
DECLARE @query
    vector(768) = '...'
    -- Embedding des
    -- Suchtexts
SELECT TOP 5 * FROM
    Books ORDER BY
    VECTOR_DISTANCE(
    'cosine',
    Embedding,
    @query);
```

Die über die Vektorsuche gefundenen Buchbeschreibungen werden anschließend zusammen mit der ursprüngli-

Listing 2: Stored Procedure UpdateTableWithEmbeddingsUniversal

```
CREATE PROCEDURE [dbo].[
    UpdateTableWithEmbeddingsUniversal]
    @TableName NVARCHAR(128), -- Name der Tabelle (z.B. 'Books')
    @IdColumnName NVARCHAR(128), -- Name der ID-Spalte (z.B. 'ID')
    @ContentColumnName NVARCHAR(128), -- Spalte mit dem Quelltext (z.B. 'Description')
    @EmbeddingColumnName NVARCHAR(128) = 'Embedding' -- Zielspalte (Standard: 'Embedding')
AS
BEGIN
    SET NOCOUNT ON;
    DECLARE @CurrentID INT;
    DECLARE @CurrentText NVARCHAR(MAX);
    DECLARE @EmbeddingResult NVARCHAR(MAX);
    DECLARE @SqlFetch NVARCHAR(MAX);
    DECLARE @SqlUpdate NVARCHAR(MAX);

    -- 1. Temporäre Tabelle erstellen, um die Arbeitsliste zu speichern
    -- Wir holen uns nur die Datensätze, bei denen das Embedding noch fehlt (IS NULL)
    IF OBJECT_ID('tempdb..#WorkList') IS NOT NULL DROP TABLE #WorkList;
    CREATE TABLE #WorkList (ID INT, Content NVARCHAR(MAX));
    SET @SqlFetch = N'INSERT INTO #WorkList (ID, Content) ' +
        N'SELECT ' + QUOTENAME(@IdColumnName) + N', ' +
        QUOTENAME(@ContentColumnName) +
        N' FROM ' + QUOTENAME(@TableName) +
        N' WHERE ' + QUOTENAME(@EmbeddingColumnName) + N' IS NULL';
    EXEC sp_executesql @SqlFetch;

    -- 2. Cursor über die temporäre Tabelle
    DECLARE work_cursor CURSOR FOR SELECT ID, Content FROM #WorkList;
    OPEN work_cursor;
    FETCH NEXT FROM work_cursor INTO @CurrentID, @CurrentText;
    WHILE @@FETCH_STATUS = 0
    BEGIN
        IF @CurrentText IS NOT NULL AND LTRIM(@CurrentText) <> ''
        BEGIN

            -- 3. Embedding über Ollama beziehen
            EXEC dbo.GetOllamaEmbeddingProc @Text = @CurrentText,
                @EmbeddingOut = @EmbeddingResult OUTPUT;

            -- 4. Dynamisches Update der Originaltabelle
            SET @SqlUpdate = N'UPDATE ' + QUOTENAME(@TableName) +
                N' SET ' + QUOTENAME(@EmbeddingColumnName) + N' =
                CAST(@emb AS VECTOR(768)) ' +
                N' WHERE ' + QUOTENAME(@IdColumnName) + N' = @id';
            EXEC sp_executesql @SqlUpdate,
                N'@emb NVARCHAR(MAX), @id INT',
                @emb = @EmbeddingResult,
                @id = @CurrentID;

        END
        FETCH NEXT FROM work_cursor INTO @CurrentID, @CurrentText;
    END
    CLOSE work_cursor;
    DEALLOCATE work_cursor;
    DROP TABLE #WorkList;
END;
```

chen Fragestellung und einem geeigneten Systemprompt an ein LLM übergeben, welches daraus die finale Antwort für den Chatbot generiert.

Exkurs

In Entity Framework Core 10 lässt sich diese Vektorsuche elegant über LINQ-Abfragen abbilden: Mit der Methode `EF.Functions.VectorDistance` (unter Angabe von `DistanceFunction.Cosine`) wird die Ähnlichkeitssuche direkt in das entsprechende SQL-Statement übersetzt, sodass Entwickler von der Typsicherheit profitieren, ohne auf die Performance der nativen SQL-Vektorfunktionen verzichten zu müssen. Der Code hierfür würde so aussehen:

```
var nearestBooks = await context.Books
    .OrderBy(b => EF.Functions.VectorDistance(
        "cosine", b.Embedding, queryVector))
    .Take(5)
    .ToListAsync();
```

Was es zu beachten gibt

Für die Darstellung der Vektorfunktionen im SQL Server wurde in diesem Artikel mit der Bücherdatenbank bewusst ein recht einfaches Beispiel gewählt. Beim Aufbau realer RAG-Systeme zeigt sich jedoch schnell, dass semantische Suche allein nicht ausreicht. Entscheidend für belastbare Ergebnisse sind darüber hinaus konsistente Embeddings, ein passendes Distanzmaß und eine sinnvolle Chunking-Strategie, die weder Kontext zerstückelt noch irrelevante Textmengen transportiert. Ebenso wichtig sind Metadaten-Filter, um thematisch ähnliche Inhalte sauber zu trennen, sowie ein verlässlicher Aktualisierungsprozess für Embeddings und die Wahl eines geeigneten Index.

Da selbst hochwertige Treffer Halluzinationen nicht vollständig verhindern, braucht es einen guten Systemprompt, klare Prompt-Regeln und Guardrails, die das LLM strikt an den gelieferten Kontext binden.

Ergänzend spielen Sicherheitsaspekte wie der Schutz persönlicher und vertraulicher Daten, Maßnahmen gegen Prompt Injection und Zugriffskontrollen eine Rolle, ebenso wie kontinuierliches Monitoring, um Qualitätsverluste durch semantische Drift frühzeitig zu erkennen. Erst das Zusammenspiel dieser Faktoren macht ein RAG-System im produktiven Einsatz wirklich robust.

Fazit

Der Microsoft SQL Server 2025 markiert einen Wendepunkt in der Evolution relationaler Datenbanken. Durch die native Unterstützung von Vektordatentypen und hocheffizienten Indizierungsverfahren wird der SQL Server vom reinen Datenspeicher zum leistungsfähigen Kern moderner KI-Architekturen.

Die Vorteile für Entwickler liegen auf der Hand: Statt komplexe Infrastrukturen mit separaten Vektor-Datenbanken und mühsamer Synchronisation aufzubauen, lassen sich semantische Suchen und RAG-Szenarien direkt dort umsetzen, wo die Geschäftsdaten liegen. Die Integration über bekannt-

te Werkzeuge wie T-SQL, `sp_invoke_external_rest_endpoint` und die Unterstützung in Entity Framework Core 10 sorgen für eine flache Lernkurve bei maximaler Performance.

Dank der Kombination aus bewährter ACID-Konformität und moderner Vektor-Power bietet der SQL Server 2025 eine pragmatische, sichere und hochperformante Basis, um Unternehmensanwendungen fit für das Zeitalter der agentischen KI zu machen. ■

- [1] Christian Kiefer, *Virtueller Kollege, KI-Sprachmodelle in Visual Studio nutzen*, *dotnetpro* 6/2024, Seite 89 ff.
- [2] Marc Müller, *Copilot Agent Mode – Konfiguration und Best Practices*, www.developer-world.de/dwx-insights/dwx-spotlight/dw-copilot-agent-mode-konfiguration-und-best-practices
- [3] Semantic Kernel auf GitHub, <https://github.com/microsoft/semantic-kernel>
- [4] Semantic Flow auf GitHub, <https://github.com/GregorBiswanger/SemanticFlow>
- [5] OllamaSharp auf GitHub, <https://github.com/awaescher/OllamaSharp>
- [6] LangChain .NET auf GitHub, <https://github.com/tryAGI/LangChain>
- [7] Microsoft Learn, *Neuerungen in SQL Server 2025*, <https://learn.microsoft.com/de-de/sql/sql-server/what-s-new-in-sql-server-2025?view=sql-server-ver17>
- [8] Holger Schwichtenberg, *Vektoren auf dem Vormarsch, Der SQL Server-Spaltentyp „Vector“ mit Entity Framework Core 10.0*, *dotnetpro* 12/2025, Seite 48 ff.
- [9] Jeremy Likness, *.NET AI Essentials – The Core Building Blocks Explained*, <https://devblogs.microsoft.com/dotnet/dotnet-ai-essentials-the-core-building-blocks-explained/>
- [10] Patrick Schnell, *Vektor-Datenbanken: Wenn Zahlen Bedeutung bekommen*, www.developer-world.de/dwx-insights/artificial-intelligence/dw-vektor-datenbanken-wenn-zahlen-bedeutung-bekommen
- [11] Ollama, <https://ollama.com>
- [12] Ollama-Modellsuche, <https://ollama.com/search>
- [13] Microsoft Learn, *CREATE VECTOR INDEX (Transact-SQL) (Preview)*, <https://learn.microsoft.com/en-us/sql/t-sql/statements/create-vector-index-transact-sql?view=sql-server-ver17>
- [14] Microsoft Learn, *What's New in EF Core 10*, <https://learn.microsoft.com/en-us/ef/core/what-is-new/ef-core-10.0/whatsnew>
- [15] Caddy, <https://caddyserver.com/download>



Christian Kiefer

lebt in der Nähe von Heidelberg und arbeitet beim Deutschen Krebsforschungszentrum (DKFZ) in Heidelberg mit dem Schwerpunkt .NET in den Bereichen Web und Backend.
christian.kiefer@bsp.de

LEGACY-TRANSFORMATION, TEIL 2

Keine Produktion ohne Automation

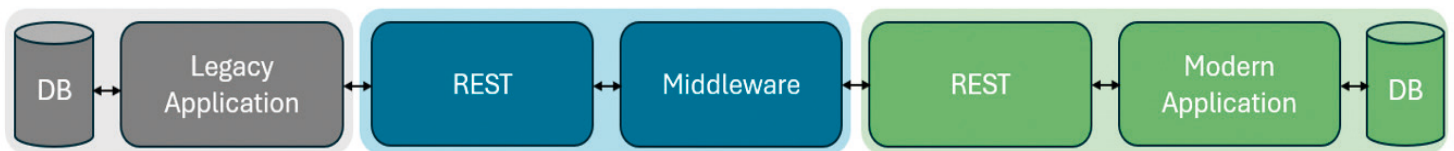
Um nicht mitten im Prozess stehen zu bleiben, schließt dieser Artikel zum Thema SQL Change Tracking das in der vorigen Ausgabe begonnene smarte Migrations-Szenario ab.

Zunächst ein kurzer Rückblick: Im Kern unserer Betrachtung steht die Idee, eine vorhandene Applikation durch eine neue, modernere Anwendung (oder gar App) zu ersetzen. Da ein architektureller Wechsel von einer monolithischen in eine deutlich mehrschichtigere Welt stattfinden soll, erscheint eine direkte Big-Bang-Migration wenig sinnvoll. Vielmehr soll das neue Produkt parallel zur Legacy-Anwendung laufen – zumindest für eine Übergangszeit. Während dieser Zeit ist es erforderlich, dass Änderungen, die durch die Legacy-Anwendung in die bisherige Datenbank finden, auch in Richtung der Ablösung kommuniziert werden, und das ist die Stelle, an der das Microsoft SQL Server Change Tracking [1] ins Spiel kommt.

etwas sehr Greifbares: Einen NotificationHandler für einen Account – auf der Basis einer tatsächlichen Datenbanktabelle.

Automation

Automation meint im ursprünglichen Wortsinn das selbstständige Denken. Mit OpenClaw stehen wir zwar kurz davor, aber für unseren Fall würde ich noch nicht so weit gehen. Seien wir daher froh, dass es weitere Wortbedeutungen gibt: Zum Beispiel die des selbstständigen Handelns (was zwar ebenfalls verdächtig nach OpenClaw klingt, aber lassen wir das). Denn das, was dieser Lösung noch fehlt, um sie näher an die Produktionsreife zu bringen, ist im Wesentlichen die



Von DB zu DB (Bild 1)

Zur Erinnerung dazu noch einmal **Bild 1** aus dem vorangegangenen ersten Teil des Artikels [2]. Das dort bisher erreichte Ergebnis setzt sich aus drei Projekten zusammen:

- SqlChangeTracker
- MockMiddleware
- CrmService

Im SqlChangeTracker findet sich eine umfängliche Lösung, die sich als Basis für beliebige SQL-Datenbanken nutzen lässt. Warum sie noch nicht Production-ready ist, dazu kommen wir gleich.

Die MockMiddleware ist genau das: Eine kleine Attrappe. Wodurch Sie Ihre Legacy-Anwendung ersetzen, ist letztlich für diesen Artikel nicht wichtig, und ebenso wenig ist von Belang, welche Art von Middleware Sie dabei als „Datendrehscheibe“ einsetzen. Dass aber irgendeine Art von Ziel für den Service existieren soll, an den etwaige Datenbankänderungen geschickt werden, ist keine Frage. Daher gibt sich der Artikel an dieser Stelle mit einem Mock zufrieden.

Der CrmService wiederum greift auf den SqlChangeTracker zurück und macht aus dem abstrakten NotificationHandler

Möglichkeit, sie autonom laufen zu lassen – und nicht interaktiv als Konsolenanwendung.

Das stimmt allerdings auch nur bedingt. Denn während der nachfolgend aufgezeigte Weg auf einen Windows-Service hinausläuft, gerät dadurch das Konsolenprojekt keineswegs ganz ins Hintertreffen. Aber der Reihe nach.

Windows-Service

Zunächst einmal wäre die Idee, dass der CrmService tatsächlich als solcher läuft: als Windows-Service, im Hintergrund. Im Lauf der Jahre gab es im Umfeld von .NET verschiedene Ansätze dafür; aktuell lautet der empfohlene Weg dazu, eine sogenannte BackgroundService-App als Basis zu verwenden.

Drumherum nutzen Sie einen Worker Service und einen Generic Host – dann klappt es auch sauber mit der Dependency Injection (interessant für zum Beispiel Logging) und einer *appSettings.json*, ohne dass Sie sich oder Ihren Code dafür verbiegen müssten.

Doch wie man das Thema dreht und wendet (und auch der neidische Blick in Richtung Linux und dessen Cron-Jobs hilft da nicht): Im Kern geht es dabei immer um eine Anwendung,

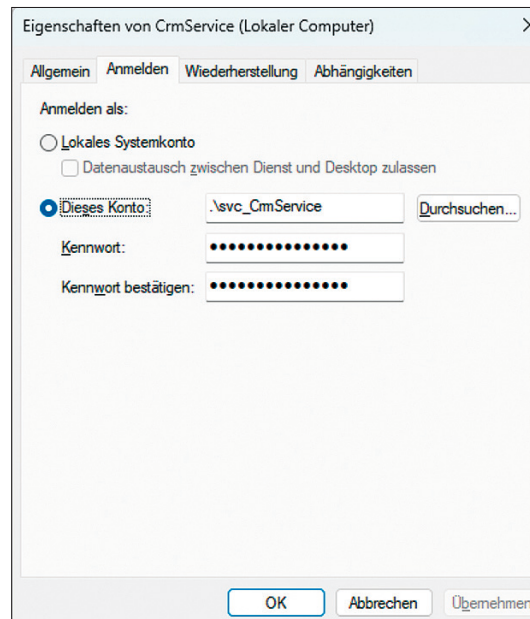
die grundsätzlich ohne Benutzerinteraktion im Hintergrund läuft und dabei vom Service Controller verwaltet wird. Das bedeutet zum Beispiel auch, dass bei der Einrichtung eines solchen Service angegeben werden muss, unter wessen Credentials der Dienst laufen soll (siehe Bild 2).

Softwareentwickler haben noch immer eine gewisse Tendenz dazu, mit administrativen Rechten zu arbeiten, und, wenn sie solche Dienste entwickeln, publishen und registrieren, sich wenig weitere Gedanken darüber zu machen. Ergebnis: Der Dienst läuft standardmäßig unter *NT AUTHORITY\Local Service*-Berechtigungen, und das fällt auch erst beim ersten Zugriff auf die Datenbank auf – insofern dieser via Integrated Security erfolgt (siehe [2]). Denn was bisher noch der Benutzer war (solange es sich beim CrmService noch um eine Konsolen-Anwendung handelte, die direkt in Visual Studio gestartet wurde), ist nun *NT Authority\Local Service*, und der hat erst einmal keine Rechte auf irgendwelche Datenbanken.

Daher noch einmal zurück zum Anfang: Der CrmService benötigt lesende Berechtigungen auf exakt die Tabellen (und deren SideTables), um die es sich beim Change Tracking dreht. Denn auch wenn es nicht unbedingt zu den Aufgaben gehört, die im direkten Fokus stehen: Es geht hier um Security!

Security

Damit ein Windows-Dienst als solcher gestartet werden kann, benötigt der Benutzer, mit dem der Dienst gestartet wird, das *SeServiceLogonRight* (was bei *NT Authority\Local Service* der Fall ist), vergleiche Kasten *SeServiceLogonRight*. Wir tauchen an dieser Stelle lieber nicht allzu sehr in die Tiefen der Berechtigungs-Infrastruktur von Windows ein. Erstellen Sie einfach einen neuen Benutzer – lokal auf Ihrem Entwickler-PC oder als Domänen-Account. Dieser dient dazu, dass Ihr künftiger Service damit gestartet wird. Folgen Sie daher den



Logon für den Service (Bild 2)

Richtlinien Ihres Unternehmens (oder Ihren eigenen) in der Benennung des Dienst-Kontos. Beispiel: *svc_CrmService*.

Als Nächstes gewähren Sie dem Konto entweder per GPO (Gruppenrichtlinienobjekt) oder lokal per *SecPol.msc* das Recht, sich als Dienst anzumelden (siehe Bild 3). Optional empfiehlt sich an dieser Stelle auch, die Möglichkeit zu verweigern, sich lokal und per Remote Desktop anmelden zu können. Abschließend benötigt dieser Account noch die korrekten Rechte auf die CRM-Datenbank. Erstellen Sie dazu im SQL Server einen entsprechenden Login sowie einen dazu passenden User in der CRM-Datenbank. Geben Sie diesem Benutzer *db_datareader*-Berechtigungen und vergessen Sie

nicht, ihm ebenfalls die korrekten Berechtigungen zu geben, um Einsicht in die gewünschten Change Tables zu erhalten (*GRANT VIEW CHANGE TRACKING*), vergleiche Kasten *Rechteprüfung*. Hier die dazu erforderlichen SQL-Queries:

```
CREATE LOGIN [COMPUTERNAME\svc_CrmService]
FROM WINDOWS;

USE [Crm];

CREATE USER [COMPUTERNAME\svc_CrmService]
FOR LOGIN [COMPUTERNAME\svc_CrmService];

ALTER ROLE [db_datareader]
ADD MEMBER [COMPUTERNAME\svc_CrmService];

GRANT VIEW CHANGE TRACKING
ON [dbo].[Account] TO [COMPUTERNAME\svc_CrmService]
```

Zwischenstand 1

Sie haben nun einen Windows-Account, der sich als Service anmeldet, und Sie haben die korrekten Berechtigungen in ►

SeServiceLogonRight

Sollten Sie Ihren Dienst bereits installiert haben (da Sie schon drei Schritte weiter sind und diesen Prozess kennen) und in dessen Einstellungen einfach ein Konto eingegeben haben, wird Ihnen aufgefallen sein, dass Windows diesem Konto in dem Fall selbstständig das Recht verleiht, sich als Dienst anmelden zu können. Das ist eine Form der Convenience, die man unter Windows selten findet.

Rechteprüfung

Falls Sie sich in einer Situation wiederfinden, in der es notwendig wird, zu prüfen, welche User Zugriffsberechtigungen auf die Side-Tables haben, bietet sich die SQL-Query aus Listing 1 an. Dabei wird konkret ermittelt, welche Konten *VIEW CHANGE TRACKING*-Berechtigungen auf der angegebenen Tabelle haben – oder Sie lassen den Filter auf die Tabelle weg und erhalten so eine Übersicht über die gesamte Datenbank.

der Datenbank gesetzt, sodass dieser sich auch damit verbinden und damit arbeiten kann.

Lassen Sie uns das prüfen, bevor Sie aus der Konsolenanwendung einen Windows-Service machen. Doch halt! Bevor Sie nun auf die Idee kommen, Visual Studio mittels *runas* als *svc_CrmService* zu starten und zu prüfen, ob die Konsolenanwendung noch funktioniert wie erwartet: Sind Sie dem optionalen Schritt von vorher gefolgt und haben die lokale Anmeldung per Policy verweigert? Gut für die Sicherheit, schlecht für die Testbarkeit. Da es aber im Moment nur darum geht, zu prüfen, ob die Datenbankberechtigungen stimmen, funktionieren auch folgende SQL-Queries, die Sie direkt im Microsoft SQL Server Management Studio ausführen können:

```
EXECUTE AS LOGIN =
    '[Computername]\svc_CrmService';

SELECT TOP (1) * FROM [dbo].[Account];
SELECT CHANGE_TRACKING_CURRENT_VERSION();

REVERT;
```

Umbau in einen Generic Host

Der grundsätzlich einfachste Einstieg in die Welt der Windows-Service-Entwicklung ist der folgende Aufruf in Ihrer Development-Konsole (vergleiche Bild 4):

```
dotnet new worker --name <Projektname>
```

Dadurch erhalten Sie auf der Basis einer dafür gedachten Projektvorlage einen Stub, auf den Sie einfach aufsetzen können.

Listing 1: VIEW CHANGE TRACKING

```
SELECT
    grantee.name AS Grantee,
    dp.state_desc AS State,
    dp.permission_name AS Permission,
    OBJECT_SCHEMA_NAME(dp.major_id) AS SchemaName,
    OBJECT_NAME(dp.major_id) AS ObjectName
FROM sys.database_permissions AS dp
JOIN sys.database_principals AS grantee
    ON dp.grantee_principal_id = grantee.principal_id
WHERE dp.permission_name = 'VIEW CHANGE TRACKING'
    AND OBJECT_SCHEMA_NAME(dp.major_id) = 'dbo'
    AND OBJECT_NAME(dp.major_id) = 'Account'
```

nen. Darin sind auch direkt die wesentlichen Unterschiede zu Ihrem aktuellen Code zu erkennen, angefangen bei der *CrmService.csproj*-Datei:

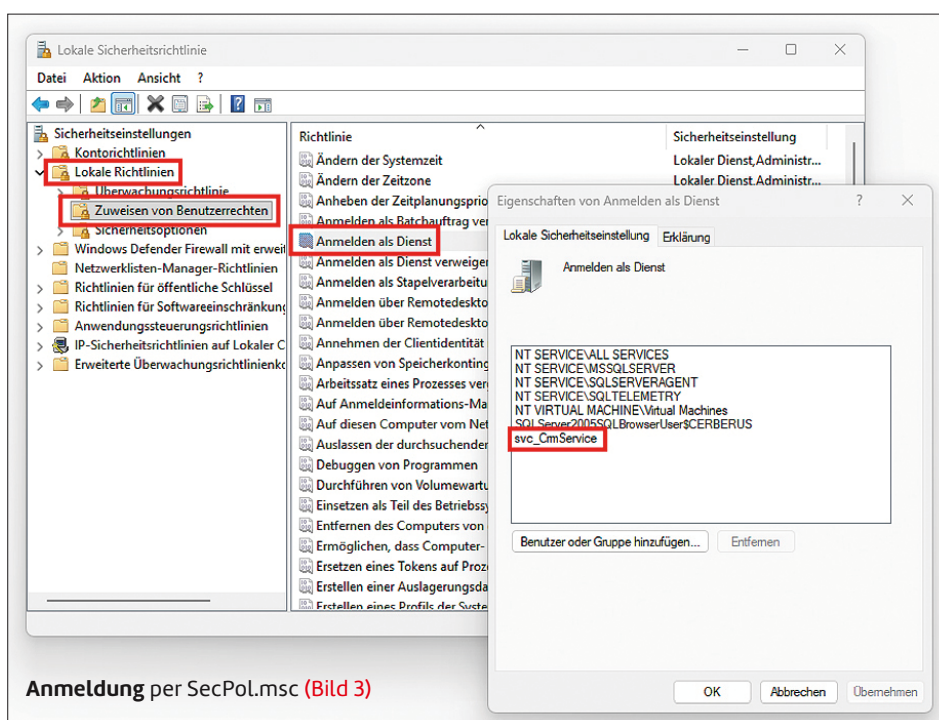
- Statt auf Microsoft.NET.Sdk basiert das Projekt auf dem SDK Microsoft.NET.Sdk.Worker (ändern Sie das).
- Der OutputType *Exe* wird nicht mehr benötigt (entfernen Sie ihn).
- Es wird eine neue *PackageReference* für das Hosting benötigt (ergänzen Sie sie).

```
<PackageReference Include="Microsoft.Extensions.Hosting"
    Version="10.0.2" />
```

Ebenfalls in dem Projekt zu erkennen: Die *Program.cs* sieht recht leer aus. Darin wird lediglich das Builder-Pattern genutzt, eine neue Klasse *Worker* als *HostedService* hinzugefügt, gebaut und dann in einem Host gestartet. Werfen Sie dabei auch einen Blick in die *Worker*-Klasse selbst. Diese leitet sich von *BackgroundService* ab und überschreibt lediglich die Methode *ExecuteAsync* – in der sich alles abspielt.

Für Ihren *CrmService* bedeutet das ein paar Änderungen. Die Details lesen Sie im Repository mit Beispieldateien zu diesem Artikel unter [3] nach; hier nur der Überblick. Sie beginnen zunächst, den *SqlChangeTracker* auf die Änderungen vorzubereiten, indem Sie ein neues Interface mit dem Namen *INotificationHandler* ergänzen:

```
namespace
    SqlChangeTracker.Interfaces;
```



```
public interface INotificationHandler
{
    Task StartAsync(CancellationTokens cancellationTokens = default);

    Task StopAsync(CancellationTokens cancellationTokens = default);
}
```

Passen Sie als Nächstes die abstrakte Basisklasse *NotificationHandler* entsprechend an:

```
public abstract class NotificationHandler<T> :
    INotificationHandler where T : class, IChangeTable,
    new()
```

Da die *Start*- und *Stop*-Methoden im Folgenden leicht umgebaut werden und es spätestens im Kontext eines Windows-Service wichtig wäre, sich um das Thema der Cancellation Tokens Gedanken zu machen, ergänzen Sie eben solche und bauen die beiden Methoden *Start* und *Stop* entsprechend der Schnittstelle um. Dazu benötigen Sie eine *CancellationTokensSource*; erstellen Sie diese in *StartAsync* und setzen Sie sie in der Methode *StopAsync* ein.

```
private CancellationTokensSource? linkedCts;
...
public virtual async Task StartAsync(
    CancellationTokens cancellationTokens = default)
{
    linkedCts = CancellationTokensSource.
        CreateLinkedTokenSource(cancellationTokens);
    ...
}
public async Task StopAsync(CancellationTokens
    cancellationTokens = default)
{
    ...
    try
    {
        linkedCts?.Cancel();
    }
    ...
}
finally
{
    ...
    try
    {
        linkedCts?.Dispose();
    }
    catch { }
    linkedCts = null;
}
}
```

Wenden Sie sich nun dem Projekt *CrmService* zu und implementieren Sie im neuen Unterverzeichnis *Services* die Klasse *NotificationHostedService*. Diese erbt von *BackgroundService* und erhält vom Host eine Liste von *NotificationHandlers* und – weil es sinnvoll ist – einen Logger injiziert (siehe [Listing 2](#)).

Abschließend müssen Sie nun noch die *Program.cs* anpassen. Fürs Erste benötigen Sie keinen *ConfigurationBuilder* mehr, da das Host-Builder-Pattern bereits mit einer *appsettings.json*-Datei umgehen kann, und die *Services* werden nicht mehr als *ServiceCollection* separat konfiguriert, sondern via *ConfigureServices* innerhalb des Hosts ergänzt. Der wichtigste Change besteht dabei im Umgang mit dem *NotificationHandler* und dem neuen *NotificationHostedService*:

```
services.AddSingleton<INotificationHandler,
    NotificationHandlerAccount>();
services.AddHostedService<NotificationHostedService>();
...
var host = builder.Build();
await host.RunAsync();
```

Wichtig dabei für alle, die noch nicht firm im Thema Dependency Injection sind: Die wahre Magie entfaltet sich in dem Moment, in dem Sie mehrere *NotificationHandler* implementieren und hier registrieren. Erinnern Sie sich an den *NotificationHostedService*: Hier werden Handler als *IEnumerable* injiziert, und in *StartAsync/StopAsync* wird einfach über sie iteriert. Das passiert praktisch alles „von allein“. Danke dafür.

Hier kommt der interessante Part: Die Anwendung lässt sich an dieser Stelle noch immer als Konsolenanwendung starten und somit komfortabel debuggen. Oder aber Sie ergänzen vor dem Aufruf *builder.Build()* noch folgende Zeilen:

```
if(!Environment.UserInteractive) ▶
```

```
PS R:\> dotnet new worker --name WorkerDemo
Die Vorlage "Workerdienst" wurde erfolgreich erstellt.

Aktionen nach der Erstellung werden verarbeitet...
Wiederherstellung R:\WorkerDemo\WorkerDemo.csproj:
Wiederherstellung erfolgreich.

PS R:\> cd .\WorkerDemo\
PS R:\WorkerDemo> ls

Verzeichnis: R:\WorkerDemo

Mode                LastWriteTime         Length Name
----                -
d-----          14.02.2026    06:51         obj
d-----          14.02.2026    06:50        Properties
-a-----          14.02.2026    06:50        137 appsettings.Development.json
-a-----          14.02.2026    06:50        137 appsettings.json
-a-----          14.02.2026    06:50        163 Program.cs
-a-----          14.02.2026    06:50        497 Worker.cs
-a-----          14.02.2026    06:50        427 WorkerDemo.csproj

PS R:\WorkerDemo> |
```

Einen neuen Worker per Template erstellen (Bild 4)

Listing 2: NotificationHostedService.cs

```
using SqlChangeTracker.Interfaces;

namespace CrmService.Services;

public class NotificationHostedService(
    IEnumerable<INotificationHandler> handlers,
    ILogger<NotificationHostedService> logger) :
    BackgroundService
{
    public override async Task StartAsync(
        CancellationToken cancellationToken)
    {
        logger.LogInformation("NotificationHostedService
            starting - starting handlers.");
        foreach (var handler in handlers)
        {
            try
            {
                await handler.StartAsync(cancellationToken);
                logger.LogInformation("Started handler
                    {Handler}", handler.GetType().Name);
            }
            catch (Exception ex)
            {
                logger.LogError(ex, "Error starting handler
                    {Handler}", handler.GetType().Name);
            }
        }

        await base.StartAsync(cancellationToken);
    }

    protected override async Task ExecuteAsync(
        CancellationToken stoppingToken)
    {
        logger.LogInformation("NotificationHostedService
            running; awaiting shutdown signal.");
        try
        {
            await Task.Delay(Timeout.Infinite,
                stoppingToken).ConfigureAwait(false);
        }
        catch (OperationCanceledException)
        {
            // expected on shutdown
        }

        public override async Task StopAsync(
            CancellationToken cancellationToken)
        {
            logger.LogInformation("NotificationHostedService
                stopping - stopping handlers.");
            foreach (var handler in handlers)
            {
                try
                {
                    await handler.StopAsync(cancellationToken);
                    logger.LogInformation("Stopped handler
                        {Handler}", handler.GetType().Name);
                }
                catch (Exception ex)
                {
                    logger.LogError(ex, "Error stopping handler
                        {Handler}", handler.GetType().Name);
                }
            }

            await base.StopAsync(cancellationToken);
        }
    }
}
```

```
{
    builder.UseWindowsService();
}
```

Genial, oder? Genau an dieser Stelle wird aus der Konsolenanwendung ein Windows-Service. Diese Extension erweitert die Anwendung um die erforderlichen Komponenten, sodass sie vom Windows Service Control Manager verwaltet werden kann. Interessiert Sie die Implementierung? Werfen Sie gerne mal einen Blick in den Code von Microsoft unter [4].

Die Installation der Anwendung als Service ist nun denkbar einfach. Nutzen Sie dafür auf der Konsole das Tool `sc.exe`. Im Repository unter [3] finden Sie dazu ein Helper-Skript mit dem Namen `Publish_and_Install_Service.ps1` (und der Vollständigkeit halber auch ein Skript mit dem Namen `Uninstall_`

`Service.ps1`). Vergessen Sie nicht, den Benutzer des Dienstes entsprechend dem anzupassen, was Sie am Beginn dieses Artikels gelernt haben.

Scrutor

Ich weiß nicht, wie es bei Ihnen ist, aber mein GitHub Copilot antwortet auf praktisch alle Prompts in Visual Studio grundsätzlich erst einmal mit verschiedenen Variationen von „Erst mal tief durchatmen“. Das ist an dieser Stelle womöglich auch angebracht. Und obwohl es Ihnen an dieser Stelle so noch nicht aufgefallen sein wird, gibt es hier einen kleinen Einschub, mit dem Sie nicht gerechnet haben werden. Das Stichwort lautet erneut: Dependency Injection.

Sie haben gerade erlebt, wie es Ihnen die Dinge erleichtert, indem Ihr `NotificationHandlerAccount` automatisch in den

```

info: CrmService.Services.NotificationHostedService[0]
      NotificationHostedService starting - starting handlers.
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (13ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT ISNULL(CHANGE_TRACKING_CURRENT_VERSION(), 0) as VersionCount
info: NotificationHandlerAccount[0]
      SqlDBNotificationService for 'AccountChangeTable' started by NotificationHandlerAccount.
info: NotificationHandlerAccount[0]
      Notification handler 'NotificationHandlerAccount' started.
info: CrmService.Services.NotificationHostedService[0]
      Started handler NotificationHandlerAccount
info: CrmService.Services.NotificationHostedService[0]
      NotificationHostedService running; awaiting shutdown signal.
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (1ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT ISNULL(CHANGE_TRACKING_CURRENT_VERSION(), 0) as VersionCount

```

Logging in
der Konsole
(Bild 5)

HostedService injiziert wurde – und nicht nur das! Hätten Sie mehrere solcher Handler, würde es genügen, sie einfach nur zu registrieren, und schon würden auch diese „automagisch“ korrekt im HostedService referenziert.

Doch haben Sie sich schon einmal Gedanken darüber gemacht, was passiert, wenn Sie – aus Gründen – wirklich viele solcher Handler zu registrieren hätten? In dem Fall schlage ich Ihnen eine Abkürzung vor: Mit Scrutor steht Ihnen ein Projekt zur Verfügung, das Ihre Automation automatisiert. Einfach als NuGet-Paket Ihrem Projekt hinzugefügt, können Sie Scrutor Ihre Assembly nach Extensions für die Dependency Injection scannen lassen; die Registrierung erfolgt dann automatisch.

Der Vorteil: Meldungen, die Ihnen erklären, eine Abhängigkeit könne nicht aufgelöst werden, gehören damit der Vergangenheit an. Probieren Sie es gerne aus. Mehr dazu auf der GitHub-Seite von Scrutor [5].

Zwischenstand 2

Da Sie zuvor bereits einen Service-Account sowohl unter Windows als auch im SQL Server mit den erforderlichen Berechtigungen eingerichtet haben, sind Sie nun in einer recht komfortablen Situation: Sie können den CrmService sowohl als Dienst starten wie auch als Konsolenanwendung – es empfiehlt sich aus naheliegenden Gründen, nicht beides gleichzeitig zu tun.

Warum also diese Unterscheidung zwischen Dienst und Konsolen-Anwendung? Der Grund ist ganz einfach: Einen Dienst zu debuggen ist etwas tricky. Beispielsweise verpassen Sie in der Regel die komplette Start-Sequenz, wenn Sie sich mit dem Visual Studio Debugger an den laufenden Prozess andocken wollen.

Es gibt zwar Möglichkeiten, das zu umgehen, aber die machen es ehrlich gesagt nicht besser. Dabei kann es zusätzlich noch zu Berechtigungsproblemen kommen, wenn der Dienst nicht unter Ihren eigenen Credentials läuft – und das wollen Sie natürlich nicht. Also lassen Sie es einfach und nutzen den für solche Fälle vorgesehenen Weg wie hier beschrieben.

Logging

Da dieser Artikel eine Annäherung an die Produktionsreife zum Ziel hat, darf das Thema „Logging“ natürlich nicht feh-

len. Und Sie sind dem hier bereits begegnet: Im *NotificationHostedService*.

Sollten Sie sich an dieser Stelle gewundert haben, warum es bereits Logmeldungen in der Konsole gab (siehe Bild 5), ohne dass Sie in *Program.cs* dafür etwas konfiguriert und als Service für die Dependency Injection hinzugefügt haben, liegt das daran, dass das Logging-Interface vom Host automatisch injiziert wird. Sie erinnern sich? Ich erwähnte bereits, dass das Builder-Pattern die *appsettings.json*-Datei automatisch verarbeitet. Interessant ist aber, dass Sie über diese Settings-Datei das Logging komfortabel konfigurieren können. Das ist wichtig, denn sobald Ihr CrmService als Windows-Service interaktionslos im Hintergrund läuft, sehen Sie keine Konsole mit Logmeldung mehr.

Während sich für realistische Produktions-Szenarien mindestens Log-Dateien für diesen Zweck eignen, besser aber noch Frameworks, die an cloudbasierte Logging-Provider berichten (zum Beispiel mittels Serilog nach DataDog oder Sentry), gäbe es da noch einen fast vergessenen Klassiker: das Windows-Eventlog.

Falls Sie allerdings nicht planen, den CrmService als Produkt zu entwickeln, das verkauft und mittels *Setup.exe* installiert werden können soll, genügt es vermutlich, eine ►

● Developers Developers Developers Developers

Erinnern Sie sich noch an den legendären Auftritt von Steve Ballmer als damaliger CEO von Microsoft bei der .NET Conference des Jahres 1999 [9]? Inspiriert durch diesen Auftritt haben die früheren Mitarbeiter von Microsoft Derek Hatchard und Dirk Primbs 2009 ein Buch mit dem gleichnamigen Titel veröffentlicht: „Developers Developers Developers Developers“ [10]. Darin enthalten: 17 Fachartikel zu verschiedenen Entwicklungsthemen von verschiedenen Autoren unter dem Label „{You shape} it“ von Microsoft. Darunter: „Creating Useful Installers with Custom Actions“ by Christian Jacob. Darin geht es um das Erstellen einer Event Source für das Windows Event Log während der Installation einer Software mit dem Microsoft Installer. Was für ein Zufall.

Ebene	Datum und Uhrzeit	Quelle	Ereigni...	Aufgab...
Informationen	14.02.2026 08:23:39	CrmService	0	Keine
Informationen	14.02.2026 08:23:39	CrmService	0	Keine
Informationen	14.02.2026 08:23:35	CrmService	0	Keine
Informationen	14.02.2026 08:23:35	CrmService	0	Keine

Ereignis 0, CrmService	
Allgemein	Details
☐ Angezeigte Ansicht ☒ XML-Ansicht	
<pre> - <Event xmlns="http://schemas.microsoft.com/win/2004/08/events/event"> - <System> <Provider Name="CrmService" /> <EventID Qualifiers="0">0</EventID> <Version>0</Version> <Level>4</Level> <Task>0</Task> <Opcode>0</Opcode> <Keywords>0x8000000000000000</Keywords> <TimeCreated SystemTime="2026-02-14T07:23:39.3310732Z" /> <EventRecordID>35548</EventRecordID> <Correlation /> <Execution ProcessID="33268" ThreadID="0" /> <Channel>Application</Channel> <Computer>Cerberus</Computer> <Security /> </System> - <EventData> <Data>Service started successfully.</Data> </EventData> </Event> </pre>	

Eventlog mit Meldungen vom CrmService (Bild 6)

Event Source im Windows-Eventlog manuell anzulegen (vergleiche Kasten **Developers Developers Developers Developers**).

Das Skript *Publish_and_Install_Service.ps1* aus dem Repository unter [3] tut dies bereits:

```

$source = $serviceName
if (-not [System.Diagnostics.EventLog]::SourceExists(
    $source)) {
    New-EventLog -LogName Application -Source $source
    Write-Host "Created EventLog source $source"
}

```

Was genau nun in das Eventlog von Windows geschrieben wird, obliegt Ihrer Konfiguration, die Sie zu diesem Zweck in der *appsettings.json* hinterlegen. Hier ein Beispiel:

```

{
  ...
  "Logging": {
    "LogLevel": {
      "Default": "Warning",
      "Microsoft": "Warning",
      "Microsoft.EntityFrameworkCore.Database.Command":
        "Warning"
    },
    "EventLog": {
      "LogLevel": {
        "Default": "Information"
      }
    }
  }
}

```

Wie Sie hieran erkennen können, sind die Standardeinstellungen in diesem Fall auf Warnungen ausgelegt – zu Testzwecken für das Eventlog mit dem Default-Loglevel *Information* aber sehr gesprächig.

Im Ergebnis finden Sie dann im Anwendungs-Log (*eventvwr.exe*) Meldungen von Ihrem CrmService wie zum Beispiel in **Bild 6** zu sehen. Eine noch bessere Trennung zwischen einer Entwicklungsumgebung, einer Testumgebung und Ihrer Produktivumgebung erreichen Sie natürlich, indem Sie die Konfiguration je Umgebung anpassen. Das gelingt Ihnen, indem Sie zusätzlich zur *appsettings.json* weitere Konfigurationsdateien nach dem Muster *appsettings.{Environment}.json* ergänzen. Visual Studio erkennt dies und ordnet die Dateien für Sie entsprechend an, und die DotNet-Laufzeitumgebung orientiert sich an Umgebungsvariablen, um zwischen den Konfigurationen unterscheiden zu können. Außerhalb von Visual Studio erreichen Sie das über das Setzen der Umgebungsvariablen *DOTNET_ENVIRONMENT* und *ASPNETCORE_ENVIRONMENT* (auf zum Beispiel *Development* oder *Prod*). Einen interessanten Artikel zum Thema Umgebungsvariablen finden Sie unter [6].

In Visual Studio selbst können Sie es sich einfach machen, indem Sie im Projekt CrmService im Verzeichnis *Properties* eine Datei mit dem Namen *launchSettings.json* ergänzen. Darin lässt sich unter anderem die Umgebung konfigurieren:

```

{
  "profiles": {
    "CrmService": {
      "commandName": "Project",
      "environmentVariables": {
        "DOTNET_ENVIRONMENT": "Development",
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}

```

Vorsicht: In Produktivumgebungen funktioniert die *launchSettings.json* natürlich nicht.

Gesundheit

Inzwischen haben Sie einen CrmService, den Sie komfortabel als Konsolenanwendung debuggen, aber auch als Windows-Service hosten können. Er kann mit mehreren Notification-Handlern zugleich umgehen und damit im Zweifel Ihre gesamte Datenbank überwachen, und er erzeugt auch Log-

Meldungen, die Ihnen dabei helfen, etwaige Probleme zu analysieren – ob nun im Konsolenfenster, im Windows-Eventlog oder bei dem Monitoring-Anbieter Ihrer Wahl.

Apropos Monitoring. Da war doch was.

Spätestens sobald Sie in Ihren internen Abstimmungsrunden erklären, demnächst den neuen Service in der Produktion ausrollen zu wollen, wird Sie (hoffentlich) jemand aus der IT darauf ansprechen, wie man den Dienst in das Monitoring aufnehmen kann, um bei etwaigen Problemen entweder automatisiert Probleme beheben zu können oder die Produktverantwortlichen via Teams, WhatsApp, Slack, MatterMost, SMS, E-Mail, Brieftaube, Rauchzeichen oder sonst etwas zu informieren.

Ihre Antwort?

„Na, greif eben auf das Eventlog zu, da gibt es eine Source für den Dienst.“

Oder aber Sie sind ein guter Entwickler und implementieren ein paar Health Checks. Los gehts!

App-Health-Checks

Microsoft schreibt dazu: „In einem verteilten System sind Health Checks regelmäßige Bewertungen des Status, der Verfügbarkeit und der Leistung einzelner Knotenpunkte oder Dienste. Mit diesen Prüfungen wird sichergestellt, dass das System ordnungsgemäß und effizient funktioniert. Integritätsprüfungen sind für die Systemzuverlässigkeit unerlässlich und werden in der Regel in regelmäßigen Abständen durchgeführt, wobei die Ergebnisse für die Entscheidungsfindung und Korrekturmaßnahmen analysiert werden“ (vergleiche [7]; dort finden Sie zudem Details zu diesem Thema).

Ebenso typischerweise werden Sie Health Checks in Webapplikationen vorfinden, weshalb die charmanteste Art, sie zu implementieren, die ist, Health-Endpunkte bereitzustellen, die externe Monitoring-Dienste abrufen können. Sprich: Sie benötigen nun zweierlei:

- einen in den Dienst integrierten Webserver,
- ein Interface für Health Checks.

Ergänzen Sie dazu im Projekt *CrmService* zunächst eine Paket-Referenz auf das NuGet-Paket *Microsoft.Extensions.Diagnostics.HealthChecks.ResourceUtilization* und zusätzlich eine Framework-Referenz auf *Microsoft.AspNetCore.App*. Damit haben Sie alles, was Sie benötigen.

Zur Demonstration implementieren Sie einen einfachen SQL Connection Health Check, dessen Aufgabe es ist, Auskunft darüber zu geben, ob eine Verbindung zur Datenbank besteht. Beginnen Sie mit einem neuen Verzeichnis *HealthChecks* und ergänzen Sie darin eine neue Klasse *SqlConnectionHealthCheck* (Listing 3). Um in Kürze das Ergebnis einer externen Anfrage in JSON-Form liefern zu können, implementieren Sie einen einfachen *HealthCheckResponseWriter* (Listing 4). Und um den Port für den Endpoint zu konfigurieren, fügen Sie folgenden Abschnitt in die *appsettings.json* ein:

```
"Health": {
  "Port": 5000
},
```

Nun konfigurieren Sie den Health-Endpoint direkt in der *program.cs* im Hostbuilder zum Beispiel direkt im Anschluss an das Logging wie folgt:

```
.ConfigureWebHostDefaults(webBuilder =>
{
    webBuilder.ConfigureKestrel((context, options) =>
    {
        var cfg = context.Configuration;
        var port = cfg.GetValue<int?>("Health:Port") ??
            5000;
        options.ListenLocalhost(port);
    });

    webBuilder.Configure(app =>
    {
```

Listing 3: SqlConnectionHealthCheck

```
using Microsoft.Data.SqlClient;
using Microsoft.Extensions.Diagnostics.HealthChecks;

namespace CrmService.HealthChecks;

public class SqlConnectionHealthCheck(IConfiguration
configuration) : IHealthCheck
{
    public async Task<HealthCheckResult>
        CheckHealthAsync(HealthCheckContext context,
            CancellationToken cancellationToken = default)
    {
        var cs = configuration.GetConnectionString(
            "DefaultConnection");
        if (string.IsNullOrEmpty(cs))
            return HealthCheckResult.Unhealthy(
                "ConnectionString missing");

        try
        {
            using var conn = new SqlConnection(cs);
            await conn.OpenAsync(cancellationToken);
            using var cmd = conn.CreateCommand();
            cmd.CommandText = "SELECT 1";
            await cmd.ExecuteScalarAsync(
                cancellationToken);
            return HealthCheckResult.Healthy("SQL OK");
        }
        catch (Exception ex)
        {
            return HealthCheckResult.Unhealthy(
                "SQL unreachable", ex);
        }
    }
}
```



```

app.UseRouting();

app.UseEndpoints(endpoints =>
{
    // Liveness (leichtgewichtig)
    endpoints.MapHealthChecks("/health");

    // Readiness (mehr Details)
    endpoints.MapHealthChecks("/hc",
        new HealthCheckOptions
        {
            Predicate = r => r.Tags.Contains("ready"),
            ResponseWriter = async (ctx, report) =>
            {
                await
                    HealthCheckResponseWriters.WriteJsonResponse(
                        ctx, report)
            });
        });
});
});
})

```

Abschließend fehlt nur noch das Registrieren des Health Checks in der Service-Konfiguration:

```

services.AddHealthChecks()
    .AddCheck<SqlConnectionHealthCheck>("sql", tags:
        new[] { "ready" })
    .AddResourceUtilizationHealthCheck(tags:
        ["ready", "resources"]);

```

Rufen Sie nun nach dem Start des CrmService den Endpunkt `http://localhost:5000/health` auf, erhalten Sie ein einfaches *Healthy* als Antwort. Hoffentlich. Falls nicht – oder nur der Vollständigkeit halber –, probieren Sie mal `http://localhost:5000/hc` (vergleiche [Bild 7](#)).

Ohne zusätzlichen Aufwand haben Sie neben dem SQL Health Check, den Sie selbst implementiert haben, auch noch Informationen zum Ressourcen-Verbrauch erhalten. Nett, oder? Werfen Sie zudem einen Blick ins Repository. Dort enthalten: Die Idee eines HeartBeats, der von den Notification-Handlern ausgeht – mit der Idee, dass bei Ausbleiben von HeartBeats ebenfalls ein Rückschluss auf die Gesundheit des Service getroffen werden kann.

Fazit und Ausblick

Falls Sie den beiden Artikeln zum Thema Change Tracking bis hierhin aufmerksam gefolgt sind, werden Ihnen drei Din-

● Listing 4: HealthCheckResponseWriters

```

using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Diagnostics.HealthChecks;
using System.Net.Mime;
using System.Text.Json;

namespace CrmService.HealthChecks;

public static class HealthCheckResponseWriters
{
    public static async Task WriteJsonResponse(
        HttpContext httpContext, HealthReport report)
    {
        httpContext.Response.ContentType =
            MediaTypeNames.Application.Json;
        var options = new JsonSerializerOptions {
            Indented = true };

        using var stream = new MemoryStream();
        using (var writer = new Utf8JsonWriter(
            stream, options))
        {
            writer.WriteStartObject();
            writer.WriteString("status",
                report.Status.ToString());
            writer.WriteStartObject("results");

            foreach (var entry in report.Entries)
            {
                writer.WriteStartObject(entry.Key);
                writer.WriteString("status",
                    entry.Value.Status.ToString());
                writer.WriteString("description",
                    entry.Value.Description ?? string.Empty);

                writer.WriteStartObject("data");
                foreach (var item in entry.Value.Data)
                {
                    writer.WritePropertyName(item.Key);
                    JsonSerializer.Serialize(writer,
                        item.Value);
                }
                writer.WriteEndObject();

                writer.WriteEndObject();
            }

            writer.WriteEndObject();
            writer.WriteEndObject();

            stream.Seek(0, SeekOrigin.Begin);
            await stream.CopyToAsync(
                httpContext.Response.Body);
        }
    }
}

```

ge nicht entgangen sein. Erstens ist die eigentliche Anforderung (das Tracken von Änderungen in SQL mit einem entsprechenden Ziel) gar nicht so komplex, und Sie könnten versucht sein, auf eine Frage zu einer Aufwandsschätzung entsprechend minimalistisch zu antworten. Insbesondere dieser Artikel hat vermutlich aber dazu beigetragen, den Aufwand etwas zu relativieren: Zu einem produktiven Einsatz gehört eben meist noch etwas mehr.

Zweitens fragen Sie sich vielleicht noch immer, was der ominöse Hinweis auf das Thema KI, GitHub-Repositories und das SQL Server Change Data Capture (CDC) im ersten Teil des Artikels [2] für einen Zweck hatte. Nun, der Autor des Repos *SQLLEFNotification* auf GitHub hat ab besagtem Punkt KI-Bots aufgegleist und diese die weitere Arbeit übernehmen lassen. Das Ergebnis ist dadurch leider vollkommen unbrauchbar geworden. Die Versionierung ist durcheinandergebracht, Anforderungen und Dokumentationen wirken nun wie von Maschinen für Maschinen geschrieben, die Beispiele funktionieren nicht mehr, und gleichzeitig hat sich die Komplexität des Codes derart gesteigert, dass er sich kaum noch nachvollziehen lässt.

Aus meiner Sicht ist das ein sehr aussagekräftiges Beispiel dafür, GitHub nicht mehr länger nur als hervorragende Quelle für wirklich wertvolle Open-Source-Projekte zu betrachten, sondern zusätzlich immer einen Blick auf die Contributor-Liste zu werfen und bei KI-Beteiligung etwas genauer draufzuschauen.

Und drittens? Der Change Context. Das war eine lange Spannungskurve, oder? Hier die Zusammenfassung: Falls Sie wollen, dass Daten, die Benutzer bereits im parallel laufenden Zielsystem erfassen, auch ihren Weg zurück in Ihre Legacy-Applikation finden (bidirektionaler Sync), dann würden Sie nicht wollen, dass diese Änderungen in der Legacy-Datenbank dazu führen, dass Ihr CrmService sie aufgreift und stante pede wieder zurückschickt. Das System also, das die Änderung in die Legacy-Datenbank schreibt, kann zu diesem Zweck um die Transaktion einen sogenannten *CHANGE_TRACKING_CONTEXT* legen, dem Sie einen Namen geben können (zum Beispiel den Grund der Änderung oder den Namen des Quell-Systems). Details zu diesem Konzept lesen Sie beispielsweise direkt bei Microsoft unter [8] nach.

In Ihrem *SqlDbNotificationService* im Projekt *SqlChangeTracker* erweitern Sie nun das SQL-Statement für den Abruf von *ChangeTable*-Informationen um einen Filter für diesen Context, und schon haben Sie die Möglichkeit geschaffen, bestimmte Änderungen zu „ignorieren“. Eine mögliche Implementierung finden Sie wie gewohnt als Beispiel im Repository zu diesem Artikel [3].

Pro-Tipp: Machen Sie diesen *ContextFilter* konfigurierbar. Überpro-Tipp: Sorgen Sie dafür, dass Änderungen in der *appsettings.json* zur Laufzeit übernommen werden können. Was? Mit einem einfachen *reloadOnChange: true* klappt das nicht? Richtig. Ist ja auch keine ASP.NET-Core-Anwendung, sondern ein Windows-Service (Stichwort: Config Monitoring). Und auch hier gilt: Werfen Sie einen Blick ins Repo unter [3]; ich habe da etwas für Sie vorbereitet, denn nach dem Artikel ist vor dem Repository.

