Kapitel 8: Coding mit LLMs und Agenten

Ein Erfahrungsbericht

Autoren: Alex Hofmann, Dr. Claas Busemann, Dr. Josef Reislhuber, Francesco La Torre, Tobias Wagner, (MaibornWolff GmbH, München, https://www.maibornwolff.de)

Sind KI-Assistenten nur bessere Autovervollständigung–oder verändern sie die Softwareentwicklung fundamental? Dieses Kapitel ist ein Erfahrungsbericht aus der Praxis eines mittelständischen IT-Dienstleisters.

Seit ca. drei Jahren–dem ChatGPT-Moment–nutzen viele Unternehmen KI-Unterstützung in Form von LLMs und agentischer KI bei der Erstellung von Code. So auch MaibornWolff. In diesem Kapitel stellen wir zunächst die Stufen vor, die man bei der Automatisierung der Codeerzeugung beobachten kann. Dazu gibt es zwei Sichten–eine schon allgemein gebräuchliche und eine weiterführende. Wir werden hier beide vorstellen. Wenn man als IT-Dienstleister Code für Projektkunden produzieren möchte, benötigt man spezielle Vorkehrungen, um sowohl das intellektuelle Eigentum (IP) seiner Kunden zu schützen als auch die Sicherheit der eigenen Umgebungen zu gewährleisten. Die wichtigste Frage, die immer wieder diskutiert wird, ist: Wie gut ist der erzeugte Code, und ist er auch langfristig wartbar? Das Kapitel schließen wir mit Beobachtungen zu Produktivitätseffekten ab, die wir bisher beobachten konnten, zur Verdichtung der Arbeit für die Entwickler und auch zu den Kosten des KI-Einsatzes.

8.1: Stufen von automatisiertem Coding mit KI

Wenn man sich die aktuelle Landschaft der KI-gestützten Softwareentwicklung anschaut, findet man eine bereits weit verbreitete Einteilung in drei Stufen: *AI-Assisted Coding* mit hoher menschlicher Kontrolle, *Vibe Coding* als kreative Kollaboration und *Agentic Coding* mit hoher Autonomie der KI (siehe Abbildung 8.1). *Vibe Engineering* wird als Meta-Kompetenz beschrieben.

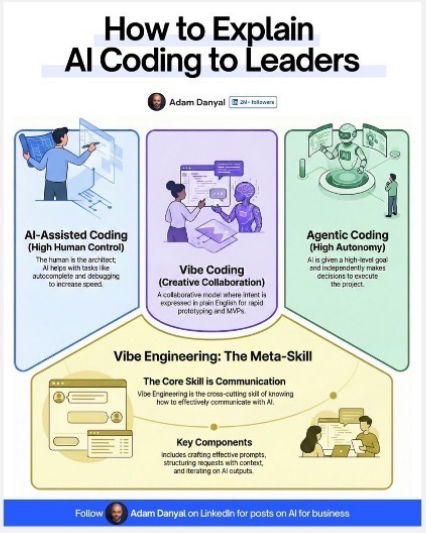

Abbildung 8.1. Gängige Darstellung der Levels von KI-Codeerzeugung

Diese Einteilung ist ein guter Einstieg, aber sie bildet nur eine Momentaufnahme ab. Wenn man einen Schritt zurücktritt und ein Bild aus einer anderen Branche sucht, drängt sich eine Analogie auf, die gut verdeutlicht, wohin die Reise geht: das autonome Fahren.

Die Automobilindustrie kennt fünf Stufen des autonomen Fahrens. Wir beobachten, dass die Softwareentwicklung gerade dieselben Autonomiestufen durchläuft (siehe Abbildung 8.2):

- **Stufe 1 – Assistiertes Coden:** Fahrassistenzsysteme unterstützen den Fahrer, übernehmen aber nicht das Steuer. Übertragen auf die Softwareentwicklung entspricht das dem Einsatz von Tools wie GitHub Copilot mit Code Completion. Die KI vervollständigt Zeilen und macht Vorschläge, aber der Entwickler behält die volle Kontrolle.