```

{
  "status": "Healthy",
  "results": {
    "sql": {
      "status": "Healthy",
      "description": "SQL OK",
      "data": {}
    },
    "container resources": {
      "status": "Healthy",
      "description": "",
      "data": {
        "CpuUsedPercentage": 30.212977930696027,
        "MemoryUsedPercentage": 0.3163003841894238
      }
    }
  }
}

```

Health Check mit Zusatzinformationen (Bild 7)

In diesem Sinne wünsche ich Ihnen weiterhin viel Erfolg bei geplanten Legacy-Migrationen und freue mich über Feedback oder auch Anfragen zu dem Thema. ■

- [1] SQL Change Tracking, <https://learn.microsoft.com/en-us/sql/relational-databases/track-changes/about-change-tracking-sql-server?view=sql-server-ver17>
- [2] Christian Jacob, Gut getrackt ist schon fast migriert, *dotnetpro* 1/2026, Seite 36 ff.
- [3] GitHub-Repository mit dem Quelltext zum Artikel, <https://github.com/Mephisztoe/dnp-sqlchangetracker>
- [4] Microsoft.Extensions.Hosting.WindowsServices, <https://github.com/dotnet/runtime/blob/main/src/libraries/Microsoft.Extensions.Hosting.WindowsServices/src/WindowsServiceLifetimeHostBuilderExtensions.cs>
- [5] Scrutor, <https://github.com/khellang/Scrutor>
- [6] DotNet Environment Variables Guide, www.mykolaaleksandrov.dev/posts/2025/08/dotnet-environment-variables-guide/
- [7] .NET app health checks in C#, <https://learn.microsoft.com/de-de/dotnet/core/diagnostics/diagnostic-health-checks>
- [8] Change Context, <https://learn.microsoft.com/en-us/sql/relational-databases/track-changes/work-with-change-tracking-sql-server?view=sql-server-ver17#set-context-information>
- [9] Steve Ballmer at NET Conference 1999, www.youtube.com/watch?v=8fcSviC7cRM
- [10] Developers Developers Developers Developers, <https://devshaped.com/book>



Christian Jacob

ist Entwickler und Co-Founder der Blue Sky Innovations GmbH. Er unterstützt Unternehmen bei der Einführung moderner Strategien sowie bei Planung und Umsetzung individueller Softwarelösungen auf Basis von Microsoft .NET.
cjacob@blueskyinnovations.de

MEHR ALS NUR PROMPTS

KI-Agenten mit Daten füttern

Wie wir Large Language Models, die auf historischen und unveränderbaren Daten trainiert wurden, ohne teures Neutraining um zusätzliches und aktuelles Wissen erweitern können.

Wer heute LinkedIn öffnet oder durch Tech-Blogs scrollt, kommt an einem Thema nicht vorbei: Künstliche Intelligenz. Es scheint, als hätte sich über Nacht eine neue Berufsgruppe gebildet – viele, die zuvor nur technische Angestellte waren, sind über Nacht zu KI-Experten mutiert, die andere Karrieristen auf LinkedIn über die Möglichkeiten der KI aufklären. Wie sehr diese Selbstvermarktung einzelner KI-Gurus ausartet, die sich berufen fühlen und sich nicht zurückhalten, uns mit Unmengen von Posts ohne nennenswerten Inhalt zu überschwemmen, zeigt sich daran, dass wir analog zu Spam bereits einen Begriff dafür haben: KI-Slop.

Neue Daten, neues Glück! Um zumindest etwas generieren zu können, was der nächste selbsternannte KI-Experte so nicht generieren könnte, stellen wir uns die Frage, wie wir LLMs mit neuen Daten füttern. Um das zu tun, müssen wir auch verstehen, was dabei unter der Haube passiert und auf welche unterschiedlichen Weisen wir diese mächtigen Sprachmodelle (LLMs) und KI-Agenten um neue Daten erweitern können, um sie dadurch auch auf Anwendungsfälle vorzubereiten, die der Standard nicht abdeckt.

Grundlagen

Ein Large Language Model (LLM) ist ein Sprachmodell, das salopp gesagt auf nahezu allen öffentlich verfügbaren Inhalten des Internets trainiert wurde, die in natürlicher Sprache verfasst wurden. Server-Logs sind nicht Teil des Korpus, wohl aber Webseiten, Social-Media-Daten und Ähnliches.

Dass Maschinen Unmengen an verfügbarem Text parsen und bearbeiten (oder, wie man es auf Englisch sagt, crawlen)

ist nichts Neues. Um das Internet abzugrasen, nutzen KI-Labore (wie die von OpenAI oder Google) Web-Crawler. Diese Bots besuchen systematisch Webseiten, folgen Links und laden Inhalte herunter. Solche Suchmaschinen existieren schon seit den späten 1990ern.

Die Grundlage moderner Sprachmodelle bildet das Paper „Attention Is All You Need“ von dem Team um Ashish Vaswani [1], das heute in fast keiner Einführung zu LLMs fehlt. Darin wurde die Transformer-Architektur vorgestellt. Dieses neue Verfahren revolutionierte die Verarbeitung riesiger Textmengen, indem es nicht mehr stur sequenziell liest, sondern kontextuelle Zusammenhänge im Satz parallel gewichtet (Attention-Mechanismus). Schauen wir uns nun an, wie auf dieser Grundlage Modelle trainiert werden.

Phase A: Pre-Training („Grundschule und Gymnasium“)

Im ersten Schritt wird ein „Basismodell“ erstellt. Die Trainingsdaten sind vorhanden. In den folgenden Schritten verarbeiten wir diesen enormen Textkorpus weiter.

- **Tokenisierung:** Der Text wird in kleine Einheiten zerlegt, sogenannte Tokens (oft Silben oder Wortteile). Das Wort „Apfelbaum“ könnte zum Beispiel zu [Ap, fel, baum] werden.
- **Next-Token-Prediction:** In diesem zentralen Teil des Trainings wird dem Modell ein Satzanfang (zum Beispiel „Der Himmel ist ...“) vorgegeben und es muss das nächste Wort vorhersagen (zum Beispiel „blau“).
- **Self-Supervised Learning:** Das Modell vergleicht seine Vorhersage mit dem tatsächlichen Text. Lag es falsch, passt es seine internen Verknüpfungen (Gewichte) an. Dies geschieht Milliarden Mal in Folge.

Am Ende des Pre-Training-Prozesses lässt sich das Verhalten des erarbeiteten Basismodells mit dem eines stereotypen Teenagers vergleichen (vergleiche Kasten **Achtung Stereotypisierung!**). Das Modell kann sich ausdrücken und Fragen beantworten. Manchmal ist es aber auch unhöflich und schroff. Bei manchen Jugendlichen kennt die Kreativität auch keine Grenzen. Sie können also etwas erfinderisch sein, um Sachverhalte so zu schildern, dass sie „elterntauglich“ erscheinen. Im Kontext von LLMs entspräche das dem Halluzinieren, also dem Erfinden von Inhalten.

• Achtung Stereotypisierung!

In den hier präsentierten Beispielen zu den Schritten des LLM-Trainings werden wiederholt Stereotype von Teenagern und Uni-Absolventen verwendet, um die Mächtigkeit von LLMs im Trainingsprozess zu beschreiben. Es ist wichtig zu betonen, dass derartige stereotype Generalisierungen natürlich nicht auf alle Teenager oder alle Uni-Absolventen zutreffen und dass sich somit niemand persönlich angesprochen fühlen sollte.

Phase B: Fine-Tuning („Studium und Berufsausbildung“)

Nachdem wir uns beim Basismodell quasi mit einem Teenager unterhalten haben, müssen wir versuchen, daraus mehr herzustellen. Wir wollen nicht, dass Anwender sich durch eine schroffe Ausdrucksweise angegriffen fühlen oder mit zu vielen Behauptungen konfrontiert werden, die zu wenig kritisch hinterfragt werden. Wir müssen das Modell also weiterentwickeln, um es für die Arbeitswelt geeignet zu machen. Folgende Schritte sind hierfür notwendig:

- **Instruction Tuning:** Menschen schreiben perfekte Frage-Antwort-Paare, um dem Modell die angemessene Form beizubringen, in der Antworten gegeben und Anweisungen ausgeführt werden. Auch hier lässt sich ein Vergleich zur realen Welt ziehen: Viele Professoren investieren Zeit, um ihren Studenten mehr kritisches Denken beizubringen. Aus einem Schüler, der bislang Aufsätze schrieb, soll jemand werden, der in der Lage ist Papers zu schreiben, die wissenschaftlichen Standards gerecht werden.
- **Reinforcement Learning from Human Feedback (RLHF):** Das Modell generiert mehrere Antworten, und ein Mensch bewertet: „Antwort A ist besser als Antwort B“. Das Modell lernt so menschliche Präferenzen (zum Beispiel Hilfsbereitschaft, Sicherheit).

Wenn Modelle trainiert werden, gibt es – analog zu den generellen Gepflogenheiten bei Open-Source- und kommerzieller Software – zwei Möglichkeiten, Informationen zu den dabei genutzten Vorgehensweisen bereitzustellen. Open-Source-Modelle machen die verwendeten Algorithmen und Daten öffentlich zugänglich. Ein Beispiel für ein Open-Source-Modell ist Llama. Open-Source-Modelle können lokal heruntergeladen und auf eigener Hardware genutzt werden. Die meisten der bekannteren Modelle (Opus, Sonnet, GPT und Gemini) sind jedoch proprietär. Geschlossene Modelle werden von den KI-Laboren, die sie generieren, gehostet. Chatbots wie ChatGPT, der Gemini-Chatbot oder Claude sind mehr als „nur“ Modelle. Um sie herum gruppiert finden sich mächtige Services, die viel Funktionalität von KI-Agenten (mehr dazu weiter unten) bereitstellen, zum Beispiel Anfragen an verschiedene Modelle verteilen und weitere Informationen zu den Anfragen laden können (vergleiche Kasten **Wider die babylonische Sprachverwirrung**). Da die Services von OpenAI, Google, Anthropic et cetera proprietär sind, haben wir als Nutzer entsprechend wenig Einsicht darin, wie Anfragen im Detail verarbeitet werden, bevor es zu einer Inferenz (also einer Auswertung) eines Modells kommt.

Metriken

Wenn man über die Leistungsfähigkeit eines Modells spricht, gibt es Metriken, mit denen man Modelle vergleichen kann. Viele Metriken sind bei geschlossenen Modellen geheim; bei Open-Source-Modellen hingegen sind sie bekannt. Gehen wir nun die wichtigsten davon durch:

- **Parameter:** Die „Gehirnmasse“ des Modells. Es sind die gewichteten Verbindungen im neuronalen Netz (Beispiel: GPT-4 hat laut Gerüchten über eine Billion Parameter). Lla-

Wider die babylonische Sprachverwirrung

Viele Begriffe werden von Experten schlampig verwendet. Also bemühen wir uns um mehr Präzision, indem wir die Zusammenhänge konkretisieren. Es gibt Firmen wie Google, OpenAI und Anthropic, die Modelle wie Gemini (Google), GPT (OpenAI) und Opus (Anthropic) bauen. Modelle unterscheiden sich in ihrer Versionierung (zum Beispiel GPT-5 versus GPT-4), aber es gibt auch Modelle, die unterschiedlich mächtig im Hinblick auf die zugrunde liegenden Trainingsdaten sind (zum Beispiel Gemini Pro versus Gemini Flash). Andere Hersteller verwenden für Modelle mit unterschiedlichen Mächtigkeiten unterschiedliche Namen. Anthropic bietet beispielsweise drei Modelle mit den Namen Sonnet, Haiku und Opus an [3]. Modelle werden von den Herstellern als allgemein zugängliche Chatbots bereitgestellt (gemini.google.com (Google), chatgpt.com (OpenAI), claude.ai (Anthropic)). Diese Chatbots nutzen die jeweiligen Modelle und zusätzlich vom Hersteller bereitgestellte Dienste (beispielsweise KI-Agenten), um Anfragen von Anwendern zu beantworten. Modelle werden von den Anbietern auch in Applikationen integriert (zum Beispiel integrieren Microsoft und Google diese in ihre Office-Applikationen), und Entwickler haben die Möglichkeit, auf diese Modelle auch über APIs zuzugreifen. Diese Firmen verwenden auch unterschiedliche Formen der Monetarisierung (zum Beispiel ein Abonnement oder Pay-as-you-go), auf die wir jedoch nicht im Detail eingehen.

ma-3 gibt es in Versionen mit 8 Milliarden (8B) oder 70 Milliarden (70B) Parametern. Mehr Parameter bedeuten meist ein komplexeres Verständnis, aber auch höhere Kosten.

- **Kontext-Fenster:** Das „Kurzzeitgedächtnis“. Diese Metrik bestimmt, wie viel Text das Modell einlesen kann, ohne dabei den Anfang zu vergessen. Diese Metrik wird in Tokens gemessen, zum Beispiel 128k Tokens.
- **Training Loss:** Eine mathematische Kurve, die zeigt, wie oft sich das Modell noch irrt. Ziel ist es, den „Loss“ möglichst weit zu senken.
- **FLOPS (Floating Point Operations):** Ein Maß für die benötigte Rechenleistung während des Trainings.

Modelle zu trainieren ist sehr teuer. Leser, die sich mit LLMs beschäftigen, kennen sicher zahlreiche Berichte, in denen geschätzt wird, wie viel Verlust ein Labor wie OpenAI verursacht. Man benötigt Tausende spezialisierter GPUs (zum Beispiel Nvidia H100). Eine einzige dieser Karten kostet zwischen 30 000 und 40 000 US-Dollar. Niemanden wundert es dann noch, dass Nvidia eines der wertvollsten Unternehmen an der Börse ist. Der Stromverbrauch eines einzigen Trainingslaufs eines großen Modells entspricht dem Jahresverbrauch von hundert Haushalten. Ein Modell zu trainieren kostet einige Millionen US-Dollar. Nur wenige Firmen können Spitzenmodelle von Grund auf trainieren. Einige Experten sehen hierin einen weiteren Grund dafür, warum die USA und China die Führerschaft in der KI haben. Während es in ►

Europa weniger Kapital gibt, um in teure und risikobehaftete Ventures zu investieren, hat das in den USA Tradition, und in China wird Zukunftstechnologie ohnedies subventioniert.

AI-Agent

Ein fertiges Modell ist nun mit einem Absolventen einer Elite-Uni vergleichbar, der das Schicksal von Lucy, gespielt von Drew Barrymore, in „50 erste Dates“ teilt. Lucy leidet an einer anterograden Amnesie und vergisst jeden Tag alles vom Vortag. Würde man einfach Zeit verstreichen lassen, wäre es nur eine Frage der gewählten Zeitspanne, bis dieses Modell versuchen würde, jedes Problem mit antiquierten Ideen zu lösen. Selbst wenn das Modell alles bis zum Knowledge-Cut-off perfekt versteht – was hilft der klügste Ratgeber, wenn er nicht in der Lage ist, neues Wissen zu lernen?

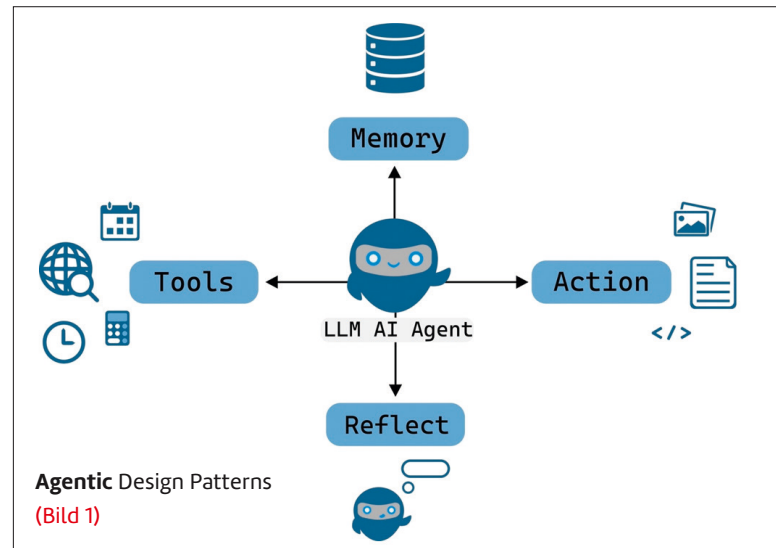
Nehmen wir an, Sie möchten etwas über den Präsidenten der Vereinigten Staaten von Amerika lernen. Doch Sie nutzen ein Modell, dessen letzter Trainingstag vor der jüngsten Wahl stattfand. Ein Modell mit veralteten Daten wird immer davon ausgehen, dass Joe Biden US-Präsident ist. Geben Sie dem Modell dagegen einen Kontext, wird dieser in die Antwort eingebaut: „Donald Trump hat die Wahlen in den USA gewonnen. Was könnte das nun für die Zukunft des Landes bedeuten?“ Ein LLM könnte in diesem Kontext zusätzliche Informationen für die Inferenz nutzen. Würde man daraufhin ohne Kontext erneut eine Anfrage stellen, wäre die Zusatzinformation nicht verfügbar. Da das Modell nicht weiß, dass es um Donald Trump geht, könnte es wieder annehmen, Joe Biden sei es (vergleiche Kasten **Alle Macht dem Kontext**).

Wenn die LLM also eine Blackbox ist, muss man sich selbst darum kümmern, das Modell leistungsstärker zu machen. Ei-

• Alle Macht dem Kontext

Der Kontext ist mächtiger, als viele sich vorstellen. Ein Kontext ist nichts anderes als eine Fülle an Zusatzinformationen zu einem Prompt, also einer Anfrage an ein LLM. Viele Design-Patterns arbeiten damit. Sicher haben Sie schon einmal LLMs für sich angepasst. Sie können einem GenAI-Chatbot Details zu Ihrer Person mitteilen. Je detaillierter Sie den Chatbot mit Informationen über sich und Ihre Ziele ausstatten, desto besser wird er seinen Output anpassen, um Themen in Ihrer Sprache zu schildern. Tragen Sie im Custom-Instruction-Feld ein, dass Sie Programmierer sind. Sie können gerne auch mehr experimentieren und Zusätze wie „Explain everything to me as if I were 5 years old“ oder „be sarkastic, funny and rude if needed“ hinzufügen. Die Sprache wird sich entsprechend verändern. Das LLM wird Ihnen folglich die Sachverhalte in der Art und Weise zurückgeben, um die Sie es bitten.

Wenn Sie nun beispielsweise den gesamten vorangegangenen Chatverlauf als Kontext mitgeben, kann ein Chatbot viel daraus ableiten, um Sie besser zu beraten. Dieser Aspekt ist wesentlich, um auch zu verstehen, wie wir mit KI-Agenten Informationen an eine KI übermitteln, die bessere Ergebnisse liefert.



ner der wichtigsten Aspekte ist hier, tagesaktuelle Informationen zu integrieren. Bei bekannten Chatbots ist es auch bereits Standard, dass sie ihr Wissen mit tagesaktuellen Daten ergänzen. Sehen wir uns einmal die Design-Patterns an.

Design-Patterns

Wer sich aktuell mit der Entwicklung von KI-Agenten beschäftigt, kommt an KI-Pionier Andrew Ng kaum vorbei. Geht es nach ihm, sollten wir uns weniger darauf konzentrieren, auf das nächste große Modell-Update (wie zum Beispiel GPT-6 – dieses Modell gab es noch nicht, während dieser Artikel verfasst wurde) zu warten, und stattdessen mehr Energie in Agentic Workflows investieren. Ein schwächeres Modell, eingebettet in einen intelligenten Workflow, schlägt oft ein stärkeres Modell, das „Zero-Shot“ (also ohne Zwischenschritte) antwortet.

Ng identifiziert dabei Kernmuster, die moderne Agentensysteme auszeichnen. Im Folgenden betrachten wir vier essenzielle Bausteine, die dabei hilfreich sind, um diese Agenten zu entwickeln (vergleiche jedoch auch den Kasten **Alternative Design-Patterns**). Diese vier Design-Patterns sind auch in **Bild 1** dargestellt.

- Reflection
- Tool Use
- Memory
- Multi-Agent Collaboration

1. Reflection (Selbstreflexion)

In einfachen LLM-Interaktionen akzeptieren wir oft den ersten Output des Modells. Bei der Reflexion wird das Modell jedoch angewiesen, seine eigene Ausgabe kritisch zu prüfen, bevor sie finalisiert wird. Das kann durch das Modell selbst erfolgen oder aber auch durch ein anderes Modell (siehe dazu auch das Design-Pattern Multi-Agent-Kollaboration).

Stellen Sie sich dies als Unit-Test oder als Code-Review für generierten Text vor. Der Agent generiert einen Entwurf, betrachtet ihn („Reflect“) und korrigiert Fehler oder Logiklücken. Dies erhöht die Robustheit und Qualität des Codes

oder der Texte massiv, da das Modell die Möglichkeit zur Selbstkorrektur erhält.

Um auch eine praktische Anwendung zu skizzieren, stellen Sie sich vor, dass Sie Anwendern Einsichten in die Finanzwelt via LLMs zur Verfügung stellen. Aus regulatorischen Gründen muss sichergestellt sein, dass der Anwender die Antwort nicht als Ratschlag, bestimmte Produkte zu kaufen, missinterpretieren kann. Wenn Sie nun jede Ausgabe mit einem Dienst prüfen – auf Englisch auch Guardrails –, wäre dies eine Anwendung des Reflection Pattern.

2. Tool Use (Werkzeugnutzung)

Ein Sprachmodell ist, isoliert betrachtet, ein Gehirn ohne zusätzliche Sensoren. Durch Tool Use (auch als Function Calling bezeichnet) verleihen wir dem Agenten Sinne, um neue Eindrücke wahrzunehmen, die zusätzlich zu seinem Erfahrungswissen verarbeitet werden können.

Der Agent erkennt, wann er sein eigenes Wissen mit weiteren Informationen anreichern muss, um externe Aktionen auszuführen – sei es eine Websuche, eine Datenbankabfrage oder die Ausführung von C#-Code in einer Sandbox. Aus einem reinen Textgenerator wird so ein handlungsfähiger Assistent, der reale Aufgaben im Systemkontext erledigen kann. Um auf das Beispiel mit dem Präsidenten zurückzukommen: Hier würde in einem Workflow eruiert werden, wer der aktuelle US-Präsident ist, und der KI-Agent würde dabei nicht die Daten verwenden, mit denen er trainiert wurde, sondern die notwendigen Informationen aus dem Internet abfragen.

3. Memory (Gedächtnis)

Klassische API-Calls an LLMs sind zustandslos. Ein echter Agent benötigt jedoch ein Gedächtnis, um komplexe, mehrstufige Aufgaben zu lösen.

Hierbei unterscheiden wir oft zwischen Kurzzeitgedächtnis (Kontext des aktuellen Threads) und Langzeitgedächtnis (zum Beispiel persistiert in Vektordatenbanken via RAG). Memory erlaubt dem Agenten, aus vergangenen Interaktionen zu lernen, Benutzerpräferenzen zu speichern und den Faden auch über lange Zeiträume hinweg nicht zu verlieren. Damit bildet das Memory Pattern das Fundament für Personalisierung und Kontinuität.

Alternative Design-Patterns

Diese vier in diesem Artikel vorgestellten Entwurfsmuster Reflection, Tool Use, Memory und Multi-Agent Collaboration sind etablierte Standards, die jedoch keineswegs in Stein gemeißelt sind. Da die Forschung an KI-Agenten ein dynamisches Feld ist, entstehen kontinuierlich neue Ansätze. Es lohnt sich daher, über diese Auswahl hinauszublicken und weitere Architekturmuster für den spezifischen Anwendungsfall zu evaluieren. Ein Entwurfsmuster könnte im Endeffekt jedes Verfahren sein, bei dem Sie anhand von Zusatzinformationen ermitteln, wie Sie ein LLM – etwa durch geeigneten Kontext – besser nutzen.

Wer mit Chatbots wie ChatGPT und Gemini arbeitet, wird merken, dass sie sich auf frühere Konversationen berufen. Das ist ein ausgezeichnetes Beispiel für die Anwendung dieses Entwurfsmusters. Auch hier gibt es wieder eine Überschneidung mit einem anderen Muster, nämlich mit Muster zwei, dem Tool Use. Wir könnten beispielsweise alle Chats in einer Datenbank oder in einer Applikation wie Notion speichern. Die Antworten unterschiedlicher Modelle können dann ebenfalls dort gespeichert werden. Wir könnten dann auch diese vergangenen Chats als Datenquelle bereitstellen.

4. Multi-Agent Collaboration (Kollaboration)

Warum einen Generalisten alles übernehmen lassen, wenn man Spezialisten haben kann? Mit diesem Muster simulieren wir ein Entwicklungsteam.

Ein Agent übernimmt die Rolle des „Software Architect“, ein anderer ist „Coder“, ein dritter „Reviewer“. Diese Agenten kommunizieren in einer Schleife miteinander. Durch diese Rollentrennung (Prompt Engineering per Persona) fokussiert sich jedes Modell auf einen Teilaspekt. Das Ergebnis ist oft präziser, da die gegenseitige Kontrolle Halluzinationen reduziert und komplexe Probleme in handhabbare Teilaufgaben zerlegt.

Auch bei den Modellen gibt es absehbar einen Trend zur Spezialisierung. Hierbei testen Benchmarks, wie gut existierende Modelle einzelne Probleme lösen können. Solche Benchmarks sind allerdings mit Vorsicht zu genießen. Die Ersteller von Modellen kennen die Testverfahren und trainieren ihre Modelle darauf, möglichst gute Resultate zu erzielen. Das ist vergleichbar mit einer Situation, in der man den Schülern die Prüfungsfragen vorab zukommen lässt. Nichtsdestotrotz zeigen sich Trends zur Spezialisierung. Anthropic baut Modelle, die sich besonders zum Erstellen von Code eignen. Viele Softwareentwickler tendieren dazu, sich ausschließlich auf die Modelle dieses Herstellers zu verlassen. Google und OpenAI liefern sich Duelle, um zu zeigen, wer die bessere Reasoning-Performance aufweist (also die Fähigkeit, zu argumentieren). In einem Anwendungsfall könnte man entsprechend den Code von einem Anthropic-Modell schreiben lassen und Gemini oder GPT für die Reflection verwenden.

Nachdem wir die ersten Design-Patterns kennengelernt haben, schauen wir uns nun an, wie wir zusätzliche Daten einbinden. Wir evaluieren dabei drei Verfahren: RAG, MCP und das Gemini-API.

Retrieval-Augmented Generation (RAG)

RAG ist eine Technik, die die Stärken eines großen Sprachmodells (LLM) wie GPT-5 mit externen privaten oder aktuellen Daten verknüpft, ohne das Modell neu trainieren zu müssen.

Vergleichen wir unser fertiges LLM erneut mit unseren Uni-Absolventen. Er weiß viel, aber sein Wissen ist auf einem alten Stand (Trainings-Cut-off), und wenn er etwas nicht weiß, erfindet er manchmal Fakten (Halluzinationen). RAG verwandelt dies in eine Open-Book-Prüfung. Der Absolvent darf, bevor er antwortet, in einem vertrauenswürdigen Lehrbuch nachschlagen, um seine Wissenslücken auszugleichen.

RAGs können auch ein weiteres Thema lösen: Natürlich dürfen beim Trainieren von Modellen keine privaten Daten ►

verwendet werden. Wenn Sie also Daten auf internen Firmen-Wikis, in E-Mails oder in Datenbanken haben, sind diese natürlich nicht in einem LLM enthalten. Sie können diese Informationen jedoch zusätzlich für LLMs bereitstellen. Gehen wir nun den Prozess durch:

- **Retrieval (Abruf):** Der Nutzer stellt eine Frage. Das System sucht in der von Ihnen bereitgestellten Datenbank nach den relevantesten Textabschnitten, die zur Beantwortung der Frage erforderlich sind.
- **Augmentation (Erweiterung):** Die gefundenen Informationen (Kontext) werden zusammen mit der ursprünglichen Frage in einen Prompt integriert.
- **Generation (Generierung):** Das LLM erhält den erweiterten Prompt und erstellt daraus eine präzise Antwort.

Um RAG zu implementieren, müssen Sie also eine Vektordatenbank aufbauen. Man könnte auch sagen, dass Sie die Informationen, die Sie hinzufügen wollen, vorkauen müssen, damit sie in einem Format vorliegen, das ein LLM versteht. Schauen wir uns den technischen Workflow im Detail an.

1. Datenvorbereitung (Ingestion und Chunking)

Da LLMs nur eine begrenzte Menge an Text auf einmal verarbeiten können (Kontextfenster), müssen Sie Ihre Dokumente (PDFs, Word, HTML) in kleine Stücke (sogenannte Chunks) zerlegen.

2. Vektorisierung (Embeddings)

Computer verstehen Text nicht direkt; sie verstehen Zahlen. Um die Daten bereits vorab zu verarbeiten, wandeln Sie die Text-Chunks mithilfe eines Embedding-Modells (zum Beispiel OpenAI oder HuggingFace) in lange Vektoren (Zahlenreihen) um.

- Ähnliche Inhalte haben ähnliche Vektoren.
- Beispiel: „Hund“ und „Welpen“ liegen im Vektorraum nah beieinander.

3. Speicherung (Vektor-Datenbank)

Diese Vektoren speichern Sie in einer Vektordatenbank (Vector Store). Bekannte Anbieter sind Pinecone (cloudbasiert), ChromaDB oder FAISS (Open Source, lokal nutzbar), Weaviate oder Milvus. Ab diesem Zeitpunkt haben wir also Daten, die wir nun nach Belieben für Inferenzen (das ist die Erstellung von Antworten, die auf einer Anfrage basieren) bereitstellen können.

Fans der TV-Serie „Doctor Who“ kennen vielleicht die Oods. Das ist eine Alien-Spezies, die sich durch ein erweitertes Gehirn auszeichnet. Und genau als das könnte man die Vektordatenbank betrachten: als ein erweitertes Gehirn, das mit dem Haupthirn zusammenarbeiten kann.

Schauen wir uns nun den Prozess an, mit dem Prompts mithilfe eines RAG verarbeitet werden. Wenn ein Nutzer eine Frage stellt, werden folgende Schritte durchgeführt:

- Die Frage wird ebenfalls in einen Vektor umgewandelt. Das ist das Standardverfahren und würde auch ohne RAG passieren.

- Statt den Vektor direkt an das LLM zu schicken, wird in der Vektordatenbank eine Ähnlichkeitssuche durchgeführt: „Welche gespeicherten Chunks liegen mathematisch am nächsten an der Frage?“ Die besten Treffer werden zurückgegeben.
- Das System schickt nun den Prompt an das LLM. Der Prompt umfasst die eigentliche Anfrage sowie den Kontext. Der Kontext enthält auch die Treffer aus der Abfrage der Objektdatenbank.

Wenn Sie eine RAG-Applikation entwickeln wollen, benötigen Sie ein Framework, das die notwendige Funktionalität für RAGs sowie für Modelle bereitstellt. LangChain ist wohl das populärste Framework, um LLMs mit Datenquellen zu verbinden. Es gibt jedoch auch andere Frameworks als Alternativen. Auch GUIs wie LangFlow oder n8n sind Beispiele für Alternativen. Gehen wir aber noch die anderen Schnittstellen durch, bevor wir uns Beispielcode anschauen.

Model Context Protocol (MCP)

MCP ist ein offener Standard, der (maßgeblich von Anthropic initiiert) entwickelt wurde, um das grundlegende Verbindungsproblem zwischen KI-Assistenten und Datenquellen zu lösen. Der „Uni-Absolvent“, also das LLM in unserer Referenz, erhält nun quasi eine Liste mit Telefon-Jokern. Das bedeutet, dass er bei jeder Anfrage Experten zu spezifischen Fragen befragen kann.

Das Telefon kann hier als einheitlicher Draht zu spezifischen Informationen gesehen werden. Bevor es MCP gab, war die Verbindung von KI mit Daten ein Chaos aus maßgeschneiderten Lösungen:

- Wenn Sie wollten, dass ChatGPT auf Ihre Google Docs zugreifen kann, brauchten Sie eine spezifische Integration.
- Wenn Sie wollten, dass Claude auf Ihre PostgreSQL-Datenbank zugreift, wäre eine vollständige Neuentwicklung nötig.
- Wenn Sie wollten, dass eine IDE (zum Beispiel VS Code) auf Jira zugreifen kann, brauchten Sie wieder etwas Neues.

Das ist das m-zu-n-Problem: Jedes KI-Modell musste für jede Datenquelle einen eigenen „Adapter“ entwickeln. MCP standardisiert diese Verbindung. Entwickler bauen einen MCP-Server für ihre Datenquelle (zum Beispiel Notion). Jede MCP-fähige KI-Anwendung (der „Client“) kann sich nun sofort mit diesem Server verbinden und die Daten nutzen, ohne dass eine spezielle Integration entwickelt werden muss. Das Protokoll besteht somit aus zwei Hauptakteuren:

- **Der MCP-Host (Client):** Das ist die Anwendung, in der Sie mit der KI sprechen (zum Beispiel die Claude-Desktop-App oder die Cursor-IDE). Diese Applikation möchte die zusätzlichen Daten haben.
- **Der MCP-Server:** Dies ist ein kleines Programm, das Zugriff auf eine bestimmte Datenquelle hat (zum Beispiel Google Drive MCP-Server oder Slack MCP-Server). Er „serviert“ die Daten in einem Format, das der Host versteht.

MCP erlaubt der KI nicht nur das Lesen, sondern definiert auch drei Interaktionsmöglichkeiten:

- **Resources (Ressourcen):** Das sind Daten, welche die KI lesen kann, konkret zum Beispiel die Inhalte einer Datei, Logs aus einer Datenbank oder die zuletzt erhaltene E-Mail. Solche Daten werden oft als Kontext für die KI herangezogen.
- **Tools (Werkzeuge):** Funktionen, die die KI ausführen kann (Action-Taking). Beispiel: „Erstelle ein Ticket in Jira“, „Speichere diese Datei“, „Führe diese SQL-Query aus“.
- **Prompts:** Das sind vordefinierte Vorlagen für Interaktionen. Beispiel: Ein „Code Review“-Prompt, der automatisch die richtigen Dateien lädt und die KI instruiert, wie sie den Code prüfen soll.

Gemini File API

Gemini File API ist ein Dienst von Google, der es Entwicklern ermöglicht, große Dateien (Dokumente, Bilder, Audio, Video) hochzuladen, damit sie von Gemini-Modellen verarbeitet werden können.

Normalerweise haben LLMs ein begrenztes Kontextfenster, und das direkte Einfügen riesiger Datenmengen in einen Prompt ist oft technisch nicht möglich oder zu teuer. Das File API löst dieses Problem, indem es die Dateien auf Google-Servern zwischenspeichert und dem Modell nur den Zugriff darauf gewährt (als Referenz), statt die rohen Daten bei jeder Anfrage mitzusenden.

Sie können das File AP nutzen, wenn Ihre Dateien zu groß sind, um sie direkt „inline“ (als Base-64-String) in den Prompt zu packen.

- **Kleine Dateien (< 20 MB)** können direkt im Request übermittelt werden (kein File-API nötig).
- **Große Dateien (> 20 MB)** müssen über das File-API hochgeladen werden.

Das API ist multimodal und versteht diverse Datentypen:

- **Dokumente:** PDF, CSV, HTML, Markdown, Text, Code (py, js et cetera)
- **Bilder:** PNG, JPEG, WEBP, HEIC
- **Audio:** MP3, WAV, AAC, FLAC
- **Video:** MP4, MOV, MPEG, AVI

Der Prozess besteht immer aus drei Schritten: Hochladen – Verarbeiten – Prompten.

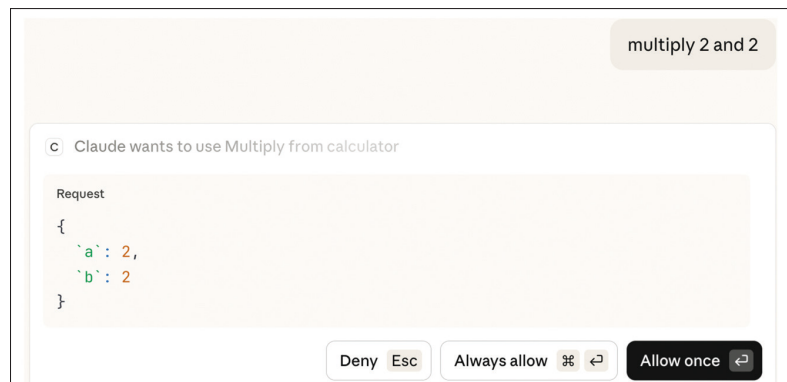
Ein deutlicher Nachteil des Google-API ist, dass nur Google-AI-Modelle zur Verfügung stehen. Zum Zeitpunkt der Abfassung des Artikels war dieses API erst wenige Monate alt. Die Zeit wird zeigen, ob es sich als Standard durchsetzt.

Codebeispiel

Für diesen Artikel wurden drei Codebeispiele erstellt, die jede Schnittstelle abdecken und in einem Git-Repo unter [2] verfügbar sind. Ziel dieser Code-Samples ist es, in möglichst einfacher Weise zu veranschaulichen, wie diese Schnittstellen funktionieren. Für die Erstellung wurde Opus 4.5 von Anthropic verwendet, und als Programmiersprache dient Python. Da Python die verbreitetste Programmiersprache für die Einbindung von KI ist, ist anzunehmen, dass für Python auch die meisten Trainingsdaten im Netz verfügbar sind. Die

Wahrscheinlichkeit ist hoch, dass die Ergebnisse in diesen Anwendungsfällen mit Python besser sind als mit weniger verbreiteten Sprachen. Beim Erstellen des Referenzcodes musste einiges nachbearbeitet werden. Versionen des generierten Codes passten nicht zusammen. Teilweise lag das auch daran, dass sich die Schnittstellen in vielen Bibliotheken wie zum Beispiel LangChain verändert haben und natürlich zahlreiche Trainingsdaten vorliegen, die noch die alten Schnittstellen nutzen. Leser, die den Code nun für ihre bevorzugten Sprachen erstellen, können deutlich unterschiedliche Ergebnisse erzielen.

Alle Beispiele sind als Jupyter Notebooks implementiert. Das heißt, man kann jeden Schritt durchführen. Allerdings werden für einzelne Beispiele API-Keys von Modellen benötigt; Leser ohne Schlüssel können diese Beispiele nicht selbst

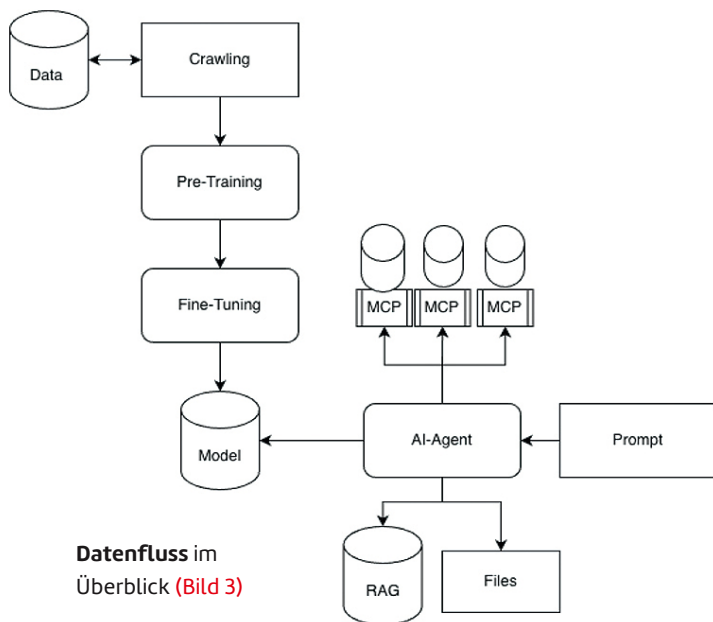


Zugriff von Claude via MCP (Bild 2)

ausführen. Wir beschreiben jedoch den Ablauf der Beispiele, und jeder Leser kann damit den Quellcode selbst ausführen.

Für das RAG-Beispiel nutzen wir LangChain. Wir senden dem LLM einen Prompt mit der Frage, wie das Wetter in Eriwan ist. Die Antwort ist, wie zu erwarten, dass das LLM keine aktuellen Wetterdaten hat. In einer bereitgestellten Textdatei steht, dass es heute in Eriwan sonnig ist. In weiterer Folge führen wir den Ablauf durch, den wir im Artikel beschrieben haben. Wir wandeln das Textfile in Tokens um und fügen die codierten Daten einer Vektordatenbank hinzu. Da wir im Beispiel nur den Ablauf veranschaulichen, verwenden wir eine In-Memory-Datenbank. Wir bauen einen Abfrage-Workflow auf, der die Vektordatenbank einbindet. Wir schicken zwei Prompts: einen mit der Frage nach dem Wetter in Eriwan und einen mit der Frage nach dem Wetter in Paris. Der zweite Prompt ist eine Kontrollinstanz. Wie erwartet bekommen wir die Antwort, dass es in Eriwan sonnig ist, weil dies in der Vektordatenbank so auf Grundlage der verarbeiteten Textdatei beschrieben ist. Wir bekommen aber keine Aussage zum Wetter in Paris.

Das MCP-Beispiel stellt einen Server bereit, auf dem wir einfache Mathematikaufgaben lösen können: Addieren und Multiplizieren. Am Code zeigt sich auch, wie stark sich das MCP-Szenario vom RAG-Szenario unterscheidet. Das Herzstück dieser Lösung ist die Datei `calculator_server.py`, die ►



diese Funktionalität bereitstellt. Schauen wir uns einen Ausschnitt der Datei an:

```
@mcp.tool()
def add(a: int, b: int) -> int:
    """Add two numbers together."""
    return a + b

@mcp.tool()
def multiply(a: int, b: int) -> int:
    """Multiply two numbers together."""
    return a * b
```

Wir sehen also, dass dieser Code Annotationen über die Methoden hat, die sie als solche für Client-Anwendungen definieren. Die einfachste Weise, diese Funktionalität zu testen, ist, diesen Server in die Claude-Desktop-Applikation einzubinden, die man auf einem Betriebssystem installieren kann. Diese Desktop-Applikation listet in der `claude_desktop_config.json`-Datei alle MCP-Server auf, die sie als Client nutzen kann. Der folgende Zusatz bindet unseren Code ein, wobei von Ihnen natürlich der Pfad zu verwenden ist, in dem Sie die Dateien abgelegt haben. In diesem Beispiel wird gezeigt, wie der Server initialisiert werden kann. Der Python-Interpreter wird gestartet und der Server als Parameter übergeben.