- **Stufe 2 – Teilautomatisiertes Coden:** Systeme können unter anderem das Steuer übernehmen, der Fahrer bleibt aber in der Verantwortung. In der Softwareentwicklung sind das die Chat-first-Coding-Tools wie Roo Code oder Claude Code. Der Entwickler führt einen Dialog mit der KI, beschreibt, was er braucht, und die KI generiert ganze Codeblöcke. Hierhin gehört auch das sogenannte *Vibe Coding*–ein Begriff, den Andrej Karpathy 2025 geprägt hat [Karpathy 2025]. In seiner reinsten Form bedeutet Vibe Coding, dass man der KI eine Absicht in natürlicher Sprache beschreibt und sich nicht darum kümmert, was genau im generierten Code passiert. Der Begriff wurde sogar zum Collins Word of the Year 2025 gewählt. Für schnelle Prototypen und Wegwerf-MVPs kann das gut funktionieren. Für Produktivsysteme funktioniert es allerdings nicht [Duyunov 2026].
- **Stufe 3 – Hochautomatisiertes Coden:** Der Fahrer kann sich in bestimmten Situationen länger vom Fahrgeschehen abwenden. In der Softwareentwicklung entspricht das dem Einsatz von Task-Level-Agents: KI-Agenten, die eigenständig Aufgaben bearbeiten, auf der Build-Pipeline sitzen und Teile des Entwicklungsprozesses autonom übernehmen.
- **Stufen 4 und 5–Vollautomatisiertes und autonomes Fahren:** Hier wird der Mensch vom Fahrer zum Passagier. Ob und wann die Softwareentwicklung diese Stufen erreicht, ist heute noch eine offene Frage–und eine kontrovers diskutierte dazu.

Wo stehen wir heute? Unsere Einschätzung: bei etwa 2,5. Die meisten Projekte bei MaibornWolff bewegen sich zwischen Stufe 2 und 3. Der Entwickler sitzt noch im „Fahrersitz“, aber die KI ist bereits deutlich mehr als ein besseres Autovervollständigungs-Tool. In manchen Projekten sind die Teams schon weiter, nutzen KI-Agenten für Codeentwicklung und Testen, in anderen noch bei Stufe 1. Und das ist auch in Ordnung: Nicht jedes Projekt und nicht jeder Kontext erlaubt denselben Autonomiegrad.

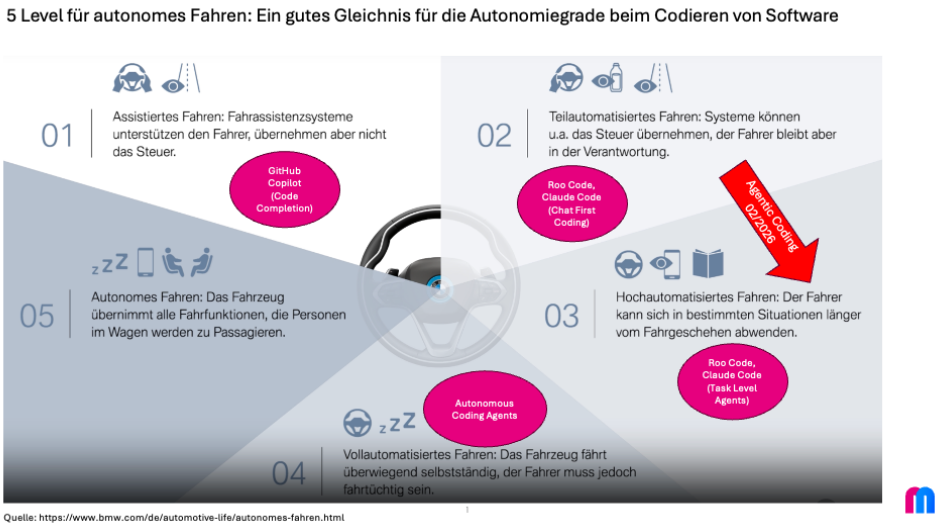

Abbildung 8.2. Langfristige Sicht auf Codeerzeugung mit KI

8.1.1: Das vierte Zeitalter der Softwaretechnik

Neben der Frage der Autonomie-Stufen lohnt sich auch ein historischer Blick. Die Softwaretechnik hat in ihrer Geschichte mehrere fundamentale Umbrüche erlebt (siehe Abbildung 8.3):

1. **Ab ca. 1960 – Von der Maschine lösen:** Hochsprachen wie COBOL und Fortran befreiten Entwickler von der direkten Maschinenprogrammierung. Man musste nicht mehr in Assembler oder Maschinencode denken.
2. **Ab ca. 1990 – Portabilität und Frameworks:** Mit Java, .NET und dem Aufkommen von Frameworks wie Spring ging es um Austauschbarkeit und Schichtenarchitekturen. SOA (Service-Oriented Architecture) war dabei ein wichtiges Thema.
3. **Ab ca. 2010 – Agile, Automatisierung, Containerisierung:** CI/CD-Pipelines, Docker, Kubernetes–die dritte Phase brachte Automatisierung des Entwicklungsprozesses selbst.
4. **Jetzt – KI- und datengetrieben:** Wir beobachten gerade den Übergang in ein viertes Zeitalter. Software wird nicht mehr primär algorithmisch codiert, sondern zunehmend *deskriptiv* beschrieben. Man beschreibt, was man haben möchte–auf einer Ebene, die näher an Requirements als

an Implementierung liegt. Und die KI erzeugt den Code. Auf der OOP-Konferenz 2026 fasste ein Referent es treffend zusammen: „Hört doch auf, Dashboards zu coden." Wenn man eine gute Datenbasis hat und ein LLM mit dem richtigen Kontext füttert, entstehen solche Sichten nahezu auf Knopfdruck.

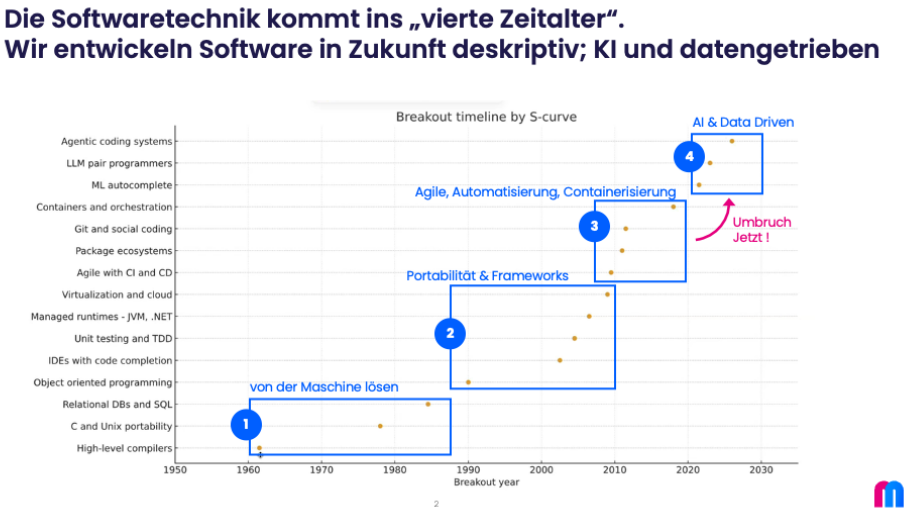

Abbildung 8.3. Abstraktionsstufen bei der Codeerzeugung

8.2: Ein neuer Software-Entwicklungsprozess

Diese Entwicklung verändert auch den Entwicklungsprozess selbst. Abbildung 8.4 zeigt den Prozess, den wir aktuell für agentisches Coding mit KI etabliert haben. Es gibt dabei zwei essenzielle neue Rollen:

- Der *Context Engineer* kümmert sich darum, den Projektkontext so aufzubereiten, dass die KI-Agenten optimal arbeiten können. Dabei geht es nicht nur um technischen Kontext wie Code, Architektur oder Build-Konfiguration, sondern auch um Domänenwissen, Fachsprache, Tickets und Testfälle.
- Der *AI Engineer* stellt sicher, dass eine funktionierende KI-Umgebung vorhanden ist. Dazu gehören z.B. ausgewählte Sprachmodelle,

Entwicklungswerkzeuge, Integration in bestehende Systemumgebungen und weiteres. Darüber hinaus stellt er den LLM-Agenten das Tooling und Datenanbindung bereit, z.B. in Form geeigneter MCP Server.

Das Thema *Context Engineering* gewinnt in KI-unterstützten Projekten aktuell eine enorme Bedeutung. Es geht darum, die Modelle nicht zu trainieren–das will und kann niemand für Frontier-Modelle –, sondern ihnen über den Kontext das nötige Wissen mitzugeben. Das geschieht über Markdown-Dateien, Rules-Files, Skills und ähnliche Mechanismen. Stefan Toth beschreibt in seinem OOP-Vortrag „Kill the Vibe?“ treffend, wie man Domänenkonzepte, Architekturprinzipien und Ist-Strukturen als Kontext hinterlegt, um die Qualität der KI-Ausgaben drastisch zu verbessern [Toth 2026]. Aktuell etablieren sich Standards, wie bspw. https://agents.md/ oder https://agentskills.io/. Skills werden ein sehr großes Thema in den kommenden Jahren.

Ein weiterer wesentlicher Aspekt des neuen Prozesses: Die Zyklen verkürzen sich enorm. Was früher ein zweiwöchiger Sprint war, schrumpft auf wenige Tage. Die Zyklen orientieren sich eher an inhaltlichen Paketen (Features) als an Zeiträumen.

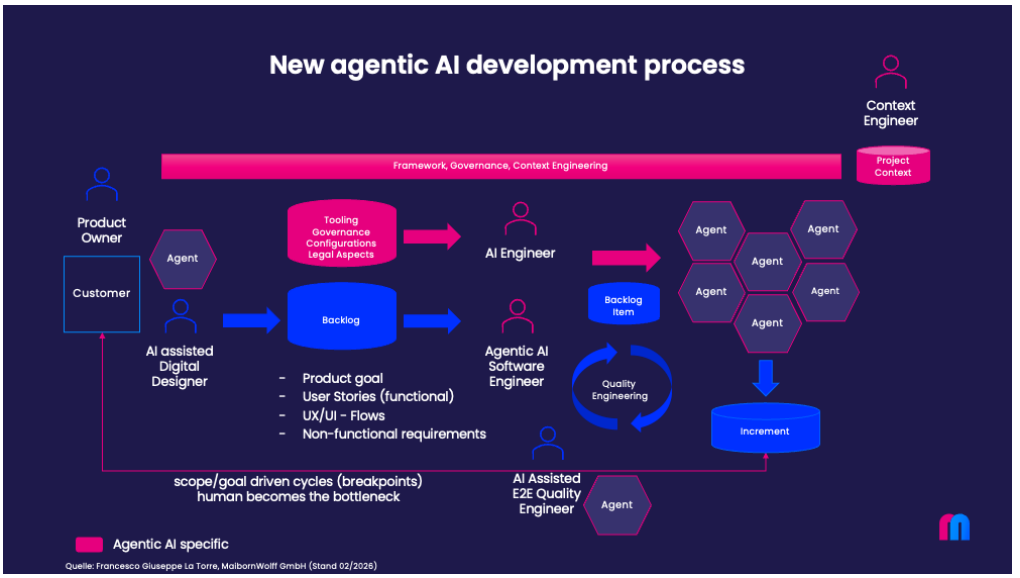

Abbildung 8.4. Prozess für Agentic AI Coding

Dabei werden Menschen im Prozess an vielen Stellen zum Bottleneck:

- Der Kunde der Software wird zum Bottleneck, weil er realistisch gesehen die Features nicht mehr so detailliert und schnell erarbeiten kann, wie die KI sie verarbeiten könnte. Weiter können die Kunden der Software im Regelfall angebotene Inkremente nicht so schnell abnehmen, wie sie erzeugt werden.
- Entwickler werden zum Bottleneck, wenn sie den generierten Quellcode in voller Tiefe und Detaillierung reviewen wollen wie bisher.
- Integrationstester werden zum Bottleneck, weil das Testen von langen Prozessketten End-To-End über Systemgrenzen hinweg länger dauert als die Entwicklung eines Features.
- Die Fachorganisation der Kunden wird zum Bottleneck, weil sie die umgesetzten Änderungen nicht schnell genug in angepasste betriebliche Prozesse und die Mitarbeiterausbildung integrieren kann.

Solche Entwicklungen kann man überall beobachten, wo agentenbasierte KI bei der Softwareproduktion im Einsatz ist.

8.3: KI-Umgebungen, welche die Daten unserer Kunden schützen

Als wir vor zwei Jahren begannen, KI-Tools in unseren Kundenprojekten einzusetzen, stießen wir auf eine klare Ablehnung: „Nein, das könnt ihr nicht machen. Wenn ihr LLM-Aufrufe absetzt, gehen unsere Daten zu Trainingszwecken raus. Der Code geht raus, die Testdaten gehen raus. Aus Sicht der Compliance ist das nicht akzeptabel."

Diese Bedenken der Kunden waren absolut berechtigt. Also haben wir eine eigene geschützte Umgebung aufgebaut: die **AI Trusted Zone** (Abbildung 8.5).

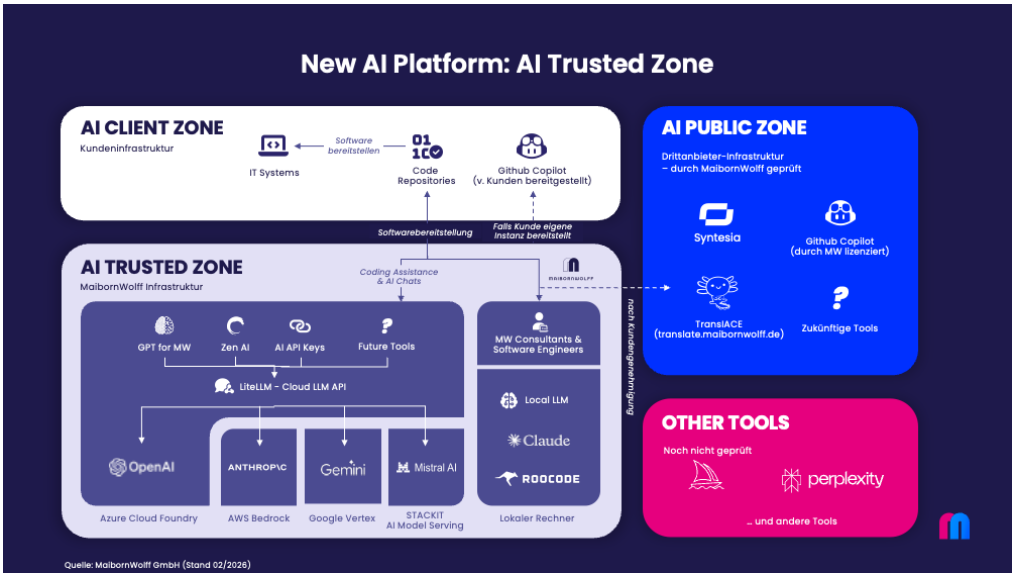

Abbildung 8.5. AI Trusted Zone

Technisch handelt es sich um eine Virtual Private Cloud (VPC), die exklusiv für MaibornWolff gehostet wird. Wir haben dafür klare Vertragsregelungen mit Microsoft getroffen. In dieser geschützten Umgebung laufen auch unsere internen MaibornWolff-Systeme. Mittlerweile haben wir auch VPC-Instanzen anderer Hyperscaler ergänzt.