```
"calculator": {
  "command": "PATH/.venv/bin/python",
  "args": ["PATH/calculator_server.py"]
},
```

Wenn Sie nun, wie in **Bild 2** gezeigt, zum Beispiel die Anfrage „Multiply 2 by 2“ stellen, erkennt das LLM, dass Sie eine Mathematikaufgabe durchführen wollen, und bindet den Server ein.

Letztlich gibt es auch noch ein Codebeispiel für Google File Gemini. Wir laden ein Bild hoch und fragen anschließend,

was das Modell sieht. Das Modell gibt zurück, dass es eine Katze sieht. Der Gemini-App-Code ist der bei Weitem am einfachsten zu verstehende, da vieles automatisiert ist. Der große Nachteil ist natürlich, dass Sie sich damit auch auf einen Anbieter einschränken, während es eine Vielzahl von Modellen gibt, die andere Standards unterstützen.

Zusammenfassung

Bild 3 fasst alles zusammen, was wir in diesem Artikel beschrieben haben. Es gibt verschiedene LLMs, die wir als Basis nutzen können. Es wurden drei Verfahren vorgestellt, über die wir eine Anfrage mit zusätzlichen Daten einreichen können. Diese drei Methoden haben unterschiedliche Vor- und Nachteile.

- **RAG** ist die Technik „Hole Daten und füttere sie ins LLM ein.“ Das bedeutet aber auch, dass wir zusätzliche Jobs brauchen, die Text in ein kompatibles Vektorformat umwandeln.
- **MCP** ist die Infrastruktur beziehungsweise Schnittstelle: ein Standardweg, über den das LLM und die Datenquelle miteinander kommunizieren. LLMs werden durch MCP also zu mehr als nur zu Modellen, die Antworten zurückliefern. MCPs ermöglichen uns auch, Aktionen in Zielsystemen auszulösen.
- **Gemini API** ist eine spezifische Schnittstelle für Gemini, über die Daten einfach hochgeladen werden können. Diese ist die einfachste der drei Schnittstellen. Sie ist aber auf Google Gemini beschränkt.

Wie Sie letztlich Ihrem LLM zusätzliche Daten beifügen, hängt auch von Ihren Anforderungen ab. Sie können beispielsweise Ihre Firmendokumente einfach über Google hochladen und durchsuchen lassen. Wenn Sie sich nicht auf Gemini als Modell einschränken wollen, können Sie die Dokumente auch in ein Vektorenformat umwandeln und dieses Ergebnis ebenfalls bereitstellen. Mit MCP würden Sie Applikationen einbinden. Sie könnte beispielsweise E-Mail-Server, Kalender und weitere Applikationen einfach mit Clients sprechen lassen und so mehr Informationen über diese Schnittstellen absaugen. ■

[1] Ashish Vaswani et al., *Attention Is All You Need*,

<https://arxiv.org/pdf/1706.03762>

[2] Beispielcode zum Artikel,

<https://github.com/StefanPapp/dnp-ai-article>

[3] Anthropic-Modellkategorien, <https://platform.claude.com/docs/en/about-claude/models/overview>



Stefan Papp

arbeitet als freiberuflicher Big-Data-Architekt und -Evangelist und begleitet zahlreiche Kunden dabei, sich mit der Analyse von neuen Datenquellen selbst neu zu definieren.

DWX ACADEMY

Hands-on Workshops
und Weiterbildung für
Software-Entwickler
und -Architekten

Software-Architekturen und

Software-Qualität

Architektur für .NET-Anwendungen	(3 Tage)
Requirements Engineering für Einsteiger	(2 Tage)
Clean Code und Software Design für .NET Developer	(3 Tage)
Clean Code für JavaScript/TypeScript-Entwickler	(3 Tage)
Softwarequalität beurteilen und verbessern	(3 Tage)
Refactoring und Unit Testing	(3 Tage)
Testen mit Playwright	(2 Tage)
Sicherheit für ASP.NET Core-Applikationen	(1 Tag)

REMOTE TRAININGS MÖGLICH!



Weitere Informationen auf developer-world.de/dwx-academy
Preise auf Anfrage!

Deine Ansprechpartnerin: Susanne Herl • +49 731 88005-8835 • susanne.herl@ebnermedia.de



WIE SOFTWAREENTWICKLUNG NICHT FUNKTIONIERT

Endlich gescheitert!

Wenn Projekte scheitern, ist das nicht wirklich erfreulich – manchmal aber doch.

Wenn ich in Kundenprojekten tätig bin, nehme ich die unterschiedlichsten Rollen ein – oftmals aber die des Architekten. So war es auch zuletzt wieder. Ein Architekt hat natürlich eine ganze Menge an Aufgaben, die sich grob unterteilen lassen in die technische und die fachliche Architektur. Die technische Variante kümmert sich um die tollen Entwürfe, Architektur- und Technologieentscheidungen und eben alles das, was Entwicklern Spaß macht. Der fachliche Teil ist aus Sicht eines Entwicklers meist langweiliger, geht es doch um Dinge wie Anforderungen, Business-Meetings und zahlreiche Abstimmungsgespräche. Als guter Architekt muss ich eine Architektur erstellen, die zu den Bedürfnissen von Kunde, Projekt und Unternehmen passt. Um das zu erreichen, muss ein Architekt stets einen guten Überblick über die Anforderungen eines Projekts haben, und das ist mit einer der Teile, die mir am meisten Spaß machen. Aus diesem Grund weiß ich in meinen Kundenprojekten meist sehr gut über die technischen, aber auch fachlichen Anforderungen Bescheid.

In besagtem Projekt lauschte ich in einem Meeting des Projektmanagements mal wieder gespannt nach neuen Anforderungen, um diese rechtzeitig ins Projekt integrieren zu können. Dabei kam ein neues Funktionspaket für einen großen Energiekunden zur Sprache, gespickt mit Anforderungen, die im Bereich von Vertrieb und Geschäftsführung wohl schon lange bekannt waren, ich aber zum ersten Mal gehört habe. Neugierig fragte ich nach, was das für ein Anforderungspaket sei und wie die Projektparameter dazu aussehen würden. Der Vertriebsleiter entgegnete mir, dass wir dieses Funktionspaket bräuchten, um einen großen Auftrag dieses Kunden im nächsten Jahr zu bekommen, und dass die Deadline schon Ende nächsten Monats sei. Der anwesende Entwickler klappte sein Notebook zu und schaute mich entnervt an. Etwas verwundert durchforstete ich unser internes Entwickler-Backlog, konnte jedoch kein einziges Ticket dazu finden. „Falls du im Backlog suchst, David, die sind da noch nicht drin, Tanja pflegt die im Lauf der Woche nach.“ Es gab also ein definiertes Aufgabenpaket, sogar eine Deadline, und weder der Kollege noch ich als Architekt hatten je davon gehört. Ich fragte nach, auf welcher Basis diese Deadline geschätzt wurde, da Schätzungen ja normalerweise von den Entwicklern oder im absoluten Notfall vom Architekten gemacht werden sollen. Noch während ich sprach, unterbrach mich der Geschäftsführer und sagte „Herr Tielke, das hier muss jetzt schnell gehen, und deshalb müssen wir da jetzt unkonventionell herangehen.“ Um die Geschichte etwas abzukürzen: Das aktuelle



Funktionspaket, welches diese Woche live gehen sollte, ist natürlich nicht fertig geworden. Die letzten Wochen waren ein absoluter Kampf, wir haben massenweise Schulden im Projekt aufgebaut und ein Entwickler hat zum Monatsende gekündigt, weil er auf diese Art des Arbeitens keine Lust mehr hat.

Darüber hinaus ist der Kunde massiv verärgert, weil das Versprechen des Vertriebs nicht eingehalten werden kann, und wird das große Projekt im nächsten Jahr nicht durchführen.

Wer ist jetzt schuld an der ganzen Misere? Natürlich mal wieder die Softwareentwicklung ...

Kompetenzen

In der professionellen Softwareentwicklung gibt es immer klare Kompetenzen. Die Fachseite und der Product Owner sind für das Erstellen der vollständigen und korrekten Anforderungen zuständig, die Entwickler ermitteln die Aufwände und verantworten die Qualität der Umsetzung. Die Funktionsliste mit den Schätzungen gehen dann an das Projektmanagement, welche daraus einen Projektplan mit Meilensteinen erstellt – so einfach ist die Softwareentwicklung. Erst wenn der Projektplan steht, wenn die Meilensteine mit den verfügbaren Ressourcen der Entwicklung abgeglichen sind, steht ein Endtermin fest, welcher dem Kunden mitgeteilt werden kann. Hier dagegen wurde – wie so oft – das Pferd von hinten aufgezäumt. Der Kunde kommt mit einer Deadline, nennt ein benötigtes Funktionspaket, und unter dem Druck von Umsatz oder Nachfolgeprojekten werden die Vertriebsabteilungen plötzlich zur One-Man-Show.

Auswirkungen

„Warum macht ihr das schon wieder?“, entfuhr es meinem Entwicklerkollegen damals in besagtem Meeting. Genau dieses „Warum?“ geistert mir immer wieder durch den Kopf, wenn ich solche Situationen erlebe. Ich schreibe in Artikeln oft: „Ich verstehe da beide Seiten ...“ – denn bei meiner Arbeit ist mir stets wichtig, beide Seiten zu verstehen und zwischen ihnen zu vermitteln. In diesem Fall hören Sie den Satz jedoch nicht von mir. Diese Art der Entwicklung erzeugt nur Probleme und ist absolut vermeidbar.

In besagtem Projekt waren sowohl das Backlog als auch der Projektplan sehr gut gefüllt, und freie Kapazitäten gab es keine. Auf Druck der Geschäftsführung musste dann umgeplant werden: Andere wichtige Kundenfunktionen wurden hintangestellt, Refactorings verschoben, Urlaube gestrichen und dringend benötigte Fortbildungen ebenfalls gestrichen. Zu-

sätzlich wurde auch die Implementierung von zahlreichen Problemen begleitet: Die eilig ermittelten Anforderungen waren unvollständig, es gab zahlreiche Änderungswünsche im laufenden Projekt, und am Ende haben wir den gewünschten Funktionsumfang nicht umsetzen können – von den erforderlichen Tests wollen wir erst gar nicht reden. Als Erstes hat der Kunde verloren, weil er ein Software-Tool bald nicht mehr einsetzen kann, da es den gesetzlichen Anforderungen nicht mehr entspricht. Der Vertrieb hat zwar mit dem aktuellen Projekt Umsätze erzielt, aber den Kunden für Folgeprojekte verloren und aufgrund der Repriorisierung zahlreiche andere Kunden verärgert. Der Geschäftsführer hat an allen Seiten verloren: Umsatz, Qualität, Kunden, Mitarbeiter.

Trotzdem werten wir aus Sicht der Softwareentwicklung dieses Desaster als Erfolg.

Als Erfolg?

Jawohl, als Erfolg! Hier haben einfach viele Stellen nicht richtig gearbeitet: der Vertrieb, der Geschäftsführer, die Product Owner und die diversen Leiter der Abteilungen: Jeder hier hätte sagen müssen: „Nein, so geht es nicht.“ Auch wenn wir von Seiten der Entwicklung mehrfach genau dieses Nein gegeben haben, wurde darauf keine Rücksicht genommen und das Projekt durch Beschluss des Geschäftsführers trotzdem begonnen. Dieses Muster erlebe ich leider sehr oft in Projekten, und oftmals versucht die Entwicklungsabteilung dann wirklich alles, um das Projekt trotz der widrigen Umstände durchzuziehen. Ist es trotzdem erfolgreich, weil viele Überstunden geleistet, an der Qualität gespart oder fadenscheinige Tricks genutzt wurden, wird genau dieses zerstörerische Verhalten in der Softwareentwicklung belohnt.

„Softwareentwicklung funktioniert so nicht“ – das war meine Warnung, als wir das Projekt auf Beschluss des Geschäftsführers trotzdem durchziehen mussten, und zu dieser Aussage stehe ich auch heute. Wenn wir Software entwickeln, gibt es klare Kompetenzen und Verantwortlichkeiten, und es darf keine Situation entstehen, bei der eine Rolle die Fehler der anderen kompensiert.

Verstehen Sie mich an dieser Stelle bitte nicht falsch: Ich sage ausdrücklich nicht, dass Sie Kollegen nicht helfen oder deren Fehler nicht ausbügeln sollen, sondern nur, dass dies nicht zur Regel werden darf. Schon in der Vergangenheit gab es in diesem Projekt ähnliche Situationen, bei denen uns zusätzliche Teilprojekte in die Entwicklung „hineingedrückt“ wurden. Wir haben dann alles unternommen um das eigentlich Unmögliche doch noch zu schaffen, haben viele Schulden aufgebaut, sind Kompromisse eingegangen und haben zahllose Überstunden gemacht. Das Ende vom Lied war, dass die Vertriebsmannschaft über den grünen Klee gelobt wurde und wir uns im nächsten Sprint bereits rechtfertigen mussten, warum wir schon wieder ein Refactoring benötigen.

Continuous Improvement

Wenn Sie treuer Leser dieser Kolumne sind, kennen Sie meine Einstellung: Es gibt keine Standardprozesse, ebenso keine Standardarchitekturen, und auch sonst sind wir mit Lösungen von der Stange in der Entwicklung nicht gut beraten. Wege

zu probieren und danach die Ergebnisse zu reflektieren und das zukünftige Verhalten anzupassen ist der einzig wahre Weg, professionelle Softwareentwicklung zu betreiben. Allerdings funktioniert dies in Projekten mit schwacher Fehlerkultur oftmals nicht besonders gut. Um besser zu werden, müssen Fehler von allen erkannt werden; dazu muss jeder einsehen, dass etwas ein Fehler war. Wird in Projekten, besonders außerhalb der Softwareentwicklung, immer alles schöngeredet, ändert sich leider am Ende des Tages nichts. Damit erzielen wir keine Verbesserungen, und die Softwareentwicklung tritt auf der Stelle. In diesem Projekt war das Scheitern der einzig richtige Ausgang für die Zukunft.

Denn hätten wir es vonseiten der Entwicklung wieder geschafft, „das Unmögliche möglich zu machen“, hätten wir in einem halben Jahr das nächste Projekt dieser Art auf dem Schreibtisch gehabt. Getreu dem Motto: „Hat letztes Mal funktioniert, also klappt es auch diesmal.“

Gift für die Entwicklung

Obwohl ich die beschriebene Situation schon mehrere Duzend Mal erlebt habe, fällt es mir noch immer schwer, die Hauptursache dafür zu finden. Manchmal ist es Unwissen darüber, wie Softwareentwicklung funktioniert. Manchmal steckt aber auch Gier dahinter und der unbedingte Wille, sich außerhalb der Entwicklungsabteilung zu profilieren, ohne Rücksicht auf die entstehenden Konsequenzen. Egal was der Hauptgrund ist: Es zerstört sehr vieles. Es zerstört Vertrauen zwischen Unternehmen und Kunden, aber auch zwischen den Abteilungen, zwischen einzelnen Rollen, und auch das Vertrauen in die eigenen Fähigkeiten, gute Software zu entwickeln. „Nein“ zu sagen zu unrealistischen Anforderungen und unerreichbaren Zeitplänen ist leider unter Entwicklern nicht sehr weit verbreitet, und ein solches „Nein“ zu akzeptieren ist noch wesentlich weniger weit verbreitet.

Fazit

Wenn wir Software entwickeln, sollten wir das mit einem gewissen Maß an Qualität tun. Damit wir das können, sollte jeder im Unternehmen, der an diesem Prozess beteiligt ist, wissen, wie man ein Projekt richtig durchführt und welche Gefahren und Stolpersteine auf diesem Weg herumliegen. Wenn ich alle Situationen in meiner Karriere zusammenfasse, die von solchen Problemen betroffen waren, war es leider fast immer so: Jemand hatte nicht verstanden, dass Softwareentwicklung so eben nicht funktioniert, und das ist schade. Ich wünsche Ihnen, dass Sie in solchen Situationen die für Sie richtigen Schlüsse ziehen – viel Erfolg! ■



David Tielke

ist freiberuflicher Berater und von Microsoft zertifizierter Trainer für die Anwendungsentwicklung auf der .NET-Plattform. Darüber hinaus hat er sich auf die Bereiche Softwarearchitektur, Softwarequalität und ALM spezialisiert.
mail@david-tielke.de

WAS KOMPLEXITÄT AUSMACHT UND WIE WIR SIE WIRKSAM REDUZIEREN

Weniger ist so viel mehr!

So können wir komplexe Projekte in der Softwareentwicklung wirklich im Griff haben.

Kein Zweifel: Softwareentwicklung ist ein komplexes Vorhaben. Doch was bedeutet das genau? Und vor allen Dingen: Wie gehen wir mit dieser Komplexität angemessen um?

Auch wenn das Thema ausgelutscht erscheint, denke ich, dass immer noch eine Reihe von Missverständnissen in den Köpfen der beteiligten Personen vorhanden sind und es vielfach an wirksamen Ideen mangelt, mit komplexen Aufgaben angemessen umzugehen.

Softwareprojekte fühlen sich oft an wie ein Kampf gegen Hydra: ein Ungeheuer aus der griechischen Mythologie. Es ist eine riesige Wasserschlange, der sofort zwei Köpfe nachwachsen, sobald sie einen verliert. Als wäre das nicht bereits schlimm genug, ist ihr mittlerer Kopf unsterblich. Ein Kampf gegen sie erscheint daher aussichtslos. Doch dem ebenfalls aus der griechischen Mythologie entsprungenen Herakles beziehungsweise Herkules gelang dieses Unterfangen mit der Unterstützung seines Neffen Iolaos.

Wer fühlt sich bei dieser Geschichte nicht an Softwareprojekte erinnert. Kaum haben wir ein Problem erfolgreich gelöst, tauchen zwei weitere auf, um die wir uns kümmern müssen. Wer hilft uns dabei? Wie gehen wir vor? Die Lösung des Herakles mit Feuer, einem unter Steinen begrabenen unsterblichen Kopf, aufgeschlitzten Körpern und diversen Kollateralschäden funktioniert hierfür nicht. Doch der Reihe nach: Was bedeutet es, dass eine Aufgabe komplex ist?

Komplex oder kompliziert?

Nicht alle Aufgaben sind komplex. Manche sind kompliziert. Was ist der Unterschied? Pragmatisch zusammengefasst: Komplizierte Aufgaben sind durch Expert:innen vorab planbar, komplexe hingegen nicht. Was zeichnet solche Expert:innen aus? Ihre Ausbildung und Erfahrung im Umgang mit dem Thema machen den Unterschied.

Ein einfaches Beispiel illustriert das. Für die Wartung unserer Heizungsanlage kommt einmal im Jahr ein Wartungstechniker. Er kennt nicht nur das Modell, sondern auch unsere konkrete Anlage seit Jahren genau. Den Aufwand für die Wartung kann er sehr gut vorab schätzen. Selbst wenn ein Verschleißteil mehr als erwartet ausgetauscht werden muss, schlägt sich das nur unwesentlich auf die Kosten für Aufwand und Material nieder.

Nun könnte ich auf die Idee kommen, das Geld zu sparen und am Wochenende die Wartung selbst zu übernehmen. Ich bin doch recht helle und handwerklich geschickt. Natürlich mache ich das nicht, denn nach einem frustrierenden Wochenende sowie dem Studium zahlreicher Dokumente und YouTube-Tutorials müsste der Techniker am Montag vermutlich dennoch kommen, um meine Fehler zu beseitigen und

die Anlage wieder zum Laufen zu bekommen. Für mich wäre diese Aufgabe sicher zu komplex. Zu viele Aspekte der Heizung, mit denen ich mich nicht auskenne, machen ihre Wartung für mich unplanbar.

Wie in **Bild 1** zu sehen, gibt es vier Komplexitätsstufen, in denen sich eine Aufgabe befinden kann:

- Einfach
- Kompliziert
- Komplex
- Chaotisch

Die Übergänge zwischen den Phasen sind eher direkt und schnell. Es gibt den Moment, an dem eine Aufgabe von kompliziert zu komplex kippt. Innerhalb einer Komplexitätsstufe wachsen die Aspekte und Parameter dann wieder fließend an.

Es ist wie mit dem Übergang von Eis zu Wasser. Von -10° C bis 0° C wird das Eis nur fühlbar wärmer. Wird weiter Energie in Form von Wärme hinzugeführt, wechselt der Aggregatzustand, und aus Eis von 0° C wird Wasser von 0° C. Wir spüren keine Erwärmung, denn die Energie fließt in den Wechsel des Aggregatzustands. Führen wir weiter Energie hinzu, wärmen wir wiederum das Wasser fühlbar an. Genauso sprunghaft verhält es sich mit den Übergängen der Komplexitätsstufen.

Kehren wir zu unserem Thema zurück: Brauchen wir also einfach nur andere, bessere Entwickler:innen, und das Problem mit den komplizierten Aufgaben wäre auch für die Softwareentwicklung gelöst? Leider nicht. Denn auch Expert:innen kommen bei weiter zunehmenden, komplizierten Aspekten irgendwann an ihre Grenzen. Die Aufgabe ist dann auch für sie komplex geworden. Auch KI hilft uns dabei nur sehr begrenzt. Es braucht grundsätzlich eine andere Herangehensweise.

Auf die verschiedenen Herangehensweisen in Abhängigkeit vom Komplexitätsgrad gehe ich in meinen Ausführungen nicht weiter ein. Das ist zum Beispiel von Dave Snowden im Cynefin-Framework sehr übersichtlich dargestellt [1]. Mir geht es in diesem Artikel um die einzelnen Komponenten, die Komplexität ausmachen, und was wir über den Umgang mit den einzelnen Komponenten im Hinblick auf die Reduktion der Komplexität unseres Projekts erreichen können.

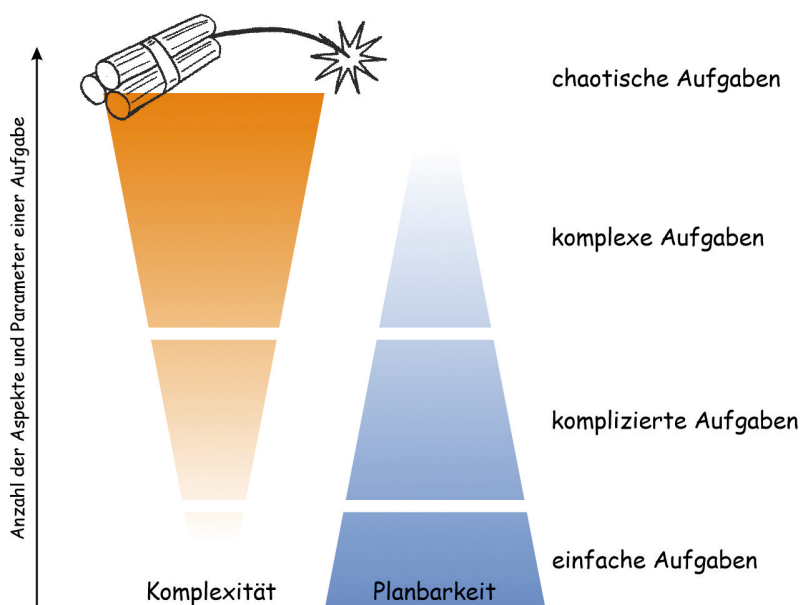
Komplexität beherrschbar machen

Grundsätzlich können wir komplexe Aufgaben nur (wieder) beherrschbar machen, indem wir die Anzahl der Parameter und Aspekte reduzieren, die die Komplexität ausmachen. In **Bild 1** bedeutet das bildlich gesprochen, die Aufgabe wieder weiter nach unten zu schieben. Typische Versuche, über Abs-

traktion eine komplexe Aufgabe im Griff haben zu wollen, werden nicht zum gewünschten Ergebnis führen.

Ist es also sinnlos, zu modellieren oder eine Struktur der Architektur eines Systems zu entwickeln? Natürlich nicht, doch zu denken, wir könnten darüber die Komplexität beherrschen, ist zwar verlockend, aber falsch. George Box sagte bereits treffend: „All models are wrong – but some are useful“. Was das bedeutet, hat Alfred Korzybski bereits 1933 beschrieben: „A map is not the territory it represents, but, if correct, it has a similar structure to the territory, which accounts for its usefulness“ [2].

Über Abstraktion, wie sie zum Beispiel über Modelle geschaffen wird, werden zahlreiche Details ausgeblendet, um übergreifende Strukturen, Dynamiken und Abhängigkeiten zu erkennen. Die ausgeblendeten Details sind aber alle noch vorhanden und werden später betrachtet. Haben wir über die abstrakten Sichten eine angemessene Strukturierung gefun-



Je mehr Parameter beziehungsweise Aspekte zu einer Aufgabe hinzukommen, desto stärker verändert sich sprunghaft ihr Charakter von einfach über kompliziert und komplex hin zu chaotisch (Bild 1)

den und Abhängigkeiten passend gesteuert, so hilft uns das, die dahinterliegenden Details besser umzusetzen. Der Grad der Komplexität einer Aufgabe wird davon aber nur marginal betroffen sein. Die Komplexität reduzieren wir dadurch nur indirekt ein wenig, falls es uns gelingt, die Abhängigkeiten global passend zu steuern.

Ein typisches Architekturmuster zur Steuerung von Abhängigkeiten ist es, zu versuchen, sie so zu steuern, dass wir von eher stabilen Teilen abhängig sind. Das reduziert einen Teil der Dynamik in unseren Aufgaben. Dieser Weg ist aber schnell ausgereizt.

Was uns bleibt, ist, diejenigen Parameter selbst aus dem Spiel zu nehmen, die die Komplexität ausmachen. Das ist schmerzhaft und daher unangenehm. Unsere Stakeholder werden nicht alles bekommen, was sie (meinen zu) brauchen.

Die Diskussionen werden mühsam sein. Doch im Endeffekt geht es um den Projekterfolg. Kommen wir mit etwas weniger Umfang ins Ziel oder scheitern wir beim Versuch, die ideale Lösung zu implementieren?

Die einzelnen Komplexitätsparameter lassen sich in sechs Bereiche zusammenfassen:

- Technische Aspekte
- Soziale und organisatorische Aspekte
- Prozesse und zeitliche Aspekte
- Fachlichkeit beziehungsweise Domänen
- Umwelt und Kontext
- Emergenz

Die Komplexität eines Projekts setzt sich aus der Komplexität der einzelnen Komplexitätsbereiche zusammen. In Bild 2 wird das schematisch durch die orange Fläche dargestellt. Über die einzelnen Parameter eines Komplexitätsbereichs können

wir die Komplexität eines Projekts wirklich reduzieren. Da ein einzelner Komplexitätsparameter meist zu wenig Einfluss auf die Gesamtkomplexität hat, betrachten wir mehrere Parameter eines Komplexitätsbereichs im Zusammenspiel.

Die ersten drei Themenblöcke (Bild 2 rechts und unten) haben wir als Projektteam weitgehend selbst in der Hand. Bei den anderen drei Themen (Bild 2 links und oben) hängen wir stärker von äußeren Gegebenheiten ab. Daher ist es sinnvoll, bei den ersten drei Themenblöcken besonders genau und gründlich die Faktoren für die Projektkomplexität zu minimieren. Gleichwohl versuchen wir das auch mit den anderen drei Themenblöcken. Hierbei haben wir jedoch deutlich mehr externe Abhängigkeiten zu betrachten.

Das Prinzip hinter diesem Vorgehen geht auf das Agile Manifest [3] zurück. Dort steht unter den zwölf Prinzipien: „Einfachheit – die Kunst, die Menge nicht getaner Arbeit zu maximieren – ist essenziell.“ Wir reduzieren unsere Tätigkeiten immer wieder auf den Kern,

die Essenz, die notwendig ist. Das allein ist bereits schwierig und technisch beziehungsweise organisatorisch herausfordernd genug.

Gehen wir nun die sechs einzelnen Komplexitätsbereiche durch und betrachten wir, was das jeweils bedeutet.

Technische Aspekte

Der technische Komplexitätsbereich betrifft die Struktur, Softwarearchitektur und Technologie der Software selbst und umfasst auch die Schnittstellen und das Datenmodell. Teile der Infrastruktur und hier insbesondere die Entwicklungsinfrastruktur inklusive der Versionsverwaltung und der CI/CD-Pipeline gehören ebenfalls dazu. Das beinhaltet auch den Einsatz von Designmustern, wenn sie übergeordnete, strukturelle Aspekte betreffen. Listen wir die Themen auf: ►

- Systemarchitektur
 - Anzahl und Art der Komponenten und deren Abhängigkeiten
 - Grundsätzliche Strukturen wie ein monolithischer Aufbau oder eine Microservice-Architektur
- Vielfalt im Technologie-Stack
 - Programmiersprachen
 - Frameworks
 - Datenbanken
 - Schnittstellen
 - Plattformen
- Anzahl und Heterogenität externer Systeme
 - APIs
 - Legacy-Systeme
 - Hardware
- Komplexität der verarbeiteten Daten
 - Umfang
 - Struktur
 - Qualität
- Qualitative beziehungsweise nichtfunktionale Anforderungen
 - Security
 - Performance
 - Skalierbarkeit
 - Zuverlässigkeit
 - Bedienbarkeit
 - Analysierbarkeit und Testbarkeit
- Zusätzliche Abhängigkeiten
 - Nutzung von Drittanbieter-Bibliotheken
 - Plattformabhängigkeiten

Sicher fehlen bei meiner Betrachtung noch Themen, die in euren Projekten eine wichtige Rolle spielen. Häufig treffen wir auf eine Vielzahl an Aspekten, die durch die Entwicklungsteams getrieben in unser Projekt gekommen sind. Bei den Entscheidungen beispielsweise für ein neues Messaging-Framework werden dann Betrachtungen der technischen Komplexität zu oft außer Acht gelassen. Häufiger wahrgenommen habe ich dagegen Aussagen wie: „Wir setzen Apache Kafka ein, weil ich das immer schon einmal ausprobieren wollte.“

Bei den technischen Aspekten ist es wichtig, diese so gering wie möglich zu halten. Das mag uncool sein und vielleicht nicht für jede Spezialaufgabe die ideal passende Lösung bieten, doch im Lifecycle des Produkts wird es sich rentieren. Gerade wenn ein umfangreiches Refactoring ansteht oder Teile unseres Produkts zu Legacy geworden sind, profitieren wir von diesem Minimalismus.

Ein Thema, das durch unsere Kunden getrieben in unsere technischen Aspekte hinzukommen und die Komplexität enorm erhöhen kann, sind eigene Frameworks oder tiefgreifende Erweiterungen von Frameworks von Drittanbietern wie zum Beispiel bei UI-Frameworks. Ja, wir haben dann zum Beispiel ein absolut einmaliges UI-Element, das voll und ganz im Sinne unseres Kunden ist. Doch die Freude währt nur bis zum nächsten umfangreichen Update unseres Drittanbieter-Frameworks. Jetzt explodieren schnell die Aufwände bei der Anpassung unserer eigenen Erweiterungen.

Um die technische Komplexität möglichst gering zu halten, gilt es grundsätzlich, die Anzahl verschiedener Tools zu minimieren und keine eigenen Frameworks zu entwickeln. Frameworks oder Tools externer Anbieter sind so einzusetzen, wie sie gedacht sind. Wenn ein Framework nicht passt, werft es raus und wählt ein passenderes Hilfsmittel aus.

Der Begriff „Legacy“ wird in dem Zusammenhang häufig abwertend benutzt. Das ist oftmals nicht richtig, denn Legacy bedeutet meistens: „Ist alt, aber funktioniert.“ Ablösungen von Legacy-Teilen sind daher nie Selbstzweck, sondern müssen stets durch einen Mehrwert für die Kunden beziehungsweise Anwender:innen begründet sein. Schaffen wir keinen sichtbaren Mehrwert für die Kunden, bleibt es so, wie es ist.

Sind verschiedene Lösungen möglich, wählt die einfachere von beiden. Muss es wirklich Kubernetes sein, oder reicht Docker aus, um die Anforderungen zu erfüllen? Brauchen wir die Funktionalität und damit verbundene Komplexität von Apache Kafka, oder reicht eine einfache Message Queue wie zum Beispiel RabbitMQ aus? Bei Weitem nicht alles, was technisch möglich oder reizvoll wäre, ist für die langfristige Wartbarkeit und Erweiterbarkeit sinnvoll.

Soziale und organisatorische Aspekte

Hier geht es um die Zusammenarbeit und Kommunikation der beteiligten Personen, die Teamorganisation und die Prozesse. Die strukturellen Aspekte sind offensichtlich, um die Komplexität zu verringern. Wir arbeiten mit möglichst wenigen Personen in der Entwicklung in möglichst kleinen Teams und mit so wenigen Teams wie möglich. Diese Teams arbeiten an einem Standort beziehungsweise mit dem geringstmöglichen Grad an Verteilung. Verteilte Teams liegen in derselben oder in direkt benachbarten Zeitzonen. So minimieren wir die Kommunikationsaufwände.

Klingt logisch, doch wie passt das in die Arbeitswelt von heute? Wo liegen die Prioritäten bei den verschiedenen genannten Aspekten? Wenn wir mit möglichst wenigen Personen arbeiten wollen, benötigen wir hervorragendes Personal. Das wird sich aber kaum an einem Standort finden lassen. Da sich Softwareentwicklung um die Menschen dreht, die sich damit befassen, bilden sie den bedeutsamsten Faktor. Die oft angestrebte Teamgröße von 7 ± 2 Personen und die minimal mögliche Anzahl an Teams bilden den Kern der Reduktion von Komplexität der sozialen und organisatorischen Aspekte.

Bei der Organisation der Arbeit legen die Teams aufgabenbezogen fest, wann sie verteilt und zu welchen Themen sie gemeinsam arbeiten wollen. So können die grundsätzliche Verteilung und die Themen „work from home“ beziehungsweise „work from anywhere“ konstruktiv betrachtet werden. Der Management-Overhead beschränkt sich auf ein Servant Leadership mit den minimalen Controlling-Aufwänden, die für die Abrechnung von Leistungen gegenüber den Kunden notwendig sind.

Ein weiterer organisatorischer Aspekt ist die Anzahl der Fachbereiche, mit denen die Entwicklung zusammenarbeitet. Hier ist es wichtig, im Projektverlauf eine zeitliche Entzerrung der Zusammenarbeit einzuplanen. Mit welchen Themen beginnen wir, und was folgt als Nächstes? Dadurch wird

die Anzahl der Fachbereiche, mit denen wir gleichzeitig arbeiten, minimiert. Die Schwierigkeit liegt darin, dies wertschätzend zu moderieren. „Dein Thema kommt später“ wird häufig als „Du bist nicht wichtig genug“ wahrgenommen.

Generell entstehen aus dem Management und der Vorgehensweise im Projekt weitere Aspekte, die die Komplexität erhöhen. Jede Hierarchie- oder Entscheidungsstufe und jedes Reporting erhöht die Anzahl der Stakeholder. Je aufwendiger der Entwicklungsprozess ist, desto mehr unterstützendes Personal wird benötigt. Von daher sind Hierarchien, Reporting und andere Kommunikationswege zu minimieren. Als Entwicklungsprozess ist das einfachste mögliche Verfahren auszuwählen. Reicht Kanban aus? Braucht es Scrum oder LeSS? Hier können Vereinfachungen organisatorisch erreicht werden, was häufig leichter zu realisieren ist als an anderer Stelle.

Wo mehrere Menschen zusammenarbeiten, kommt es früher oder später zu Konflikten. Meist geht es dabei um technische oder fachliche Entscheidungen beziehungsweise Aspekte des Vorgehens. Diese aufzulösen ist tägliches Projektmanagement. Ein zu langes Aufschieben führt dazu, dass die Entscheidungen durch das Faktische getroffen werden. Hier sind alle Beteiligten gefragt, Entscheidungen zum notwendigen Zeitpunkt zu treffen beziehungsweise die Vorbereitung von Entscheidungen voranzutreiben, um die daraus resultierenden Auswirkungen auf die Komplexität zu minimieren.

Besonders kritisch können Zielkonflikte zwischen einzelnen beteiligten Personen sein. Sie lassen sich meist nur auf Zeit lösen und führen früher oder später dazu, dass sich die Teams personell verändern. Jeder Wechsel von eingearbeiteten Personen führt zu Ablenkungen, neuer Einarbeitung und Zeitverlust. Ganz abgesehen von den negativen Auswirkungen auf das Team, während der Zielkonflikt am Köcheln ist. Die Fähigkeit der verantwortlichen Personen, mit den Konflikten konstruktiv umzugehen, ist essenziell, um längere Störungen und damit zusätzliche Komplexität zu vermeiden.

Ein weiterer Konfliktherd besteht im Stakeholder-Management. Die Vielfalt der unterschiedlichen Sichten und Bedürfnisse ist ein erheblicher Komplexitätsaspekt. Dabei steht nicht nur das Auflösen von Zielkonflikten im Mittelpunkt. Die unterschiedlichen Anforderungen können manchmal einfach nicht sinnvoll in einer Lösung umgesetzt werden. Dann ist es besser, das Produkt und die Teams aufzuteilen, auch wenn das zu erheblichen Code-Redundanzen führt. Die Komplexität wird für alle deutlich geringer. Die mentale Last in der Entwicklung verringert sich spürbar, was unterm Strich zu

schnelleren Ergebnissen führt. Die Code-Redundanzen sind demgegenüber das deutlich kleinere Problem.

Generell gilt es, die Kommunikationswege direkt und eindeutig zu gestalten. Dabei werden wir bestimmt von den Abhängigkeiten in der Architektur, wie sich auch die Struktur unserer Entwicklungseinheiten in der Software widerspiegelt. Die Stichworte sind hier Conways Law beziehungsweise Team Topologies. Die Schnittstellen zwischen Teams sind mit besonderer Achtsamkeit zu gestalten, um die Komplexität unserer Software und unseres Vorgehens zu minimieren.

Prozesse und zeitliche Aspekte

Was kommt noch zu den sozialen und organisatorischen Aspekten hinzu, wenn wir die Prozesse und den zeitlichen Rahmen analysieren? Zeitdruck und die Änderungsdynamik an

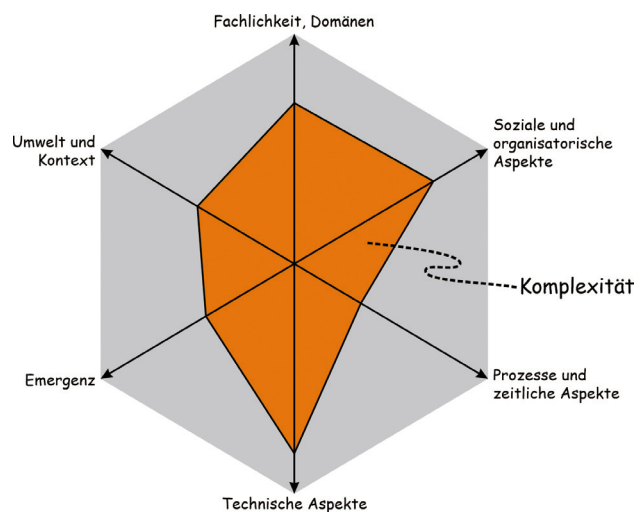
bestehender Software sind Aspekte, die zu einer Parallelisierung von Aktivitäten führen und damit die Komplexität des Projekts erhöhen. Enge Deadlines erhöhen die Risiken und führen in der Regel zu einem Qualitätsverlust der Software. Auch entstehen mehr technische Schulden, für deren Beseitigung wir erst recht keine Zeit haben.

Auch hier ist absolut notwendig, die Menge an gleichzeitig umzusetzenden Anforderungen zu minimieren und das Projekt zeitlich zu strecken. Ein stark inkrementelles Vorgehen ist hierbei nützlich, um den inhaltlichen Projektfortschritt tatsächlich nutzbar zu machen. Das ist nicht nur wirtschaftlich not-

wendig, sondern liefert auch wertvolles Feedback zurück in die Produktentwicklung.

Der Preis, den wir dafür zahlen, ist eine komplexe Toolandschaft für die Continuous Integration und Continuous Delivery (CI/CD). Dieser Teil lässt sich meist ausreichend aus dem Projektkern herausziehen und durch einen tiefgehenden DevOps-Ansatz lösen. Die Komplexität wird zu Expert:innen verschoben, für die die häufigen Releases oft nur eine komplizierte Aufgabe darstellen.

In diesem Zusammenhang stellt der Grad der Testautomatisierung einen interessanten Aspekt dar. Je höher der Umfang der Testautomatisierung auf den verschiedenen Ebenen wie Unit, Service beziehungsweise Komponente und System ist, desto höher ist der Automatisierungsgrad und damit die Geschwindigkeit der Rückmeldungen. Dieser positive Effekt auf die Komplexität unseres Projekts kann aber schnell in sich zusammenfallen, wenn die Wartungs- und Anpassungsaufwände der Testautomatisierung zu umfangreich werden und den automatisierten Prozess regelmäßig unterbrechen. ►



Die sechs Komplexitätsbereiche von Softwareprojekten – die Komplexität des Projekts entspricht der Fläche im Diagramm (orange) (Bild 2)

Die Tests sind auf allen Ebenen entsprechend gut wartbar und robust zu designen, damit wir einen Vorteil daraus ziehen können.

Es gilt wie so oft, die passende Balance zu finden. Da eine solche Balance unter anderem von den Projektumständen, der Fortschrittsgeschwindigkeit und dem Können der Teammitglieder abhängt, handelt es sich um eine labile Balance. Das bedeutet, wir müssen sie immer wieder neu finden. Damit ist die Dynamik der gefundenen Balancen ebenfalls ein Komplexitätsparameter.

Die letzten Komplexitätsfaktoren aus diesem Bereich, die ich betrachten möchte, sind sich ändernde Rahmenbedingungen. Hier haben wir aus dem Projekt heraus kaum direkte Einflussmöglichkeiten. Ein angemessener Weg, damit umzugehen, sind kurze Entwicklungszyklen. Ein streng iteratives Vorgehen bietet Chancen, dass solche sich ändernden Rahmenbedingungen erst in der nächsten Iteration berücksichtigt werden müssen. Das führt dann dazu, dass die laufende Iteration davon nicht betroffen ist und die sich ändernde Rahmenbedingung für die laufende Entwicklung die Komplexität nicht erhöht.

Auch wenn ein radikales inkrementell-iteratives Vorgehen kombiniert mit konsequentem DevOps die Komplexität in Vergleich zu eher linearen Vorgehensweisen erhöht, werden wir unterm Strich davon bei komplexen Projekten überwiegend Vorteile haben. Wir können darüber die Projektkomplexität besser reduzieren beziehungsweise die auslösenden Faktoren zeitlich strecken.

Fachlichkeit beziehungsweise Domänen

Die Fachlichkeit hinter den Anforderungen zu verstehen und sie passend für das Anwendungsgebiet der Softwarelösung zu modellieren ist elementar für den Erfolg eines Softwareprojekts. Wir können die eleganteste, wartbarste und performanteste Software programmieren, doch ohne das tiefe Verständnis der Fachlichkeit und eine angemessene Transformation in eine für die Softwareentwicklung nutzbare Struktur wird ihr der Erfolg am Markt verwehrt werden.

Wenn etwas für uns in der Entwicklung komplex ist, können wir davon ausgehen, dass die Anforderungen, Geschäftsregeln und Fachlichkeit auch für die Fachexpert:innen komplex sind. Das hat zur Folge, dass uns zu Beginn eines Projekts die Anforderungen nicht vollständig bekannt sein können. Wir haben es daher mit drei Arten von Anforderungen zu tun:

- vollständig bekannte Anforderungen,
- nur grob bekannte und im weiteren Entwicklungsprozess noch zu verfeinernde Anforderungen,
- zu Beginn noch unbekannte Anforderungen.

Das ist einer der Gründe, modular und inkrementell vorzugehen. Das (vermeintliche) Gesamtbild existiert nur auf einer abstrakten Ebene, um Geschäftsprozesse und fachliche Domänen zu identifizieren. Wir verfeinern und implementieren nur die Teile, die bereits ausreichend bekannt sind. Meist handelt es sich dabei um Kernelemente der Fachlichkeit und damit auch der späteren Softwarelösung. Ein solches Vorgehen ist also grundsätzlich sinnvoll.

Es hilft uns auch dabei, die Komplexität des Projekts zeitlich zu entzerren. Wir können gar nicht sinnvoll viele Dinge parallel machen. Auf Basis bereits implementierter Fachlichkeit können dann die nächsten anstehenden Anforderungen verfeinert oder überhaupt erst identifiziert werden. Hieraus ergibt sich der Bedarf nach einem inkrementell-iterativen und agilen Vorgehen, um mit der Komplexität angemessen umgehen zu können.

Leider gibt es noch eine Reihe weiterer Aspekte, die die Komplexität der Fachlichkeit ausmachen. Zum einen ist die Domänenlogik selbst zu analysieren: Wie komplex sind die Geschäftsregeln und -prozesse? Welche unterschiedlichen Anwendungsfälle, Nutzergruppen oder Konfigurationen gibt es? Und vor allen Dingen: Wo können wir ohne große Einschränkungen in der Anwendbarkeit der Software Vereinfachungen vornehmen oder Aspekte ganz herausnehmen? Häufig reicht die 80-Prozent-Lösung aus und macht es auch den Anwender:innen einfacher, ihrer Arbeit nachzukommen. Nicht jede Anwendergruppe braucht ihre eigene Sonderlösung. Nicht jedes Detail muss konfigurierbar sein. Bei jeder Anforderung hinterfragen wir, ob der Nutzen den Aufwand und die Erhöhung der Komplexität mit ihren Auswirkungen auf die spätere Wartbarkeit und Erweiterbarkeit rechtfertigt.

Regulatorische Anforderungen wie rechtliche Vorgaben zum Beispiel zum Datenschutz, bei Medizin- oder Finanzprodukten nehmen dabei eine Sonderstellung ein. Wir dürfen sie nicht wegdiskutieren, da uns ansonsten mit unserem Produkt der Eintritt in den entsprechenden Markt verwehrt wird. Auch hier gilt es genau zu analysieren, welche Anforderungen für welchen Anwendungsbereich im Detail gelten. So unterscheiden sich die Ausprägungen der Anforderungen zum Beispiel bei Medizinprodukten zwischen reiner Forschung und Diagnostik voneinander. Zugegeben, die Unterschiede sind gering, können jedoch in der Entwicklung einen signifikanten Unterschied machen.

Aus regulatorischen Rahmenbedingungen entsteht eine Reihe zusätzlicher Anforderungen, die oft architekturrelevant sind und auf keinen Fall vernachlässigt werden dürfen. Die Vorgaben werden genau erfasst und beschrieben, sodass ihre Auswirkungen auf den Projektverlauf einerseits und die Softwarearchitektur beziehungsweise das High-Level-Design andererseits bestmöglich abgeschätzt werden können. Wir schaffen für solche Anforderungen höchstmögliche Transparenz, damit wir ihre Auswirkungen auf unser Projekt steuern können.

Umwelt und Kontext

Die letzten direkt greifbaren Komplexitätsparameter, die wir hier betrachten, betreffen die externen Faktoren, die ein Projekt beeinflussen. Der Markt, auf dem wir aktiv sind, wandelt sich je nach Branche zwar unterschiedlich schnell, doch er wandelt sich. Abhängigkeiten von Technologien wie Tools und Frameworks, allgemeine Anforderungen zu Datenschutz oder Internet-Sicherheit oder Hardware- beziehungsweise Plattformabhängigkeiten wie Betriebssysteme, Treiber oder Cloud-Umgebungen können uns das Projektleben schwer machen.

Auch hier gilt dasselbe Handlungsmuster: Reduziere die Anzahl der Komplexitätsparameter, wo es nur geht! Verwende Standards und minimiere die Anzahl der Tools, Lieferanten und Kanäle. Und vor allen Dingen: Fokussiere deinen Markt! Wer genau bildet deine Zielgruppe? Welche Kunden können noch ohne weitere Erhöhung der Komplexität mitgenommen werden? Und ganz wichtig: Welche Kunden wollen wir nicht haben? Klarheit über unser Marktsegment minimiert die notwendigen Produktvarianten und damit die Komplexität unseres Projekts. Damit maximieren wir vielleicht nicht unseren Umsatz, aber unsere Marge.

Gutes Produktmanagement ist der Schlüssel, um in unserer Projektumwelt die Parameter zu minimieren, die unsere Komplexität im Projekt hochtreiben! Kenne deine Kunden, ihre Probleme und Bedürfnisse besser, als sie sie selbst kennen!

Emergenz, also nichtlineare Effekte

So viel zu den harten Fakten, die planbar und damit steuerbar sind. Der letzte Komplexitätsbereich ist dagegen deutlich schwerer zu greifen: Emergenz.

Damit wird die Eigenschaft komplexer Systeme bezeichnet, neue Eigenschaften oder Verhaltensweisen zu zeigen, die sich nicht offensichtlich auf einzelne Elemente des Systems zurückführen lassen [4]. Oder wie es Aristoteles bereits rund 340 Jahre vor Christus in seiner „Metaphysik“ auf den Punkt brachte: Das Ganze ist mehr als die Summe seiner Teile!

Emergenz wird in der Gruppendynamik eher positiv betrachtet. Sie wird dort als Effekt gesehen, der die Gruppenleistung höher werden lässt als die Summe der einzelnen Leistungen der Teammitglieder. Doch wenn kleine Änderungen große Auswirkungen haben können, müssen diese nicht notwendigerweise positiv sein. Unvorhergesehene Seiteneffekte nach Änderungen an unserer Software, die vorher noch stabil lief, oder überraschend entstandene Konflikte zwischen Personen sind Beispiele dafür. Selbst bei scheinbar einfachen Systemen kann das Zusammenspiel vieler Teile zu unerwartetem Verhalten führen. Diese neuen Verhaltensweisen verhalten sich nicht linear zu vermeintlich ursächlichen Auslösern und sind damit nicht vorab planbar.

Emergente Wechselwirkungen gibt es zwischen Modulen unserer Software wie auch zwischen den Menschen beziehungsweise den Teams in der Entwicklung. Wie gehen wir damit angemessen um?

Hier gilt es, die Anzahl der gleichzeitigen Änderungen zu minimieren. Idealerweise führen wir stets nur eine Anpassung oder Änderung zu einem Zeitpunkt durch. Danach stabilisieren wir das System wieder, bevor wir eine weitere Änderung vornehmen. So behalten wir die Auswirkungen besser im Griff. Doch dieser Ansatz passt leider gar nicht bei Continuous Integration mit mehreren Teams, wobei meist mehrere Änderungen zeitgleich parallel eingeführt werden.

Hier sind aussagekräftige Integration Stages wichtig. Erst nachdem alle Tests bei den Entwickler:innen erfolgreich gelaufen sind und das Ganze auf der Ebene der Teams mit den anderen Änderungen der Teammitglieder ebenfalls erfolgreich integriert worden ist, erfolgt der Weg in die große Systemintegration mit ihren Tests. Die Aussagekraft der einzel-

nen Stages hängt direkt von der Aussagekraft der automatisierten Tests ab. Kraftvolle, wirksame Tests auf den einzelnen Integrationsstufen sind unser Mittel, ungewünschte Emergenz in unserer Software in den Griff zu bekommen. Der Wert des Testens auf der Ebene der Komponenten beziehungsweise Services eines Teams sowie auf der darüberliegenden Ebene der Systemintegration und des Systems selbst kann hier nicht hoch genug bewertet werden.

Wie kommen wir zu solchen wirksamen Tests? Zum einen leiten wir sie direkt aus den fachlichen Anforderungen und den Akzeptanzkriterien ab. Über diesen Mechanismus erweitern wir unsere Anforderungen mit konkreten Aspekten. Zum anderen sichern wir Risiken ab und definieren darüber die nichtfunktionalen beziehungsweise Qualitätsanforderungen. Last- und Antwortzeitverhalten kann emergenten Einflüssen unterliegen. Über Tests messen wir die relevanten Parameter und betrachten ihre Trends über die Zeit. Dazu bringen wir das System an seine Grenzen und darüber hinaus. Nur so finden wir heraus, wie sich unser Produkt verhält, wenn Teile überlastet oder ausgefallen sind, und können entsprechend reagieren.

Fazit

Wir müssen lernen, Nein zu sagen, wenn wir komplexe Projekte zum Erfolg führen wollen! Nur die Reduktion der parallel wirksamen Komplexitätsparameter verringert die Komplexität unseres Softwareprojekts. Das Prinzip der Einfachheit – die Kunst, die Menge nicht getaner Arbeit zu maximieren – ist dafür essenziell!

Negative emergente Wirkungen können fast ausschließlich nur über dynamische Tests aufgedeckt werden. Mit Akzeptanzkriterien und den Ergebnissen aus den regelmäßig durchgeführten Risikomanagement-Workshops liefern wir nicht nur den dafür notwendigen Input, sondern erweitern unseren Blick auf das Projekt und seine Komplexitätsfaktoren. ■

[1] Dave Snowden & Friends, *CYNEFIN – weaving sense-making into the fabric of our world*, Cognitive Edge, 2020, ISBN 978-1-73537-990-6

[2] Alfred Korzybski, *Science and Sanity – An Introduction to Non-Aristotelian Systems and General Semantics*, 6. Auflage, Institute of General Semantics, 2023, ISBN 978-1-970164-23-7

[3] Beck et al.: *Prinzipien hinter dem agilen Manifest*, <https://agilemanifesto.org/iso/de/principles.html>

[4] Uwe Vigenschow, *Lernende Organisationen*, dpunkt.verlag, 2021, ISBN 978-3-86490-798-2



Uwe Vigenschow

ist Principal QA Specialist bei der Nelta GmbH und freiberuflicher Fachautor. Er bringt dabei seine Erfahrung aus über 30 Jahren als Softwareentwickler, Berater und Führungskraft für den Erfolg seiner Kunden und Leser:innen ein.

uvigenschow@nelta.de

CLEAN CODE UND ARCHITEKTUR, TEIL 5

Prinzipien für Architektur

Manche Clean-Code-Developer-Prinzipien lassen sich auch auf Architektur übertragen, wie etwa DRY. Aber gibt es auch reine Architekturprinzipien?

Clean-Code-Prinzipien beziehen sich auf Code. Das ergibt sich bereits aus der Bezeichnung. Im Code arbeiten wir mit den Modulen Methode und Klasse beziehungsweise Datei. Architektur arbeitet primär mit den darüberliegenden Modulen Bibliothek, Paket, Komponente und Microservice. Wenn wir also versuchen, Clean-Code-Prinzipien auf Architektur zu übertragen, müssen wir uns fragen, welche Bedeutung ein Prinzip auf Bibliotheken et cetera haben könnte.

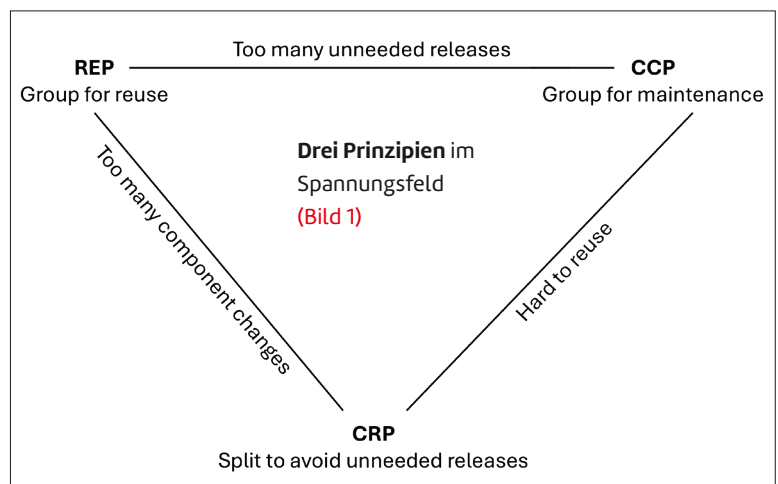
Die Übertragung von Clean-Code-Prinzipien auf den Bereich Architektur funktioniert sicher nicht für alle Prinzipien. Beispielsweise bezieht sich das Liskov Substitution Principle (LSP) auf die Vererbung in objektorientierten Sprachen. Architektur hat aber nichts mit einem bestimmten Paradigma oder einer konkreten Programmiersprache zu tun. Nicht einmal bei Programmiersprachen, die nicht im objektorientierten Paradigma liegen, ist das LSP anwendbar. Für Bibliotheken existiert das Konzept der Vererbung nicht, daher ergibt der Versuch, das LSP darauf zu übertragen, keinen Sinn.

Wir können somit zunächst die Unterscheidung treffen zwischen Clean-Code-Developer-Prinzipien, die sich auch auf Architektur übertragen lassen, und solchen, für die das nicht sinnvoll ist. Mit dem LSP haben wir ein Beispiel, welches sich nicht auf Architektur übertragen lässt. Das DRY-Prinzip (Don't repeat yourself) dagegen lässt sich übertragen, da es auch im Bereich der Softwarearchitektur sinnvoll ist, sich zu fragen, ob eine bestimmte Dopplung schädlich ist oder nicht. Auch das Single Responsibility Principle (SRP) lässt sich auf Architektur übertragen. Hier gilt dann, dass Bibliotheken, Pakete, Komponenten und Microservices jeweils eine klare Verantwortlichkeit haben müssen. Es darf auch auf dieser Ebene nur einen Grund für Veränderung geben. Die Granularität ist auf diesen höheren Ebenen eine andere als auf der Ebene einer Methode oder Klasse. Dennoch gilt, dass auch diese Module einen klaren Fokus haben müssen.

Neben den Prinzipien, die wir seinerzeit in die Clean Code Developer Initiative aufgenommen haben, gibt es weitere, die wir nicht in unseren Katalog aufgenommen haben, weil sie uns nicht relevant erschienen. Ferner gibt es Prinzipien, die aus einer anderen Richtung als Code kommen: Architekturprinzipien. Sechs davon sollen in diesem Beitrag behandelt werden. Betrachtet werden drei Prinzipien zur Strukturierung von Bibliotheken sowie drei Prinzipien zur Kopplung zwischen Bibliotheken.

Begriffe

An dieser Stelle ist es erneut wichtig, die Begriffe klar zu definieren. In seinem Buch „Clean Architecture“ verwendet Bob C. Martin den Begriff Component und schreibt an einer Stelle „a JAR file or DLL“. In der in Teil 1 dieser Serie [1] vorgestellten Begrifflichkeit bezeichne ich eine DLL allerdings als Bibliothek beziehungsweise Library. Eine Komponente beziehungsweise Component hätte einen separaten Kontrakt. Ein einzelnes JAR-File oder eine einzelne DLL verfügt



jedoch nicht über einen separaten Kontrakt, denn der müsste in einer eigenen DLL liegen. Insofern handelt es sich, übertragen auf die vorgestellten und verwendeten Begriffe, im Folgenden also um Prinzipien zur Gestaltung von Bibliotheken oder im Englischen um Library Principles. In der Literatur wird man allerdings eher den Begriff Component Principles finden, auch wenn mir dies inkonsequent erscheint.

Da die Begriffe nicht einheitlich definiert sind, muss man jeweils vor der Verwendung verdeutlichen, mit welcher Bedeutung der Begriff verwendet wird. An dieser Unschärfe führt leider kein Weg vorbei.

Was kommt rein, was nicht?

Eine Bibliothek ist eine binäre Einheit. Bei den kompilierten Sprachen wie Java oder C# sind es tatsächlich binäre Dateien, die den Quellcode nicht mehr enthalten. Bei interpretierten Sprachen wie Python oder JavaScript mag der Quellcode noch vorliegen, er interessiert den Verwender jedoch nicht.

Das konkrete Format einer Bibliothek könnte auch geändert werden, ohne dass ich als Verwender etwas davon merken würde, weil ich eben nicht in die Bibliotheken hineinschaue.

Beim Einsatz von Bibliotheken zur Strukturierung einer Anwendung stellt sich die Frage, was denn eigentlich sinnvoll in einer Bibliothek zusammengefasst werden sollte. Im Fall von .NET ist es so, dass eine Bibliothek technisch gesehen einer DLL-Datei entspricht. Diese entsteht durch Übersetzung eines Visual-Studio-Projekts, also einer `.csproj`-Datei. Hierbei ist es unerheblich, ob das Projekt mit Visual Studio, Rider oder irgendeiner anderen IDE bearbeitet und übersetzt wird. Die Eins-zu-eins-Zuordnung von `.csproj`-Datei zu DLL und somit Bibliothek hat sich über die Jahre etabliert, auch wenn technisch anderes denkbar und möglich wäre.

Bezogen auf .NET stellt sich demnach die Frage: Welche Prinzipien können herangezogen werden, um zu entscheiden, welche C#-Dateien (Klassen und Interfaces) in einem Projekt zusammengefasst werden und somit in derselben Bibliothek, in derselben DLL-Datei landen?

Zu dieser Frage gibt es drei Prinzipien:

- Release/Reuse Equivalence Principle (REP)
- Common Closure Principle (CCP)
- Common Reuse Principle (CRP)

Bevor die Prinzipien nun im Einzelnen betrachtet werden, noch ein paar Wort zum Scope: Vorgestellt werden die Prinzipien hier im Kontext eines „normalen“ Softwaresystems. Man kann die Prinzipien auch auf die Erstellung von Frameworks anwenden. Allerdings werden die wenigsten Entwickler regelmäßig an einem Framework wie etwa dem .NET Framework oder Avalonia arbeiten. Natürlich besteht diese Möglichkeit in Zeiten von Open Source. Ich betrachte es allerdings eher als Sonderfall, dass ein Entwickler tatsächlich an einem Framework arbeitet. Die meisten von uns erstellen Anwendungen.

Der wichtigste Unterschied liegt hier in der Wiederverwendung. Frameworks sind ausgelegt auf maximale Wiederverwendung. Dies erfordert einen speziellen Blick auf die Schnittstellen und den Zuschnitt der Bibliotheken, Klassen und Interfaces. Innerhalb einer Anwendung mag es das ein oder andere Mal zur Wiederverwendung einer Klasse kommen. Im Rahmen der Anwendungsentwicklung ist dies allerdings eher die Ausnahme, und der Fokus liegt darauf, eine Anwendung zum Laufen zu bringen, und nicht darauf, maximal viele und dazu noch unbekannte Verwender durch Erstellen eines Frameworks zu bedienen.

Release/Reuse Equivalence Principle (REP)

Group for Reuse – Gruppieren im Sinne der Wiederverwendung: „Die Granularität der Wiederverwendung entspricht der Granularität des Release.“

Die Idee des Release/Reuse Equivalence Principle ist, dass jede Veröffentlichung einer Bibliothek für die Verwender Sinn ergibt. Dabei stoßen wir als Erstes auf die Frage, ob das Prinzip sinnvoll anwendbar ist, wenn eine Bibliothek lediglich innerhalb einer einzigen Anwendung verwendet wird. Schon aufgrund des Begriffs „Reuse“ innerhalb der Bezeichnung

des Prinzips ergibt sich, dass es um Wiederverwendung geht. Findet diese nicht statt, könnte das Prinzip eventuell nicht sinnvoll anwendbar sein. Denkt man den Gedanken weiter, könnte man sogar infrage stellen, ob es für eine Anwendung überhaupt Sinn ergibt, sie auf mehrere Projekte/Bibliotheken aufzuteilen, solange Teile davon nicht in anderen Anwendungen wiederverwendet werden.

Ich lege das Prinzip allerdings nicht ganz so streng aus. Auch ohne konkrete Wiederverwendung kann uns das Prinzip Hinweise geben, welche Klassen und Interfaces wir in einem Projekt zusammenfassen sollten. Die Kernidee: Wenn eine Bibliothek in einer neueren Version an einen Verwender ausgeliefert wird (Release), dann sollte diese Bibliothek für den Verwender Sinn ergeben (Reuse). Fasse ich beispielsweise alle Klassen der Benutzerschnittstelle einer Anwendung in einer Bibliothek zusammen, kann diese Bibliothek in einer anderen Anwendung wiederverwendet werden. Es fehlt nichts, und es ist nicht zu viel in der Bibliothek. Damit wird auch das Single Responsibility Principle (SRP) eingehalten: Es gibt nur einen Grund, etwas an dieser Bibliothek zu ändern, nämlich genau dann, wenn sich etwas an der Benutzerschnittstelle verändert.

Liegen dagegen in einer Bibliothek Klassen, die zu einem anderen Aspekt gehören, kann auch durch diese Klassen ein Grund für einen neuen Release entstehen. In dem Fall hätte ein Verwender der Bibliothek jedoch eventuell keinen Nutzen aus dem neuen Release, sofern er diesen anderen Aspekt der Bibliothek gar nicht verwendet. Das Release/Reuse Equivalence Principle sorgt somit dafür, dass ein neuer Release für den Verwender grundsätzlich sinnvoll sein kann, weil sich dadurch eine Änderung oder Ergänzung an dem von ihm tatsächlich verwendeten Zweck der Bibliothek ergeben hat.

Common Closure Principle (CCP)

Group for Maintenance – Gruppieren im Sinne der Wandelbarkeit/ Wartung: „Fasse zusammen, was sich aus denselben Gründen gemeinsam ändert.“

Eine Bibliothek sollte so zusammengesetzt sein, dass sie alle Dinge enthält, die zusammen benötigt werden. Als Beispiel kann wieder eine Bibliothek dienen, in der die Benutzerschnittstelle einer Anwendung abgelegt wird. Werden alle benötigten Klassen gemeinsam in einer Bibliothek abgelegt, wirken sich Änderungen an der Benutzerschnittstelle jeweils nur lokal auf diese Bibliothek aus. Das führt dazu, dass diese Bibliothek nur dann neu veröffentlicht werden muss, wenn sich etwas an der Benutzerschnittstelle ändert. Bei Änderungen aus anderen Gründen bleibt diese Bibliothek unangetastet, was insgesamt die Wartung der Anwendung vereinfacht.

Durch das Gruppieren nach Gemeinsamkeiten wird die Wandelbarkeit des Gesamtsystems verbessert. Ein Entwickler findet sich schneller zurecht, sobald die Verantwortlichkeiten der einzelnen Bibliotheken – und damit im Fall von .NET der einzelnen Visual-Studio-Projekte – bekannt sind. Es muss nicht nach den Dateien gesucht werden, die zu ändern sind, sondern die erste Grobselektion ergibt sich aus der Zuständigkeit der einzelnen Projekte beziehungsweise Bibliotheken. In erster Näherung können die Kernaspekte eines ►

Softwaresystems herangezogen werden, um danach die Bibliotheken zu schneiden:

- Benutzerschnittstelle
- Domänenlogik
- Ressourcenzugriff

Da diese drei Aspekte meist bestimmte Datentypen oder auch Interfaces gemeinsam verwenden, ergibt sich noch der vierte Aspekt der Kontrakte. Auch diese können in einer Bibliothek zusammengefasst werden. Daraus ergibt sich dann auch gleich ein Vorteil bei der Zusammenarbeit mit mehreren Entwicklern: Contract-first wird auf diese Weise unterstützt.

Natürlich ist es in einem ausreichend großen Softwaresystem denkbar, die Kernaspekte nochmals weiter aufzuteilen. So ergeben sich vielfach innerhalb der Domänenlogik weitere Möglichkeiten einer Aufteilung nach größeren Teilaspekten der Domäne. In einem Buchhaltungssystem können das beispielsweise die Bereiche Kernbuchhaltung, Reports, Steuermeldungen et cetera sein. Gleiches ist auch bei der Benutzerschnittstelle und den Ressourcenzugriffen denkbar, wenn man diese weniger technisch, sondern stattdessen mehr an der Domäne ausgerichtet denkt.

Common Reuse Principle (CRP)

Split to avoid unneeded releases – Trenne zur Vermeidung unnötiger Veröffentlichungen: „Zwingt Verwender nicht in nicht benötigte Abhängigkeiten.“

Bei diesem Prinzip geht es um die Frage, was man nicht zusammenlegen sollte. Naheliegender ist, dass es nicht sinnvoll ist, Dinge zusammenlegen, die keinerlei Zusammenhang haben. Das Extrembeispiel könnte sein, die Benutzerschnittstelle mit den Datenbank-Zugriffsfunktionen zusammenzulegen. Es mag im Ablauf des Entwicklungsprozesses einen gewissen logischen Zusammenhang geben: Ein neues Feld in einem Dialog führt gegebenenfalls zu einem neuen Feld in der Datenbank. Technisch gesehen gehören die Aspekte jedoch nicht zusammen. Dieses Spannungsfeld aus Anordnung nach Technologie versus Anordnung nach Domänenaspekten versuchen die sogenannten Vertical Slices zu lösen, in denen alles in einem Projekt oder Verzeichnis zusammengefasst wird, was unter Domänengesichtspunkten zusammengehört. So könnte zum Beispiel alles, was benötigt wird, um den Umzug eines Kunden im System zu erfassen, in einem Verzeichnis abgelegt werden: ein Stück Benutzerschnittstelle zum Erfassen der neuen Anschrift, den Teil Domänenlogik zum Validieren der neuen Anschrift, Datenbanklogik zum Persistieren der Information und so weiter. Auch hier wird versucht, Dinge zu trennen, die nicht zusammengehören. Denn wenn ich als Entwickler an den Anforderungen arbeite, um den Umzug eines Kunden zu erfassen, bin ich bei der vertikalen Zusammenfassung nicht mit Dingen konfrontiert, die nicht zu diesem Thema gehören. Es geht mir hier allerdings nicht darum, unterschiedliche Möglichkeiten der Anordnung von Dateien beziehungsweise Klassen zu diskutieren. Stattdessen stehen die Prinzipien im Vordergrund, die solche Entscheidungen leiten können. Das Common Reuse Principle macht keine Aussage darüber, wie wir Dateien in einem Projekt

organisieren. Es weist lediglich darauf hin, dass die Organisation so erfolgen soll, dass unnötige Veröffentlichungen vermieden werden.

Spannungsfeld

Die drei Prinzipien zur Struktur von Bibliotheken bilden ein Spannungsfeld, in dem man die eigenen Projekte in den unterschiedlichen zeitlichen Phasen jeweils verorten muss. Dazu ist es hilfreich, sich zunächst Folgendes klarzumachen:

- REP und CCP sind inkludierende Prinzipien.
- CRP ist ein exkludierendes Prinzip.

Die beiden ersten Prinzipien geben Hinweise darauf, welche Klassen und Interfaces in einer Bibliothek zusammengefasst werden sollen. Das dritte Prinzip besagt, was getrennt werden soll.

Bild 1 zeigt das Spannungsfeld, in dem sich die drei Prinzipien befinden. Nimmt man jeweils zwei der Prinzipien in den Fokus und berücksichtigt das dann verbleibende dritte Prinzip demgegenüber etwas weniger, ergibt sich daraus eine bestimmte Konsequenz.

- **REP und CCP:** Folgt man vor allem dem Reuse/Release Equivalence Principle (REP) und dem Common Closure Principle (CCP), ergeben sich daraus potenziell zu viele nicht benötigte Releases. Das ist nachvollziehbar, denn der Fokus liegt dann auf „Group for Reuse“ und „Group for Maintenance“. Eine Aufteilung, die gegebenenfalls überflüssige Releases reduzieren würde, findet dann nicht statt. Zu Beginn eines Softwareprojekts wird es jedoch noch keine Wiederverwendung geben. Insofern ist diese Kombination der beiden Prinzipien zu Beginn eher zu vermeiden.
- **CCP und CRP:** Werden das Common Closure Principle (CCP) und das Common Reuse Principle (CRP) in den Vordergrund gestellt, ergeben sich Schwierigkeiten in der Wiederverwendung. Unnötige Releases werden zwar vermieden und es wird nach Maintenance-Gesichtspunkten gruppiert, doch die Gruppierung zum Zwecke der Wiederverwendung fehlt. In der Folge ist die Wiederverwendung der Bibliotheken dadurch erschwert. Aber Achtung: Zu Beginn eines neuen Projekts gibt es noch keine Wiederverwendung, so dass das REP zunächst vernachlässigt werden kann.
- **REP und CRP:** Die dritte Variante entsteht beim Fokus auf das Reuse/Release Equivalence Principle (REP) und das Common Reuse Principle (CRP). In diesem Fall entstehen zu oft Änderungen an Bibliotheken, da der Aspekt „Group for Maintenance“ nicht ausreichend berücksichtigt wird. Auch hier gilt, dass dies nicht zwingend problematisch sein muss, da zu Beginn die Wartung noch nicht im Vordergrund steht. Und solange ohnehin immer das gesamte System deployt wird, braucht man keinen speziellen Fokus auf die Reduktion unnötiger Releases einzelner Bibliotheken.

Component Coupling

Der nächste Satz von Prinzipien dreht sich um die Frage der Kopplung. Kopplung entsteht, wenn Dinge voneinander abhängig sind. Hier geht es wieder um „Components“ beziehungsweise Bibliotheken. Auch Bibliotheken können vonei-

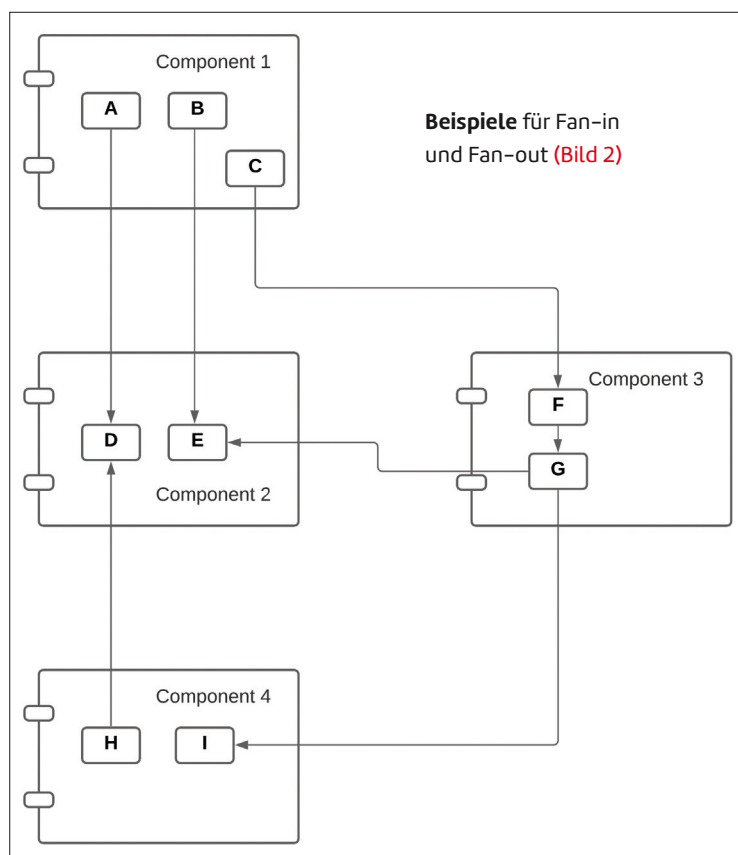
inander abhängig sein und damit eine Kopplung eingehen. Folgende drei Prinzipien werden betrachtet:

- Acyclic Dependency Principle (ADP)
- Stable Dependencies Principle (SDP)
- Stable Abstraction Principle (SAP)

Acyclic Dependency Principle (ADP)

„Abhängigkeiten zwischen Bibliotheken müssen kreisfrei sein.“

Das Acyclic Dependency Principle (ADP) besagt, dass es in der Struktur der Abhängigkeiten keine Kreise geben darf. Das ist heutzutage selbstverständlich und wird bei Verwendung moderner Entwicklungsumgebungen durch diese verhindert. In Visual Studio, Visual Studio Code oder Rider können nur Abhängigkeiten zwischen Projekten – und damit



zwischen Bibliotheken – hergestellt werden, die kreisfrei sind.

Die Herausforderung bei Kreisen in der Abhängigkeitsstruktur liegt im Build-Prozess. Wenn Projekt A von Projekt B abhängig ist und Projekt B wiederum von Projekt A abhängig, gibt es keine Möglichkeit, jedes Projekt im Build-Prozess genau einmal zu übersetzen. Übersetzt man zuerst A, kann dies zu Problemen führen, weil es auf B basiert. Folglich müsste zuerst B übersetzt werden. Das geht aber ebenfalls nicht, weil dazu zuerst A übersetzt werden müsste.

Wir können es also dabei belassen: Das Acyclic Dependency Principle ist so relevant, dass Verstöße bereits durch die Entwicklungsumgebungen verhindert werden.

Stable Dependencies Principle (SDP)

„Abhängigkeiten müssen in der Richtung der Stabilität verlaufen.“

Beim Stable Dependencies Principle (SDP) ist der Sachverhalt etwas komplizierter. Es besagt, dass sich die Richtung der Abhängigkeiten an der Stabilität der beteiligten Bibliotheken orientieren muss. Dazu müssen wir allerdings zunächst genauer bestimmen, was mit der Stabilität einer Bibliothek gemeint ist.

Die Stabilität einer Bibliothek wird definiert über ihre eingehenden und ausgehenden Abhängigkeiten. Dabei werden mit Fan-in die eingehenden Abhängigkeiten bezeichnet und entsprechend mit Fan-out die ausgehenden Abhängigkeiten. Betrachtet werden dazu die in einer Bibliothek enthaltenen öffentlichen Interfaces, abstrakten Klassen und konkreten

Klassen. Bild 2 zeigt einige Beispiele. Die Unterscheidung zwischen konkreten Klassen auf der einen Seite sowie Interfaces und abstrakten Klassen auf der anderen ist hier noch nicht relevant, das ändert sich jedoch im weiteren Verlauf.

In Bild 2 enthält die Bibliothek *Component 1* drei Klassen A, B und C. Diese sind abhängig von Klassen in anderen Bibliotheken. Da die Abhängigkeiten nur aus dieser Bibliothek herausführen, hat sie ein Fan-out von 3.

Bei der Bibliothek *Component 2* ist dies genau umgekehrt: Sie verfügt über vier eingehende Abhängigkeiten, ihr Fan-in beträgt somit 4.

Die Stabilität *I* einer Bibliothek berechnet sich wie folgt:

$$I = \text{Fan-out} / (\text{Fan-out} + \text{Fan-in})$$

Berechnet wird also das Verhältnis der ausgehenden Abhängigkeiten zu allen Abhängigkeiten, sowohl ein- als auch ausgehend. Der berechnete Wert *I* wie Instability besagt, wie stabil eine Bibliothek ist. Es ergeben sich Werte im Bereich von 0 bis 1:

$I == 0$, Instability == false, => maximal stabil

$I == 1$, Instability == true, => maximal instabil

Die Bibliothek *Component 1* ist somit maximal instabil (Instability == 1):

$$\text{Fan-in} == 0, \text{Fan-out} == 3 \Rightarrow 3 / (0 + 3) == 1$$

Bei der Bibliothek *Component 2* ist es umgekehrt, sie ist maximal stabil (Instability == 0):

$$\text{Fan-in} == 4, \text{Fan-out} == 0 \Rightarrow 0 / (4 + 0) == 0$$

Das Beispiel der Bibliothek *Component 3* soll verdeutlichen, dass es bei der Berechnung ausschließlich um Abhängigkeiten geht, die von außen betrachtet werden. Die Abhängigkeit von *F* nach *G* ist bei der Berechnung nicht relevant:

$$\text{Fan-in} == 1, \text{Fan-out} == 2 \Rightarrow 2 / (1 + 2) == 2/3$$

Diese Bibliothek ist somit eher instabil, da sie über mehr ausgehende als eingehende Abhängigkeiten verfügt.

Die Berechnung der Stabilität mag auf den ersten Blick kompliziert erscheinen. Die Idee hinter dem Prinzip ist jedoch leicht nachvollziehbar: Wenn eine Bibliothek überwiegend von anderen abhängig ist, wird sie dadurch instabil. Das ergibt sich aus der Tatsache, dass diese Bibliothek gegebenenfalls angepasst werden muss, wenn sich Änderungen an den Bibliotheken ergeben, von denen sie abhängig ist.

Umgekehrt muss eine Bibliothek stabil sein, von der viele andere Bibliotheken abhängig sind. Das beste Beispiel ist das .NET Framework. Es muss maximal stabil sein, da sehr viele Bibliotheken vom .NET Framework abhängig sind.

Somit liegen in einer Hierarchie von Bibliotheken am unteren Ende die stabilen Bibliotheken, da diese von keinen an-

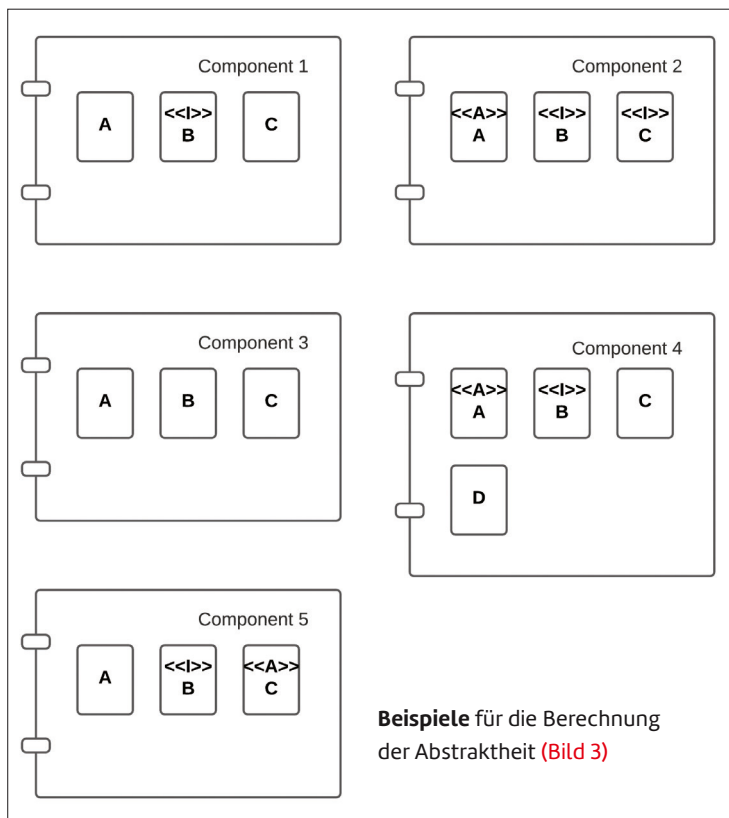
der Bibliothek. Der Grad der Abstraktheit einer Bibliothek hängt davon ab, wie viele abstrakte Elemente im Verhältnis zur Gesamtzahl sie enthält. Betrachtet werden dabei Interfaces und abstrakte Klassen sowie konkrete Klassen. Die Abstraktheit wird wie folgt berechnet:

- N_A – Anzahl der Interfaces und abstrakten Klassen
- N_C – Anzahl der Interfaces, abstrakten und konkreten Klassen
- $A = N_A / N_C$
- $A == 0$, Abstraktheit == false, => maximal konkret
- $A == 1$, Abstraktheit == true, => maximal abstrakt

Auch bei dieser Metrik liegen die Werte im Bereich von 0 bis 1.

In **Bild 3** enthält die Bibliothek *Component 1* das Interface *B* sowie die beiden konkreten Klassen *A* und *C*. Daraus ergibt sich der Wert der Abstraktheit wie folgt:

$$A = 1 / 3, \text{ eher konkret}$$



Die Bibliothek *Component 2* enthält ausschließlich abstrakte Klassen und Interfaces, ist somit also maximal abstrakt. Bei *Component 3* ist es genau umgekehrt: Sie enthält ausschließlich konkrete Klassen und ist daher maximal konkret.

Die Aussage des Stable Abstractions Principle (SAP) besagt, dass Abstraktheit und Stabilität einer Bibliothek im gleichen Verhältnis stehen sollen. Abstrakte Bibliotheken müssen somit stabil sein, während konkrete Bibliotheken instabil sein können. Dazu ist es erforderlich, wieder die Abhängigkeitsstruktur der Bibliotheken zu betrachten, weil sich ansonsten keine Aussage über ein- und ausgehende Abhängigkeiten treffen lässt.

Main Sequence

Eine interessante Erkenntnis ergibt sich, wenn man Abstraktheit und Stabilität von Bibliotheken ins Verhältnis setzt. Dazu berechnet man den Abstand einer Bibliothek von der sogenannten Main Sequence:

$$D = |A + I - 1|$$

$D == 0$: Bibliothek liegt auf der Main Sequence

$D == 1$: Bibliothek liegt maximal entfernt von der Main Sequence

deren Bibliotheken abhängig sind. Je weiter wir in der Struktur nach oben schauen, desto instabiler werden die Bibliotheken, da sie durch die Abhängigkeiten von Änderungen an darunterliegenden Bibliotheken betroffen sind.

Das Stable Dependencies Principle (SDP) besagt somit, dass die Abhängigkeiten von instabilen zu stabilen Bibliotheken verlaufen müssen. Der Wert I für die Instabilität muss somit in Richtung der Abhängigkeiten sinken.

Stable Abstractions Principle (SAP)

„Eine Bibliothek soll so abstrakt sein, wie sie stabil ist.“

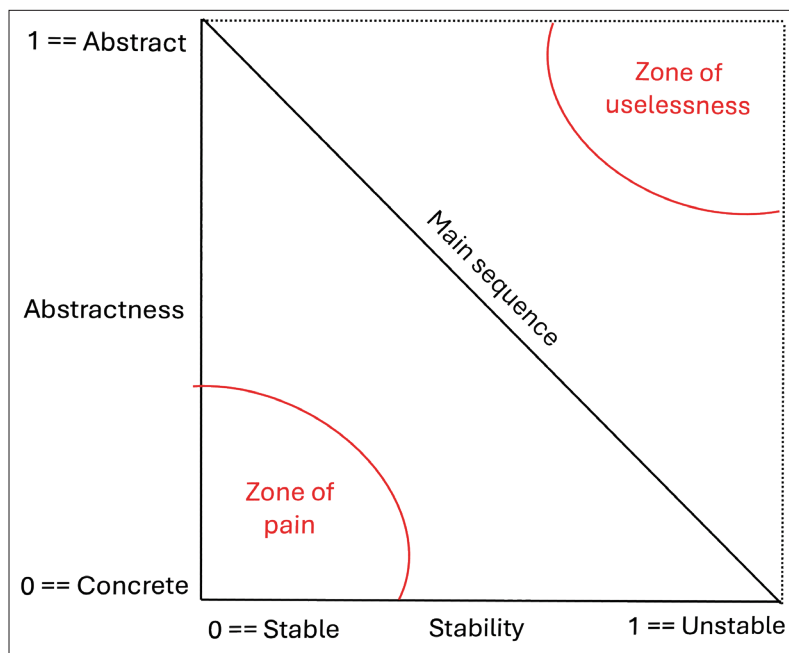
Auch beim Stable Abstractions Principle (SAP) ist eine Metrik erforderlich. In diesem Fall eine Metrik für den Grad der

Der Abstand wird berechnet als der absolute Wert aus der Summe von A und I minus 1. Stellt man die beiden Werte für Abstraktheit und Stabilität in einem Diagramm dar, kann man die beiden Problembereiche leicht identifizieren. Auf der x-Achse sehen wir die Stabilität, auf der y-Achse die Abstraktheit (**Bild 4**). Interessant sind nun die beiden Extreme links unten und rechts oben.

Zone of Pain

Links unten in **Bild 4** befindet sich die sogenannte Zone of Pain. Hier sind Stabilität und Abstraktheit jeweils 0. Das bedeutet: Die Bibliotheken in diesem Bereich sind stabil und konkret. Stabil bedeutet, dass von der Bibliothek viele ab-

hängig sind und die Bibliothek selbst gleichzeitig über keine Abhängigkeiten verfügt. Das ist zunächst nichts Ungewöhnliches. Schwierig wird es jedoch, wenn diese Bibliothek gleichzeitig auch sehr konkret ist. Dann verfügt sie über keine Abstraktionen. Das bedeutet: Änderungen am Verhalten oder den Schnittstellen führen zwangsläufig zu Änderungen bei den abhängigen Bibliotheken. Es folgt daraus, dass stabile Bibliotheken über Abstraktionen verfügen müssen.



Main-Sequence-Diagramm (Bild 4)

Am Beispiel der Entwicklung des .NET Frameworks kann man das gut nachvollziehen. In den Zeiten der ersten .NET-Framework-Versionen verfügten die meisten darin enthaltenen Klassen nicht über Interfaces. Darüber hinaus waren sie auch noch sealed, konnten also nicht beerbt werden. Das bedeutete für automatisierte Tests, dass man es sehr schnell mit konkreten Abhängigkeiten zu tun hatte, die man nicht durch Dependency Injection und Attrappen ersetzen konnte. Microsoft hat jedoch dazugelernt, und in den aktuellen Versionen verfügen viele Infrastrukturklassen über Interfaces.

Zone of Uselessness

In der rechten oberen Ecke von Bild 4 finden wir die sogenannte Zone of Uselessness. Hier liegen Bibliotheken, die instabil und abstrakt sind. Das bedeutet, diese Bibliotheken sind von anderen abhängig, wodurch sie instabil werden. Gleichzeitig verfügen sie praktisch nur über Interfaces und abstrakte Klassen. Ohne Verwender sind Interfaces und abstrakte Klassen jedoch nutzlos. Aus der Instabilität folgt, dass niemand von diesen Bibliotheken abhängig ist, es somit keine Verwender gibt.

Den Bereich der Zone of Uselessness wird man eher nicht versehentlich erreichen. Die Zone of Pain ist im Vergleich dazu schon kritischer.

Fazit

Bei der Gestaltung von Bibliotheken können wir uns bei den sechs dargestellten Prinzipien einige Anregungen holen. Mein wichtigster Rat lautet: Wenn du eine Anwendung entwickelst, baue kein Framework. Es gibt fundamentale Unterschiede zwischen dem Entwickeln einer Anwendung und dem eines Frameworks. Ein Framework zeichnet sich dadurch aus, dass es viele Verwender gibt. Somit müssen die Bibliotheken, aus denen das Framework besteht, stabil sein und über Abstraktionen verfügen. Das führt dazu, dass die öffentliche Schnittstelle eines solchen Frameworks gut durchdacht sein will. Im Rahmen einer Anwendungsentwicklung fallen solche Überlegungen nicht an, da es keine Wiederverwendung gibt. Wiederverwendung ist kein Ziel oder Wert. Stattdessen ergibt sie sich manchmal aus der Tatsache, dass man Dinge zum wiederholten Male implementiert. Aus eigener Erfahrung kann ich sagen, dass es sehr schwer ist, Funktionalität, die man in mehreren Anwendungen gebrauchen kann, in ein Framework herauszuziehen. Ein Framework muss als Paket abgelegt werden, benötigt also einen geordneten Versionierungsprozess. Stellt man ad hoc fest, dass man im eigenen Framework eine zusätzliche Funktionalität benötigt oder bestehende Funktionalität verändern möchte, sind plötzlich mehrere Abhängige davon betroffen. Oft ist es dann einfacher, mit gewissen Dopplungen im Code zu leben.

Die zweite wichtige Frage im Zusammenhang mit Bibliotheken ergibt sich aus der Teamstruktur. Arbeite ich alleine an einer Anwendung, komme ich zu einem anderen Schnitt von Bibliotheken als bei der Arbeit in einem Team. Arbeiten sogar mehrere Teams gemeinsam an einer Anwendung, ergeben sich wiederum andere Aspekte für den Zuschnitt von Bibliotheken. Es gibt also nicht die eine richtige Aufteilung von Klassen in Bibliotheken. Stattdessen entwickelt und verändert sich dies im Zeitverlauf einer Anwendungsentwicklung.

Die Metriken für Stabilität und Abstraktheit können bei Bestandscode helfen, problematische Strukturen zu erkennen. Natürlich sollte man dazu Tools verwenden, mit denen die Metriken bestimmt werden. ■

[1] Stefan Lieser, *Mehr Struktur im Begriffschaos*, dotnetpro 6-7/2025, Seite 100 ff.



Stefan Lieser

sucht ständig nach Verbesserung und neuen Wegen, um die innere Qualität von Software zu optimieren. Gemeinsam mit Ralf Westphal hat er die Clean Code Developer Initiative (<https://clean-code-developer.de>) ins Leben gerufen. <https://ccd-akademie.de>

SYSTEMARCHITEKTUREN IN AZURE

Strategien für zukunfts-sichere Cloud-Lösungen

Wer Azure-Lösungen baut, muss in Systemen denken, nicht in einzelnen Services.

Viele Cloud-Projekte starten mit einer sauberen technischen Basis. Die Anwendung ist modular aufgebaut, folgt etablierten Architekturmustern und wird automatisiert bereitgestellt. Continuous Integration und Continuous Deployment sind etabliert, Infrastruktur wird deklarativ beschrieben, Observability ist eingerichtet. Auf den ersten Blick wirkt das System robust.

Doch die Produktionsrealität folgt anderen Regeln als Architekturdiagramme. Last steigt nicht linear, sondern sprunghaft. Externe Abhängigkeiten verhalten sich nicht deterministisch. Sicherheitsanforderungen ändern sich unter regulatorischem Druck. Organisationen restrukturieren Teams, verschieben Verantwortlichkeiten oder integrieren neue Geschäftsbereiche. Und gelegentlich fällt schlicht eine komplette Region aus. In solchen Momenten zeigt sich, ob eine Lösung lediglich funktional korrekt implementiert wurde oder ob sie als System tragfähig entworfen ist.

Der Unterschied ist subtil, aber entscheidend. Eine Anwendung kann nach allen Regeln der Clean Architecture gebaut sein und dennoch als Gesamtsystem versagen. Eine perfekt gekapselte Domänenschicht hilft wenig, wenn alle Requests synchron an einen einzelnen Engpass gebunden sind. Ein skalierbares API nützt nichts, wenn dessen Abhängigkeiten nicht mithalten können. Ein formal abgesichertes Deployment-Modell verliert an Wert, wenn Sicherheitskonzepte nicht mit der Organisationsstruktur skalieren.

Zukunftssichere Lösungen zeichnen sich nicht dadurch aus, dass sie heute alle Anforderungen perfekt erfüllen. Ihr Hauptmerkmal ist, dass sie mit Unsicherheit umgehen können. Sie absorbieren Lastspitzen, ohne in sich zusammenzufallen. Sie tolerieren Teilausfälle, ohne vollständig unbrauchbar zu werden. Sie erlauben organisatorische Veränderungen, ohne massive Reibungsverluste in der Infrastruktur zu erzeugen. Und sie tun all das, ohne wirtschaftlich aus dem Ruder zu laufen.

Der Kern dieses Artikels ist daher nicht die Vorstellung einzelner Azure-Services. Im Mittelpunkt stehen architektonische Strategien auf Systemebene: Wie lassen sich Cloud-Lösungen so gestalten, dass sie Resilienz, Skalierbarkeit, Sicherheit und Kostenkontrolle nicht als nachgelagerte Optimierung behandeln, sondern als integrale Eigenschaften des Designs?

Wer als Solution Architect, Senior Engineer oder technischer Entscheider Azure-Systeme verantwortet, muss lernen, in Wechselwirkungen zu denken. Jede Entscheidung zu Kommunikationsmustern, Datenhaltung, Identität oder De-

ployment hat systemische Effekte. Diese Effekte sind es, die über Zukunftsfähigkeit entscheiden.

Die folgenden Abschnitte analysieren diese Zusammenhänge und leiten konkrete Strategien ab, wie sich belastbare, anpassungsfähige Cloud-Systeme entwerfen lassen.

Zukunftssicherheit als architektonische Zielgröße

Der Begriff „zukunftsicher“ wird im Projektalltag häufig verwendet, bleibt jedoch oft unscharf. Gemeint ist selten eine konkrete technische Eigenschaft. Stattdessen fungiert er als Platzhalter für ein diffuses Sicherheitsgefühl: Die gewählte Plattform ist modern, die Architektur folgt aktuellen Mustern, die Infrastruktur ist automatisiert. Damit sollte man „für die nächsten Jahre gut aufgestellt“ sein.

Diese Annahme greift zu kurz. Zukunftssicherheit ist keine Frage technologischer Modernität. Sie ist eine Systemeigenschaft. Ein System ist dann zukunftsicher, wenn es auf Veränderung reagieren kann, ohne dass grundlegende Struktur-entscheidungen permanent revidiert werden müssen. Veränderung ist dabei kein Sonderfall, sondern Normalzustand.

Veränderung als konstanter Faktor

In modernen Softwarelandschaften manifestiert sich Veränderung auf mehreren Ebenen gleichzeitig. Last steigt sprunghaft, Integrationen kommen hinzu oder fallen weg, Sicherheitsanforderungen verschärfen sich, Teams und Verantwortlichkeiten verschieben sich. Gleichzeitig entwickeln sich Plattformdienste kontinuierlich weiter. Ein System, das unter diesen Bedingungen stabil bleibt, ist nicht perfekt geplant. Es ist anpassungsfähig konstruiert.

Zukunftssicherheit bedeutet daher vor allem drei Dinge:

- **Erstens:** Entscheidungen müssen mit Blick auf Reversibilität getroffen werden. Architektur sollte Korrekturen ermöglichen, ohne fundamentale Umbauten zu erzwingen. Klare Schnittstellen, entkoppelte Komponenten und deklarativ beschriebene Infrastruktur schaffen Spielräume. Starre, implizite Abhängigkeiten verbauen sie.
- **Zweitens:** Stabilität unter Stress ist wichtiger als Optimierung im Idealzustand. Systeme müssen mit Lastspitzen, Teilausfällen und verzögerten Abhängigkeiten umgehen können. Ein System, das im Normalbetrieb effizient ist, unter Last jedoch unkontrolliert eskaliert, ist nicht zukunftsfähig.
- **Drittens:** Wirtschaftlichkeit ist Teil der Architektur. In Cloud-Umgebungen wie Microsoft Azure sind Kosten ein direktes

Resultat von Designentscheidungen. Überprovisionierung, falsche Kommunikationsmuster oder ungeeignete Persistenzmodelle wirken sich unmittelbar auf das Budget aus. Nachhaltige Architektur berücksichtigt dieses Kostenverhalten von Anfang an.

Zukunftssicherheit entsteht somit nicht durch maximale Vorsorge oder technologische Breite. Sie entsteht durch bewusste Systemgestaltung, klare Annahmen und die Bereitschaft, Architektur als kontinuierlichen Prozess zu verstehen und nicht als einmalige Entwurfsphase.

Architektur auf der richtigen Ebene denken

In vielen Projekten wird Architektur primär im Kontext des Codes diskutiert. Es geht um saubere Schichten, Dependency Inversion, modulare Strukturen oder die Entscheidung zwischen modularem Monolithen und Microservices. Diese Diskussion ist wichtig, sie greift jedoch zu kurz, sobald eine Lösung als Cloud-System betrieben wird. Applikationsarchitektur beantwortet die Frage, wie fachliche Logik strukturiert ist. Systemarchitektur beantwortet die Frage, wie Komponenten unter realen Betriebsbedingungen interagieren.

Dieser Perspektivenwechsel ist entscheidend. Ein Service kann intern vorbildlich implementiert und dennoch als Teil eines Gesamtsystems problematisch sein. Typische Symptome zeigen sich nicht im Code, sondern im Zusammenspiel:

- Latenzketten durch synchrone Aufrufe über mehrere Services hinweg
- Unerwartete Lastverstärkung durch Retries entlang einer Abhängigkeitskette
- Zentrale Persistenz als impliziter Single Point of Failure
- Geteilte Infrastrukturressourcen als versteckte Kopplung

Systemarchitektur betrachtet daher nicht nur Komponenten, sondern auch deren Verhalten. Sie analysiert Abhängigkeiten, Failure Domains und Kommunikationsmuster. Gerade in Plattformen wie Microsoft Azure entstehen Abhängigkeiten häufig nicht nur zwischen eigenen Services, sondern auch zu Managed Services. Diese sind hochverfügbar, aber nicht unfehlbar. Ihre Einbindung definiert maßgeblich das Stabilitätsprofil des Gesamtsystems. Der entscheidende Unterschied: Während Applikationsarchitektur primär statische Struktur beschreibt, adressiert Systemarchitektur dynamisches Verhalten unter Last, Ausfall und Veränderung.

Design for Failure: Ausfälle sind Normalzustand

Verteilte Systeme sind inhärent fehlertoleranzbedürftig. Netzwerke sind nicht zuverlässig, Time-outs gehören zum Alltag, Abhängigkeiten reagieren verzögert oder inkonsistent. Cloud-Plattformen abstrahieren Infrastruktur, eliminieren jedoch keine Fehler. Design for Failure bedeutet, Ausfälle als normalen Betriebszustand zu akzeptieren.

Architektur muss davon ausgehen, dass jede Abhängigkeit temporär nicht erreichbar sein kann. Das beginnt bei scheinbar trivialen Aspekten wie Time-outs. Ein fehlender Time-out ist in einem verteilten System kein Komfortmerkmal, sondern ein Risiko. Blockierende Threads, erschöpfte Connection

Pools und steigende Latenzen sind typische Folgewirkungen. Ebenso kritisch sind unkontrollierte Retry-Strategien. Ein einzelner fehlgeschlagener Request, der automatisch mehrfach wiederholt wird, mag harmlos erscheinen. In einem System mit hoher Parallelität kann dies jedoch zu Lastlawinen führen. Mehrere Services, die gleichzeitig auf eine instabile Abhängigkeit mit aggressiven Retries reagieren, verstärken die Störung selbst.

Resilienzmechanismen müssen daher systemisch betrachtet werden. Circuit Breaker verhindern, dass fehlerhafte Abhängigkeiten dauerhaft mit Traffic überflutet werden. Bulkheads begrenzen die Auswirkungen eines Fehlers auf klar definierte Ressourcenpools. Idempotenz stellt sicher, dass wiederholte Requests keine inkonsistenten Zustände erzeugen.

Wichtig ist die fachliche Dimension: Nicht jeder Fehler muss transparent für den Endnutzer sein. Systeme sollten gezielt degradieren können. Ein temporär nicht verfügbarer Empfehlungsservice darf nicht den gesamten Bestellprozess blockieren. Diese Priorisierung ist keine technische, sondern eine fachliche Entscheidung und damit Teil der Architektur.

Kommunikationsmuster: Synchrone Bequemlichkeit versus systemische Stabilität

Synchrone HTTP-Kommunikation ist in .NET und Azure einfach umzusetzen. Sie entspricht dem mentalen Modell klassischer Client-Server-Systeme. Ein Request wird gestellt, eine Antwort erwartet, danach wird fortgefahren.

In isolierten Szenarien funktioniert dieses Modell hervorragend. In komplexen Systemlandschaften entstehen jedoch Ketten von Abhängigkeiten. Service A ruft Service B auf, dieser wiederum Service C. Jeder zusätzliche synchrone Call erhöht die Gesamtlatenz und erweitert die Failure Domain.

Das Problem ist weniger der einzelne Aufruf als die Kaskade. Fällt Service C aus, blockieren B und anschließend A. Retries, die nicht abgestimmt sind, verschärfen die Situation.

Asynchrone Kommunikation reduziert diese Kopplung. Messaging-basierte Integrationen ermöglichen es, Last zu puffern und Komponenten zeitlich zu entkoppeln. Ein Auftrag kann entgegengenommen werden, auch wenn die nachgelagerte Verarbeitung verzögert erfolgt. Eventual Consistency wird akzeptiert, um Stabilität zu gewinnen.

Diese Entscheidung hat Konsequenzen. Konsistenzmodelle werden komplexer, Monitoring anspruchsvoller, Fehleranalyse diffiziler. Dennoch ist die bewusste Entkopplung oft der Preis für Skalierbarkeit und Resilienz. Architektur bedeutet hier, fachliche Anforderungen gegen technische Stabilität abzuwägen. Nicht jede Interaktion muss synchron sein. Aber jede synchrone Interaktion sollte bewusst begründet werden.

Skalierung als architektonische Eigenschaft

Cloud-Plattformen suggerieren, Skalierung sei eine Konfigurationsfrage. Instanzen werden automatisch hinzugefügt, Ressourcen dynamisch angepasst. Dieses Modell funktioniert nur dann, wenn die Architektur skalierungsfähig ist.

Horizontale Skalierung setzt Zustandslosigkeit voraus oder zumindest klar kontrollierte Zustandsverwaltung. Gemeinsame Datenbanken, In-Memory-Caches oder Session-Stores ►

können Skalierungsgrenzen definieren, die durch bloßes Hinzufügen von Compute-Ressourcen nicht überwunden werden.

Ein häufiger Engpass ist die Datenbank. Solange alle Services auf eine zentrale Instanz zugreifen, bleibt diese der limitierende Faktor. Skalierungsstrategien müssen daher Datenmodelle einbeziehen: Partitionierung, Sharding, Trennung von Lese- und Schreibmodellen oder gezielte Denormalisierung können erforderlich werden. Auch externe Abhängigkeiten definieren Skalierungsgrenzen. APIs mit Rate Limits oder Identity-Provider mit begrenztem Durchsatz können das gesamte System ausbremsen. Skalierung ist daher nicht nur eine Frage eigener Ressourcen, sondern auch des Umgangs mit fremden Beschränkungen. Architektonisch relevant ist zudem die Frage der Laststeuerung. Ein System, das jede eingehende Anfrage ungeprüft akzeptiert, kann sich selbst überlasten. Backpressure-Mechanismen, Warteschlangen oder gezielte Drosselung sind legitime Mittel, um Stabilität zu sichern.

Skalierung ist damit keine rein technische Optimierung, sondern eine strukturelle Eigenschaft des Systems.

Hochverfügbarkeit: Mehr als Prozentwerte

Service Level Agreements definieren Verfügbarkeitszusagen einzelner Dienste. In der Praxis ist die effektive Systemverfügbarkeit jedoch das Produkt aller Abhängigkeiten. Mehrere Services mit jeweils 99,9 Prozent Verfügbarkeit ergeben keine 99,9 Prozent Systemverfügbarkeit. Abhängigkeiten multiplizieren die Risiken. Architektonisch bedeutet Hochverfügbarkeit, Failure Domains klar zu definieren und Redundanz gezielt einzusetzen. Verfügbarkeitszonen innerhalb einer Region reduzieren das Risiko lokaler Infrastrukturprobleme. Regionale Redundanz schützt vor großflächigen Ausfällen.

Multi-Region-Architekturen erhöhen die Resilienz, bringen jedoch erhebliche Komplexität mit sich. Datenreplikation über Regionen hinweg wirft Fragen zu Konsistenz, Konfliktlösung und Failover-Strategien auf. Automatisiertes Failover ist anspruchsvoll, manuelles Failover erfordert trainierte Prozesse.

Nicht jedes System benötigt globale Redundanz. Entscheidend ist die fachliche Bewertung: Welche Ausfallzeiten sind akzeptabel? Welche Datenverluste tolerierbar? Daraus leiten sich Recovery Time Objective und Recovery Point Objective ab. Erst dann lassen sich sinnvolle Architekturentscheidungen treffen.

Hochverfügbarkeit ist kein Feature, das aktiviert wird. Sie ist das Ergebnis konsistenter Architekturentscheidungen entlang aller Systemebenen.

Sicherheit als integraler Bestandteil der Systemarchitektur

In modernen Cloud-Systemen verschiebt sich der Sicherheitsfokus von Netzwerkperimetern hin zu Identität und Zugriffskontrolle. Dienste kommunizieren über verschlüsselte Verbindungen, Workloads laufen in isolierten Umgebungen. Die entscheidende Frage lautet: Wer darf was?

Identity wird zur zentralen Steuergröße. Service-to-Service-Kommunikation sollte nicht über statische Secrets abgesichert werden, sondern über kurzlebige, verwaltete Identitäten. Rollen- und Rechtekonzepte müssen so gestaltet sein,

dass sie sowohl minimal notwendige Berechtigungen als auch organisatorische Veränderungen unterstützen.

Governance ist dabei keine rein administrative Disziplin. Richtlinien für Ressourcenbereitstellung, Naming Conventions, Tagging oder Netzwerksegmente beeinflussen die Architektur unmittelbar. Werden diese Aspekte nicht früh berücksichtigt, entstehen heterogene Landschaften, die schwer kontrollierbar sind.

Security by Design bedeutet, Sicherheitsanforderungen nicht als nachgelagerten Prüfpunkt zu behandeln, sondern als Entwurfsparameter. Jede neue Komponente muss sich in das bestehende Identitäts- und Berechtigungsmodell einfügen. Architektur und Sicherheitskonzept sind untrennbar.

Observability: Architektur sichtbar machen

Ein System, das sich nicht beobachten lässt, kann nicht stabil betrieben werden. Logs, Metriken und Traces sind keine Zusatzfunktion, sondern elementare Architekturbestandteile.

Gerade in verteilten Systemen ist die Korrelation von Requests entscheidend. Ohne durchgängige Trace-Informationen bleibt unklar, welche Abhängigkeit Latenz verursacht oder wo Fehlerketten entstehen. Monitoring dient dabei nicht nur der Alarmierung, sondern der Validierung architektonischer Annahmen. Lasttests und Chaos-Experimente liefern wertvolle Erkenntnisse über Systemverhalten unter Stress. Sie machen sichtbar, ob Retry-Strategien sinnvoll greifen, ob Circuit Breaker korrekt konfiguriert sind und ob Skalierungsmechanismen tatsächlich wie erwartet reagieren.

Observability ist damit ein Feedback-Mechanismus für Architekturentscheidungen. Sie schließt den Kreis zwischen Design und Betrieb.

Die Referenz: Ein belastbares Azure-System

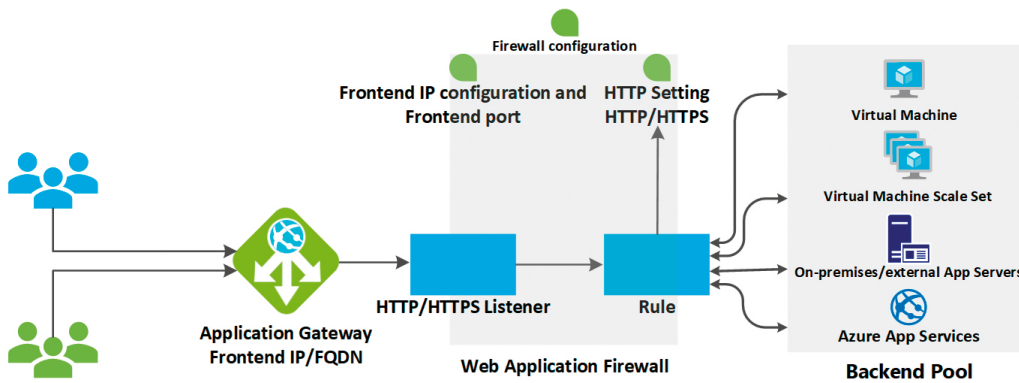
Eine Referenzarchitektur für ein belastbares Cloud-System auf Microsoft Azure ist kein starres Baukastendiagramm. Sie ist ein strukturierter Denkraum, der typische Verantwortlichkeiten sauber trennt, Failure Domains definiert und nicht-funktionale Anforderungen explizit adressiert.

Die konkrete Auswahl einzelner Dienste, ob App-Service oder Container-Plattform, ob relationale oder dokumentenorientierte Datenbank, ist dabei zweitrangig. Entscheidend ist, wie die Bausteine zusammenspielen und welche systemischen Eigenschaften daraus entstehen.

Im Folgenden wird eine typische Referenzarchitektur beschrieben, die sich in vielen produktiven Szenarien bewährt hat. Sie dient als Ausgangspunkt für Diskussion und Anpassung, nicht als Dogma.

API-Gateway als kontrollierter Einstiegspunkt

Jedes belastbare System benötigt einen klar definierten Eintrittspunkt. Ein öffentlich erreichbares API-Gateway übernimmt diese Rolle. Seine Aufgaben gehen über simples Routing hinaus. Es terminiert TLS-Verbindungen, validiert Tokens, setzt Rate Limits, erzwingt Quotas und bildet eine erste Schutzschicht gegen Überlast oder Missbrauch. Darüber hinaus kapselt es interne Strukturen gegenüber externen Konsumenten.



Übersicht über die Funktionsweise von Azure Application Gateway (Bild 1)

Architektonisch erfüllt das Gateway mehrere Funktionen:

- **Begrenzung der Angriffsfläche:** Interne Services sind nicht direkt exponiert, sondern nur über klar definierte Schnittstellen erreichbar.
- **Versionierung und kontrollierte Evolution** für externe APIs, ohne interne Implementierungen offenzulegen.
- **Ort für zentrale Policies:** Beispiele dafür sind Throttling oder IP-Filter. Diese gehören nicht in jeden einzelnen Service.

Kritisch ist hier die Frage nach der Verfügbarkeit. Das Gateway ist zwangsläufig eine zentrale Komponente, siehe Bild 1. Wird es zum Single Point of Failure, ist das gesamte System nicht erreichbar. Entsprechend muss es zonenredundant betrieben und in die allgemeine Resilienzstrategie eingebettet werden.

Stateless Backend-Services als Skalierungseinheit

Hinter dem Gateway stehen fachlich geschnittene Backend-Services. Ihre wichtigste Eigenschaft in einer Cloud-Referenzarchitektur ist Zustandslosigkeit zumindest in Bezug auf flüchtigen, pro Request relevanten Zustand.

Statelessness ist keine akademische Forderung, sondern eine Voraussetzung für horizontale Skalierung. Instanzen müssen beliebig hinzugefügt oder entfernt werden können, ohne dass inkonsistente Sitzungszustände entstehen. Persistenter Zustand wird daher explizit ausgelagert: in Datenbanken, Caches oder Messaging-Systeme. Lokaler Speicher oder In-Memory-Sessions sind in verteilten Szenarien problematisch.

Architektonisch relevant ist hier die Granularität der Services. Zu fein geschnittene Microservices erhöhen die Anzahl synchroner Abhängigkeiten und damit Latenz sowie Fehlerisiko. Zu grobe Services erschweren unabhängige Skalierung und Deployment.

Eine tragfähige Referenzarchitektur wählt Servicegrenzen entlang fachlicher Verantwortlichkeiten und beobachtbarer Lastprofile. Skalierungseinheiten müssen technisch sinnvoll und fachlich konsistent sein.

Asynchrone Verarbeitung als Entkopplungsmechanismus

Nicht jede Interaktion muss synchron erfolgen. Eine belastbare Architektur nutzt asynchrone Kommunikation gezielt, um Last zu puffern und Komponenten zu entkoppeln.

Hier übernehmen Messaging-Systeme eine zentrale Rolle. Sie dienen als Puffer zwischen Eingangslast und nachgelagerter Verarbeitung. Ein eingehender Auftrag wird angenommen und in eine Warteschlange gestellt, selbst wenn die eigentliche Verarbeitung zeitverzögert erfolgt. Der Vorteil liegt in der Entkopplung von Verfügbarkeit. Fällt ein nachgelagerter Service temporär aus, bleiben eingehende Requests nicht zwingend blockiert. Die Last wird gepuffert und kontrolliert abgearbeitet. Diese Architekturentscheidung erfordert jedoch klare fachliche Definitionen. Eventual Consistency muss akzeptabel sein. Geschäftsprozesse müssen so modelliert werden, dass Zwischenzustände tolerierbar sind.

Ein Messaging-System selbst ist wiederum eine kritische Komponente. Es muss hochverfügbar betrieben und hinsichtlich Durchsatz und Backpressure-Strategien sorgfältig dimensioniert werden. Warteschlangen dürfen nicht unkontrolliert wachsen. Monitoring ist hier essenziell.

Ein Messaging-System selbst ist wiederum eine kritische Komponente. Es muss hochverfügbar betrieben und hinsichtlich Durchsatz und Backpressure-Strategien sorgfältig dimensioniert werden. Warteschlangen dürfen nicht unkontrolliert wachsen. Monitoring ist hier essenziell.

Persistenz mit klarer Partitionierungsstrategie

Datenhaltung ist häufig der limitierende Faktor in Cloud-Systemen. Eine Architektur darf Persistenz nicht als monolithischen Block behandeln. Zentrale Fragen lauten:

- Welche Daten sind transaktional konsistent erforderlich?
- Welche Daten lassen sich eventually consistent replizieren?
- Wie entwickelt sich das Datenvolumen über die Zeit?
- Welche Zugriffsmuster dominieren? Viele Reads, viele Writes, analytische Abfragen?

Eine einzelne relationale Datenbank kann für viele Szenarien ausreichend sein. Problematisch wird es, wenn sie implizit zur globalen Synchronisationsstelle des Systems wird. Partitionierung ist daher frühzeitig zu bedenken. Sie kann fachlich motiviert sein, etwa nach Mandanten, Regionen oder Aggregaten oder technisch nach Lastprofilen. Read- und Write-Modelle lassen sich trennen, um Skalierung gezielt zu ermöglichen.

Single Points of Failure entstehen häufig auf Datenebene. Selbst wenn Compute horizontal skaliert, bleibt eine zentrale Datenbankinstanz kritisch. Replikations- und Failover-Strategien müssen daher integraler Bestandteil der Architektur sein (Bild 2). Entscheidend ist, Konsistenzanforderungen explizit zu definieren. Nicht jede Operation benötigt starke Transaktionsgrenzen. Wo diese jedoch fachlich zwingend sind, müssen sie auch technisch abgesichert werden.

Zentrale Identitäts- und Zugriffssteuerung

Identität ist das Rückgrat moderner Cloud-Systeme. Eine Referenzarchitektur definiert daher ein zentrales Identitätsmodell für Benutzer und Dienste. Extern authentifizierte Benutzer erhalten Tokens, die über das API-Gateway validiert ►

werden. Die interne Service-zu-Service-Kommunikation erfolgt über verwaltete Identitäten statt über statische Secrets. Rollen- und Berechtigungsmodelle werden konsistent angewendet.

Architektonisch relevant ist die Frage, wo die Autorisierung stattfindet. Eine grobe Vorprüfung am Gateway ist sinnvoll, ersetzt jedoch keine feingranulare Autorisierung innerhalb der Services. Fachliche Regeln gehören in die Domäne, nicht in die Infrastruktur. Auch hier gilt: Der Identity-Provider ist eine kritische Abhängigkeit. Caching-Strategien für Tokenvalidierung und resiliente Konfiguration sind erforderlich, um Authentifizierungsdienste nicht zum Bottleneck werden zu lassen.

Durchgängige Observability als verbindendes Element

Alle zuvor beschriebenen Komponenten entfalten ihren Wert nur, wenn ihr Verhalten transparent ist. Observability ist daher kein separates Modul, sondern eine Querschnittseigenschaft der Architektur. Jeder Request erhält eine Korrelations-ID. Logs sind strukturiert und maschinenlesbar. Metriken bilden Latenzen, Fehlerraten und Ressourcenauslastung ab. Verteiltes Tracing macht Abhängigkeitsketten sichtbar.

Besonders wichtig ist die Beobachtung systemischer Effekte. Steigt die Retry-Rate? Wachsen Warteschlangen kontinuierlich? Nimmt die Latenz einzelner Downstream-Services zu? Solche Signale deuten auf strukturelle Probleme hin – nicht nur auf operative. Observability ermöglicht zudem gezielte Chaos-Tests. Komponenten können bewusst deaktiviert werden, um das Verhalten des Systems zu analysieren. Eine Referenzarchitektur ist erst dann belastbar, wenn sie auch unter absichtlich herbeigeführten Störungen stabil bleibt.

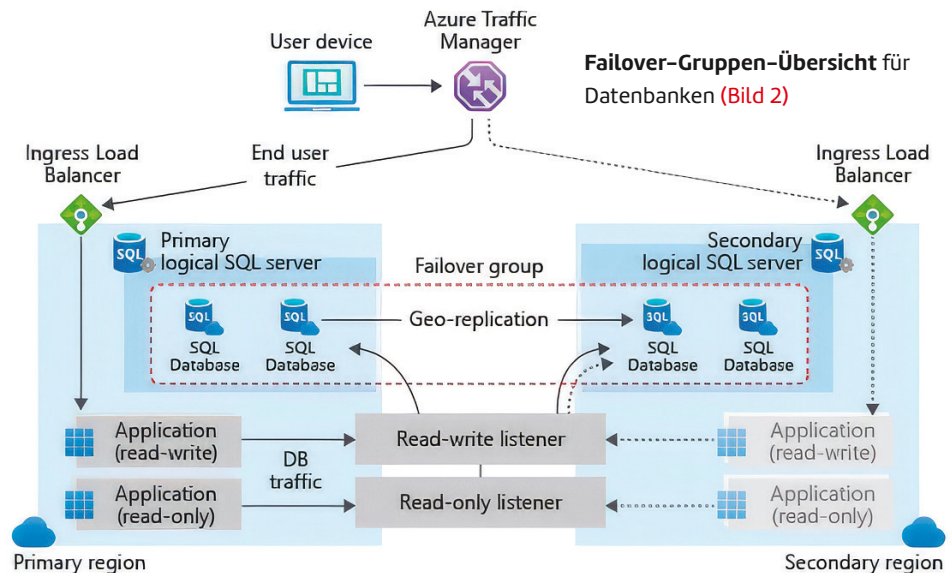
Kritikalität und Failure Domains bewusst definieren

Eine Referenzarchitektur wird erst dann konkret, wenn Kritikalität explizit bewertet wird.

- Welche Komponenten sind geschäftskritisch?
- Welche dürfen temporär ausfallen, ohne Kernfunktionen zu beeinträchtigen?
- Wo existieren implizite Single Points of Failure?

Nicht jede Funktion benötigt maximale Verfügbarkeit. Empfehlungsdienste, Reporting-Komponenten oder sekundäre Integrationen können im Zweifel degradieren. Zahlungsabwicklung oder Authentifizierung hingegen sind häufig kritisch.

Diese Differenzierung beeinflusst Redundanzstrategien, Deployment-Modelle und Monitoring-Schwellenwerte. Architektur wird dadurch priorisiert, statt pauschal auf maximale Verfügbarkeit zu optimieren.



Failure Domains müssen klar abgegrenzt sein. Ein Fehler in einem Hintergrundprozess darf nicht das gesamte API blockieren. Ressourcentrennung, etwa durch separate Compute-Pools oder isolierte Queues, ist ein bewusstes Designmittel.

Begründete Entscheidungen statt Diagramm-Ästhetik

Eine Referenzarchitektur ist kein ästhetisches Artefakt. Sie ist ein Set begründeter Entscheidungen.

- Warum wurde Messaging an dieser Stelle eingeführt?
- Warum wird diese Datenbank partitioniert?
- Warum ist Multi-Region hier notwendig – oder warum bewusst nicht?
- Warum ist ein bestimmter Service als kritisch eingestuft?

Diese Begründungen müssen sowohl technisch als auch fachlich nachvollziehbar sein. Architektur lebt von Transparenz. Nur so lassen sich Trade-offs diskutieren und bei veränderten Rahmenbedingungen neu bewerten.

Ein belastbares Azure-System entsteht nicht durch die Aneinanderreihung moderner Services. Es entsteht durch konsistente Prinzipien, klare Verantwortlichkeiten und eine bewusste Bewertung von Risiken.

Die Referenzarchitektur ist dabei kein Endzustand, sondern ein Orientierungsrahmen. Ihre Stärke liegt nicht in der Vollständigkeit, sondern in der Klarheit ihrer Annahmen.

Typische Fehlannahmen in Azure-Projekten

Architekturprobleme entstehen selten durch fehlendes Wissen über einzelne Dienste. Sie entstehen durch falsche Annahmen über deren Wirkung im Gesamtsystem. Einigen dieser Annahmen begegnet man in Projekten immer wieder, unabhängig von Branche oder Unternehmensgröße.

Eine besonders verbreitete Vorstellung lautet, die Plattform kümmere sich bereits um Hochverfügbarkeit. Azure stellt hochverfügbare Bausteine bereit, oft mit beeindruckenden SLA-Werten. Daraus wird schnell abgeleitet, das eigene System sei damit automatisch resilient. Tatsächlich garantiert ein SLA lediglich die Verfügbarkeit eines einzelnen Dienstes un-

ter definierten Bedingungen. Es garantiert nicht, dass das Zusammenspiel mehrerer Dienste stabil bleibt. Werden mehrere Abhängigkeiten seriell kombiniert, sinkt die effektive Systemverfügbarkeit. Hinzu kommen logische Fehler, Konfigurationsprobleme oder falsche Retry-Strategien, Aspekte, die kein SLA abdeckt. Hochverfügbarkeit entsteht durch Architekturentscheidungen entlang klar definierter Failure Domains, nicht durch das Vertrauen auf Plattformversprechen.

Ebenso hartnäckig hält sich die Annahme, Microservices seien per se skalierbar. Die Realität zeigt ein differenziertes Bild. Kleine, unabhängig deploybare Services erhöhen zwar organisatorische Flexibilität. Skalierbarkeit ergibt sich jedoch nur, wenn Datenhaltung, Kommunikationsmuster und Zustandsmanagement darauf abgestimmt sind. Ein fein geschnittenes Service-Set, das synchron auf eine zentrale Datenbank zugreift, skaliert nicht besser als ein Monolith. Es verteilt lediglich die Komplexität.

Auch Security wird häufig unterschätzt. Die Idee, Sicherheitsaspekte ließen sich später „ergänzen“, führt in verteilten Systemen fast zwangsläufig zu strukturellen Altlasten. Identitätsflüsse, Berechtigungsmodelle und Netzwerksegmente prägen die Architektur von Beginn an. Werden sie erst nachträglich eingeführt, müssen Kommunikationswege, Tokenflüsse und Zugriffspfade oft grundlegend umgebaut werden. Sicherheit ist kein Add-on, sondern eine Entwurfsentscheidung.

Ein weiteres Missverständnis betrifft Kosten. Sie werden oft als operatives Thema betrachtet, das im Betrieb optimiert werden kann. In Cloud-Umgebungen sind Kosten jedoch eine direkte Folge von Designentscheidungen. Chatty Kommunikationsmuster, überdimensionierte Ressourcen oder ungünstige Speicherstrategien lassen sich nicht allein durch nachgelagerte Optimierung kompensieren. Wirtschaftlichkeit ist eine architektonische Eigenschaft.

Diese Fehlannahmen haben eines gemeinsam: Sie verschieben Verantwortung von der Architektur auf die Plattform oder den Betrieb. Genau diese Verschiebung rächt sich im produktiven Alltag.

Lessons Learned aus realen Projekten

Theoretische Prinzipien gewinnen erst dann an Schärfe, wenn sie mit realen Projekterfahrungen abgeglichen werden. Bestimmte Muster zeigen sich dabei immer wieder.

Lasttests sind keine Performance-Spielerei, sondern Architekturtests. Sie zeigen nicht nur, wie schnell ein System ist, sondern wie es unter Druck reagiert. Eskalieren Fehlerraten exponentiell? Blockieren Threads durch fehlende Time-outs? Wachsen Warteschlangen unkontrolliert? Solche Effekte offenbaren strukturelle Schwächen, die im Normalbetrieb verborgen bleiben.

Ein häufiger Fehler sind unkoordinierte Retry-Strategien. Mehrere Services, die bei einem Downstream-Fehler aggressiv erneut anfragen, erzeugen zusätzliche Last genau dort, wo bereits Instabilität herrscht. Ohne Back-off und Begrenzung können Retries selbst zur Ursache eines Ausfalls werden. Resilienzmechanismen müssen systemweit abgestimmt sein.

Multi-Region-Architekturen werden oft als ultimative Hochverfügbarkeitslösung verstanden. In der Praxis entsteht

jedoch schnell Scheinsicherheit. Wenn Failover-Prozesse nicht automatisiert und regelmäßig getestet werden, bleibt die zweite Region ein theoretisches Konstrukt. Im Ernstfall fehlen erprobte Abläufe, Datenreplikation ist nicht konsistent, Konfigurationen weichen ab. Redundanz ohne Übung ist kein Sicherheitsnetz.

Fehlende Telemetrie verlängert Incident-Zeiten dramatisch. Ohne durchgängige Korrelation von Requests, strukturierte Logs und belastbare Metriken bleibt Ursachenanalyse spekulativ. Teams reagieren symptomatisch statt ursächlich. Observability ist kein Luxus, sondern Voraussetzung für kontrollierten Betrieb.

Gleichzeitig zeigt die Praxis, dass zu frühe Optimierung für Extremszenarien Komplexität erzeugt, die lange nicht benötigt wird. Globale Multi-Region-Setups, komplexe Sharding-Strategien oder hochgradig generische Plattformarchitekturen binden Ressourcen und erhöhen die kognitive Last. Architektur muss realistische Annahmen treffen und iterativ wachsen.

Zukunftssichere Systeme entstehen daher nicht durch einmalige „große“ Entwürfe. Sie entwickeln sich schrittweise. Annahmen werden validiert, Engpässe identifiziert, Strukturen angepasst. Architektur ist ein kontinuierlicher Prozess, technisch wie organisatorisch.

Fazit: Cloud Development als Systemdisziplin

Cloud Development ist keine Ansammlung moderner Dienste. Es ist eine architektonische Disziplin. Plattformen wie Microsoft Azure bieten leistungsfähige Bausteine für Skalierung, Resilienz und Sicherheit. Diese Bausteine entfalten ihren Wert jedoch nur im Kontext einer durchdachten Systemarchitektur.

Zukunftssichere Lösungen zeichnen sich nicht durch maximale Komplexität oder technologische Breite aus. Sie entstehen durch bewusste Reduktion von Kopplung, explizite Behandlung von Fehlern und klar definierte Failure Domains. Skalierung wird systemisch gedacht, nicht konfigurationsgetrieben. Sicherheitskonzepte sind integraler Bestandteil des Designs. Kostenverhalten ist transparent und steuerbar. Observability verbindet Entwurf und Betrieb.

Am Ende entscheidet nicht, wie modern die verwendeten Services sind, sondern wie robust das System auf reale Belastungen reagiert. Architektur zeigt ihre Qualität nicht im Idealzustand, sondern unter Stress.

Wer in Systemen denkt, akzeptiert Unsicherheit als Grundbedingung. Genau daraus entstehen Lösungen, die auch dann tragfähig bleiben, wenn sich Rahmenbedingungen ändern – technisch, fachlich oder organisatorisch. ■



Florian Lenz

ist Microsoft MVP und Experte für Cloud-native und Serverless-Lösungen in Azure. Er unterstützt Unternehmen dabei, ihre Cloud-Landschaften zu optimieren und Architekturen aufzubauen, die nachhaltigen Erfolg garantieren.

www.florian-lenz.io

SICHERHEITSRISIKEN FÜR WEBANWENDUNGEN: OWASP TOP TEN 2025

Auffrischkurs Web-Security

Die neue OWASP Top Ten ist da – die erste Ausgabe seit vier Jahren.

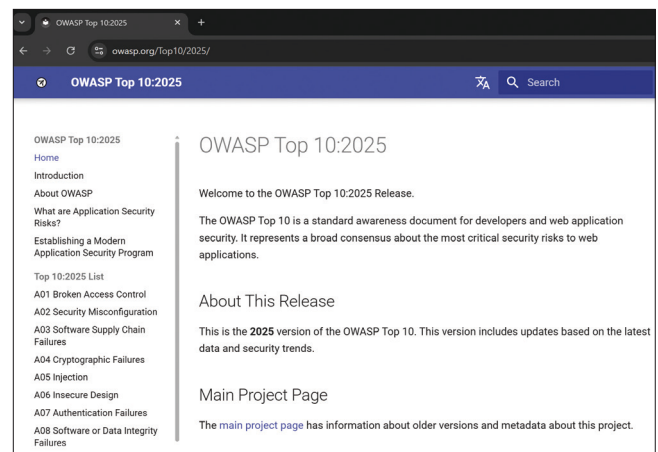
Was bedeutet das für ASP.NET Core?

Gegründet wurde die OWASP [1] im September 2001. Das Kürzel stand damals für „Open Web Application Security Project“. Dieser Web-Fokus der Organisation wurde mehr als 20 Jahre später etwas erweitert – nicht nur klassische Webanwendungen sind Sicherheitsrisiken ausgesetzt. Mittlerweile ist die OWASP das Open Worldwide Application Security Project. Die Mission ist also weiterhin die Sicherheit von Anwendungen, die heute quasi alle irgendwie mit Webtechnologien arbeiten, und wenn auch nur irgendetwas davon mit HTTP ist.

Die Angebote der OWASP sind schier endlos. Das Projektinventar [2] verzeichnet über 400 verschiedene Einträge mit natürlich stark unterschiedlichem Reifegrad. Doch neben einigen schlecht bis gar nicht gepflegten Initiativen gibt es auch viele nützliche Angebote und Informationen: Checklisten, Standards, technische Dokumentationen, Sicherheitssoftware – für fast alle Aspekte rund um die Software-Sicherheit findet sich etwas bei der OWASP.

Dabei ist es schon fast ein wenig schade, dass in der öffentlichen Wahrnehmung ein Projekt mit Abstand den größten Bekanntheitsgrad hat: die OWASP Top Ten [3]. Diese Liste der zehn größten Sicherheitsrisiken für Webanwendungen wurde erstmals 2003 veröffentlicht. Aus dem ursprünglich jährlichen Aktualisierungsrhythmus wurden irgendwann drei und schließlich vier Jahre, die zwischen Updates der Liste gewartet werden musste. Mit der vorherigen OWASP Top Ten von 2021 haben wir uns in der dotnetpro bereits vor Jahren beschäftigt [4]. Und seit Anfang 2025 waren sowohl Autor als auch Redaktion in Lauerstellung, schließlich wurde die Liste für 2025 schon frühzeitig angekündigt. Mal für die erste Jahreshälfte, mal für Spätsommer/Frühherbst, doch nichts passierte. Dann, kurz vor Torschluss, gab es Anfang November 2025 einen Release-Kandidaten der Liste. Der Hintergrund war ziemlich profan: In der ersten Novemberwoche fand die Konferenz OWASP Global AppSec USA [5] statt. Die neue OWASP Top Ten stand auf dem Programm, und größere OWASP-Veranstaltungen gab es danach nicht mehr im Jahr 2025.

Das Projektteam hat abgeliefert, zumindest größtenteils. Veröffentlicht wurde, wie bereits erwähnt, eine Vorabversion, nebst der Bitte, doch Feedback zu geben. Dieses trudelte eher spärlich ein, die Jahresuhr tickte weiter. Am 24. Dezember 2025 (!) schließlich wurde dann die finale Version der OWASP Top Ten 2025 (Bild 1) veröffentlicht, ohne große Fanfaren, nicht



Was lange währt: Die OWASP Top Ten 2025 (Bild 1)

einmal der offizielle X-Account [6] wurde aktualisiert (Bild 2), geschweige denn die eigens für die Top Ten angelegte Domain owasptopten.org [7]. Wen das Datum verwundert: Dass am 24. Dezember eine etwaige Weihnachtsbescherung stattfindet, ist eine Spezialität des deutschsprachigen Raums und skandinavischer Länder. In den USA beispielsweise, Hauptsitz der OWASP-Organisation, ist das ein Tag wie jeder andere – Geschenke gibt es, wenn überhaupt, erst am 25. Dezember. Insofern war der 24. 12. eine der letzten Möglichkeiten für eine Veröffentlichung der Liste. Das Social-Media-Team war da offenbar bereits im Urlaub.

Aber genug des Zynismus. Die Liste von 2025 ist da, also behalten wir die Tradition bei und werfen einen Blick auf den Entstehungsprozess und dann alle zehn Punkte. Dabei haben wir selbstredend die Microsoft-Brille auf: Was bedeuten die aufgeführten Sicherheitsrisiken für Webanwendungen mit ASP.NET Core und Blazor? Gibt es technologiespezifische Sicherheitsmaßnahmen?

Bevor es losgeht, ein essenzieller Disclaimer: Die OWASP Top Ten ist eine Awareness-Liste, es geht also darum, Bewusstsein für Risiken zu schaffen. Keinesfalls ist die OWASP Top Ten eine Art Checkliste für Audits oder die Reihenfolge der Listenelemente eine unumkehrbare Wahrheit. Alle Einträge der Liste sind wichtig, manche mehr, manche weniger, auch abhängig vom gewählten Stack und der Art der Applikation. Von den circa zwei Dutzend Top Tens der Organisa-

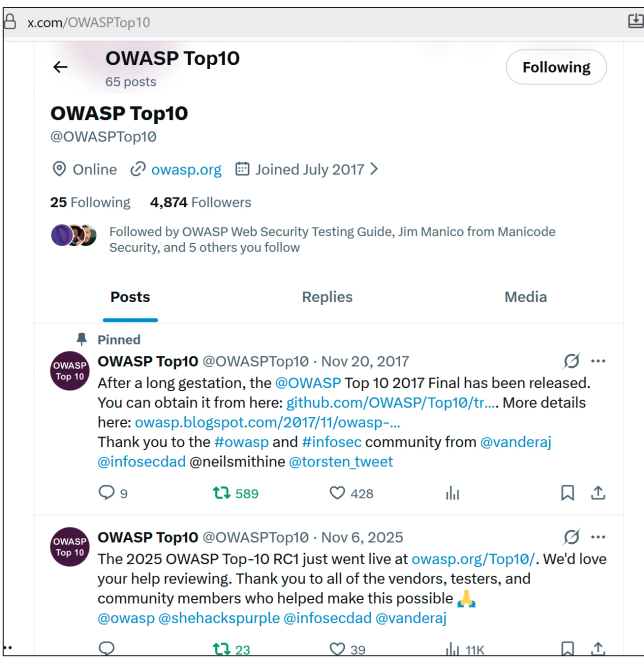
tion ist die „OWASP Top Ten“ diejenige mit dem wissenschaftlichsten Ansatz: sehr datengetrieben mit ordentlicher Basis. Dennoch lässt sich über die Liste und ihre Punkte wunderbar streiten (am Ende des Artikels thematisieren wir dies noch mal kurz). Doch das ist nicht entscheidend. Zur Erinnerung: Es geht um Awareness, und das leistet die OWASP Top Ten seit etwa 25 Jahren wunderbar.

Entstehungsprozess

An der ersten Stelle einer jeden OWASP Top Ten steht ein Datenaufwurf. Wer Security-Audits durchführt, egal ob Firma, Softwaredienst oder Individuum, wird um strukturierte Kennzahlen der Überprüfungsergebnisse gebeten. Der zugrunde liegende Zeitraum beginnt mit dem Stichtag der dann noch aktuellen Top Ten. Diese erschien im September 2021. Obwohl auch noch Last-Minute-Datenspenden von Ende 2021 angenommen wurden, galt für die 2025er Top Ten, dass alle Audits ab 2022 akzeptiert wurden. Am anderen Ende des gültigen Zeitintervalls stand 2024. Aufgrund der Verzögerungen in der Entstehung der Liste wurden zwar bis Ende Juni 2025 noch Einreichungen akzeptiert, aber zumindest offiziell sollten nur Audits bis Ende 2024 gültig sein. Für die nächste Liste – 2029 wäre der normale Turnus – kann also von Daten der Jahre 2025 bis 2028 ausgegangen werden.

Selbstverständlich beinhalten die Daten keine konkreten Funde, schon gar nicht mit Codeauszügen oder Exploits. Stattdessen handelt es sich um reines Zahlenwerk. Für die Erstellung der OWASP Top Ten bedient sich das Projektteam eines Nummerierungsschemas, das jedem Sicherheitsmangel oder -risiko eine Zahl zuweist: CWE. Das steht für Common Vulnerabilities and Exposures; auf der Website der US-Regierungsorganisation MITRE findet sich eine komplette Liste, zum Redaktionsschluss im März 2026 waren es 944 Stück, die teilweise hierarchisch angeordnet sind. Die meisten dieser CWEs sind ziemlich spezifisch. Unter der sehr niedrigen Nummer 11 findet sich beispielsweise *ASP.NET Misconfiguration: Create Debug Binary* und behandelt Risiken, wenn ein im Debug-Modus kompiliertes Binary eingesetzt wird. Ein anderes Beispiel: CWE-256 behandelt das Speichern von Passwörtern im Klartext.

Die Daten aus Sicherheitsaudits enthalten im Wesentlichen diese CWE-Nummern: Wie oft wurden die entsprechenden



Zeitverzug: Der X-Account der Top Ten weiß auch im März 2026 noch nichts vom finalen 2025er Release (Bild 2)

Risiken oder Schwachstellen gefunden, und wie viele Anwendungen wurden untersucht. Damit lassen sich relative und absolute Häufigkeiten ermitteln, eine solide Grundlage für die spätere OWASP Top Ten.

Die Datenspenden mussten dabei in einem vorgegebenen Format eingereicht werden, entweder als JSON- oder als Excel-Datei, obwohl die vom Projektteam zur Verfügung gestellte Vorlage im CSV-Format vorlag, aber das ist wohl nicht entscheidend. Bild 3 zeigt eben dieses CSV – in Excel.

Ergebnis dieser Bemühungen ist ein ansehnlicher Datenkorpus. Die vorherigen Ausgaben der OWASP Top Ten haben gezeigt, dass allein die Menge der eingereichten Auditergebnisse sehr stark variiert, die Bandbreite reicht von Einzelkämpfern bis hin zu automatisierten Tools. Man könnte also darüber diskutieren, dass die Zahlenwerte nicht ganz repräsentativ sein könnten beziehungsweise dass einzelnen Einreichungen zu viel Gewicht zukäme. Dem steht eine der Lehren aus der Cybersecurity entgegen: Niemand spricht darüber. Insofern sind die Audit-Daten wohl die beste Quelle für eine seriöse Untersuchung.

Die Einreichungsfrist wurde mehrfach verlängert. Ob das an (zu) wenig Daten lag, kann bezweifelt werden: Als Daten eingeflossen sind die Untersuchungsergebnisse von über 2,8 Millionen Webanwendungen – natürlich sind da auch die Ergebnisse automatischer Tools mit dabei. Doch allein aus ►

A screenshot of an Excel spreadsheet titled 'sample-data-submission'. The table has columns: NumberofAppsTested, CWE, NumberofAppsPer, TimePeriod, ContributorName, ContributorContactEmail, TypeofTesting, PrimaryLanguage, and Region. The data rows show audit results for different contributors and time periods.

	A	B	C	D	E	F	G	H	I
	NumberofAppsTested	CWE	NumberofAppsPer	TimePeriod	ContributorName	ContributorContactEmail	TypeofTesting	PrimaryLanguage	Region
1	100	20	53	2021	Name	Email	TAH	.NET	North Ar
2	100	20	10	2022	Name	Email	TAH	.NET	North Ar
3	100	20	30	2023	Name	Email	TAH	PHP	North Ar
4	100	425	5	2021	Name	Email	TAH	.NET	North Ar

Zahlen des Versagens: Die Vorlage zum Einreichen der Auditergebnisse (Bild 3)

den Audit-Statistiken lässt sich noch keine OWASP Top Ten erstellen. Es gibt eine dreistellige Anzahl von CWEs, aber nur zehn Listenplätze. In früheren Ausgaben der Liste (bis einschließlich 2017) war die Menge der „erlaubten“ CWEs begrenzt, aber das ist freilich wenig sinnvoll, gerade im Hinblick auf neu aufkommende Gefahren und Angriffe. Somit müssen mehrere CWEs zusammengruppiert werden.

Dies ist vielleicht der größte methodische Schwachpunkt an dem Erstellungsprozess der OWASP Top Ten, denn hier kann Subjektivität ins Spiel kommen. Letzten Endes schauen alle auf die zehn Kategorien der Liste, aber nur im zweiten Schritt auf die CWEs. Durch den Zuschnitt der Listenpunkte lassen sich also bestimmte Themen gewichten und herausstellen. Das ist nicht vorwurfsvoll gemeint, denn letzten Endes verschwinden keine (relevanten) Daten aus der Audit-Erhebung; manche sind nur sichtbarer als andere.

Das Ergebnis der Gruppierung bestand aus zwölf Kategorien. Wegen der namensgebenden Begrenzung einer Top-Ten-Liste musste also aussortiert werden. Aus diesem Grund erstellte das Projektteam eine Umfrage, in der die „Wackelkandidaten“ unter den Kategorien bewertet werden konnten. Bild 4 zeigt einen Ausschnitt aus der Umfrage – neben den Kategorienamen wurden auch die zugehörigen CWE-Nummern angegeben. Bei der 2021er OWASP Top Ten übrigens wurden nur acht Listenplätze auf Basis des Datenkorpus vergeben; zwei weitere stammten aus der Umfrage. Dies führt unter anderem dazu, dass der relevante, aber doch spezielle Angriff Server-Side Request Forgery (SSRF) einen eigenen Eintrag in der Liste erhalten hat, trotz geringem Vorkommen in den Audit-Ergebnissen. Dies sollte bei der Neuausgabe der Top Ten vermieden werden, so kam es zur Änderung im Prozess.

Am Ende entstand eine Liste mit folgenden zehn Punkten:

- Broken Access Control
- Security Misconfiguration
- Software Supply Chain Failures
- Cryptographic Failures
- Injection
- Insecure Design
- Authentication Failures
- Software or Data Integrity Failures
- Security Logging and Alerting Failures
- Mishandling of Exceptional Conditions

Wir werfen jetzt einen Blick auf jeden dieser Punkte, bei manchen Themen sehr und bei manchen weniger ausführlich, abhängig unter anderem von der Menge der CWEs und natürlich von der Relevanz für Webanwendungen mit ASP.NET Core.

#1: Broken Access Control

Das laut OWASP Top Ten 2025 wichtigste Sicherheitsrisiko ist dasselbe wie 2021 (2017 war der entsprechende Punkt noch auf Platz 5): Broken Access Control. Salopp gesprochen: Wenn es in einer Anwendung eine Sicherheitslücke gibt, dann hatte ein Angreifer wohl unerwünscht Zugriff auf etwas. Irgendwie passt „kaputte Zugriffskontrolle“ auf fast alles. Das zeigt sich auch beim Blick auf die zugehörigen CWEs: 40 Stück. Auf

	#1	#2	#3
Mishandling of Exception Conditions (CWE-209, 215, 476, 391, ...)	☐	☐	☐
Logging & Alerting Failures (CWE-117, 221, 223, 532, 778))	☐	☐	☐
Software or Data Integrity Failures (CWE-494, 502, 353, 829, 830, ...)	☐	☐	☐
Software Supply Chain Failures (CWE-488, 1035, 1104, 1329, 1395))	☐	☐	☐
Lack of Application Resilience (CWE-400, 401, 404-410, 770, ...)	☐	☐	☐
Memory Management Errors (CWE-119-122, 124-127, 787, 788, ...)	☐	☐	☐

Mut zur Lücke: OWASP-Umfrage zur Bewertung von Sicherheitsrisiken (Bild 4)

konzeptioneller Ebene lässt sich hier schon viel erreichen: Autorisierung überall, bei jedem (schützenswerten) Endpunkt. Das ist in ASP.NET Core mit dem `[Authorize]`-Attribut „out of the box“ verfügbar, muss aber konsequent durchgezogen werden. Typisches Muster aus diversen Audits: Autorisierung findet nur für bestimmte HTTP-Methoden statt, Marke „wer (im Beispiel eines API) keinen Zugriff auf den `GET`-Endpunkt hat, der kommt gar nicht erst auf die Struktur der `DELETE`-Endpunkte zum Löschen, also bedarf es bei Letzteren keines Zugriffsschutzes“. Das ist natürlich ein naives Vorgehen. Zugriffsrechte müssen immer und überall geprüft werden.

Dasselbe gilt für jede Form der Validierung. Diese ist in ASP.NET Core relativ simpel, wenn Data Annotations zum Einsatz kommen. Hier ein typisches Beispiel:

```
using System.ComponentModel.DataAnnotations;

public class Person
{
    [Required]
    public string FirstName { get; set; } = string.Empty;
    [Required, MinLength(3)]
    public string LastName { get; set; }
}
```

Beim Einsatz von Model Binding werden die Annotationen entsprechend überprüft. Eine gut bekannte, aber trotzdem gemeine Falle: Dies schlägt sich in der Eigenschaft `ModelState.IsValid` nieder, verhindert aber nicht, dass innerhalb der `Action`-Methode die Daten automatisch abgelehnt werden würden. Aus diesem Grund ist stets ein Vorgehen nach folgendem Muster notwendig:

```
[HttpPost]
public async Task<IActionResult> Create([Bind(
    "FirstName,LastName")] Person person)
{
    if (ModelState.IsValid)
    {
        _context.Add(person);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    return View(person);
}
```

Apropos Datenvalidierung: Hier gab es in .NET 10 einige Neuerungen. Die Validierungsmöglichkeiten in Blazor wurden erweitert und erlauben jetzt unter anderem auch die automatische Überprüfung von Collections. Minimal APIs wiederum unterstützen erst seit dieser .NET-Version überhaupt die Validierung mit Data Annotations. Für beide Anwendungsfälle ist jedoch der explizite Einsatz einer speziellen Middleware erforderlich:

```
builder.Services.AddValidation();
```

Etwas versteckt in „Broken Access Control“ ist ein Klassiker, der bis zur OWASP Top Ten 2013 eine eigene Kategorie innehatte: Cross-Site Request Forgery, kurz CSRF. In den meisten Varianten fußt dieser Angriff auf von einem Client unfreiwillig erzeugten HTTP-Requests, typischerweise aufgrund einer Weiterleitung oder eines in eine andere Website eingebetteten JavaScript-Codes. Das häufige Problem: Bei diesen HTTP-Anfragen werden automatisch die Cookies des Clients mitgeschickt, der Server kann dann also nutzerspezifisch diese HTTP-Anfragen verarbeiten und im Namen des Opfers Aktionen ausführen. Im skizzierten – und natürlich etwas vereinfacht dargestellten – Szenario kann der Angreifer den HTTP-Request präzise vorhersagen, mit Ausnahme des Cookies, aber das kommt ja automatisch mit.

Verwendet eine Webanwendung ASP.NET Core, ist das – aus Angriffssicht – nicht mehr ganz so einfach. Bei *POST*-Formularen, erzeugt per Form Tag Helper oder *@Html.BeginForm*, wird automatisch ein zusätzliches „unsichtbares“ Formularfeld ergänzt. Wird ein Formular ohne Tag- oder

HTML-Helper generiert, kann das Zusatzfeld alternativ durch *@Html.AntiForgeryToken()* eingefügt werden. Das neue Feld sieht dann wie folgt aus:

```
<input name="__RequestVerificationToken" type="hidden"
    value="CfD..." /></form>
```

Gleichzeitig schickt der Webserver, sofern noch nicht vorhanden, ein Cookie namens *.AspNetCore.Antiforgery.<individuelle ID>*. Dessen Wert passt zum Inhalt des Formularfelds – sie sind nicht identisch, aber der Server kann entscheiden, ob sie zusammengehören. Die Idee dahinter ist folgende: Findet ein CSRF-Angriff statt, wird zwar möglicherweise das Cookie automatisch an den Server übermittelt, aber nicht der Inhalt des versteckten Formularfelds. Der ist individuell und nicht vom Angreifer vorhersehbar. Auf dem Server lässt sich also ermitteln, ob wirklich das Formular verschickt oder lediglich ein HTTP-Request von fremder Hand erzeugt wurde. Damit sind CSRF-Angriffe verhinderbar.

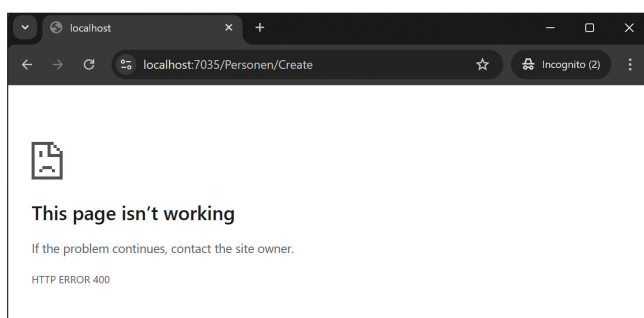
Der kleine Haken: Das muss man aktiv so wollen. Das Attribut *[ValidateAntiForgeryToken]* sorgt dafür, dass diese Überprüfung nicht nur stattfindet, sondern bei einem Fehlschlag auch die *Action*-Methode verlassen wird und der Client den HTTP-Statuscode 400 erhält: Bad Request (Bild 5). In der vorher gezeigten *Create()*-Methode fehlt dieses Attribut, hier wäre also kein Schutz vorhanden.

Gemäß dem Motto *Defense in Depth* – Tiefenverteidigung; jede Schutzmaßnahme kann theoretisch ausgehebelt werden, also schadet es nicht, mehrere zu haben – lohnt es sich, die Verteidigungsanstrengungen gegen CSRF zu erweitern. Das *SameSite*-Flag für Cookies gibt an, ob bei einem Cross-Site-Request (wie er bei CSRF-Angriffen typisch ist; das Opfer wird von einer fremden Website „umgeleitet“) überhaupt Cookies mitgeschickt werden. Bei der Einstellung *Lax* geschieht dies nur bei *GET*-Requests, die zumindest laut REST-Prinzip keine Aktionen auf dem Server auslösen dürfen. Im *SameSite*-Modus *Strict* wiederum wird das entsprechende Cookie bei Cross-Site-Requests gar nicht mitgeschickt und somit eine der Prämissen vieler CSRF-Attacken ausgehebelt. Das Anti-CSRF-Cookie von ASP.NET Core setzt standardmäßig auf *Strict*.

Einen ähnlichen Namen wie CSRF hat der Angriff *Server-Site Request Forgery* oder kurz SSRF. Diesen hatten wir vorher schon kurz erwähnt – er hatte in der OWASP Top Ten 2021 einen eigenen Listenplatz inne, allerdings ausschließlich wegen der Umfrage. Bei diesem Angriff geht es darum, dass nicht wie bei CSRF ein Client einen verhängnisvollen HTTP-Request macht, sondern ein Server-System. Wenn das funktioniert und das Ziel der HTTP-Anfrage den Request wie vom Angreifer gewünscht verarbeitet, ist das nichts anderes als Broken Access Control. Aus diesem Grund ist SSRF nun „nur noch“ Teil von Top-Ten-Eintrag Nummer 1.

#2: Security Misconfiguration

Eine sicherheitstechnisch unzureichende Konfiguration ist einer der Aufsteiger in der OWASP Top Ten 2025 – in der Vorgängerausgabe befand sich dieser Punkt noch auf Platz 5. ►



Schlechter als 404: „Bad Request“, wenn der CSRF-Schutzmechanismus anschlägt (Bild 5)

Ein möglicher Grund für den Aufstieg: Bestimmte Konfigurationsoptionen lassen sich trefflich testen. Beispielsweise HTTP-Header, die die Webanwendung setzt. Es gibt eine ganze Reihe davon, die bestimmte Sicherheitsfeatures in Browsern aktivieren. Hier ein kleiner Ausschnitt von verfügbaren Optionen:

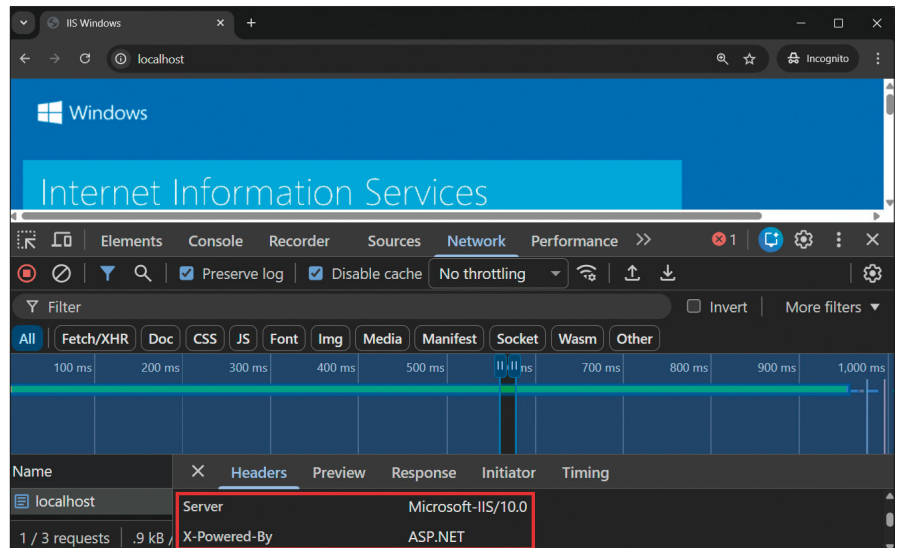
- **Content-Security-Policy** [9]: Beschränkt, welche Ressourcen eine Webseite laden darf – bei Listenpunkt 5 wird das noch einmal relevant.
- **Permissions-Policy** [10]: Konfiguriert, welche JavaScript-APIs (etwa: Geolocation oder Bluetooth-Zugriff) verwendet werden dürfen.
- **Referrer-Policy** [11]: Gibt an, in welchen Szenarien der Referer-HTTP-Header geschickt wird. Dieser enthält salopp gesprochen den URL der aktuell im Browser geladenen Seite, was möglicherweise die Zielwebsite nichts angeht.
- **X-Content-Type-Options** [12]: Einst eine Microsoft-Erfindung, die verhindern kann, dass Browser den Inhalt einer Ressource aufgrund der ersten paar Bytes „erraten“, was in sehr speziellen Szenarien ein Sicherheitsrisiko darstellt.
- **X-FRAME-OPTIONS** [13]: Beschränkt das Laden der aktuellen Seite in einem Iframe (oder, für die Älteren unter uns, Frame). Damit kann beispielsweise Clickjacking verhindert werden. Eine ähnliche Funktionalität ist aber auch mit Content Security Policy möglich.

Außer Headern gibt es noch andere in einer HTTP-Response auftretende Sicherheitsoptionen wie bestimmte Cookie-Flags – *SameSite* etwa hatten wir zuvor schon behandelt, aber es gibt noch weitere:

- **secure**: Sorgt dafür, dass ein Cookie nur per HTTPS übermittelt werden darf.
- **HttpOnly**: Schützt ein Cookie vor JavaScript-Zugriff.
- **__Host**: Cookies, deren Name hiermit beginnt, benötigen das *secure*-Flag und dürfen das *domain*-Flag nicht verwenden.
- **__Http**: Cookies, deren Name hiermit beginnt, benötigen das *HttpOnly*- und das *secure*-Flag.
- **__Host-Http**: Dieser Namenspräfix kombiniert die Voraussetzungen von *__Host* und *__Http*.
- **__Secure**: Dieser Namenspräfix erfordert das *secure*-Flag.

All dies lässt sich sehr einfach testen, auch automatisiert. Positiver Nebeneffekt: Sind die Header und Einstellungen alle gesetzt, ist der erzeugte Prüfbericht automatisch kürzer.

An dieser Stelle ein kurzer, aber wichtiger Einschub: Die Kunst ist nicht notwendigerweise das Finden von Lücken oder Identifizieren von Risiken, sondern das Management dieser Vorgänge. Ist ein fehlender HTTP-Header in allen Szenarien ein großes Problem? Typische Antwort aus der Beratung: „Hängt davon ab.“ Sprich, es ist stets eine Bewertung

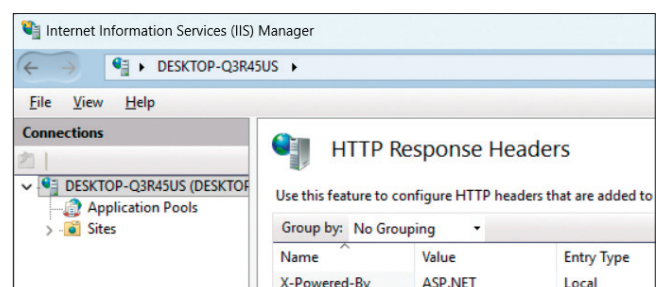


Verräter: HTTP-Header, die zu viel über das System berichten (Bild 6)

vorzunehmen, wie mit einem Risiko umgegangen wird. Auch ein „Wir ignorieren es“ kann eine valide Reaktion sein.

Neben HTTP-Headern für mehr Sicherheit gibt es auch solche, die vermieden werden sollten – nämlich die, die Informationen über das zugrunde liegende System verraten (Bild 6). Klassiker sind *Server*, der den verwendeten Webserver samt Version enthalten kann, sowie *X-Powered-By*, das von manchen Frameworks gesetzt wird. Je mehr ein Angreifer über sein Ziel weiß, desto höher sind die Chancen auf Erfolg. Deswegen ist es ratsam, diese Header zu entfernen. *X-Powered-By* beispielsweise wird von ASP.NET Core gar nicht gesetzt, lediglich vom „alten“ .NET Framework, lässt sich aber in der IIS-Konfiguration per Einstellung entfernen (Bild 7).

Von HTTP-technischen Einstellungen einmal abgesehen gibt es natürlich eine weitere Reihe von Optionen, die – unsicher – gesetzt werden können. Das gilt sowohl für die Serversoftware (inklusive virtualisierter Systeme) als auch für Applikationsframeworks. Die Liste an neuralgischen Punkten ist lang, wobei gerade in den letzten Jahren viele Systeme in zahlreichen Aspekten „secure by default“, also standardmäßig sicher sind. Mit anderen Worten: Die Grundeinstellungen sind ziemlich solide. Dies verdeutlicht beispielsweise folgender Auszug aus der generierten *Program.cs* beim Erstellen einer Anwendung mit ASP.NET Core 10 Razor Pages:



Erwischt: Ein unerwünschter Header kann in der IIS-Administration entfernt werden (Bild 7)

```
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    // ...
}
```

```
// ...
app.MapStaticAssets();
app.MapRazorPages()
    .WithStaticAssets();
```

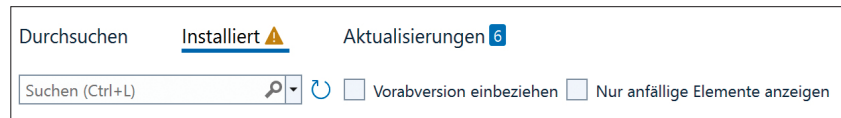
Wenn also die Anwendung nicht im Entwicklungsmodus läuft, wird ein spezieller Exception-Handler verwendet – eine Seite, auf der keine technischen Informationen über aufgetretene Ausnahmen angezeigt werden. Im Entwicklungsmodus wiederum erscheint in solchen Fällen ein kompletter Stacktrace samt weiteren Angaben. Diese Einstellung ist sinnvoll – und fraglos besser als ein „Jede und jeder bekommt eine ausführliche technische Fehlermeldung angezeigt“.

Weiter unten sehen wir dann, dass mit `MapStaticAssets()` und `WithStaticAssets()` die Unterstützung für statische Inhalte erst aktiviert werden muss. Standardmäßig ist dieses Feature ausgeschaltet! Dies ist prinzipiell ein guter Ratgeber für Sicherheit: Nur das aktivieren, was auch wirklich benötigt wird.

Ein weiterer häufiger Fehler besteht darin, wenn Zugangsdaten wie Passwörter in einer Konfigurationsdatei vorgehalten werden – beispielsweise der `appconfig.json`. Wird diese in eine Versionsverwaltung eingchecked, sind diese Credentials möglicherweise dauerhaft dort vorhanden (und werden von Tools wie Gitleaks [14] simpel aufgespürt). Für ASP.NET (auf Basis des .NET Framework) gab es hier sogar einen eigenen CWE-Eintrag (13), aber generischer ist CWE-260, „Password in Configuration File“. Passwörter gehören geschützt, beispielsweise über einen Vault-Mechanismus, in Azure könnte der Azure Key Vault eine mögliche Lösung sein. Alternativ helfen Umgebungsvariablen auf den jeweiligen Serversystemen.

#3: Software Supply Chain Failures

Auch das Versagen in der „Software-Lieferkette“ ist ein „Gewinner“ der OWASP Top Ten 2025. Einst hieß die Kategorie „Using Components with Known Vulnerabilities“, aber das ist zu eng gefasst; die komplette Supply Chain sieht sich Risiken gegenüber. Die Wichtigkeit und Relevanz dieses Listeneintrags wird durch zahlreiche großflächige Sicherheitsvorfälle der letzten Monate bestätigt. Im September 2025 beispielsweise wurden durch den Wurm „Shai-Hulud“ über 500 NPM-Pakete mit Schadcode versehen [15]; zwei Mo-



Sicherheitsleck: Verwundbare NuGet-Pakete in Visual Studio finden (Bild 8)

nate später erwischte „Shai-Hulud 2.0“ über 800 Pakete [16]. NuGet ist da in der Vergangenheit vergleichsweise verschont geblieben, was allerdings auch an den recht frühzeitig eingeführten Sicherheitsmaßnahmen [17] liegen könnte. So ist beispielsweise der Einsatz von Zwei-Faktor-Authentifizierung notwendig, um die Übernahme eines Maintainer-Kontos (samt Ausspielen neuer NuGet-Paketversionen inklusive Schadcode) zu erschweren.

Werfen wir also als Nächstes einen Blick auf einige NuGet-spezifische Sicherheitsaspekte. In Visual Studio selbst gibt es im Rahmen des NuGet-Paket-Managers eine Erkennung verwundbarer Software-Versionen; die Checkbox aus Bild 8 dient zum Filtern. Achtung, das gelbe Warndreieck (auch in Bild 8 zu sehen) deutet nicht notwendigerweise eine verwundbare Version an, sondern erscheint beispielsweise auch bei veralteten Paketversionen.

Ein ähnliches Ergebnis lässt sich auch aus der Konsole (und damit besser automatisierbar) erzielen. Das Tool `dotnet-outdated` [18] untersucht Projektdateien (`.csproj` und Konsorten) und liefert dann eine Liste veralteter Versionen. Dies entspricht grob dem Tab im NuGet-Paket-Manager, in dem verfügbare Updates angezeigt werden. Es findet also keine Beschränkung auf Versionen mit Sicherheitslücken statt. Aber da veraltete Abhängigkeiten im Code technische Schulden sind, ist es absolut sinnvoll, regelmäßig nach Updates zu suchen. Freilich sollte das mit einem hohen Grad an automatisierten Tests flankiert werden, um nach Aktualisierung eines NuGet-Pakets die Gewissheit zu haben, dass die Anwendung weiterhin funktioniert.

So wird das Werkzeug auf dem System global installiert:

```
dotnet tool install --global dotnet-outdated-tool ▶
```

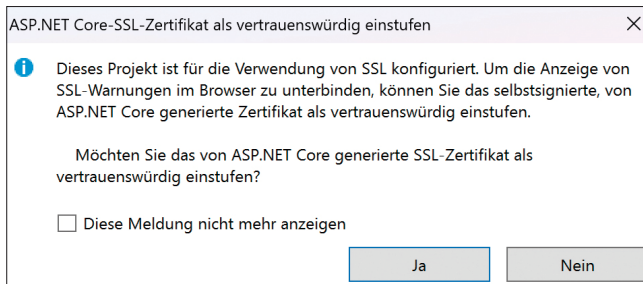
```
C:\Windows\System32\cmd.e  x  +  v

> dotnet outdated AspNetCore8Demos.BlazorCRUD.csproj
Discovering projects...
Analyzing project(s)...
Analyzing dependencies...
» AspNetCore8Demos.BlazorCRUD
   [net8.0]
Microsoft.AspNetCore.Components.QuickGrid.EntityFrameworkAdapter 8.0.2 -> 8.0.24
Microsoft.EntityFrameworkCore.SqlServer 8.0.2 -> 9.0.13
Microsoft.EntityFrameworkCore.Tools 8.0.2 -> 10.0.3
Microsoft.VisualStudio.Web.CodeGeneration.Design 8.0.1 -> 9.0.0

Version color legend:
<red> : Major version update or pre-release version. Possible breaking changes.
<yellow>: Minor version update. Backwards-compatible features added.
<green>: Patch version update. Backwards-compatible bug fixes.

You can upgrade packages to the latest version by passing the -u or -u:prompt option.
Elapsed: 00:00:03.0790124
```

Altersdiskriminierung: dotnet-outdated ermittelt zu aktualisierende NuGet-Pakete (Bild 9)



Vertrauen Sie mir, ich bin (kein) Arzt: Visual Studio will uns ein Zertifikat schmackhaft machen (Bild 10)

Ein typischer Aufruf sieht dann wie folgt aus (Bild 9 zeigt eine mögliche Ausgabe):

```
dotnet outdated MyProject.csproj
```

#4: Cryptographic Failures

Das „kryptografische Versagen“, wie es im vierten Punkt der OWASP Top Ten von 2025 heißt, ist meist ein Verzicht auf HTTPS. Dabei macht es ASP.NET Core einem sehr einfach: Bereits bei der Projektanlage ist „Verwendung von HTTPS“ standardmäßig aktiviert, und die *Program.cs* lädt sogar die Middleware, mit der HTTP-Requests auf HTTPS umgeleitet werden:

```
app.UseHttpsRedirection();
```

Auch ansonsten gibt es heutzutage kaum einen Grund, nicht auf HTTPS zu setzen. Etwas ärgerlich ist nur das lokale Entwickeln: Das selbstsignierte Zertifikat für *localhost* sorgt für Probleme, außer es wird global auf dem System akzeptiert (wozu einen Visual Studio stets ermuntert, siehe Bild 10). Dann aber gilt das auch auf dem kompletten System für jeden (HTTPS-)Server auf *localhost*, egal auf welchem Port. Dies ist möglicherweise unerwünscht.

Ab .NET 10 geht das besser: Das .NET-Team hat offenbar den RFC 6761 [19] genau gelesen, in dem (unter anderem) *localhost* als spezielle, lokale Domain reserviert wird. In einigen Projektanlageassistenten, Bild 11 zeigt als Beispiel Blazor, wird auf Wunsch statt auf *localhost* auf *<Name-DesProjekts>.dev.localhost* gehostet. Das ist zwar immer noch das lokale System, auch das Zertifikat ist selbstsigniert, aber es ist eben ein individueller Host.

Um HTTPS tatsächlich zu erzwingen, kann auch erwogen werden, den HTTP-Endpunkt abzuschalten. Als etwas sanfterer, aber trotzdem sehr effektiver Weg bietet sich der Einsatz von HSTS an – HTTP Strict Transport Security [20]. Per HTTP-Header *Strict-Transport-Security* wird der Browser angewiesen, nur noch HTTPS-Requests an den aktuellen Host zu schicken:

```
Strict-Transport-Security: max-age=31536000;
    includeSubDomains
```

HTTP-Anfragen werden dann automatisch umgeschrieben, sodass kein unverschlüsselter Datenverkehr mehr mit dem Server stattfindet. Die *Program.cs* von ASP.NET-Core-Projekten sieht das bereits vor, nur nicht im Entwicklungsmodus (was eine sinnvolle Unterscheidung ist, weil Browser HSTS-Einstellungen cachen können):

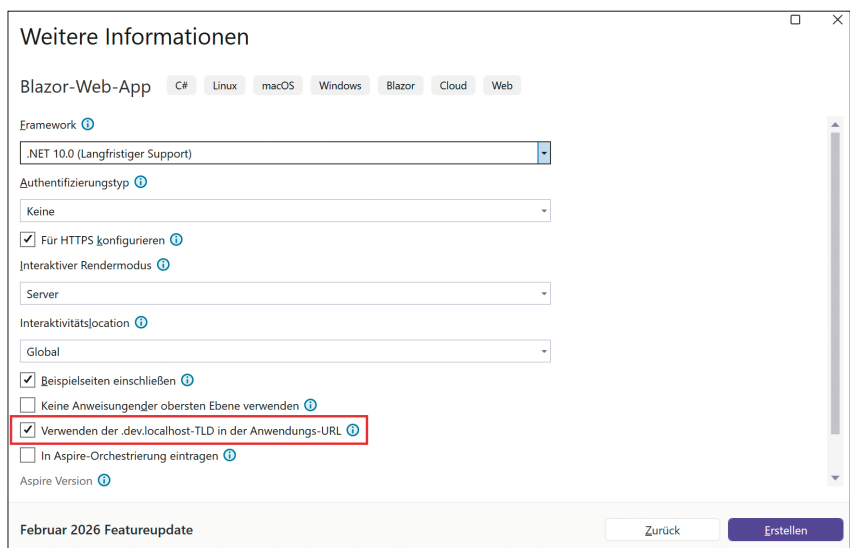
```
if (!app.Environment.IsDevelopment())
{
    // ...
    app.UseHsts();
}
```

Die so aktivierte Middleware sendet den *Strict-Transport-Security*-Header aus, allerdings nur mit einer Gültigkeitsdauer von 30 Tagen. Es ist empfehlenswert, auf einen höheren Wert zu gehen, indem die Middleware vor dem Aufruf von *UseHsts()* entsprechend konfiguriert wird. Hier ein Beispiel für ein Jahr (sofern es kein Schaltjahr ist):

```
builder.Services.AddHsts(options =>
{
    options.IncludeSubDomains = true;
    options.MaxAge = TimeSpan.FromDays(365);
});
```

#5: Injection

Das nächste Listenelement, Injection, ist akkurat betitelt, aber nur, wenn die Hintergründe beleuchtet werden. Hier liegt nämlich eine Art Taschenspielertrick der OWASP zugrunde. Aber der Reihe nach: Von der ersten Ausgabe der OWASP Top Ten bis einschließlich zur 2017er Edition war Platz 1 stets von SQL Injection belegt. In der Tat: In den früheren Jahren des Web war das Problem noch nicht überall



Individualismus: Die Anwendung läuft lokal auf einem eigenen Hostnamen (Bild 11)

bekannt und Code mit Einschleusemöglichkeiten für SQL weithin verbreitet. Das hat sich aber sukzessive geändert. Insofern haben viele – auch der Autor dieses Artikels – regelmäßig die Top-Platzierung für diesen Angriff kritisiert, wurde er doch routinemäßig in Sicherheits-Audits zu einem Exoten. Die OWASP begründete ihre Prioritätenreihenfolge damit, dass SQL Injection vielleicht selten sein mag, aber wenn sie stattfindet, ist sie eine Katastrophe.

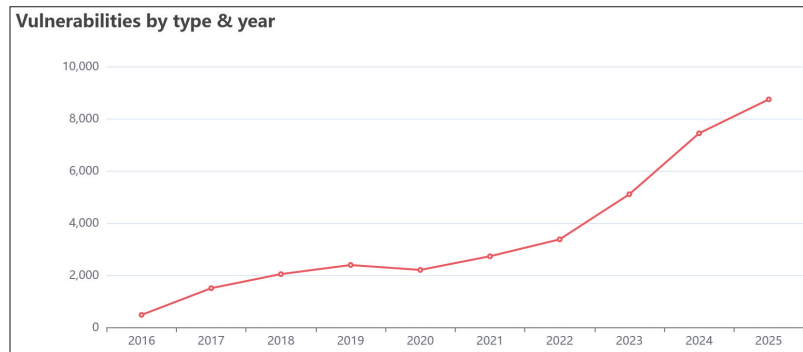
Eine weitere wohlbekannte Sicherheitslücke ist Cross-Site Scripting (XSS), das meist in Form von eingeschleustem JavaScript-Code daherkommt. Auch dieser Angriff war viele Jahre lang ein „Stammgast“ in der OWASP Top Ten, fiel dann aber stetig nach unten. Das deckte sich nicht mit der Praxiserfahrung vieler Auditoren, die sich nicht über einen Mangel an XSS-Funden beklagen konnten. Auch die statistische Auswertung von CVEs (Common Vulnerabilities and Exposures) auf cvedetails.com [21] verzeichnet seit Jahren eine aufsteigende Tendenz – Bild 12 zeigt den Stand Ende 2025. Natürlich gibt es hier ebenfalls eine Begründung der OWASP: XSS möge zwar häufig sein, aber so schlimm sei bössartiger Fremdcode doch auch nicht.

Bei der Datenanalyse für die OWASP Top Ten 2021 stellte sich heraus, dass die vermeintlich so wichtige SQL Injection es zahlenmäßig kaum in die Liste schaffen würde. Aber ist XSS nicht auch eine Form von Injection – es wird etwas eingeschleust, eben JavaScript-Code (und manchmal auch „nur“ HTML)? Somit wurde aus der Not eine Tugend gemacht: Die neue Kategorie „Injection“ beinhaltet alle Formen von eingeschleusten Inhalten, inklusive SQL Injection und eben auch Cross-Site Scripting. Positiver Nebeneffekt aus OWASP-Sicht: SQL Injection war weiterhin dabei. Negativer Nebeneffekt aus Sicht des Autors: Ohne diesen Kontext könnte ein oberflächlicher Blick auf die Top Ten den Eindruck erwecken, XSS sei nicht mehr relevant.

Wir werfen natürlich einen Blick auf diese Risiken hinsichtlich ihrer Anwendbarkeit bei ASP.NET Core und Blazor. Bei SQL Injection sieht es insofern gut aus, als Microsoft im Großteil der Tutorials und Beispiele auf Entity Framework Core setzt. Als OR-Mapper wird hier gar nicht mit SQL gearbeitet, sondern mit einem API. Der typische Fehler bei SQL Injection – Clienteingaben werden eins zu eins in das SQL-Kommando geschrieben und können somit selbst Befehle sein – ist hier größtenteils nicht mehr möglich, beispielsweise weil sich das API um die korrekte Handhabung von Sonderzeichen kümmert. Es ist lediglich wichtig, diejenigen Methoden von Entity Framework Core zu meiden, bei denen tatsächlich SQL zur Ausführung übermittelt werden kann:

- `ExecuteRawQuery()`
- `FromSqlRaw()`

Solche Anfragen gehören parametrisiert, sprich in den SQL-Kommandos stecken Platzhalter, denen – per Methodenaufruf, nicht per String-Konkatenation – Werte zugewiesen werden. Derselbe Ansatz gilt im Übrigen auch bei der Arbeit mit



The trend is not your friend: Steigende Vorkommen von XSS (Bild 12)

SQL via ADO.NET oder ähnlichen APIs. Also theoretisch ein gelöstes Problem.

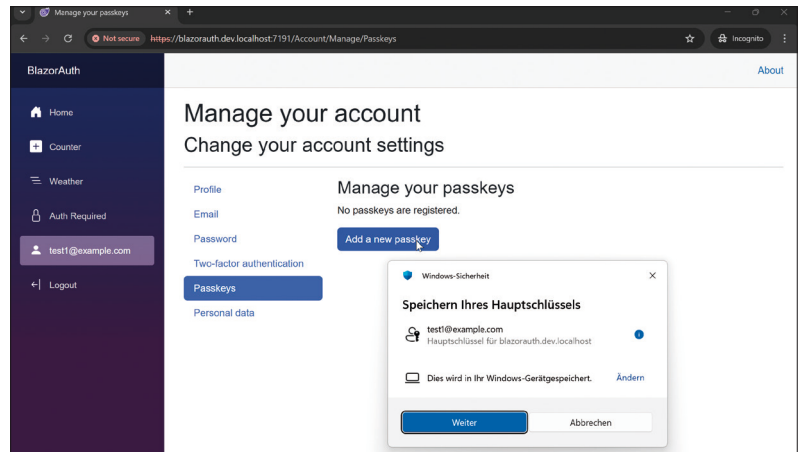
Ebenfalls gelöst ist eigentlich Cross-Site Scripting. Prinzipiell ist es einfach: Wann immer nicht vertrauenswürdige Daten ins HTML gepackt werden, müssen die relevanten Sonderzeichen maskiert werden: `<`, `>`, `,`, `&`. Bereits im klassischen ASP gab es dafür die Methode `HTMLEncode()`; in .NET heißt sie `HtmlEncode()` und steckt in `System.Web.HttpUtility`. In vielen Fällen ist ihr expliziter Einsatz aber gar nicht nötig, weil das Framework „mitdenkt“. Bei der Ausgabe in einer Razor-View oder Razor-Komponente mit `@` – es sind also Razor Pages, MVC und auch Blazor-Anwendungen erfasst – findet diese Sonderzeichenbehandlung automatisch statt. Überspitzt formuliert: Auch ohne XSS-Grundwissen sind viele Anwendungen sicher. Gerade aus diesem Grund ist es wichtig, die Ausnahmen und Sonderfälle zu kennen. Hier die wichtigsten:

- `@` ersetzt nur HTML-Sonderzeichen. In einem anderen Inhaltskontext, beispielsweise innerhalb eines JavaScript-Strings, gibt es weitere Zeichen, auf die geachtet werden muss. Insbesondere für das genannte Beispiel steht mit `JavaScriptStringEncode()` eine spezifische Escaping-Methode zur Verfügung.
- `@Html.Raw()` entwertet keine HTML-Sonderzeichen. Hierfür gibt es legitime Anwendungsfälle (etwa: Einsatz von HTML-Fragmenten), aber bei jedem einzelnen Aufruf muss belegt werden, dass die verwendeten Daten vertrauenswürdig sind und keinesfalls vom Client stammen.
- `@` wandelt nur bei herkömmlichen Strings Sonderzeichen um. Wird ein Wert vom Typ `HtmlString` übergeben, findet keine Konvertierung statt, weil ASP.NET Core davon ausgeht, dass dieser Inhalt bereits vorher ungefährlich gemacht wurde (was aber nicht so sein muss).

Eine zusätzliche Absicherung gegen XSS-Angriffe kann mit der vorher schon erwähnten Content Security Policy (CSP) erreicht werden, einem W3C-Standard [9], bei dem der Webserver dem Browser per HTTP-Header `Content-Security-Policy` vorgibt, von wo Ressourcen geladen werden dürfen. Ein potenzieller Effekt von mehreren: Wird per XSS-Lücke Code von einem anderen Server eingeschleust, wird dieser dank Policy gar nicht erst geladen.

Wie eine solche Content Security Policy auszusehen hat, hängt sehr stark davon ab, wie die Anwendung aufgebaut ►

ist. Die Grundregel lautet: Je weniger Inhalte von einem Third-Party-Host kommen und je weniger mit Inline-Code und -Styles gearbeitet wird, desto besser. Für Blazor gelten einige zusätzliche Anforderungen. Microsoft bietet in der Dokumentation eine Ausgangsbasis an, mit der gestartet werden kann. [22] Ein wichtiges Feature von CSP ist ein Benachrichtigungsmechanismus, wenn es zu einer Policy-Verletzung kommt: Auf Wunsch werden alle Details per HTTP-POST-Request im JSON-Format an einen in der Policy spezifizierten Endpunkt geschickt: Welcher Teil der Policy wurde verletzt, an welcher Stelle, durch welchen Code. Das hilft dann enorm bei der Fehlersuche.



Schlüsselbund: Passkeys-Administration mit ASP.NET Core Identity (Bild 13)

#6: Insecure Design

Der vielleicht unkonkreteste Eintrag in der OWASP Top Ten von 2025 befindet sich auf Nummer 6. Es liegt natürlich auf der Hand, dass ein unsicheres Softwaredesign ein Risiko sein kann. Aber mit derselben Begründung könne auch „unsicherer Code“ einen Listenplatz verdienen. Was damit ausgedrückt werden soll: Im Unterschied zu anderen Listenelementen sind die Handlungsoptionen nicht so offensichtlich.

Die Liste der CWEs ist lang – volle 39 Stück werden angeboten. Daraus spezifische und nicht zu generische Empfehlungen abzuleiten ist nicht einfach. Prinzipiell hilft es natürlich, sich schon beim Design einer Anwendung über das Thema Sicherheit Gedanken zu machen. Auch der Softwareentwicklungsprozess selbst soll unter dem Schirm eines Sicherheitskonzepts stehen – der von Microsoft einst initiierte und auch außerhalb des Redmonder Konzerns eingesetzte Security Development Lifecycle (SDL) [23] kann eine gute Basis darstellen. Und natürlich gehört Sicherheits-Know-how ins gesamte Team hineingetragen. Schulungen und Awareness sind essenziell für eine langfristig erfolgreiche Sicherheitsstrategie. Die OWASP Top Ten kann hier ein Baustein sein.

#7: Authentication Failures

Der nächste Punkt der OWASP Top Ten dreht sich um „Versagen“ bei der Authentifizierung – im Unterschied zum ersten Listenelement geht es also nicht um den Zugriff auf Ressourcen und Funktionalitäten (Autorisierung beziehungsweise Zugriffskontrolle), sondern um den Schritt davor, die Erkennung eines Nutzers. Auch an dieser Stelle wurden sehr viele unterschiedliche CWEs zusammengetragen, insgesamt drei Dutzend. Viel dreht sich um schwache Passwörter. Wer auf ASP.NET Core Identity setzt, hat hier möglicherweise weniger Sorgen: Die Zugangsdaten werden sicher gespeichert (und zwar in Form von Hashes), und es gibt auch eine Reihe von Passwortregeln, die standardmäßig gelten:

- Mindestens ein Groß- und Kleinbuchstabe
- Mindestens eine Ziffer
- Mindestens ein nicht alphanumerisches Zeichen
- Mindestlänge sechs Zeichen

Der letzte Punkt ist natürlich nicht State of the Art, und die Forderung von Sonderzeichen und Ziffern sorgt erfahrungsgemäß dafür, dass diese schlicht am Ende des Passworts stehen, was sicherheitstechnisch nicht viel bringt. Die Länge ist entscheidend. Beispielsweise schlagen sowohl das NIST [24] als auch das BSI [25] als Minimum acht Zeichen vor – mehr geht freilich immer. Die Passwortvorgaben lassen sich entsprechend anpassen, hier ein Beispiel:

```
builder.Services.Configure<
    IdentityOptions>(options =>
    {
        options.Password.RequiredLength = 8;
    });
```

Multi-Faktor-Authentifizierung wird mehr und mehr zum Standardfall und ist auf jeden Fall in Betracht zu ziehen. ASP.NET Core Identity ist bereits für den Einsatz

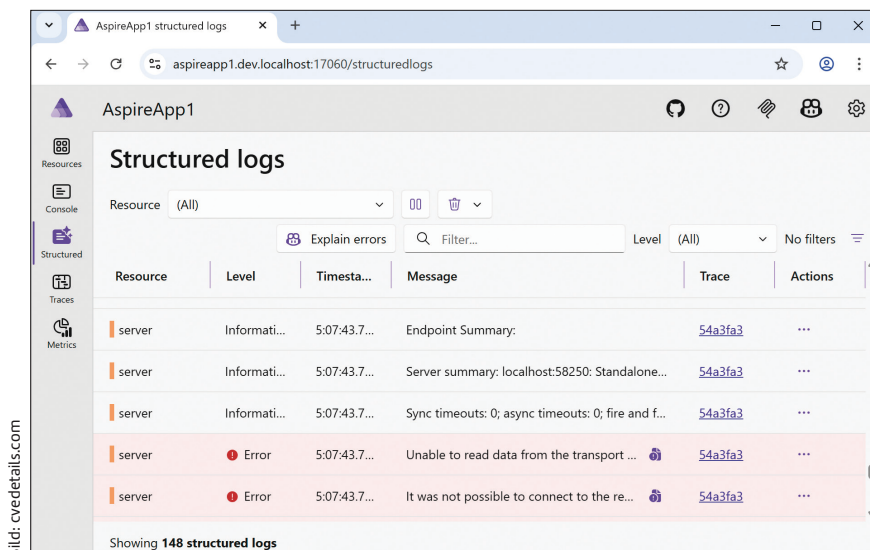


Bild: cvedetails.com

Strebsam: Aspire bietet auch eine grafische Log-Ansicht (Bild 14)

von Authenticator-Apps vorbereitet; ein wenig schade nur, dass die Dokumentation für das Erstellen eines QR-Codes eine 2015 (!) letztmals aktualisierte Bibliothek empfiehlt [26]. Das heißt freilich nicht, dass das nicht gut funktionieren würde. Auch Passkeys sind seit .NET 10 Teil der Templates und auch der Code-Generierung für ASP.NET Core Identity. Nach dem Login in eine Anwendung erscheint in der Nutzerverwaltung eine Option für Passkeys, über die sie angelegt und konfiguriert werden können (Bild 13). Ganz ohne Passkeys sollte der Zugriff freilich weiterhin möglich sein, denke man nur daran, dass ein Nutzer das Ökosystem wechselt oder bei einem der großen Anbieter ausgesperrt wird, dessen Identität aber dort hängt.

Ganz allgemein gilt: Wenn es für sicherheitskritische Aspekte Standards und/oder Referenzimplementierungen gibt, ergibt es Sinn, sich auf diese draufzusatteln und das Rad nicht neu zu erfinden. Der Mechanismus von ASP.NET Core Identity wird, gerade wenn es um manche Anpassungen geht, von einigen als komplex und eher mäßig dokumentiert beschrieben, aber er ist eine sehr solide Basis, sowohl für klassische Webanwendungen als auch für (Blazor-)SPAs.

#8: Software or Data Integrity Failures

Punkt 8 der OWASP Top Ten hieß in der Vorgängerausgabe noch anders: Damals stand anstelle des „oder“ im Titel ein „und“. Die Umbenennung ergibt Sinn, denn sowohl Daten- als auch Softwareintegrität sind permanent sicherzustellen.

Bei der Datenintegrität war lange Jahre die Deserialisierung ein Schwachpunkt vieler Anwendungen. Allzu oft wurde zu optimistisch vorgegangen; eingehende Daten – etwa an REST- oder früher SOAP-Endpunkten – wurden fröhlich in .NET-Objekte umgewandelt. Im schlimmsten Fall war so das Einschleusen von Code möglich. Im .NET Framework war der *BinaryFormatter* besonders berüchtigt, denn den konnte Microsoft nicht abdichten. Was tut man, wenn es etwas Unsicheres gibt, das aber in vielen Anwendungen steckt? Zunächst gibt man eine Warnung aus, und tatsächlich gibt es hierfür die Codeanalyse-Regel CA2300 [27]. In .NET 9 wurde der *BinaryFormatter* dann entfernt.

Ein Aspekt von Softwareintegrität wiederum besteht darin, dass bestehender Code nicht nachträglich modifiziert wird. Je nach Einsatzszenario bietet es sich an, zu überprüfen, ob beispielsweise am Ende eines Verarbeitungsschritts (etwa einer bestimmten Build-Pipeline) der Code immer noch derselbe ist wie vorher. Auch das Build-System könnte ja betroffen sein. Für Webinhalte bietet sich unter Umständen Sub-Resource Integrity an (SRI) [28]. Dieses Feature erlaubt es Browsern, beim Laden von CSS-Styles oder JavaScript-Code die Inhalte vom Server per Hash-Wert zu überprüfen. Dieser wird im *integrity*-Attribut des *<link>*- beziehungsweise *<script>*-Tags angegeben. Hier ein Beispiel für Bootstrap:

```
<link href="https://cdn.jsdelivr.net/npm/bootstrap
@5.3.8/dist/css/bootstrap.min.css" rel="stylesheet"
integrity="sha384-sRI14..." crossorigin="anonymous">
<script src="https://cdn.jsdelivr.net/npm/bootstrap
@5.3.8/dist/js/bootstrap.bundle.min.js"
```

```
integrity="sha384-FKyoE..." crossorigin="anonymous">
</script>
```

Stimmt der Hash der geladenen Ressource nicht mit dem *integrity*-Wert überein, wird das CSS beziehungsweise der JavaScript-Code nicht ausgeführt.

#9: Security Logging and Alerting Failures

Sie merken es schon – je weiter wir in der OWASP Top Ten kommen, desto kürzer werden die zugehörigen Erläuterungen. Nicht, dass es nicht jede Menge zu sagen gäbe, aber die Relevanz nimmt tatsächlich ein wenig ab. Und manche Listenpunkte sind auch ein wenig selbsterklärend, Nummer 9 etwa: Logging ist wichtig, insbesondere, wenn es sich um sicherheitsrelevante Ereignisse handelt. Aber ein Log allein ist noch keine Lösung. Warten, bis die Festplatte vollläuft, ist keine tragfähige Strategie. Insofern ist es wichtig, einen Prozess zu haben, der die Logs regelmäßig überprüft. Bei bestimmten Vorkommnissen ist auch eine automatische Benachrichtigung (das OWASP-Listenelement verwendet das etwas plakativere Wort „Alarm“) angemessen.

Nehmen wir als Beispiel eine Content Security Policy, wie sie zuvor schon als mögliches Gegenmittel für Cross-Site Scripting thematisiert wurde. Wenn es zu einer Policy-Verletzung kommt, gibt es eigentlich nur zwei mögliche Ursachen: Fehler in der Policy (ein legitimer Teil der Anwendung funktioniert nicht mehr) oder eine aktuell ausgenutzte Sicherheitslücke (was durch die Policy gestoppt wird). In beiden Fällen will das Entwicklungsteam sofort Bescheid wissen. Aus diesem Grund ist das Reporting-Feature von Content Security Policy so wichtig. Diese Meldungen sollen dann aber nicht in einem Datengrab landen, sondern direkt innerhalb des Teams eskaliert werden.

Eine häufig stattfindende Diskussion dreht sich darum, ob nicht etwa Aspire [29] (ehemals: .NET Aspire, aber Microsoft hat die Hoffnung, dass die Technologie auch außerhalb der eigenen Blase Zuspruch findet) diese Aufgabe erfüllen könnte. Ein schickes Dashboard zeigt dort entsprechende Log-Einträge (Bild 14). In der Praxis scheint Aspire nach anekdotischen Erfahrungen des Autors primär während der Entwicklung im Einsatz zu sein, gibt es doch für Produktivsysteme mögliche Alternativen (etwa Grafana [30] oder die entsprechenden Cloud-Überwachungstools, in Microsofts Cloud etwa Azure Monitor Insights [31]). Technologische Grundlage für viele dieser Ansätze ist OpenTelemetry, ein offener Standard, um mit Telemetriedaten zu arbeiten. Unter [32] gibt es weitere Informationen, wie sich das in .NET einsetzen lässt.

#10: Mishandling of Exceptional Conditions

Ein neues Listenelement gibt es tatsächlich in der OWASP Top Ten von 2025. Dieses klingt aber ziemlich profan: Es geht darin um die Ausnahmebehandlung. Sollte das nicht eigentlich offenkundig sein? Exceptions müssen vollständig abgefangen werden. Außerdem ist es empfehlenswert, beim Auftreten einer Ausnahme auch zu überprüfen, ob noch weitere Aktionen notwendig sind, beispielsweise das Schließen von Datenbankverbindungen, das Zurückrollen von ►

Transaktionen oder das Löschen bestimmter temporärer Dateien.

In der praktischen Umsetzung ist das nicht ganz so allgemein lösbar, wie es in der Theorie klingt. Gerade bei ASP.NET hilft ein Blick auf die Art und Weise, wie Ausnahmen in die Projekttemplates einkonfiguriert sind. Die *Web-App*-Vorlage enthält folgenden Code in der Datei *Program.cs*; bei Blazor sieht es ähnlich aus:

```
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    // ...
}
```

Außerhalb des Entwicklungsmodus wird also der URI */Error* als globale Fehlerseite verwendet. Dieser legt beim Laden eine Request-ID an, anhand derer die aktuelle Anfrage möglichst eindeutig zugewiesen werden kann:

```
RequestId = Activity.Current?.Id ??
    HttpContext.TraceIdentifier;
```

Diese ID wird dann in der Oberfläche ausgegeben, wie **Bild 15** zeigt. Wichtig: Der darunterliegende Text weist auf die Vorteile des Entwicklungsmodus hin. Dieser hat natürlich auf dem Produktsystem nichts zu suchen, also entfernen Sie ihn am besten gleich aus der Datei *Error.cshtml*.

Je nachdem, was für eine Art von Fehler auftritt, kann die Handhabung unterschiedlich sein. HTTP 400 bei ungültigen Anfragen (etwa falsches Anti-CSRF-Token) ist möglicherweise schwerwiegender als „lediglich“ eine nicht gefundene Datei (HTTP 404). Eine auf dem Server aufgetretene Exception (HTTP 500) wiederum ist wieder besorgniserregend. Die entsprechenden HTTP-Statuscodes geben also die Marschrichtung vor.

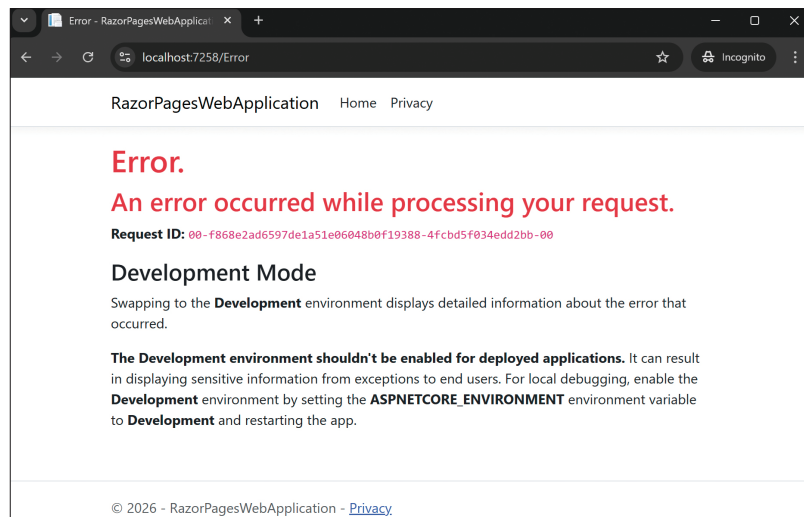
Folgende Middleware sorgt dafür, dass bei einem Fehler-Statuscode (also alles, was mit 4 oder 5 beginnt), zu einem Endpunkt weitergeleitet wird. Dieser wird als Parameter übergeben, im Beispiel heißt er *Error-400*, *Error-404* oder *Error-500*, abhängig vom Statuscode.

```
app.UseStatusCodePagesWithRedirects("/Error-{0}");
```

Was fehlt?

Damit sind wir am Ende der OWASP Top Ten 2025 angelangt. Bleibt nur noch eine Frage: Was hat keine Aufnahme in die Liste gefunden, hätte aber ganz gut gepasst? Wie versprochen hier die subjektiven Punkte 11 bis 13:

- Web-Security ist nur nicht technisch, sondern auch organisatorisch. Ein fehlender, unzureichender oder nicht gelebter Sicherheitsprozess konterkariert alle Bemühungen.
- Code ist das eine, Infrastruktur das andere. Schutz vor Ressourcenknappheit durch übermäßige Requests gehört ebenfalls in eine Sicherheitsstrategie.



Fehlerkultur: Angreifen werden keine Informationen über die Exception verraten (**Bild 15**)

- KI kann ein Segen sein – allerdings auch für Angreifer. Die Möglichkeiten von LLMs und Co. sorgen auch für ungeahnte Sicherheitsrisiken. Dieser Punkt würde die OWASP Top Ten gut ergänzen; fairerweise muss erwähnt werden, dass es von der OWASP unter anderem eine dedizierte Top Ten für Large Language Models [33] gibt.

Ungefähr 2029, mit der nächsten OWASP Top Ten, wird sich zeigen, ob diese Wünsche erfüllt wurden. ■

- [1] Das Open Web Application Security Project, <https://owasp.org/>
- [2] Liste aller OWASP-Projekte, <https://owasp.org/projects/>
- [3] Die OWASP Top Ten 2025, <https://owasp.org/Top10/2025/>
- [4] Christian Wenz, *No risk, more fun, dotnetpro 12/2021*, Seite 54 ff.
- [5] OWASP Global AppSec USA 2025, <https://owasp.glueup.com/event/owasp-2025-global-appsec-usa-washington-dc-131624/>
- [6] Die OWASP Top 10 auf X, ehemals Twitter, <https://x.com/OWASPTop10>
- [7] Die separate (selten aktualisierte) Website der OWASP Top 10, www.owasptopen.org
- [8] CWE-Website der MITRE, <https://cwe.mitre.org/>
- [9] Content Security Policy, www.w3.org/TR/CSP/
- [10] Permissions-Policy, www.w3.org/TR/permissions-policy/
- [11] Referrer-Policy, www.w3.org/TR/referrer-policy/
- [12] X-Content-Type-Options als Teil der Beta 2 des Internet Explorer 8, <https://learn.microsoft.com/en-us/archive/blogs/ie/ie8-security-part-vi-beta-2-update>
- [13] X-FRAME-OPTIONS, <https://datatracker.ietf.org/doc/html/rfc7034>
- [14] Gitleaks, <https://github.com/gitleaks/gitleaks>
- [15] Der Wurm Shai-Hulud, www.cisa.gov/news-events/alerts/2025/09/23/widespread-supply-chain-compromise-impacting-npm-ecosystem

- [16] Daniel Abeles, GitLab discovers widespread npm supply chain attack, <https://about.gitlab.com/blog/gitlab-discovers-widespread-npm-supply-chain-attack/>
- [17] Schutzmaßnahmen in NuGet, <https://devblogs.microsoft.com/dotnet/building-a-safer-future-how-nuget-is-tackling-software-supply-chain-threats/>
- [18] dotnet-outdated, <https://github.com/dotnet-outdated/dotnet-outdated>
- [19] RFC 6761 (.localhost und andere Spezialdomains), <https://datatracker.ietf.org/doc/html/rfc6761>
- [20] HTTP Strict Transport Security (HSTS), <https://datatracker.ietf.org/doc/html/rfc6797>
- [21] CVE-Datenbank, www.cvedetails.com
- [22] CSP-Empfehlung für Blazor-Apps, <https://learn.microsoft.com/en-us/aspnet/core/blazor/security/content-security-policy?view=aspnetcore-10.0#server-side-blazor-apps>
- [23] Security Development Lifecycle (SDL), www.microsoft.com/en-us/securityengineering/sdl
- [24] NIST-Passwortrichtlinien, <https://pages.nist.gov/800-63-3/sp800-63b.html>
- [25] BSI-Passwortrichtlinien, www.bsi.bund.de/DE/Themen/Verbraucherinnen-und-Verbraucher/Informationen-und-Empfehlungen/Cyber-Sicherheitsempfehlungen/Accountschutz/Sichere-Passwoerter-erstellen/sichere-passwoerter-erstellen_node.html
- [26] QR-Codes zum Einrichten einer Authenticator-App mit ASP.NET Core, <https://learn.microsoft.com/en-us/aspnet/core/security/authentication/identity-enable-qrcodes>
- [27] Codeanalyse versus BinaryFormatter, <https://learn.microsoft.com/en-us/dotnet/fundamentals/code-analysis/quality-rules/ca2300>
- [28] Sub-Resource Integrity (SRI), www.w3.org/TR/sri-2/
- [29] Aspire, <https://aspire.dev>
- [30] Grafana, <https://grafana.com>
- [31] Azure Monitor Insights, <https://learn.microsoft.com/de-de/azure/azure-monitor/visualize/insights-overview>
- [32] OpenTelemetry für .NET, <https://learn.microsoft.com/en-us/dotnet/core/diagnostics/observability-with-otel>
- [33] OWASP Top Ten für LLMs, <https://genai.owasp.org/llm-top-10/>

**Christian Wenz**

ist Softwarearchitekt, Berater und Spezialist für die Themen Performance und Web Security. Seit 2004 ist er Microsoft MVP. Seine neuesten Bücher sind „ASP.NET Core Security“ (erschienen bei Manning) und „C# und .NET 8“ (Hanser). www.christianwenz.de

DWX ACADEMY

Hands-on Workshops
und Weiterbildung für
Software-Entwickler
und -Architekten

SICHERHEIT FÜR ASP.NET CORE-APPLIKATIONEN

- Architektur des Web und mögliche Schwachstellen
- Verschlüsselungs- und Authentifizierungsverfahren
- Hashingverfahren
- OAuth und OpenId Connect (OIDC)
- OWASP Top Ten
- Härtung des ASP.NET Core Backends
- Härtung des HTML / JavaScript Frontends
- Härtung über die Infrastruktur

NEU!
2 TAGE

**ROLAND KÖNIG**

Deine Ansprechpartnerin: Susanne Herl
+49 731 88005-8835 • susanne.herl@ebnermedia.de

developer-world.de/dwx-academy

RUST UND WASM

Ein starkes Team

In Kombination mit Rust ist WebAssembly ein mächtiges Werkzeug.

WebAssembly (Wasm) wird häufig als „Gamechanger für das Web“ bezeichnet. Gleichzeitig bleibt es für viele Entwickler abstrakt: Man hört von nahezu nativer Performance im Browser, von sicheren Sandboxes und von der Möglichkeit, Systemprogrammiersprachen wie Rust im Web einzusetzen – doch was bedeutet das konkret? Im Rahmen dieses Artikels wird ein kleiner, aber dennoch leistungsfähiger BASIC-Interpreter in Rust entwickelt, der als WebAssembly-Modul im Browser ausgeführt werden kann.

WebAssembly ist eine portable Ausführungsumgebung, die in Laufzeitumgebungen, etwa den gängigen Browsern oder in Node.js, verfügbar ist. Wasm erlaubt es, rechenintensive oder sicherheitskritische Logik aus dem klassischen JavaScript-Kontext herauszulösen und in einer klar definierten, kontrollierten Sandbox auszuführen. WebAssembly ist aber keine Konkurrenz zu JavaScript, sondern eine Ergänzung. JavaScript bleibt für Orchestrierung und UI zuständig, während WebAssembly gezielt dort eingesetzt wird, wo Struktur, Performance und Vorhersagbarkeit entscheidend sind.

WebAssembly: ein Überblick

Man kann sich Wasm wie eine sehr einfache, strikt reglementierte virtuelle Maschine vorstellen. Sie kennt keine Benutzeroberflächen, keine Browser-APIs und keine komfortablen Standardbibliotheken. Dafür bietet sie etwas anderes: Vorhersagbarkeit. Code, der in Wasm ausgeführt wird, verhält sich deterministisch, läuft in einer Sandbox und kann von außen nur über explizit definierte Schnittstellen angesprochen werden. Genau diese Eigenschaften machen WebAssembly interessant, sobald Logik komplexer wird.

JavaScript ist hervorragend darin, Benutzerinteraktionen zu steuern, Events zu verarbeiten und APIs anzusprechen. Sobald es jedoch um komplexere Funktionalitäten geht, ist der Code schwer zu überblicken. Dynamische Typisierung, implizite Konvertierungen und ein Laufzeitmodell, das stark vom JIT-Compiler abhängt, machen solche Systeme fehleranfällig. Das ist kein Vorwurf an JavaScript, sondern eine Folge seiner ursprünglichen Zielsetzung.

WebAssembly schließt diese Lücke. Es erlaubt, Logik in Programmiersprachen zu schreiben, die von Anfang an für strukturierte Programme, explizite Speicherverwaltung und Performance entworfen wurden. Der Browser oder eine andere Laufzeitumgebung stellt lediglich die Ausführungsplattform bereit, während der eigentliche Code in einer isolierten Umgebung läuft. Dadurch entsteht eine klare Trennung: hier die Orchestrierung und Darstellung, dort die Rechenlogik.

An dieser Stelle kommt Rust ins Spiel. Rust ist keine Systemprogrammiersprache im historischen Sinn, sondern eine

moderne Antwort auf altbekannte Probleme. Sie kombiniert niedrige Abstraktionsebenen mit einem strengen Typsystem und einem Ownership-Modell, das Speicherfehler bereits zur Kompilierungszeit verhindert. In Rust gibt es keinen Garbage Collector, keine versteckten Laufzeitkosten und keine impliziten Speicherzugriffe. Was im Code steht, ist das, was tatsächlich passiert.

Diese Eigenschaften passen erstaunlich gut zu WebAssembly. Da Wasm selbst keine Speicherverwaltung mitbringt, profitieren Programme enorm von einer Programmiersprache, die Sicherheit ohne zusätzliche Laufzeit garantiert. Rust füllt genau diese Lücke. Das Ergebnis ist robuster und zugleich effizienter Code, der sich sehr gut in eine Wasm-Umgebung einfügt. Für Entwickler bedeutet das weniger Überraschungen und mehr Kontrolle – zwei Punkte, die gerade bei komplexer Logik entscheidend sind.

Doch warum habe ich ausgerechnet einen BASIC-Interpreter als Beispiel für diesen Artikel gewählt? Weil ein Interpreter ein ideales Lehrstück ist. Er zwingt dazu, eine Programmiersprache formal zu beschreiben, Eingaben zu analysieren, einen internen Zustand zu verwalten und Schritt für Schritt Anweisungen auszuführen. All das passiert ohne grafische Oberfläche, ohne Netzwerk und ohne externe Abhängigkeiten. Die gesamte Komplexität liegt in der Logik selbst.

BASIC wiederum ist bewusst einfach. Die Sprache ist historisch gewachsen, klar strukturiert und kommt mit sehr wenigen Konzepten aus. Genau das macht sie ideal für ein Beispielprojekt. Der Interpreter, den wir in diesem Artikel bauen, ist kein vollständiges BASIC und erhebt auch keinen Anspruch darauf. Er ist ein minimaler Sprachkern, der gerade genug Funktionalität bietet, um die wesentlichen Ideen zu demonstrieren. Am Ende soll nicht nur klar sein, wie WebAssembly technisch funktioniert, sondern vor allem, warum die Kombination mit Rust so gut funktioniert. Ziel ist es, ein Gefühl dafür zu entwickeln, wann dieser Ansatz sinnvoll ist – und wann nicht. WebAssembly ist kein Allheilmittel. Aber in Kombination mit der richtigen Sprache und einem klar abgegrenzten Problemraum ist es ein erstaunlich mächtiges Werkzeug.

Die Grundlagen

Bevor wir unsere erste Zeile Rust-Code schreiben, lohnt es sich, kurz innezuhalten und zu klären, was WebAssembly überhaupt ist. Viele Missverständnisse entstehen, weil versucht wird, es mit bekannten Konzepten gleichzusetzen. WebAssembly ist weder eine Programmiersprache noch eine klassische virtuelle Maschine wie die JVM oder das .NET Framework. Es ist etwas eigenständiges – bewusst minimalistisch und gerade deshalb so wirkungsvoll.

Der Kern von WebAssembly ist ein binäres Ausführungsformat. Dieses Format wurde so entworfen, dass es von Browsern und anderen Laufzeitumgebungen effizient geladen, validiert und ausgeführt werden kann. Wasm definiert exakt, welche Operationen erlaubt sind, wie Speicher aussieht und wie Code mit seiner Umgebung kommuniziert.

Ein hilfreiches Bild ist das einer sehr einfachen CPU, die in einer Sandbox läuft. Diese CPU kennt nur einige Datentypen, arbeitet Stack-basiert und greift auf einen linearen Speicherbereich zu. Es gibt keine Objekte, keine Klassen, keine Ausnahmen und keinen Garbage Collector. Alles, was über diese Grundfunktionen hinausgeht, muss entweder vom Compiler der Ursprungssprache oder vom Host bereitgestellt werden.

Der Stack ist dabei zentral. Wasm ist eine Stack-basierte Maschine, das heißt: Operationen nehmen ihre Operanden vom Stack und legen ihre Ergebnisse wieder dort ab. Eine Addition etwa besteht aus zwei Stack-Pops gefolgt vom Push des Ergebnisses. Dieses Modell mag zwar auf den ersten Blick ungewohnt wirken, ist aber extrem gut für Validierung und Optimierung geeignet. Es erlaubt der Laufzeitumgebung, sicherzustellen, dass ein Modul korrekt und sicher ist, bevor es überhaupt ausgeführt wird.

bereit. Die Sicherheit entsteht durch die Kombination mit einer geeigneten Programmiersprache.

Ein weiteres zentrales Konzept sind Importe und Exporte. Wasm-Module sind grundsätzlich isoliert. Kein Zugriff auf das DOM, kein Dateisystem, kein Netzwerk. Wenn ein Modul mit der Außenwelt interagieren soll, muss diese Interaktion explizit definiert werden. Funktionen, die das Modul nach außen bereitstellt, werden exportiert. Funktionen oder Ressourcen, die es von der Umgebung benötigt, werden importiert.

Dieses Modell hat weitreichende Konsequenzen. Es zwingt zu klaren Schnittstellen. Ein Wasm-Modul ist kein monolithischer Block, der irgendwo im Hintergrund läuft, sondern eine klar abgegrenzte Komponente. Für unseren BASIC-Interpreter bedeutet das: Er wird keine direkte Ausgabe machen können. Stattdessen liefert er seine Ergebnisse an den Host zurück, der dann entscheidet, wie diese weiterverarbeitet oder angezeigt werden.

Ein besonders wichtiger Aspekt von Wasm ist die Sandbox. Jedes Modul läuft in einer isolierten Umgebung. Selbst fehlerhafter oder bösartiger Code kann keine Probleme verursachen, solange die Laufzeitumgebung korrekt implementiert ist. Speicherzugriffe sind auf den eigenen linearen Speicher