Der Aufbau folgt dem Prinzip eines **AI Gateways**–einem Konzept, das in Kapitel 4 (*Agent-Gateways: Die Rache der SOA im KI-Zeitalter*) ausführlich beschrieben wird. Im Kern funktioniert es so: Alle LLM-Anfragen aus den IDEs der Entwickler laufen über einen zentralen Proxy (LiteLLM). Dieser routet die Anfragen an das jeweils gewünschte Modell–OpenAI GPT, Anthropic Claude über AWS Bedrock, Google Gemini, Mistral. Für besonders sensible Anwendungsfälle nutzen wir den Anbieter STACKIT, dessen Rechner in Deutschland stehen.

Als wir dieses Umgebungskonzept unseren Kunden gezeigt haben und die AI Trusted Zone vorgestellt haben, war die Antwort: „Go. Könnt ihr machen."

Neben dem IP-Schutz hat diese Architektur einen zweiten entscheidenden Vorteil: **Kostenmonitoring**. Über den zentralen Proxy (LiteLLM) können wir genau nachvollziehen, welche Kosten entstehen, Budgets pro Entwickler zuweisen und im Projekt anfallende Kosten an Kunden verrechnen. Darauf kommen wir im Abschnitt über Kosten noch zurück.

Unsere Botschaft an andere Unternehmen: Jede Firma, die KI ernsthaft in der Softwareentwicklung einsetzen will, benötigt eine solche geschützte KI-Umgebung–mit klarer Governance, Kostenmonitoring und der Möglichkeit, zwischen verschiedenen Modellen und Providern zu wählen. Das ist auch eine Frage der digitalen Souveränität in Bezug auf die Nutzung von KI.

8.4: Weitere Sicherheitsaspekte

Die AI Trusted Zone schützt die Daten unserer Kunden vor dem Abfluss an LLM-Provider. Aber es gibt weitere Sicherheitsthemen, die beim Einsatz von KI-Agenten in der Softwareentwicklung relevant sind.

Prompt Injection ist ein reales Risiko: Wenn ein Agent im Web recherchiert oder externe Datenquellen einbindet, könnte er manipulierte Anweisungen aufnehmen und ausführen. Die großen Modelle haben inzwischen eigene Schutzmechanismen dagegen, aber ein Restrisiko bleibt. Wer sich für die Details interessiert, findet in den Kapiteln 9 (*Inhärente Risiken von LLMs*) und 10 (*Sicherheit von Anwendungen auf der Basis von LLMs*) dieses Buches eine ausführliche Darstellung der Risiken und Abwehrstrategien. Wir haben das gelöst, indem die LLMs/Agents nicht mehr direkt im Internet suchen, sondern einen Index-und-Suchdienstleister nutzen (Google), der kuratierte und stets aktuelle Suchindizes unterhält und diese gegen Prompt Injection schützt. Die Suchanfragen kommen gar nicht mehr bei den echten Seiten an, welche infiziert sein könnten und damit eine für Prompt Injection verursachen würden.

MCP-Server (Model Context Protocol) bieten die Möglichkeit, KI-Agenten mit externen Systemen wie Jira, GitHub oder Datenbanken zu verbinden. Wir haben uns im Sommer 2025 bewusst dagegen entschieden, MCP-Server als neues Kommunikationsprotokoll in unserer Coding-Umgebung einzusetzen, weil es für uns sicherheitstechnisch damals noch ein zu großes Einfallstor war. Aktuell erarbeiten wir Konzepte, um lokal betriebene MCP-Server gegen gängige Angriffsvektoren abzusichern. Außerdem erstellen wir eine MCP Whitelist.