```

~/dev/GitHub/SQLpassion/osdev/main64 on main rustup target add wasm32-unknown-unknown
info: downloading component 'rust-std' for 'wasm32-unknown-unknown'
info: installing component 'rust-std' for 'wasm32-unknown-unknown'
~/dev/GitHub/SQLpassion/osdev/main64 on main brew install wasm-pack
Warning: Calling 'depends_on macos: :high_sierra' is deprecated! There is no replacement.
Please report this issue to the powershell/homebrew-tap tap (not Homebrew/* repositories):
/opt/homebrew/Library/Taps/powershell/homebrew-tap/Formula/powershell.rb:33

=> Fetching downloads for: wasm-pack
✓ Bottle Manifest wasm-pack (0.13.1)
✓ Bottle Manifest rustup (1.28.2)
✓ Bottle wasm-pack (0.13.1)
✓ Bottle rustup (1.28.2)
=> Installing wasm-pack dependency: rustup
=> Pouring rustup--1.28.2.arm64_tahoe.bottle.tar.gz
🍺 /opt/homebrew/Cellar/rustup/1.28.2: 27 files, 11.5MB
=> Pouring wasm-pack--0.13.1.arm64_tahoe.bottle.tar.gz
🍺 /opt/homebrew/Cellar/wasm-pack/0.13.1: 10 files, 6.2MB
=> Running 'brew cleanup wasm-pack'...
Disable this behaviour by setting 'HOMEBREW_NO_INSTALL_CLEANUP=1'.
Hide these hints with 'HOMEBREW_NO_ENV_HINTS=1' (see 'man brew').
~/dev/GitHub/SQLpassion/osdev/main64 on main

```

Die ersten Schritte: Rust und Wasm werden bekannt gemacht und das Wasm-Pack installiert (Bild 1)

Neben dem Stack gibt es noch den linearen Speicher. Dieser ist nichts weiter als ein zusammenhängendes Byte-Array. Es gibt keine Zeiger im klassischen Sinn, sondern nur Offsets in diesem Speicherbereich. Aus Sicht von WebAssembly ist dieser Speicher vollständig typfrei.

Erst die Sprache, aus der der Code stammt – hier Rust – legt fest, wie dieser Speicher interpretiert wird. Genau hier liegt ein wichtiger Punkt: WebAssembly schützt nicht vor logischen Speicherfehlern, sondern stellt lediglich die Grundlage

beschränkt, Funktionsaufrufe sind streng typisiert, und es gibt keine Möglichkeit, willkürlich Code auszuführen. Diese Sicherheitsgarantien sind einer der Hauptgründe, warum WebAssembly im Browser überhaupt möglich ist.

Gleichzeitig ist WebAssembly erstaunlich portabel. Das gleiche Modul kann im Browser, in einer Serverumgebung oder in anderen spezialisierten Laufzeitumgebungen ausgeführt werden. Die CPU-Architektur spielt dabei keine Rolle, da Wasm bewusst hardwareunabhängig ist. ►


```

~/Documents/source/wasm-rust-demo/rust-wasm-lib wasm-pack build --release --target web
[INFO]: Checking for the Wasm target...
[INFO]: Compiling to Wasm...
Finished `release` profile [optimized] target(s) in 0.00s
[INFO]: Installing wasm-bindgen...
[INFO]: Optimizing wasm binaries with `wasm-opt`...
[INFO]: Optional fields missing from Cargo.toml: 'description', 'repository', and 'license'. These are not necessary, but recommended
[INFO]: Done in 0.23s
[INFO]: Your wasm pkg is ready to publish at /Users/klaus/Documents/source/wasm-rust-demo/rust-wasm-lib/pkg.
~/Documents/source/wasm-rust-demo/rust-wasm-lib

```

Der Rust-Code wird für Wasm kompiliert (Bild 2)

Das bedeutet auch, dass Performance nicht durch aggressive JIT-Optimierungen entsteht, sondern durch ein klar definiertes, vorhersagbares Ausführungsmodell. Wasm-Code verhält sich auf unterschiedlichen Plattformen sehr ähnlich – ein großer Vorteil für ein reproduzierbares Verhalten.

Wichtig ist auch, zu verstehen, was WebAssembly nicht ist. Es ist kein Ersatz für JavaScript, kann keine Events abonnieren, kein UI zeichnen und keine APIs ansprechen. All das bleibt Aufgabe des Hosts. WebAssembly ist ein Werkzeug für klar abgegrenzte Logik, nicht für Anwendungssteuerung. In diesem Zusammenspiel liegt seine Stärke.

Ein einfaches Beispiel

Bevor wir uns an die Implementierung eines BASIC-Interpreters wagen, ist es wichtig, ein erstes greifbares Erfolgserlebnis zu haben. Erst wenn tatsächlich ein Stück Rust-Code im Browser ausgeführt wird, entsteht ein klares Bild davon, wie die einzelnen Teile zusammenspielen. Genau darum geht es in diesem Abschnitt.

Wir formulieren eine kleine Funktion, die etwas berechnet oder einen Text verarbeitet und deren Ergebnis aus JavaScript heraus aufgerufen wird.

Bevor es losgeht, werden Rust und Wasm miteinander bekannt gemacht (Bild 1). Das geschieht mit diesem Befehl:

```
rustup target add wasm32-unknown-unknown
```

Außerdem müssen Sie das sogenannte *wasm-pack* installieren. Auf einem Mac können Sie dafür *brew* verwenden:

```
brew install wasm-pack
```

Rust ist grundsätzlich in der Lage, direkt nach WebAssembly zu kompilieren. Mit dem entsprechenden Target (*wasm32-unknown-unknown*) erzeugt der Rust-Compiler eine reine *.wasm*-Datei. Diese enthält ausführbaren Code, Funktionen, Speicherezugriffe und Kontrollstrukturen.

Aus Sicht von WebAssembly ist dieses Ergebnis vollkommen ausführbar. Aus Sicht eines Browsers ist es jedoch nur die halbe Lösung. Der Browser kann WebAssembly laden und ausführen, erfasst aber nicht die internen Datenstrukturen oder Typen von Rust.

Genau hier entsteht eine zentrale Lücke. Rust kennt komplexe Typen wie Strings, Referenzen und strukturierte Rückgabewerte, Wasm dagegen nur sehr einfache Datentypen. Ohne zusätzliche Hilfe müssten Sie den Speicher manuell verwalten, Strings selbst serialisieren und eine eigene JavaScript-Schicht schreiben, die mit Offsets und Zeigern arbeitet. Das ist zwar technisch möglich, praktisch aber zu viel Aufwand.

wasm-pack schließt genau diese Lücke. Es ist kein eigener Compiler, sondern ein Werkzeug, das den gesamten Build-Prozess koordiniert. Intern ruft es den Rust-Compiler auf, erzeugt das Wasm-Modul und führt anschließend ein weiteres Werkzeug aus, das für die eigentliche Magie sorgt: *wasm-bindgen*.

Dieses Werkzeug analysiert das erzeugte Wasm-Modul, erkennt die explizit freigegebenen Rust-Funktionen und generiert den notwendigen JavaScript-Code, um die Funktionen komfortabel aus dem Browser heraus aufrufen zu können.

Das Ergebnis dieses Prozesses ist nicht nur eine *.wasm*-Datei, sondern ein kleines Paket aus WebAssembly und JavaScript. Die JavaScript-Datei übernimmt das Laden des Moduls, initialisiert den Speicher und kümmert sich um die Übersetzung zwischen JavaScript-Typen und Rust-Typen. Für den Entwickler fühlt sich das Ergebnis dadurch fast so an, als würde er eine normale JavaScript-Bibliothek verwenden, obwohl die eigentliche Logik in Rust implementiert ist.

Das erste Beispiel ist eine einfache Rechenfunktion. Sie nimmt zwei Zahlen entgegen, addiert diese und liefert das Ergebnis zurück. In Rust sieht dieser Code unspektakulär aus.

```

use wasm_bindgen::prelude::*;
// Ein sehr einfaches Beispiel: Addition zweier i32.
#[wasm_bindgen]
pub fn add(a: i32, b: i32) -> i32 {a + b}

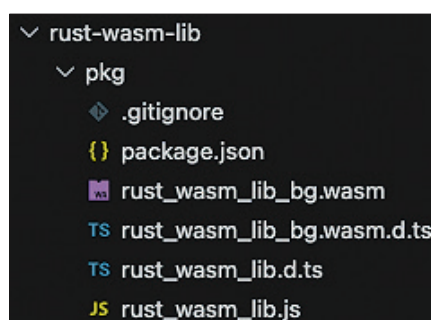
```

Über das Attribut *[wasm_bindgen]* wird die Funktion *add()* exportiert und kann somit innerhalb des JavaScript-Codes aufgerufen werden. Die aufgerufene Funktion läuft nicht mehr als JavaScript, sondern als kompiliertes Modul mit einem klar definierten ABI (Application Binary Interface [1]). Der folgende Befehl (Bild 2) kompiliert die Rust-Library und erzeugt daraus Wasm-Code:

```

wasm-pack build --release
--target web

```



Die erzeugten Dateien (Bild 3)

```

~/Documents/source/wasm-rust-demo python3 -m http.server 8080
Serving HTTP on :: port 8080 (http://[::]:8080/) ...
::1 - - [11/Jan/2026 21:04:48] "GET /rust-wasm-lib/pkg/rust_wasm_lib.js HTTP/1.1" 304 -
::1 - - [11/Jan/2026 21:04:48] "GET /rust-wasm-lib/pkg/rust_wasm_lib_bg.wasm HTTP/1.1" 304 -
::1 - - [11/Jan/2026 21:06:58] "GET /web/ HTTP/1.1" 304 -
::1 - - [11/Jan/2026 21:06:58] "GET /rust-wasm-lib/pkg/rust_wasm_lib.js HTTP/1.1" 304 -
::1 - - [11/Jan/2026 21:06:58] "GET /rust-wasm-lib/pkg/rust_wasm_lib_bg.wasm HTTP/1.1" 304 -
::1 - - [11/Jan/2026 21:06:58] "GET /web/ HTTP/1.1" 304 -
::1 - - [11/Jan/2026 21:06:59] "GET /web/ HTTP/1.1" 304 -

```

Starten eines Browsers mit Python (Bild 4)

Durch den Aufruf von *wasm-pack* werden im Ordner *pkg* die in Bild 3 gezeigten Dateien erzeugt. Die wichtigste ist die *.wasm*-Datei, das eigentliche WebAssembly-Modul. Die Datei

- enthält den kompilierten Rust-Code,
- besteht aus WebAssembly-Bytecode,
- läuft in der WebAssembly-Runtime des Browsers (oder einer anderen Wasm-Runtime) und
- kennt keine JavaScript-Typen, kein DOM und kein I/O.

Bei der *.js*-Datei handelt es sich um das Bindeglied zwischen JavaScript und WebAssembly. Sie

- übernimmt das Laden der *.wasm*-Datei,
- initialisiert den Speicher,
- übersetzt zwischen JavaScript- und Rust-Datentypen und
- stellt ein benutzerfreundliches JavaScript-API zur Verfügung.

Die *.ts*-Dateien enthalten Typdefinitionen für TypeScript, Parameter und Rückgabewerte. Listing 1 zeigt den für den Aufruf des Wasm-Moduls erforderlichen JavaScript-Code.

Nachdem das Wasm-Modul kompiliert wurde, können Sie mit Python einen lokalen Webserver starten, der lokal über den Port 8080 zu erreichen ist:

```
python3 -m http.server 8080
```

Der Zugriff auf die Webanwendung, die das Wasm-Modul mithilfe von JavaScript ausführt (siehe Bild 4 und Bild 5), erfolgt mit *http://localhost:8080/web/index.html*.

Dieses einfache Beispiel zeigt deutlich, was WebAssembly nicht tut. Es kümmert sich nicht um Benutzerinteraktionen, nicht um das Layout und auch nicht um Ereignisse. Es ist kein Ersatz für JavaScript, sondern ein Baustein, der gezielt dort eingesetzt wird, wo er sinnvoll ist.

Im nächsten Schritt wird dieses Grundprinzip erweitert. Wir beginnen damit, eine kleine Laufzeitumgebung für den BASIC-Interpreter zu bauen. Anstelle eines simplen Rückgabewerts werden wir mit Programmen arbeiten, die aus mehreren Anweisungen bestehen. Doch dies klappt wie bereits geschildert: ein klar abgegrenztes Wasm-Modul, geschrieben in Rust, eingebettet in eine JavaScript-Umgebung.

Architektur des Interpreters

Zuerst wird die Struktur des Systems definiert. Ein Interpreter ist weniger eine Ansammlung einzelner Funktionen als das Zusammenspiel mehrerer klar abgegrenzter Komponenten. Im Zentrum steht ein sauberer Datenfluss. Ein BASIC-Pro-

gramm wird als Text an den Interpreter übergeben. Dieser Text wird zunächst analysiert, anschließend in eine interne Repräsentation (Abstract Syntax Tree, AST) überführt und schließlich Schritt für Schritt ausgeführt. Jede dieser Phasen ist klar voneinander getrennt, was den Code nicht nur übersichtlich macht, sondern auch didaktisch leichter zugänglich.

Der erste Schritt ist das Zerlegen des Programms in einzelne Zeilen. BASIC ist traditionell zeilenorientiert, und wir nutzen diese Eigenschaft konsequent. Jede Zeile beginnt mit einer Zeilennummer, gefolgt von genau einer Anweisung. ►

Listing 1: Zugriff auf Rust von JavaScript

```

import init, { add, shout } from
  "../rust-wasm-lib/pkg/rust_wasm_lib.js";

async function run()
{
  // initialisiert das Wasm-Modul
  await init();

  document.getElementById("sum").
    textContent = String(add(7, 35));
  document.getElementById("shout").
    textContent = shout("webassembly");
}

run().catch((err) =>
{
  console.error(err);
  document.body.insertAdjacentHTML(
    "beforeend",
    '<pre style="color:red;">${String(err)}</pre>'
  );
});

```

Rust + WebAssembly Minimal Demo

```

add(7, 35) = 42
shout("webassembly") = WEBASSEMBLY!!!

```

Ausführen des Rust-Codes im Browser (Bild 5)

Diese Struktur erlaubt es, den Kontrollfluss sehr einfach abzubilden. Die Zeilennummern dienen später als Sprungmarken für *GOTO*-Anweisungen und bestimmen die Reihenfolge der Ausführung.

Auf dieser Grundlage arbeitet der Lexer. Seine Aufgabe ist es, jede Zeile in eine Folge von Tokens zu zerlegen. Tokens sind die kleinsten bedeutungstragenden Einheiten der Sprache: Schlüsselwörter wie *LET*, *PRINT*, *IF*, *THEN*, *GOTO* oder *END*, Operatoren wie *+*, *-*, *=*, *<* und *>* sowie Literale und Variablenamen. Da unser BASIC-Subset sehr eingeschränkt ist, bleibt auch der Lexer überschaubar.

Aus den Tokens entsteht dann eine strukturierte Darstellung der Anweisung. Statt mit rohen Zeichenketten zu arbeiten, wandeln wir jede Zeile in eine klar typisierte interne Form. In Rust bietet sich hierfür eine Kombination aus *enum* und *struct* an. Jede BASIC-Anweisung wird zu einer eigenen Variante, etwa für Zuweisungen, Ausgaben oder Sprünge. Diese interne Darstellung ist der eigentliche Schlüssel zur späteren Ausführung, da sie die syntaktischen Details vollständig vom Laufzeitverhalten trennt.

Ein zentraler Bestandteil der Architektur ist das Variablenmodell. Um die Laufzeit möglichst einfach und effizient zu halten, verwenden wir ein festes Variablenschema. Integer-Variablen werden ausschließlich durch die Buchstaben A bis Z repräsentiert. String-Variablen folgen dem klassischen BASIC-Stil und tragen ein Dollarzeichen als Suffix, also A\$ bis Z\$.

Der Typ einer Variable ist allein durch ihren Namen festgelegt und kann sich zur Laufzeit nicht ändern. Diese Entscheidung erlaubt es, Variablen intern als zwei einfache Arrays abzubilden: eines für Integer-Werte und eines für Strings. Auf komplexe Datenstrukturen oder Hash-Maps kann vollständig verzichtet werden. Die Ausführung des Programms erfolgt über einen einfachen Program Counter (PC). Dieser zeigt stets auf die aktuell auszuführende Zeile. Abgearbeitet

● Listing 2: String-Literale im Tokenizer

```
if c == '"' {
    // string literal
    i += 1;
    let start = i;
    while i < chars.len() && chars[i] != '"' { i += 1; }

    if i >= chars.len() {
        return Err("Unterminated string literal"
            .to_string());
    }

    let s: String = chars[start..i].iter().collect();
    i += 1; // closing quote
    tokens.push(format!("{}", s));
    continue;
}
```

● Listing 3: Sprungtabelle

```
// Jump table + duplicate detection
let mut jump = HashMap::new();
for (i, l) in parsed.iter().enumerate() {
    if jump.insert(l.line_no, i).is_some() {
        return Err(format!(
            "Duplicate line number: {}", l.line_no));
    }
}
```

wird das Programm von der kleinsten bis zur größten Zeilennummer. Befehle wie *GOTO* oder *IF... THEN* können diesen Ablauf gezielt verändern, indem sie den Program Counter auf eine andere Zeile setzen. Existiert die angesprungene Zeile nicht, handelt es sich um einen Laufzeitfehler.

Der unterstützte Befehlssatz ist minimal, deckt aber alle wesentlichen Konzepte ab. Mit *LET* lassen sich Integer- oder String-Variablen zuweisen. Integer-Ausdrücke dürfen einfache Additionen und Subtraktionen enthalten, während String-Zuweisungen ausschließlich mit String-Literalen arbeiten.

PRINT erlaubt die Ausgabe eines einzelnen Ausdrucks, entweder eines Integer-Werts, einer String-Variable oder eines String-Literals. Die Ausgabe selbst erfolgt nicht direkt im Interpreter, sondern wird gesammelt und an die JavaScript-Seite zurückgegeben.

Mit *IF... THEN* führen wir eine einfache Form der bedingten Ausführung ein. Vergleiche sind auf Gleichheit sowie größer und kleiner beschränkt und dürfen nur zwischen gleichartigen Typen stattfinden. Der Teil nach *THEN* besteht genau aus einer einzelnen Anweisung, etwa einem *PRINT* oder einem *GOTO*. Auf verschachtelte Bedingungen oder mehrzeilige Blöcke verzichten wir bewusst.

Der Befehl *GOTO* ermöglicht explizite Sprünge zu einer bestimmten Zeilennummer und macht damit Schleifen und nichtlineare Programmabläufe möglich. Den Abschluss bildet *END*, das die Programmausführung sofort beendet.

Aus Sicht von WebAssembly passt diese Architektur hervorragend in das Ausführungsmodell. Der Interpreter ist reine Logik ohne direkten Zugriff auf die Außenwelt. Er benötigt weder Dateisystem noch Netzwerk noch Benutzeroberfläche. Alle Ein- und Ausgaben erfolgen über klar definierte Schnittstellen. Genau diese Eigenschaften machen ihn zum idealen Kandidaten für die Ausführung in einer Wasm-Sandbox.

Mit dieser Architektur als Grundlage können wir nun in die konkrete Implementierung einsteigen. Zunächst sehen wir uns an, wie Tokenizer und Parser in Rust umgesetzt werden, bevor wir zur Laufzeitumgebung und der eigentlichen Programmausführung übergehen.

Die Implementierung

Ziel dieses Abschnitts ist es, zu verstehen, wie aus einem Textprogramm eine ausführbare Struktur wird und warum die Aufteilung in Tokenizer, Parser und Interpreter sinnvoll ist.

Der Weg eines BASIC-Programms durch unser System ist klar definiert: Zuerst wird jede Zeile in Tokens zerlegt, dann werden diese Tokens in eine strukturierte Repräsentation übersetzt, und schließlich wird diese Struktur ausgeführt. Jede dieser Phasen übernimmt genau eine Aufgabe – und der Code dazu liegt in jeweils einer eigenen Rust-Quelldatei.

Der Tokenizer (*tokenizer.rs*) ist die erste Instanz, die mit dem Programmtext in Berührung kommt. Seine Aufgabe ist bewusst begrenzt: Er versteht keine Grammatik und keine Bedeutung, sondern zerlegt eine Zeile lediglich in sinnvolle Einheiten – in sogenannte Tokens. Der zentrale Einstiegspunkt ist diese Methode:

```
pub fn tokenize_line(&self, line: &str)
    -> Result<Vec<String>, String>
```

Diese Funktion nimmt eine einzelne BASIC-Zeile entgegen und liefert eine Liste von Tokens zurück. Dabei werden Leerzeichen ignoriert, Kommentare verworfen und bestimmte Zeichen gezielt isoliert.

Ein sehr wichtiger Sonderfall sind String-Literale. Alles zwischen doppelten Anführungszeichen muss als ein einzelnes Token behandelt werden, siehe [Listing 2](#). Mithilfe des Tokenizers wird aus *LET A\$ = "Hello World"* eine Tokenfolge wie:

```
["LET", "A$", "=", "\"Hello World \""]
```

Auch Operatoren wie *+*, *-*, *=*, *<* und *>* werden gezielt als eigene Tokens erzeugt. Das erleichtert es dem Parser später, Ausdrücke und Bedingungen zu erkennen, ohne Zeichenketten analysieren zu müssen. Wichtig ist hierbei, dass der Tokenizer keine Entscheidung trifft, ob ein Token gültig ist. Er liefert nur Material – die Bedeutung entsteht erst im nächsten Schritt.

Der Parser (*parser.rs*) ist das eigentliche Herzstück der Sprachdefinition. Hier wird entschieden, was eine gültige Anweisung ist und wie sie intern dargestellt wird. Der Einstiegspunkt für den gesamten Parser ist:

```
pub fn parse_program(
    &self, program: &str) -> Result<(Vec<ProgramLine>,
    HashMap<i32, usize>), String>
```

Der Parser liest das Programm zeilenweise, generiert die Tokens für jede Zeile und erwartet immer dieselbe Grundstruktur:

Zeilennummer + genau eine Anweisung

Diese Zeilennummern sind später entscheidend für den Kontrollfluss. Nach dem Parsen werden alle Zeilen sortiert und in eine Sprung-Tabelle überführt, siehe [Listing 3](#). Damit kann der Interpreter später einen Befehl wie *GOTO 30* in konstanter Zeit auflösen, ohne das gesamte Programm durchsuchen zu müssen. Statt Tokens während der Ausführung erneut zu interpretieren, übersetzt der Parser jede Anweisung in eine klar typisierte Struktur. Der Kern davon ist das Enum *Stmt* (*ast.rs*), siehe [Listing 4](#).

Alle syntaktischen Details sind damit erledigt. Der Interpreter sieht später keine Keywords, keine Tokens und keine Zeichenketten mehr – nur noch klar benannte Varianten.

Ein besonders eleganter Aspekt ist das Variablenmodell. Da Integer-Variablen ausschließlich A bis Z heißen dürfen und String-Variablen A\$ bis Z\$, kann der Parser Variablen direkt in Indizes übersetzen ([Listing 5](#)). Damit wird aus A automatisch der Index 0, aus Z der Index 25.

Integer-Ausdrücke bestehen aus Literalen, Variablen sowie den Operatoren *+* und *-*. Der Parser verarbeitet sie links-assoziativ ([Listing 6](#)). Das Ergebnis ist ein kleiner Ausdrucksbaum, der später direkt ausgewertet werden kann.

Für Strings bleibt alles bewusst einfach: Ein String-Ausdruck ist entweder ein Literal oder eine Variable – genau ein Token, keine Verkettung, keine Funktionen. Beim Parsen einer *IF*-Anweisung sucht der Parser explizit nach dem Schlüsselwort *THEN* ([Listing 7](#)). Nach Tokenizing und Parsing ist die Syntax vollständig verschwunden. Der Interpreter arbeitet nur noch mit:

- einer sortierten Liste von Programmzeilen,
- einer Sprungtabelle,
- zwei Variablen-Arrays und
- einem Programm-Counter.

Die Hauptschleife des Interpreters (*interpreter.rs*) ist bewusst einfach gehalten, siehe [Listing 8](#). Hier zeigt sich der größte Vorteil der vorangegangenen Schritte: Die Ausführung ist ein ►

● Listing 4: Enum Stmt

```
#[derive(Debug, Clone)]
pub enum Stmt {
    LetInt { var: usize, expr: IntExpr },
    LetStr { var: usize, expr: StrExpr },
    PrintInt(IntExpr),
    PrintStr(StrExpr),
    Goto(i32),
    IfThen { cond: Cond, then_stmt: Box<Stmt> },
    End,
    Nop }
```

● Listing 5: Parsen von Variablen

```
fn parse_int_var(s: &str)
    -> Result<usize, String> {
    if !is_int_var(s) {
        return Err(format!(
            "Invalid int variable '{}'. Use A..Z.", s));
    }
    let c = s.chars().next().unwrap().
        to_ascii_uppercase();
    Ok(((c as u8 - b'A') as usize)
    }
```


Listing 6: Integer parsen

```
fn parse_int_expr(&self, tokens: &[String])
-> Result<IntExpr, String> {
    // left-associative: term ( (+|-) term )*
    let mut i = 0;
    let mut expr =
        self.parse_int_term(tokens, &mut i)?;

    while i < tokens.len() {
        let op = tokens[i].as_str();
        if op != "+" && op != "-" {
            return Err(format!("Unexpected token in
                int expression: '{}'", tokens[i]));
        }

        i += 1;
        let rhs = self.parse_int_term(
            tokens, &mut i)?;
        expr = if op == "+" {
            IntExpr::Add(Box::new(expr),
                Box::new(rhs))
        } else {
            IntExpr::Sub(Box::new(expr),
                Box::new(rhs))
        };
    }
    Ok(expr)
}
```

einfaches *match*-Statement. Es gibt keine String-Vergleiche, keine Syntaxprüfung und keine Mehrdeutigkeiten mehr. Die

Ausgabe des Interpreters wird nicht direkt an den Browser geschickt, sondern gesammelt:

```
self.output.push_str(&format!("{}", v));
```

Listing 7: Parsen von IF-Statements

```
fn parse_if(&self, tokens: &[String])
-> Result<Stmt, String> {
    // IF <cond> THEN <stmt>
    let then_pos = tokens
        .iter()
        .position(|t| t.to_ascii_uppercase() ==
            "THEN")
        .ok_or_else(|| "IF requires
            THEN".to_string())?;

    if then_pos == 0 {
        return Err("IF missing condition".to_string());
    }

    if then_pos + 1 >= tokens.len() {
        return Err(
            "IF THEN missing statement".to_string());
    }

    let cond_tokens = &tokens[..then_pos];
    let then_tokens = &tokens[then_pos + 1..];

    let cond = self.parse_cond(cond_tokens)?;
    let then_stmt =
        self.parse_stmt(then_tokens)?;

    Ok(Stmt::IfThen {
        cond, then_stmt: Box::new(then_stmt),
    })
}
```

Am Ende wird dieser String an JavaScript zurückgegeben. Das passt perfekt zum Wasm-Modell und macht den Interpreter leicht testbar. Um Endlosschleifen zu vermeiden, zählt der Interpreter seine Schritte und bricht nach einem festen Limit ab. Das schützt das UI davor, durch fehlerhafte Programme eingefroren zu werden.

An dieser Stelle wird nun auch deutlich, wie sauber die Grenze zwischen WebAssembly und der Weboberfläche verläuft. Der Interpreter selbst weiß nichts über das DOM, Buttons oder Textareas – er nimmt lediglich einen Programmtext entgegen und liefert am Ende einen einzigen String zurück. Alles, was „Web“ ist, passiert ausschließlich auf der JavaScript-Seite: Eingabe lesen, WebAssembly initialisieren, Interpreter aufrufen, Ausgabe anzeigen.

Das funktioniert, weil *wasm-pack* neben der Wasm-Datei auch eine kleine JavaScript-Datei generiert, die als Binde-

Listing 8: Interpretierung

```
while pc < self.lines.len() {
    let stmt = self.lines[pc].stmt.clone();
    match stmt {
        Stmt::LetInt { var, expr } => { ... }
        Stmt::PrintInt(expr) => { ... }
        Stmt::Goto(target) => { ... }
        Stmt::IfThen { cond, then_stmt } => { ... }
        Stmt::End => break,
        _ => pc += 1,
    }
}
```

● Listing 9: Der JavaScript-Code

```
import init, { run } from "../pkg/rust_basic.js";

const $code =
  document.getElementById("code");
const $out = document.getElementById("out");
const $run = document.getElementById("run");
const $status =
  document.getElementById("status");

$code.value =
  '10 LET A = 1
  20 LET A$ = "COUNT"
  30 PRINT A$
  40 PRINT A
  50 LET A = A + 1
  60 IF A < 5 THEN GOTO 30
  70 END
  ';

async function main() {
  await init();
  $status.textContent = "wasm ready";
  $run.disabled = false;

  $run.addEventListener("click", () => {
    const program = $code.value;
    const output = run(program);
    $out.textContent = output;
  });

  $run.disabled = true;
  main().catch((e) => {
    console.error(e);
    $status.textContent = "failed to load wasm";
    $out.textContent = String(e);
  });
}
```

glied zwischen den beiden Welten dient. In dieser Datei steckt die ganze Magie: das Laden des Wasm-Moduls, das Anlegen und Verwalten des Speichers und – ganz wichtig – die Konvertierung zwischen JavaScript-Strings und Rust-Strings. Für unseren Code fühlt sich das danach so an, als würden wir ganz normale JavaScript-Funktionen importieren.

Im UI-Code sieht der Ablauf bewusst simpel aus: Zuerst wird das Modul initialisiert, dann wird beim Klick auf *Run* der Inhalt der Textarea an die Funktion *run()* übergeben und die Rückgabe in ein `<pre>`-Element geschrieben (Listing 9).

Hier sehen Sie auch sehr schön, warum wir im Rust-Teil die Ausgabe sammeln und als String zurückgeben: JavaScript muss nicht mit Streams, Callbacks oder komplexen Host-APIs hantieren. Das Ergebnis ist deterministisch und sofort verfügbar. Der Interpreter läuft aus Sicht von JavaScript synchron – der einzige asynchrone Schritt ist die Initialisierung (*init()*), weil das Wasm-Binary erst geladen werden muss.

Diese Aufteilung hat einen angenehmen Nebeneffekt: Die Weboberfläche bleibt extrem schlank. JavaScript übernimmt nur die Orchestrierung – Programm rein, Ausgabe raus – während die gesamte Fachlogik (Tokenizing, Parsing, Ausführung, Step-Limit) vollständig in Rust gekapselt bleibt. Genau so spielt Wasm seine Stärken aus: als isoliertes, portables Modul für klar abgegrenzte Logik, das sich in einer Web-App wie eine gewöhnliche Bibliothek verwenden lässt. Und innerhalb des Rust-Codes bilden der Tokenizer, der Parser und der Interpreter eine klar getrennte, leicht verständliche Pipeline. Jeder Teil ist klein, überschaubar und erfüllt genau eine Aufgabe.

Fazit

Am Beispiel eines kleinen, aber vollständigen BASIC-Interpreters (Bild 6) wurde gezeigt, wie sich Rust und WebAssembly in der Praxis ergänzen. Rust liefert eine klare, sichere und

BASIC Interpreter (Rust + WebAssembly)

```
10 LET A = 1
20 LET A$ = "COUNT"
30 PRINT A$
40 PRINT A
50 LET A = A + 1
60 IF A < 5 THEN GOTO 30
70 END
```

Run wasm ready

Output