Sandboxing: Wenn ein KI-Agent Code generiert und ausführt, muss sichergestellt sein, dass er nicht unkontrolliert auf das System zugreifen kann. Tools wie Claude Code bringen inzwischen zwar eigene Sandbox-Mechanismen mit, jedoch haben wir aus Sicherheitsgründen beschlossen, eine

eigene Sandbox-Lösung basierend auf Docker aufzubauen. Zwei wesentliche Anforderungen haben uns dabei geleitet: Zum einen darf der Agent lokal nicht mit der Kennung und den Rechten des Entwicklers laufen. Zum anderen muss der Netzwerkzugang durch die Sandbox massiv eingeschränkt werden. Auf dem lokalen Entwicklerrechner sollte man zusätzlich vorsorgen–regelmäßige Backups, Versionskontrolle mit Git und eine saubere Isolation von Credentials und Zugriffsrechten sind Pflicht und nicht verhandelbar.

8.5: Das Wichtigste: Die Qualität des generierten Codes

Es gibt beeindruckende Erfolgsberichte über KI-generierten Code. Anthropic behauptet, dass es intern Projekte gibt, in denen kein Code mehr von Hand geschrieben wird. In vielen Unternehmen berichten Entwickler von massiven Produktivitätsgewinnen. Wir freuen uns über diese Entwicklung–und sind trotzdem vorsichtig. **Wir akzeptieren keinen Code, der nicht normale Reviews durchlaufen hat: sowohl maschinelle Reviews als auch menschliche.**

Essenziell für guten Code ist dabei die Einrichtung von Feedback-Schleifen (auch Backpressure genannt). Wichtig ist dabei, dass Folgendes erfüllt ist, noch bevor ein menschlicher Reviewer einen Code-Review durchführt:

- Eine sehr gute Testabdeckung, wobei Integrationstests eine besonders wichtige Rolle zukommt.
- Automatisierte Reviewstufen mit sehr strengen statischen Codeanalyse-Regeln. Diese fallen noch deutlich strenger aus, als dies bisher in rein menschengetriebenen Entwicklungsprozessen üblich war.
- Automatisierte Architektur- und Sicherheitschecks

Erst wenn man alle diese Stufen der Qualitätssicherung aufgebaut hat, gewinnt man wirklich an Geschwindigkeit. Menschliche Code-Reviews fokussieren sich dann eher auf Architekturentscheidungen. Dies ist sinnvoll, weil ein Mensch der riesige Mengen Code fundiert gegen kleinteilige Regeln reviewen soll, sowieso zum „Durchwinken“ neigen würde. Also muss man dafür sorgen, dass für qualifizierte Menschen nach Möglichkeit die hochwertigen Aufgaben übrig bleiben.

8.5.1: Erfahrungen aus der Praxis

Was sind unsere konkreten Erfahrungen nach mehreren Monaten intensivem Einsatz? In einem Wort: durchwachsen–aber mit klar aufsteigender Tendenz. **KI wird das Software-Engineering auf die nächste Stufe heben, da sind wir uns absolut sicher.**

Bei einem unserer Projekte, einem großen, gewachsenen System mit einer umfangreichen Codebasis, liegt die Akzeptanzrate des generierten Codes bei **80 bis 90 Prozent**. Das klingt gut, heißt aber auch: Jede Zeile muss ein erfahrener Entwickler reviewen. Die restlichen 10 bis 20 Prozent enthalten Probleme: Manchmal sind dies überkomplexe Lösungen, wo einfachere gereicht hätten. Manchmal generierte Tests, die auf den ersten Blick gut aussehen, aber wertlos sind. Zum Beispiel, weil Null-Werte als Testdaten verwendet werden, die den Test immer grün machen, ohne etwas Sinnvolles zu prüfen.