```
COUNT
1
COUNT
2
COUNT
3
COUNT
4
```

Der Interpreter in Aktion (Bild 6)

gut strukturierte Implementierung, während WebAssembly eine portable, isolierte Ausführungsumgebung bereitstellt, die sich nahtlos in eine Webanwendung integrieren lässt. ■

[1] *Application Binary Interface*, https://en.wikipedia.org/wiki/Application_binary_interface

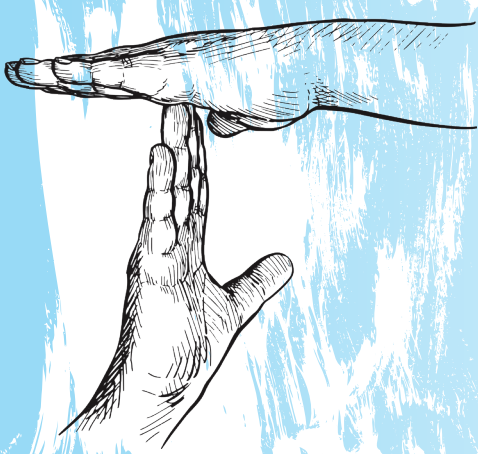


Klaus Aschenbrenner

berät Unternehmen in Europa beim Einsatz des Microsoft SQL Server und beschäftigt sich mit Windows-Programmierung und .NET. Zweimal zeichnete ihn Microsoft für sein Engagement als Microsoft MVP aus. Er ist zu erreichen über www.SQLpassion.at.

TIME-OUT

Wer entscheidet hier eigentlich?



Sind wir immer rationale Wesen? Oder fahren wir eher „auf Autopilot“?
Wer weiß, wie Menschen entscheiden, kann gezielter führen.

Als Entwickler glauben wir besonders gern an rationale Entscheidungen. Das liegt nahe: Code ist logisch, deterministisch, testbar. Wenn etwas nicht funktioniert, gibt es eine Ursache. Und (meist) eine dazu passende Lösung. Wenn Code so funktioniert, warum also nicht auch wir selbst? Versuchen wir nicht immer alles zu verstehen, damit wir besser handeln können? Frei nach dem Motto: Fakten rein, nüchterne Entscheidung und richtiges Verhalten raus. Ist das so?

Leider stimmen hier unsere subjektiven Erfahrungen und die Ergebnisse der Hirnforschung erstaunlich gut überein, denn beide sagen uns: Wohl eher nicht. Es ist eher chaotischer. Wie bei einem Verliebten, der nur noch rosa Wolken sieht, sind unsere Entscheidungen meist „getrübt“. Mit dem Unterschied, dass jeder von uns seinen eigenen „Pressesprecher“ hat, der die gefallenen Entscheidungen bestens nach außen verkaufen kann. So einen Pressesprecher brauchen wir – wir wollen ja nicht als unberechenbare Egoisten dastehen, sondern als verlässliche Kollegen und treue Staatsbürger.

Die Wissenschaft ist da erstaunlich präzise in ihren Aussagen: In Wirklichkeit fällt die Entscheidung, bevor der Kopf weiß, was los ist. Benjamin Libet hat bereits 1983 nachgemessen und festgestellt, dass Entscheidungen im Gehirn getroffen werden, bevor wir sie bewusst erleben – circa 300 bis 500 Millisekunden vorher [1][2]. Und dann dauert es noch mal 200 Millisekunden, bis unsere Begründung dafür da ist – die in den meisten Fällen überhaupt nichts mit unserer echten Motivation zu tun hat. Begründungen sind sehr rational, sie dienen dazu, dass wir in einem Team verlässlich zusammenarbeiten können. Sie sind aber eben nur vom Pressesprecher formuliert, nicht vom „Entscheider“.

Und Jonathan Haidt beschreibt zwei Jahrzehnte später, dass wir zu Urteilen kommen, bevor wir argumentieren [3][4]. Unser Verstand wird durch unsere Intuitionen in eine Richtung gelenkt, so wie moderne Scheinwerfer beim Auto auf Lenkbewegungen reagieren, und dann wird nur noch in diese Richtung ausgeleuchtet und nachgedacht. Denken in der Box. Oder mit klaren Vorgaben.

Das klingt alles sehr deprimierend, es ist aber wichtig, weil sehr viele Entscheidungen schnell passieren müssen. Daniel Kahneman nennt es das „schnelle Denken“ [5] und benennt klar den Vorteil: Es ist energiesparend. Und meistens richtig. Nicht immer, aber das „langsame Denken“ können wir uns auf Dauer nicht immer leisten. Nur wenn es Konflikte gibt zwischen mehreren „Lagern“ in unserem Bauch, dann kommt uns etwas zu Bewusstsein, und dann kann das langsame Denken einsetzen.

Das bedeutet: In der Regel ist die Entscheidung gefallen, der Code geschrieben, und erst dann kommt unsere Commit-Message. Damit die anderen sehen, was uns da geritten hat. Damit wir nicht ganz so unsozial wirken und nicht so, als täten wir nur, was uns Spaß macht – auch wenn das vielleicht die eigentliche Motivation war. Wie gesagt, nicht alle Entscheidungen sind so. Wir haben ein kleines Zeitfenster, also grundsätzlich die Möglichkeit, gegen die eigene Entscheidung ein Veto einzulegen (ein paar Millisekunden zwischen Entscheidung und Ausführung). Und bei komplexen Entscheidungen wie dem Wechsel des Arbeitgebers bleibt uns viel Zeit, das Für und Wider abzuwägen, wenngleich wir meist im Bauch schon die Entscheidung getroffen haben.

Im Alltag mag das so aussehen: Man sieht Code. Er funktioniert. Und trotzdem zuckt es. Diese eine Methode ist zu lang, und diese Klasse macht zu viel auf einmal. Passt nicht. Fühlt sich nicht gut an. Rational weiß man: Lass es, das Ticket ist eigentlich erledigt. Aber trotzdem refaktoriert man, weil dieses Gefühl zu stark ist. Der Verstand liefert die Erklärung dann hinterher: „Wartbarkeit“, „Clean Code“, „technische Schulden“. Alles richtig. Aber nicht der Auslöser. Nur die Erklärung.

Wir tun gerne so, als würden Architekturentscheidungen sauber hergeleitet, als würden wir Optionen vergleichen, Vor- und Nachteile abwägen, um dann die Entscheidung zu treffen. In Wirklichkeit läuft es oft anders. Man mag ein Pattern. Man vertraut einem Framework. Man hat gute Erfahrungen gemacht – oder schlechte. Die Entscheidung ist meist emotional. Das ist kein Vorwurf. So etwas ist menschlich.

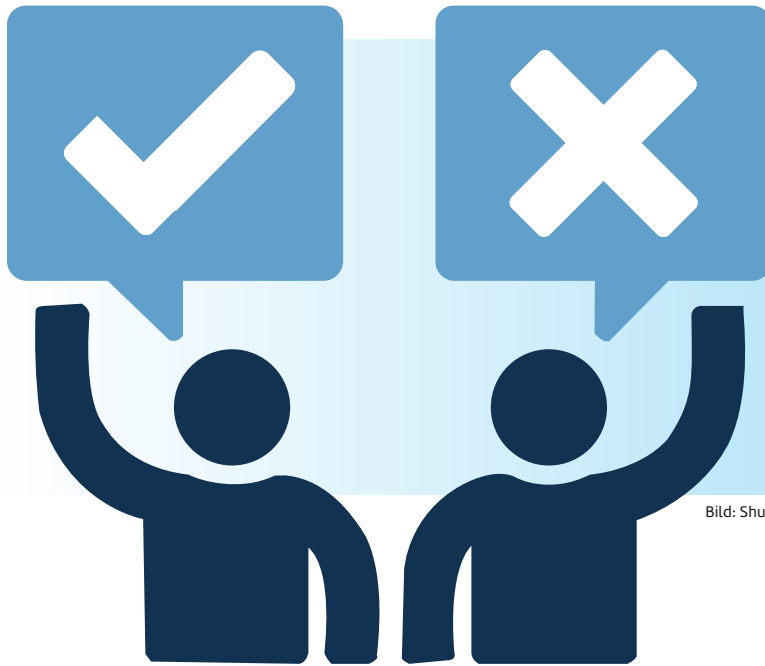


Bild: Shutterstock / Creative Sonda

Oder diese ewigen Diskussionen um Tests. Ja, man nimmt sich vor, das jetzt anzugehen – aber dann passiert wieder nichts. Der Verstand sagt: Ergibt Sinn. Der Bauch: Nicht jetzt.

Viele Entscheidung sind längst getroffen, bevor man diskutiert. Aus einem Sicherheitsbedürfnis, aus Zeitdruck, aus Angst vor Komplexität oder schlicht aus Bequemlichkeit. Der Verstand diskutiert. Die Intuition blockiert.

Was bedeutet das dann für Entwicklerteams?

1. Codequalität entsteht nicht durch Regeln allein, sondern durch ein gemeinsames Gefühl dafür, was akzeptabel ist. Durch Gewohnheit. Durch Akzeptanz. Wenn niemand schlechten Code kommentiert, ist er plötzlich normal.
2. Reviews wirken nur, wenn sie sicher sind. Ein Review, das beschämt, erzeugt defensiven Code. Ein Review, das Sicherheit bietet, erzeugt bessere Entscheidungen. Dazu braucht es eine Fehlerkultur, die das Zugeben von Fehlern wertschätzt – als Maßnahme zu mehr Qualität. Entwickler entscheiden intuitiv, ob sie etwas ansprechen oder lieber verstecken. Psychologische Sicherheit ist kein Wohlfühlthema. Sie ist eine Entscheidungsarchitektur.
3. Technische Schulden sind kein rationales Abwägen. Niemand entscheidet rational, jetzt Schulden zu machen. Man entscheidet intuitiv für kurzfristige Erleichterung. Und die hat Gründe, die man ändern kann und sollte.

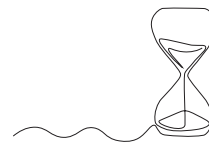
Führung in der Entwicklung heißt: Rahmen bauen, damit der Bauch besser entscheiden kann, nicht Argumente sammeln, um zu überzeugen. Wenn Entscheidungen intuitiv entstehen, dann ist die wichtigste Frage nicht: Wie erkläre ich es besser? Sondern: Wie fühlt sich die richtige Entscheidung im Alltag an? Ist Refactoring erlaubt? Darf man Tickets infrage stellen? Darf man sagen: Das schaffe ich nicht sauber?

Das entscheidet darüber, wie Code entsteht. Wir schreiben keinen Code, den wir für richtig halten. Wir schreiben Code, der sich im Moment richtig anfühlt. Der Rest ist Erklärung. Und der freie Wille? Vielleicht besteht unsere Freiheit nicht

darin, anders zu entscheiden. Sondern darin, die Bedingungen zu verändern, unter denen wir entscheiden. Eine offenere Kultur. Klarere Leitplanken. Weniger Angst. Mehr Zeit an den richtigen Stellen. Das ist nicht heroisch, aber wirksam.

In diesem Sinne: Happy Coding! ■

- [1] Benjamin Libet et al., *Time of conscious intention to act in relation to onset of cerebral activity*, *Brain*, 106 (3) 1983, Seite 623–642, <https://doi.org/10.1093/brain/106.3.623>
- [2] Benjamin Libet, *Do we have free will?*, *Journal of Consciousness Studies*, 6 (8–9) 1999, Seite 47–57, <https://spot.colorado.edu/~tooley/Benjamin%20Libet.pdf>
- [3] Jonathan Haidt, *The emotional dog and its rational tail: A social intuitionist approach to moral judgment*, *Psychological Review*, 108 (4) 2001, Seite 814–834, <https://doi.org/10.1037/0033-295X.108.4.814>
- [4] Jonathan Haidt, *The Righteous Mind, Why Good People Are Divided by Politics and Religion*, Penguin Books, 2013, ISBN 978-0-14-103916-9
- [5] Daniel Kahneman, *A perspective on judgment and choice: Mapping bounded rationality*, *American Psychologist*, 58 (9) 2003, Seite 697–720, <https://doi.org/10.1037/0003-066X.58.9.697>



Dipl.-Theol. Bernhard Pichler

war 20 Jahre Geschäftsführer der Softwarefirma informare, bevor er 2020 zum Leiter Entwicklung und Produktmanagement der DPS BS ernannt wurde. Die DPS BS ist Deutschlands größter Softwarepartner für betriebswirtschaftliche Software für den Mittelstand.

ARBEITSMARKT

TRENDS UND JOBS FÜR ENTWICKLER

Amadeus Fire

Gehälter in der IT

Für den IT-Gehaltsreport 2026, den Amadeus Fire in Kooperation mit Gehaltsreporter erstellt hat, wurden die Gehälter von über 9000 IT-Fachkräften ausgewertet – von Berufseinsteigern bis zu Führungskräften, von der Anwendungsadministration bis zur Softwareentwicklung. Ermittelt wurden die Daten anhand von Online-Gehaltschecks. Sie ergeben daher ein gutes Bild der Gehaltsrealität in Deutschlands IT-Landschaft. In **Bild 1** finden Sie einige der Gehaltsauswertungen für Softwareentwickler.

Die pro Berufsbild ermittelten Gehaltsdaten wurden in vier Quartile unterteilt. Wiedergegeben wurden das erste Quartil (Q1), der Medianwert (Q2) und das dritte Quartil (Q3). Der Wert für Q1 besagt, dass 25 Prozent aller beobachteten Gehälter unterhalb dieses Werts liegen. Q1 markiert also den Übergang vom unteren Viertel zum mitt-

leren Bereich. Der Medianwert (Q2) besagt, dass die Hälfte der Daten darunter und die andere Hälfte darüber liegt. Q3 markiert den Übergang vom mittleren Bereich zum oberen Viertel.

Beeinflusst wird die Höhe der einzelnen Gehälter vor allem durch die Berufserfahrung, aber auch durch die Firmengröße (Großunternehmen zahlen meist mehr) und die Region des Arbeitsplatzes. Besonders viel gezahlt wird in Regionen mit hohen Lebenshaltungskosten wie München, Frankfurt, Stuttgart und Hamburg. Außerdem werden Positionen mit Personalverantwortung meist besser bezahlt. In **Bild 2** sehen Sie den Einfluss der Berufserfahrung bei NET-/C#-Entwicklern.

Bitkom

Frauen in IT-Berufen

Der Softwareentwickler, der Programmierer oder der Cloud-Engineer – das Klischee von der Männerdomäne IT hält sich hartnäckig: Auch 2026 sind

noch 43 Prozent der Unternehmen der Ansicht, Männer seien für IT- und Digitalberufe besser geeignet. In fast jedem zweiten Unternehmen herrscht die Meinung vor, IT- und Digitalberufe würden Frauen abschrecken.

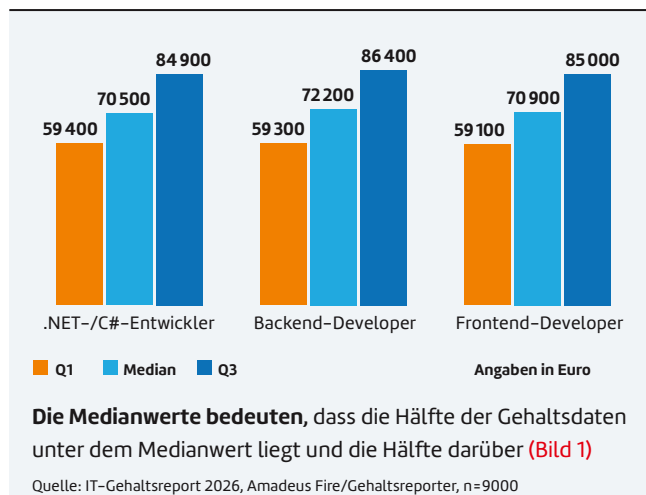
Das sind Ergebnisse einer repräsentativen Befragung von mehr als 600 Unternehmen aller Branchen im Auftrag des Bitkom.

In der beruflichen Praxis bleiben Frauen weiterhin unterrepräsentiert. In 58 Prozent der deutschen Unternehmen sind weniger als die Hälfte der Beschäftigten weiblich. Ein Viertel der Unternehmen berichtet von einem annähernd ausgeglichenen Verhältnis von Frauen und Männern. 13 Prozent beschäftigen mehr Frauen als Männer. Deutlich größer ist das Ungleichgewicht in IT- und Digitalfachbereichen. Bei diesen Tätigkeiten beschäftigt keines der repräsentativ befragten Unternehmen mehr Frauen als Männer. In 89 Prozent der Unternehmen sind weniger als

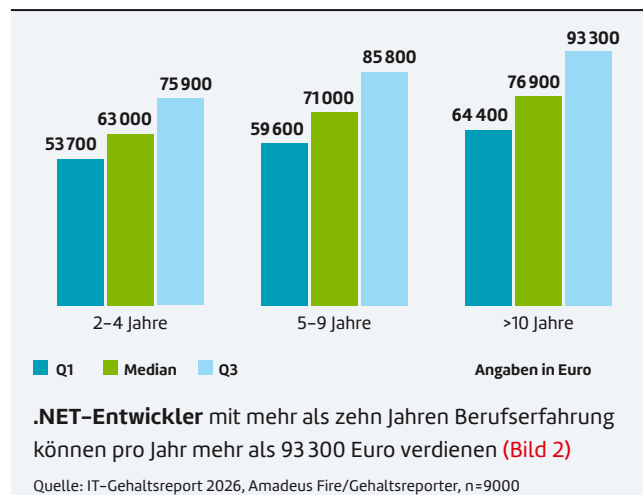
die Hälfte der Stellen in IT- und Digitalfachbereichen mit Frauen besetzt, und lediglich 9 Prozent berichten von einem annähernd ausgeglichenen Geschlechterverhältnis (**Bild 3**).

Aus Sicht der Wirtschaft sind die Gründe für den niedrigen Frauenanteil in IT- und Digitalberufen vielfältig. Die Hälfte der Unternehmen sieht Hürden beim Wiedereinstieg, etwa fehlende Weiterbildungen während der Elternzeit, als zentralen Faktor. 48 Prozent nennen eine unzureichende Sensibilisierung von Führungskräften mit Personalverantwortung. Zudem spielen Stereotype innerhalb der Unternehmen eine Rolle: 47 Prozent sehen männlich geprägte Rollenbilder als Hemmnis. 39 Prozent sprechen von einer „gläsernen Decke“, also einer strukturellen Benachteiligung von Frauen bei gleicher Qualifikation und Leistung. 35 Prozent geben an, Frauenförderung werde von Teilen der männlichen Belegschaft als ungerecht empfunden (**Bild 4**).

● Gehälter für Softwareentwickler



● NET-/C#-Entwickler: Einfluss der Berufserfahrung



Zahl des Monats

In einer Umfrage von YouGov sagten **34 Prozent** der Befragten in Deutschland, Angst zu haben, den eigenen Arbeitsplatz an eine Künstliche Intelligenz (KI) zu verlieren. **62 Prozent** gaben an, diese Sorge nicht zu teilen. Weitere fünf Prozent trauten sich in dieser Frage keine Einschätzung zu.

Quelle: YouGov-Umfrage im Auftrag der Deutschen Presse-Agentur

Über die Hälfte der Unternehmen sieht Hürden beim Quereinstieg, 48 Prozent nennen Defizite bei der Betreuungsinfrastruktur und 40 Prozent eine klischeebehaftete Berufsorientierung als Gründe für den geringen Frauenanteil.

IU Internationale Hochschule

Lieber weniger als mehr

Ausweitung der Arbeitszeit auf 48 Stunden, Abschaffung von Teilzeit ohne Betreuungspflichten: Arbeitszeitmodelle stehen aktuell im Fokus der politischen Debatte. Laut der neuen repräsentativen Studie der IU Internationalen Hochschule „Arbeitszeitmodelle der Zukunft: Mehrarbeit versus Lebensqualität?“ geben 73,5 Prozent der Befragten an, dass sich eine 48-Stunden-Woche „eher negativ“ oder „sehr negativ“ auf ihr Leben auswirken würde.

„Die zentrale Frage ist nicht, ob wir mehr oder weniger arbeiten, sondern wie wir Arbeit zu-

kunfts-fähig gestalten können. Längere Arbeitszeiten führen nachweislich zu mehr Fehlern – mehr Stunden bedeuten nicht automatisch mehr Produktivität. Die Ergebnisse zeigen deutlich, dass eine Ausweitung der Arbeitszeit abgelehnt wird. Für die Zukunftsfähigkeit sind daher eher effizientere Prozesse, eine moderne Arbeitskultur und flexible Modelle entscheidend, die sowohl wirtschaftliche Anforderungen erfüllen als auch Beruf, Privatleben und die physische sowie psychische Gesundheit berücksichtigen“, ordnet Professor Dr. Malte Martensen, Professor für Allgemeine BWL mit Schwerpunkt Personalmanagement und Organisation an der IU Internationalen Hochschule, die Studienergebnisse ein. Auf die Frage, welches Arbeitszeitmodell am besten zur eigenen Lebenssituation passt, nennen knapp 45 Prozent das klassische Vollzeitmodell mit 35 bis 40 Wochenstunden. Mit rund einem Drittel folgt bereits an zweiter

Stelle die 4-Tage-Woche – ein Modell mit 32 Stunden an vier Tagen bei vollem Gehalt und Vollzeit-Anforderungen an das Arbeitsergebnis. Eine klassische Teilzeit-Anstellung mit weniger als 35 Stunden pro Woche bevorzugt knapp ein Fünftel der Befragten (18,5 Prozent).

Forward College

Open Bachelor

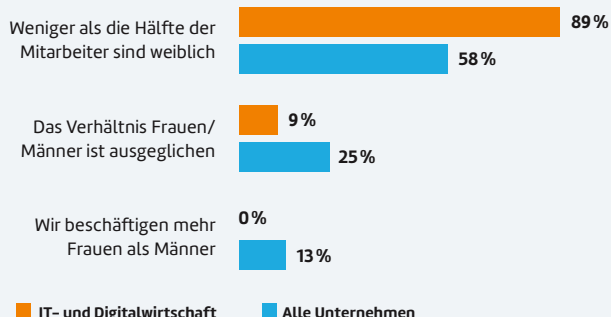
Mit dem Open Bachelor hat das paneuropäische Forward College (<https://forward-college.eu>) einen in Frankreich staatlich anerkannten und in der gesamten EU gültigen Studiengang eingeführt. Die Struktur ist interdisziplinär angelegt, um junge Menschen auf eine Arbeitswelt vorzubereiten, in der technologische und gesellschaftliche Veränderungen parallel verlaufen. Im ersten Studienjahr erkunden die Studierenden mehrere Fachbereiche – darunter Wirtschaft, Politik & Internationale Beziehungen, Psychologie, Management sowie Data Science

und KI – bevor sie sich im weiteren Verlauf spezialisieren. Das pädagogische Modell knüpft an das internationale Profil des College an: Drei Jahre Studium an drei verschiedenen Standorten – Lissabon, Paris und Berlin – kombiniert mit kleinen Lerngruppen, persönlichem Coaching, jährlichen Feedbacks und strukturiertem Leadership-Programm.

Eine besondere Rolle spielt der souveräne Umgang mit künstlicher Intelligenz. Die Studierenden erlernen den verantwortungsvollen Einsatz generativer KI und entwickeln ab dem zweiten Jahr eigene KI-Agents über den instituts-eigenen digitalen „AI Hub“.

Der Abschluss richtet sich an junge Menschen, die eine fundierte fachliche Vertiefung mit internationaler Erfahrung, analytischer Stärke und menschlicher Kompetenz verbinden möchten – Fähigkeiten, die in Wissenschaft, Wirtschaft und öffentlichem Sektor zunehmend gefragt sind.

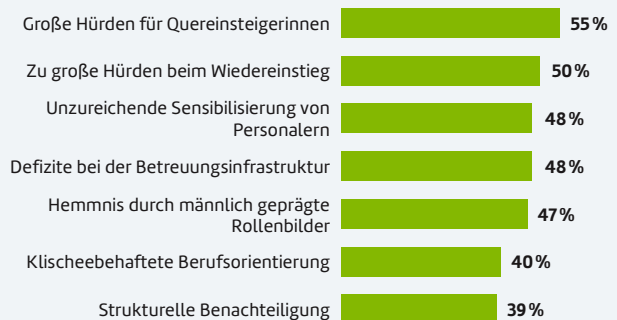
● Frauenanteil in IT und Wirtschaft



Ein ausgeglichenes Verhältnis von Männern und Frauen gibt es lediglich in neun Prozent der IT- und Digitalunternehmen. In der Gesamtwirtschaft sind es 25 Prozent (Bild 3)

Quelle: Bitkom Research, n=603, repräsentativ

● Gründe für den Frauenmangel in der IT



Als Gründe für den Frauenmangel nennen die befragten Unternehmen unter anderem die unzureichende Sensibilisierung von Führungskräften mit Personalverantwortung (Bild 4)

Quelle: Bitkom Research, n=603, repräsentativ



Alle Fotos: Sofija De Mitri | Patrizio De Mitri | Event Wave

DWX 2026

Wissen trifft Community

Wohlfühlzeit mit intensivem Lernen für Entwickler:innen auf der DWX in Mannheim.

Das Schlimmste an der DWX (Developer World Experience) sei, dass sie nur einmal im Jahr stattfinden würde, sagt man. Perfekte Fortbildung gepaart mit intensivem Community-Feeling finde sich nur hier, sagt man.

Fakt ist, DWX heißt vier Tage intensive Wissensvermittlung gepaart mit Wohlfühlzeit für Developer. Fortbildung in Zeiten von KI ist wichtiger denn je: Grundlagen in allen Entwicklungsbelangen, gepaart mit dem Know-how zum Einsatz von KI-Tools – damit sicherst du dir deinen Job. Umgeben von über 2000 gleichgesinnten Entwicklerinnen und Entwicklern bist du endlich unter Menschen, die verstehen, wovon du redest und welche Probleme du täglich zu meistern hast.

Dann ist die DWX vorbei und du musst wieder ein Jahr auf die nächste DWX warten. Das ist schlimm. Schlimmer trifft es nur den, der gar nicht erst zur DWX kommen konnte. Deshalb der Appell direkt am Anfang: Hol dir gleich dein Ticket.

● DWX 2026 im Überblick

- 29. Juni bis 2. Juli 2026
- m:con Congress Center Rosengarten, Mannheim
- 27 Tracks, 160 Sessions: Das volle Programm findest du unter developer-world.de/dwx/dwx-programm
- Hol dir dein Ticket unter developer-world.de/dwx/dwx-tickets

Das Wichtigste zuerst: Die Menschen

Technisch gesehen ist die DWX eine Konferenz. Praktisch gesehen ist sie ein Familientreffen der Entwickler-Community – mit besseren Talks als auf den meisten Familienfeiern und deutlich mehr Diskussionen über Softwarearchitektur.

Das, was die DWX wirklich ausmacht, lässt sich nicht nur im Programm abbilden. Es passiert auch zwischen den Sessions. Es passiert, wenn du den Speaker nach dem Vortrag ansprichst, weil du eine Folgefrage hast – und er nimmt sich wirklich Zeit. Nicht nur zwei Minuten. Weil das hier keine Schnellabfertigung ist, sondern ein echtes Gespräch unter Leuten, die das gleiche Handwerk lieben.

Du sprichst plötzlich mit jemandem über das Architekturproblem, das dich seit Monaten beschäftigt – und nach zehn Minuten hast du drei neue Ideen und eine Visitenkarte. Du sitzt beim Mittagessen neben jemandem, dessen GitHub-Repo du schon seit Monaten auswendig kennst, und fragst einfach: „Hey, bist du das wirklich?“

Das ist kein Zufall. Das ist System. Das Konferenzteam baut diese Möglichkeiten aktiv ein: Kaffeepausen mit genug Zeit für die Deep Talks, eine Expo, die zum Verweilen einlädt, Abende, bei denen niemand hetzen muss und Spaß und Spiel nicht zu kurz kommen. Keine Ellenbogen-Netzwerkerei. Kein Visitenkarten-Tausch-Theater. Einfach Leute, die ihr Handwerk lieben, und Zeit, darüber zu reden.

Die Basis für die Community-Interaktion ist in erster Linie die wunderbare Möglichkeit, sich fortzubilden. 160 Sessions in

27 Tracks. Das klingt nach viel – weil es auch viel ist. Und hinter jedem dieser Tracks steckt ein Track Chair, der oder die aus über 1500 Einreichungen die absoluten Perlen herausgefiltert hat. Ehrenamtlich. Weil sie es gerne machen. Ein riesiges Dankeschön an alle, die sich diese Aufgabe aufgehängt haben.

Die Köpfe dahinter – klingende Namen, echtes Know-how

Eine Konferenz steht und fällt mit ihren Sprecherinnen und Sprechern. Als Wissensquelle sorgen sie für die Fortbildung. Alle hier aufzuführen würde ein bisschen zu weit führen. Aber Expertinnen und Experten wie Jörg Neumann, Maria Korneeva, David Tielke, Christian Giesswein, Manfred Steyer, Dr. Carola Lilienthal, Torsten Strauß oder Jonah Andersson sprechen für die hohe Qualität der Konferenz.

Ein paar Beispiele seien aber doch genannt. Olaf Lischke bringt dir bei, wie du die Stolperfallen von EF Core vermeidest. Eberhard Wolff fragt, ob KI wirklich ein Werkzeug für Architektur ist. Spoiler: Die Antwort wird differenzierter ausfallen als erwartet. Dr. Gernot Starke nimmt sich kognitive Verzerrungen im Software Engineering vor: Warum treffen gute Architektinnen und Architekten manchmal ganz schön schlechte Entscheidungen? Weil sie Menschen sind. Ramona Schwing zeigt gleich zweimal, wie Sicherheit und Testing zusammengedacht werden müssen. Thomas Claudius Huber nimmt C# 14 auseinander – Pflichtprogramm für alle, die im



Pausen für Leib und Leidenschaft: Vorzügliche Verpflegung und reger Austausch mit Gleichgesinnten zwischen den Sessions.

.NET-Ökosystem zu Hause sind. Martin Försch und Thomas Endres – bekannt für spektakuläre Talks – nehmen sich diesmal KI in Bild-, Audio- und Videoproduktionen vor. Golo Roden erklärt, warum „die 90er angerufen haben und ihr Backend zurückwollen“. Und Shawn Wildermuth gibt praktische KI-Tipps für Developer.

Die Sprecher:innen kommen aus dem deutschsprachigen Raum, den Niederlanden, Belgien, Skandinavien, den USA und weit darüber hinaus. Die Mischung aus erfahrenen Veteranen und aufstrebenden neuen Stimmen ist jedes Jahr ►

● 27 Tracks, 27-mal die Welt der Softwareentwicklung

Künstliche Intelligenz:

KI ist der Elefant im Raum – und die DWX schaut ihm direkt in die Augen, gleich mit vier Tracks. AI Technologies (Gregor Biswanger) und Agentic AI (Jörg Neumann) laufen am Dienstag, AI Developer Tools (Neno Loje) am Mittwoch und AI Development Strategies (Marco Richardson) am Donnerstag.

Daten & Qualität:

Daten sind das neue Öl – aber nur, wenn sie gut gepflegt sind. Data Analysis (Constantin Klein), Data Quality (Stephan Rossbach) und Databases (Thorsten Kansy) zeigen, wie du aus dem Rohstoff Daten echten Mehrwert machst.

Softwarequalität & -sicherheit:

Gute Software entsteht nicht zufällig – sie wird gebaut. Software Architecture & Design (Dr. Carola Lilienthal), Software Testing (Ramona Schwing), Software Quality (David Tielke) und Software Security (Christian Wenz) liefern das Handwerkszeug dafür.

Menschen & Teams:

Code schreiben können viele. Aber Teams führen, sich selbst reflektieren, wachsen – das ist eine andere Liga. Personal Development (Tom Wendel) und Team Leadership (Stefan Mintert) adressieren genau die Fähigkeiten, die im Alltag oft entscheiden, ob ein Projekt wirklich fliegt.

.NET & Frameworks:

Einer der Kerne des DWX-Universums: C# & .NET (Christian Giesswein), .NET Frameworks (Florian Lenz), .NET Frontend

(Thomas Claudius Huber) und .NET Backend (Dennis Doomen). Von C# 14 über .NET MAUI 10 bis zu .NET Aspire in der Praxis – wer im Microsoft-Ökosystem entwickelt, wird hier rundum satt.

Cloud:

Die Cloud ist nicht mehr die Zukunft – sie ist dein Alltag. Cloud Platforms (Stefan Rapp) und Cloud Strategies (Dr. Damir Dobric) vermitteln praxisnah, wie du das Beste aus Azure, Kubernetes und Co. herausholst: von Landing Zones über serverlose Architekturen bis zu nachhaltiger Cloud-Entwicklung.

DevOps & Automatisierung:

Ohne funktionierende Pipelines ist der beste Code wertlos. DevOps Foundations & Strategies (Michael Kaufmann), CI/CD (Marc Müller) und Git & GitHub (Marko Beelmann) zeigen, wie du Deployments, Workflows und Source Control auf Profi-Niveau bringst.

Web Development:

Web Frontend (Maria Korneeva), Web Backend (Nils Hartmann) und Web Languages (Peter Kröner) decken alles ab: von modernem CSS und TypeScript über GraphQL und REST bis zu High-Performance-Web-Apps.

Methoden & Prozesse:

Agilität: Totgesagte leben länger. Agile (Ulf Mewe) und Product Ownership & Management (Ina Einemann) zeigen, was moderne Methoden wirklich leisten – und wo sie an ihre Grenzen stoßen.

eine der größten Stärken der DWX. Manchmal sind es gerade die unbekannten Namen, die die stärksten Sessions abliefern.

Workshops: Für alle, die es wirklich wissen wollen

Sessions sind großartig. Eine Stunde Input, Aha-Momente inklusive. Wer aber wirklich in ein Thema reinspringen, es durchkauen und selbst ausprobieren will, ist richtig bei den halb- und ganztägigen Workshops am Montag, den 29. Juni. Laptop mitbringen, loslegen, debuggen. Am Ende hast du nicht nur Folien, sondern funktionierenden Code auf deinem Rechner.

Ganztätig gibt es unter anderem: Test-Driven Development mit und ohne KI (David Tanzer), AI for Developer Productivity (Daniel Sogl), moderne KI-Backends mit Tools, Workflows und Agenten (Sebastian Springer), Custom MCP Server auf Azure für .NET-Entwickler (Jonah Andersson), eine Master Class zu Custom Copilots (Dr. Damir Dobric), moderne Softwarearchitekturen (David Tielke), SQL Server Performance Optimization (Torsten Strauß) und .NET Aspire für verteilte Web-Anwendungen (Johan Smarius).

Halbtags geht es nicht weniger zur Sache: Angular-Apps mit lokaler, offlinefähiger generativer KI (Christian Liebel), UX-Mindset für Entwickler mit dem gewissen Augenzwinkern (Vicky Pirker), EF Core Performance Deep Dive (Christian Giesswein) und Web Application Security Bootcamp auf Basis der OWASP Top Ten (Christian Wenz). Nachmittags dann Gregor Biswanger mit „Bye Vibe, Hello Vise Coding“ und GitHub Copilot, Blazor ohne Magie (nochmal Christian Giesswein), Cloud Architecture Kata (Tobias Voß) und Automation mit GitHub Actions & Azure DevOps (Leo Visser).

Abendevents: Wenn der Code offline geht

Die Abende der DWX gehören der Community – und die weiß, wie man sie nutzt. Hier stehen Night Sessions und die Casino Night auf dem Programm.

Die Night Sessions bieten Softwareentwicklung, aber auf unterhaltsame Art und Weise: spezielle Projekte, Mitmach-Aktionen, Spiel und Spaß. Wer 2025 dabei war, erinnert sich sicher daran, wie Tom Wendel das komplette Auditorium in ein Smartphone-Orchester verwandelt hat. Plötzlich erfüllte die Filmmusik aus „Fluch der Karibik“ den Saal. Auch wenn es bis dahin mehrerer Anläufe bedurfte: Das Ergebnis klang erstaunlich gut.

Was ist denn wohl 2026 geplant? Wir verraten nur so viel: Super Mario trifft auf TypeScript: Manfred Steyer einführt mit neuester Technik in die Nostalgie eines alten Spiels.

Und dann ist die Night Session vorbei und es startet die legendäre Casino Night. Echte Chips, professionelle Dealer, Entwicklerinnen und Entwickler mit ihren besten Pokerfaces – eine Kombination, die funktioniert.

Die Stimmung ist ausgelassen, die Gespräche intensiv, und die Vernetzung passiert ganz nebenbei – weil man eben zusammen am Roulettetisch steht.

Die Fachmesse: Mehr als bunte Stände

Die Aussteller sind seit 2013 fester Bestandteil des DWX-Erlebnisses. Während der gesamten Konferenz haben die Partner die Möglichkeit, ihre Lösungen in ungezwungener Atmo-



Ausklang: Nach einem Tag mit intensivem Lernen gibt es für Teilnehmerinnen und Teilnehmer Spiel und Spaß bei der Casino Night.

sphäre zu zeigen – ohne Zeitdruck, ohne Terminstress. Echte Gespräche zwischen Menschen, die wissen, was sie brauchen, und Unternehmen, die wissen, was sie anbieten.

Für 2026 haben schon die folgenden Partnerunternehmen zugesagt: fecher, Xebia, INNOQ, combit, exxeta, oose, sepp.med, Textcontrol. Wer selbst als Partner dabei sein möchte: Christoph Teubert hilft weiter – sales@developer-world.de.

Wer mehreren Kolleginnen und Kollegen die Fortbildung öffnen will, kann sich wegen eines Gruppenrabatts gerne an Susanne Herl wenden (susanne.herl@ebnermedia.de).

Essen & Trinken: Bits und Bytes machen nicht satt

Kurze Anmerkung für alle, die schon mal Konferenz-Catering erlebt haben, das nach Kantinen-Mittwoch schmeckt: Hier ist das anders. Der Rosengarten macht das ordentlich. Das Mittagbuffet ist kein Standardfutter, die Kaffeepausen sind echte Pausen – lang genug, um wirklich zu essen, zu trinken und dabei ein Gespräch anzufangen, das mehr als zwei Sätze lang ist.

Und nach der Konferenz liegt Mannheims Innenstadt direkt vor der Tür. Die Planken, die Fressgasse, zahlreiche Restaurants und Bars in Laufweite – wer nach einem langen Konferenztag noch Energie hat, wird sie hier loswerden.

Neun Sessions gleichzeitig – und trotzdem nichts verpassen

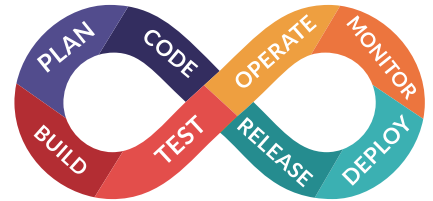
Das große Luxusproblem der DWX: Neun Sessions laufen gleichzeitig, aber du kannst nur eine besuchen. Acht anderen gibst du einen Korb. Das klingt frustrierend – ist aber lösbar. Je nach Ticketart bekommst du Zugang zu den Aufzeichnungen aller Sessions für sechs beziehungsweise zwölf Monate. Dazu alle Slides und Materialien, die die Sprecherinnen und Sprecher zur Verfügung stellen. Du kaufst also nicht nur ein Ticket. Du kaufst eine Wissensbibliothek.

Also: Was hält dich noch?

Du willst endlich wissen, welche KI-Tools dir am besten helfen? Check. Du willst deine Architektur-Skills kritisch hinterfragen? Check. Du willst wissen, wie andere Teams ihre CI/CD-Pipeline aufgebaut haben? Check. Du willst einfach mal wieder mit Menschen reden, die verstehen, warum Tabs versus Spaces keine triviale Frage ist? Triple Check. Komm zur DWX. Wir sehen uns in Monnem. Tilman Börner ■

KI gehört in deinen Workflow

Die Infinite AI Conference zeigt dir, wie du KI in jedem Schritt des Software-Lifecycle richtig einsetzt.



Wie nutzt du KI in deinem Entwickleralltag? Vermutlich so wie die meisten: ein bisschen Copilot hier, ein ChatGPT-Prompt da, und zwischendurch noch ein bisschen Google und Stackoverflow. Das ist gut und schön, aber als „Integration der KI in den Workflow“ kann man das nicht bezeichnen. Denn der Unterschied zwischen „KI benutzen“ und „KI orchestrieren“ ist riesig. Wer KI isoliert einsetzt, kratzt nur an der Oberfläche dessen, was möglich ist. Wer sie dagegen systematisch in den gesamten Software-Lifecycle einbettet – von der ersten User Story bis zum Monitoring –, hat einen echten Wettbewerbsvorteil.

Genau das ist der Ansatz der Infinite AI Conference, die am **30. Juni und 1. Juli 2026** im **m:con Congress Center Rosengarten in Mannheim** stattfindet. Die Konferenz hat sich ein klares Ziel gesetzt: KI als integralen Bestandteil des gesamten Entwicklungsprozesses zu etablieren. Handwerkszeug für deinen Entwickleralltag.

Den Lifecycle konsequent mit KI durchdenken

Das Programm ist deshalb bewusst entlang der Phasen eines Softwareprojekts aufgebaut. Von Requirements über Architektur und Entwicklung bis hin zu Tests, Deployment und Betrieb – jede Phase wird beleuchtet, jede Disziplin bekommt ihren Platz. Zwei Tage, nach denen du weißt, wie du das Maximale aus der KI holst.

Tag eins:

Michael Schemel eröffnet die Infinite AI Conference mit einer Keynote, die gleich zur Sache kommt: „Software im Zeitraffer. KI, die Beschleunigung von Delivery – und die Konsequenzen für den SDLC.“ Wer bisher dachte, KI mache Entwicklung nur ein bisschen schneller, wird hier wohl umdenken müssen. Michael taucht außerdem in einer weiteren Session tiefer in die Materie ein und beleuchtet, wie sich Softwarearchitekturen, Patterns und Tool Chains durch KI rasant neu formieren.

Martin Buskes Session deckt den ersten Schritt des Lifecycle ab: „Prompts statt PRD: Wie AI die Requirements-Praxis umkrempelt.“ Sind klassische Product Requirements Documents (PRD) bald Geschichte? Und wenn ja – was kommt danach?

Spannend geht es am Nachmittag des ersten Tages weiter: *Olena Borzenko* und *Marie-Theresa Borsig* fragen in ihrer gemeinsamen Session „You are here: How software development is changing in an AI-native future“ – eine Standortbestimmung, die vermutlich für einige unbequeme Wahrheiten sorgen wird. Den Abschluss bildet *Patrick Schnell*, der sich der Frage widmet, ob wir uns dem letzten echten Coding-

Jahr nähern – mit einer Session über den Einfluss von Claude Code auf die Softwareentwicklung.

Tag zwei:

Die Keynote von *Dehla Sokenou* und *Harald Störrle* stellt die unbequeme These auf, dass sich in Sachen Softwarequalität alles ändern muss – damit am Ende alles gleich bleibt. Klingt paradox, ergibt aber Sinn, wenn man versteht, wie stark KI die klassischen Qualitätssicherungsprozesse auf den Kopf stellt.

Alexander Rampp untersucht in seiner Session, ob KI beim Testen wirklich Gamechanger ist. Spoiler: Es muss differenzierter betrachtet werden, als viele Social-Media-Posts behaupten. *Luca Ortiz* taucht in den Systemtest ein, bevor *Emmanuel Bartolesi* und *Yuriy Varshavskyy* das große Bild zeigen: Wie KI den gesamten Software Delivery Lifecycle neu schreibt. Den Abschluss machen *Jonathan David* und *Harald Binkle* mit einem Thema, das bei KI-Diskussionen meist zu kurz kommt: AIOps und die KI-gestützte Wartung von Softwaresystemen.

Spaß haben auf der Casino Night

Zwischen zwei intensiven Tagen des Lernens liegt noch die legendäre DWX Casino Night am Dienstagabend. Poker, Roulette, Drinks und Essen, gemeinsam mit der Developer-Community der DWX. Als Teilnehmerin oder Teilnehmer der Infinite AI Conference bist du automatisch dabei.

Für wen ist die IAIC?

Was die Infinite AI Conference aus dem Meer an AI-Events heraushebt, ist der konsequente Ansatz. Statt Tool-Parade oder „Die 100 besten Prompts für Entwickler“ bietet die IAIC einen strukturierten Durchgang durch den gesamten Entwicklungsprozess – mit dem klaren Versprechen: Wer hier rausgeht, weiß, wie man KI systematisch in allen Phasen der Softwareentwicklung einsetzt.

Die Infinite AI Conference richtet sich an alle, die Software bauen und Verantwortung dafür tragen. Das schließt Entwicklerinnen und Entwickler ebenso ein wie Architekt:innen, DevOps-Engineers, Tester:innen und Requirements Engineers. Und das Beste: Die Konferenz ist Stack-unabhängig. Ob du mit .NET, Java, Python oder JavaScript arbeitest – die Konzepte und Tools lassen sich auf jeden Stack übertragen.

Das Extra zum Schluss: Wer ohnehin zur DWX will, kann Kombitickets nutzen, etwa um die DWX-Workshops oder den dritten DWX-Konferenztag zu besuchen. Also hol dir dein Ticket!

www.developer-world.de/infinite-ai

Tilman Börner ■



LERNSTOFF

NEUE BÜCHER UND MEHR

KI-TRAINING

PyTorch

Die Python-Programmbibliothek PyTorch ist ein essenzielles Framework für maschinelles Lernen. Der Titel von Bert Gollnick liefert eine fundierte Einführung in die Verfahren von Deep Learning, die sich damit realisieren lassen. Zum Einstieg werden Grundbegriffe wie neuronale Netzwerke, Aktivierungsfunktionen und Tensoren behandelt. Die praktische Bewährung wird mit einem ersten Training an einem Modell ausprobiert, das mit Daten über den Zusammenhang von Schlaf- und Angststörungen arbeitet, von der Datenbereitstellung bis zum getrennten Speichern von Trainings- und Validierungsdaten.

Die folgenden Kapitel behandeln das Training von einfachen Modellen, den umfangreichen Bereich der Klassifizierung von Bildern mittels Convolutional Neural Networks (CNNs), den Stiltransfer zwischen Bildern und die Objekterkennung in Bildern. Eigene Themen bilden Empfehlungssysteme und der Bereich der Autoencoder, die beispielsweise zur Datenbereinigung eingesetzt werden. Die Vorhersage von



Bert Gollnick

PyTorch
KI-Modelle trainieren, tunen und einsetzen

Rheinwerk, 2026, 447 Seiten,
Deutsch, ISBN 978-3-367-11129-9,
39,90 Euro

Zielgruppe: Modellentwicklung

Voraussetzungen: Python-Kenntnisse

Leseprobe:

<https://tinyurl.com/GollnickPyTorch>

Zeitreihen ist Anwendungsgebiet von Recurrent Neural Networks (RNNs), entwickelt, um sequenzielle Daten zu verarbeiten. Das Kapitel über große Sprachmodelle geht ausführlich auf das Natural Language Processing (NLP) mit PyTorch ein. Es behandelt den Einsatz von Parametern, die unterschiedlichen Nachrichtentypen, die Rolle der Prompts, der Chains und der Transformer. Die verbleibenden Kapitel befassen sich mit Fine-Tuning, Validierung und Monitoring und abschließend mit dem Deployment. (vh) ■

CCD AKADEMIE

Clean Code und KI

Anfang 2023 wechselte das Thema Künstliche Intelligenz vom Nischendasein in die Mainstream-Medien. Inzwischen ist die KI als Werkzeug für die Softwareentwicklung so leistungsfähig, erklärt CCD-Chef Stefan Lieser, dass er selbst praktisch seinen gesamten Code von KI generieren lässt. Ist Clean Code somit am Ende? Nein, alles andere als das, sagt Lieser. Die Clean-Code-Prinzipien sind wichtiger denn je.

Profunde Kenntnisse als Entwickler sind nach wie vor zwingend. Doch es geht immer weniger um technisches Know-how, sondern darum, die KI-Tools in einem Entwicklungsprozess zielführend einzusetzen. Selbstverständlich ist auch technische Expertise erforderlich, um Security, Skalierbarkeit, UI/UX und andere Themen umsetzen zu können. Doch das kann man nun wenigen Spezialisten überlassen.

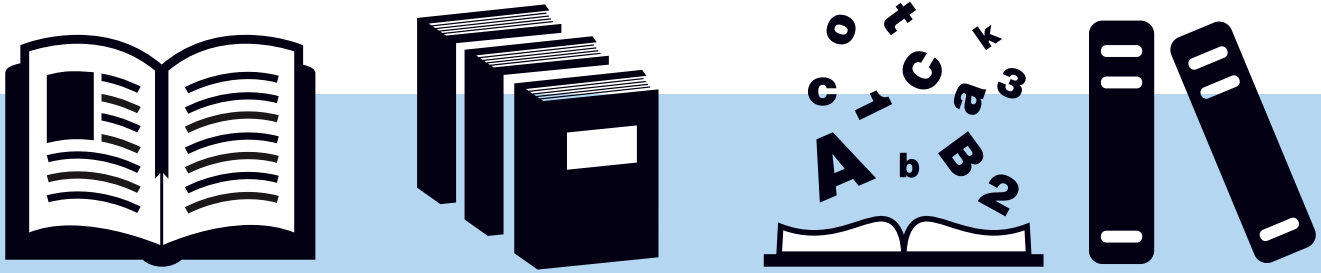
Für Entwickler gilt es vor allem, den von der KI generierten Code zu bewerten. Dafür sind fundierte Clean-Code-Kenntnisse erforderlich.

Für Stefan Lieser ist klar: „Jeder Softwareentwickler muss in der Lage sein, KI-Tools in seinen Prozess zu integrieren. Wer heute noch ohne diese Tools unterwegs ist, verliert sehr schnell den Anschluss. Die Produktivitätsgewinne sind enorm, nicht selten um den Faktor fünf bis zehn.“

Aus diesem Grund richtet die CCD Akademie [1] jetzt alle Trainings auf KI aus. Sie vermittelt weiterhin Prinzipien wie das Integration Operation Segregation Principle (IOSP). Im Fokus steht aber auch, wie die Tools optimal genutzt werden können, um produktiver zu werden, ohne dass die Werte Wandelbarkeit und Korrektheit in Gefahr geraten.

Derzeit setzt die Akademie auf Claude Code und Open-Code. Lieser: „Irgendwo muss man anfangen. Es ist im ersten Schritt wichtig, die KI in den Prozess zu integrieren und täglich damit zu arbeiten. Von dort aus ist es leicht, andere Tools zu verwenden.“ (bl) ■

[1] <https://ccd-akademie.de>

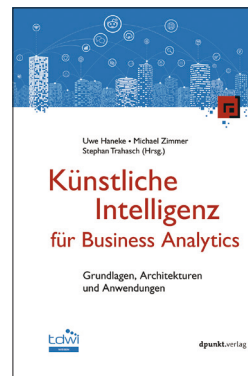


WIE ES GEHT

KI fürs Business

Viele Unternehmen fragen sich, wie sie KI am besten nutzen können. Die Herausgeber dieses Titels haben kompetente Stimmen dazu zusammengeführt. Der erste Teil liefert eine verständliche Einführung in maschinelles Lernen, Natural Language Processing, Computer Vision und KI-gestützte Robotik. Der zweite Teil geht auf die strategischen und operativen Anforderungen ein, auf die notwendigen Anpassungen von Geschäftsmodellen gegenüber den oft disruptiven Folgen des KI-Einsatzes und auf die Zukunft der Arbeit. Dabei geht es immer auch um den tatsächlichen Beitrag zur Wertschöpfung.

Im Bereich Business Analytics erlauben KI-Analysen auf der Basis von Mustererkennung in großen Datenmengen fundiertere Prognosen, Natural Language Processing (NLP) vereinfacht Datenabfragen, KI-gestützte Dashboards liefern intelligenter Handlungsempfehlungen. Die Voraussetzung ist aber eine entsprechende Aufbereitung der Daten, die in einem eigenen Kapitel behandelt wird.



Uwe Haneke, Michael Zimmer, Stephan Trahasch (Hrsg.)

Künstliche Intelligenz für Business Analytics. Grundlagen, Architekturen und Anwendungen
dpunkt, 2026, 342 Seiten, Deutsch, ISBN 978-3-98889-050-4, 69,90 Euro

Zielgruppe: KI-Entwicklung

Voraussetzungen: keine

Leseprobe (englisch):

<https://tinyurl.com/dpunktKI4BA>

Von vornherein stellt sich die Frage, was KI darf und was nicht. Der EU AI Act wird deshalb einer kritischen Analyse unterzogen. Zudem werden rechtliche und ethische Aspekte bei der Verwertung von Texten, Bildern und Filmmaterial ausführlich behandelt. Wie all dies in der Praxis aussieht und welche Hürden dabei zu meistern sind, betrachtet der dritte Teil anhand konkreter Praxisfälle unterschiedlicher Branchen – unter anderem der Energiewirtschaft, Anwendungen im Klimaschutz und in der Robotik. (vh) ■

● Kurz vorgestellt

Carsten Vogt

Von Java zu Kotlin

Einführung und praktischer Einstieg in den Java-Nachfolger Kotlin. Das Buch zeigt die Vorteile von Kotlin und illustriert Sprachkonzepte anhand von Beispielen. Zudem werden die relevanten Aspekte und Techniken diskutiert. Mit Code-Beispielen.

Hanser, 2026, 273 Seiten, Deutsch,

ISBN 978-3-446-48372-9, 39,99 Euro

Marcel Küppers

Informations- und Cybersicherheit CISO

Der Praxis-Leitfaden für CISOs und Security-Entscheider liefert Ansätze für den strategischen Aufbau moderner Sicherheitsprogramme, Handlungsanleitungen für CISOs bei Governance-, Risiko- und Compliance-Themen sowie einen Überblick der aktuellen Technologien, Frameworks und Best Practices.

mitp, 2026, 688 Seiten, Deutsch,

ISBN 978-3-7475-0679-0, 59,99 Euro

Thomas Roßner, Mario Winter, Andreas Spillner, Thomas Linz
Praxiswissen Softwaretest – Testmanagement

Das Standardwerk wurde abgestimmt auf den Lehrplan „Certified Tester – Advanced Level – Testmanagement“ in Version 3.0. Mit drei durchgehenden Fallbeispielen, Tipps und Exkursen. Die fünfte Auflage berücksichtigt sequenzielle, hybride und agile Ansätze.

O'Reilly, 5. Auflage 2026, 570 Seiten, Deutsch,

ISBN 978-3-86490-763-0, 44,90 Euro

Samuel Lolagar

OSINT

Für Open Source Intelligence steht das Kürzel OSINT, und es bezeichnet die Kunst, Informationen aus offenen Quellen zu gewinnen und auszuwerten. Der Leitfaden zeigt Methoden und Tools für umfassende Recherchen in Datenbanken, sozialen Netzwerken und anderen Quellen.

Rheinwerk, 2026, 448 Seiten, Deutsch,

ISBN 978-3-367-10374-4, 34,99 Euro

Highlights in der Ausgabe 3/26



Die nächste
Ausgabe
erscheint am
18.06.2026

Focus:

Softwareentwicklung in Zeiten
künstlicher Intelligenz

Weitere Themen:

- Azure Development
- SQL Server 2025 und Microsoft Azure
- Azure Bicep und native IaC
- Microsoft AI und Purview
- Cyber Resilience Act
- UX goes Development mit Figma
- ... und vieles mehr

Bild: Shutterstock / Iurii Motov

developer-world.de



Inserentenverzeichnis

Ebner Media Group

www.ebnermedia.de

2, 15, 17, 23, 55, 95, 127, 147, 148

Impressum

dotnetpro by DWX

MEDIENHAUS

Ebner Media Group GmbH & Co. KG
Olgastraße 121/1
89073 Ulm

info@ebnermedia.de
www.ebnermedia.de

GESCHÄFTSFÜHRUNG

Annel Ebner, Marco Parrillo

DE147041097
Sitz: Ulm, Amtsgericht Ulm, HRA 1900

Persönlich haftende Gesellschafterin der
Ebner Media Group GmbH & Co. KG
ist die Ebner Ulm MGV GmbH, Ulm.

Sitz: Ulm, Amtsgericht Ulm, HRB 576

OFFICE

Ebner Media Group GmbH & Co. KG
Office Köln
Helmholtzstraße 29 – 31
50825 Köln
Telefon: +49 731 88005-8000
www.dotnetpro.de

VERANTWORTUNG CONTENT

Chefredaktion

Fernando Schneider
fernando.schneider@ebnermedia.de

Tilman Börner (ab 1. April 2026)
tilman.boerner@ebnermedia.de

Externe Autor*innen

Benjamin Abt, Klaus Aschenbrenner,
Andreas Erben, Christian Giesswein,
Thomas Hafermalz, Christian Jacob, Thorsten Kansy,
Christian Kiefer, Dr. Veikko Krypczyk, Bernhard Lauer,
Florian Lenz, Stefan Lieser, Stefan Papp,
Bernhard Pichler, Tobias Richling, Patrick Schnell,
Dr. Holger Schwichtenberg, Frank Simon,
David Tielke, Uwe Vigenschow, Helmut Vonhoegen,
Christian Wenz

Art Direction

Maria-Luise Steinkühler
malu.steinkuehler@ebnermedia.de

Gestaltung

EMG DESIGN UNIT
DesignUnit@ebnermedia.de

Publisher

Marco Parrillo
marco.parrillo@ebnermedia.de

MEDIA

Media Sales extern

Christoph Teubert
sales@developer-world.de

Client Success (Disposition)

Anja Büttner
clientsuccess@ebnermedia.de

HERSTELLUNG, VERTRIEB & KUNDENSERVICE

Head of Commercial Operations, Production & Customer Experience

Thomas Heydn
thomas.heydn@ebnermedia.de

Kundenservice

Telefon: +49 731 88005-8205
kundenservice@ebnermedia.de

Bezugspreise

Die jeweils gültigen Bezugspreise für unsere
Print- und Digitalausgaben sowie für eventuelle
Abonnements entnehmen Sie bitte den Angaben auf
unserer Website oder den jeweiligen Bestellseiten.
Alle Preise verstehen sich inklusive der gesetzlichen
Mehrwertsteuer und zuzüglich ggf. anfallender
Versandkosten, sofern nicht anders angegeben.

Preisänderungen bleiben vorbehalten. Bei
bestehenden Abonnements gelten die bei
Vertragsschluss vereinbarten Konditionen, sofern
keine anderweitige Regelung getroffen wurde.

Widerrufsrecht

Sie haben das Recht, binnen vierzehn Tagen ohne
Angabe von Gründen diesen Vertrag zu widerrufen.

Die Widerrufsfrist beträgt vierzehn Tage ab dem Tag,
an dem Sie oder ein von Ihnen benannter Dritter,
der nicht Beförderer ist, die erste Ware in Besitz
genommen haben.

Um Ihr Widerrufsrecht auszuüben, genügt
eine eindeutige Erklärung, mit der Sie uns über
Ihren Entschluss, diesen Vertrag zu widerrufen,
informieren. Bitte richten Sie diese an:

Ebner Media Group GmbH & Co. KG
Office München
Brienner Straße 45a-d
80333 München
Telefon: +49 731 88005-8205
E-Mail: hilfe@ebnermedia.de

Sie können dafür z. B. einen Brief oder eine E-Mail
verwenden. Zur Wahrung der Widerrufsfrist reicht es
aus, dass Sie die Mitteilung über die Ausübung des
Widerrufsrechts vor Ablauf der Frist absenden.

Urheberrecht / Copyright

Alle Inhalte, Werke und Publikationen – ob online
oder im Print – der Ebner Media Group GmbH & Co. KG
unterliegen dem deutschen Urheberrecht.

Jegliche Vervielfältigung, Bearbeitung, Verbreitung,
öffentliche Zugänglichmachung oder sonstige
Nutzung außerhalb der gesetzlichen Schranken des
Urheberrechts ist ohne ausdrückliche schriftliche
Genehmigung durch die Ebner Media Group GmbH
& Co. KG nicht gestattet. Downloads, Scans oder
Kopien sind ausschließlich für den privaten, nicht
kommerziellen Gebrauch erlaubt.

Soweit Inhalte nicht vom Verlag selbst erstellt
wurden, werden die Urheberrechte Dritter beachtet
und entsprechend gekennzeichnet.

DRUCK

Silber Druck GmbH & Co. KG
Otto-Hahn-Straße 25
34253 Lohfelden

.NET DEVELOPER CONFERENCE

Die Konferenz für .NET-Entwickler:innen

**Jetzt
Blind Price
sichern!**

**23. BIS 26.
NOVEMBER 2026**
Köln

.NET Core | .NET Frontend | .NET Backend | Cloud & Infrastructure
AI Fundamentals | AI Developer Tools | Software Craftmanship
Data & Data Management





Die Konferenz für AI, Cloud, Web
und .NET Development

29. JUNI BIS 2. JULI 2026

m:con Congress Center
Rosengarten, Mannheim

JETZT MIT
DNPaboSpecial
10% VORTEIL
AUF DEIN TICKET
SICHERN

4 Tage Konferenz

für Entwickler:innen, Architekt:innen & Tech Leads,
die Qualität, Skalierbarkeit und AI-Readiness ernst nehmen.

**Das erwartet Dich –
praxisnah, tiefgehend, relevant:**

- Software Design & Architecture | Software Quality |
Testing & Security
- AI Technologies | Agentic AI | AI Developer Tools & Strategies
- C# & .NET | Frontend | Backend | Frameworks
- Cloud Platforms | DevOps | CI/CD | Databases
- Web Technologies | Agile Development |
Product Ownership
- Personal Skills | Team Leadership



**Werde Teil der Community,
die Software wirklich weiterdenkt.**



www.developer-world.de/dwx | **#DWX**