Als Einsatzgebiete, bei denen KI besonders hilfreich ist, konnten wir Folgendes beobachten:

- **Code-Exploration und Onboarding:** Das Verständnis einer fremden Codebasis beschleunigt sich dramatisch. Man kann die KI fragen, wo bestimmte Funktionalitäten implementiert sind, sich Aufrufhierarchien generieren lassen und die Architektur verstehen, ohne jede Datei einzeln durchlesen zu müssen.
- **Refactoring:** Klassen zusammenfassen, lange Methoden kürzen, Redundanzen entfernen–hier kann man der KI oft vertrauen.
- **Testgenerierung:** Ist ein exzellenter Startpunkt. Die KI generiert Unit-Tests und Subsystem-Tests, die als Grundgerüst dienen. Man muss sie aber prüfen und nachschärfen.
- **Story-Implementierung:** Ganze Features lassen sich aus einem Ticket-Text heraus umsetzen. In einem unserer Projekte wurden nur noch ca. 5% des Codes von Hand geschrieben–der Rest kam von der KI, wurde aber intensiv gesteuert und kontrolliert.

Bei folgenden Einsatzgebieten beobachten wir bei der KI noch Raum für Verbesserungen:

- **Autonomes Bugfixing:** Hier hilft KI als Kompass und Navigator, zeigt, wo man Probleme suchen muss, und liefert Hypothesen. Entscheidend

ist, dass die KI auf alle relevanten Informationen zugreifen kann. Das eigentliche Debugging erfordert nach wie vor menschliche Expertise.
- **Architekturentscheidungen:** Die KI kann Architektur beschreiben, dokumentieren und lässt sich ebenfalls sinnvoll als Sparringspartner für Architekturentscheidungen einsetzen. Die Beurteilung, ob eine Architektur für einen konkreten Fall angemessen ist, bleibt jedoch schwierig.

8.5.2: Was sagt die Forschung?

Die Frage, ob KI-generierter Code schlechter wartbar ist, beschäftigt nicht nur Praktiker, sondern auch Forscher. Eine kontrollierte Studie mit 151 professionellen Entwicklern kommt zu einem beruhigenden Ergebnis [Borg et al. 2025]: Entwickler, die Code weiterentwickeln sollten, der mit KI-Unterstützung entstanden war, waren **nicht langsamer und produzierten keine schlechtere Codequalität** als Entwickler, die rein manuell erstellten Code weiterentwickelten. Gleichzeitig bestätigt die Studie die Produktivitätsgewinne: eine mediane Reduktion der Entwicklungszeit um ca. 30%, bei erfahrenen KI-Nutzern sogar um fast 56%.

Der Trade-off, den viele Teams befürchten–schneller entwickeln, aber schlechtere Wartbarkeit hinterlassen–materialisierte sich in dieser Studie nicht. Allerdings mit einer wichtigen Einschränkung: Die Teilnehmer der Studie haben nicht ihr Denken ausgelagert. Sie haben das fachliche Problem durchdacht, es in kleinere Teile zerlegt und gute Engineering-Praktiken angewandt, um die KI zu steuern.

8.5.3: Reines Vibe Coding sollte nicht für Produktionscode eingesetzt werden

Dmitry Duyunov bringt es auf den Punkt [Duyunov 2026]: Vibe Coding funktioniert hervorragend für schnelle Prototypen, Proofs of Concept und einmalige Scripts. Für Produktivsysteme scheitert es fast immer an fehlenden Invarianten, fragilen Zustandsübergängen, versteckten Seiteneffekten und technischen Schulden, die sich nach wenigen Iterationen aufbauen. Die KI spürt weder den Betriebskontext noch On-Call-Verantwortung, noch Langzeit-Wartbarkeit. Das ist nicht Aufgabe der KI. Es ist unsere Verantwortung.

8.5.4: Elephant Code und die Wartbarkeitsfalle

Michael Stal warnt in seinem JavaSPEKTRUM-Editorial vor einem weiteren Phänomen [Stal 2026]: LLMs neigen dazu, vollständige Lösungen zu generieren, selbst wenn minimale Lösungen genügt hätten. Codebasen schwellen an, die Wartbarkeit sinkt. Er nennt es „Elephant Code". Wir nennen es „Back to Wartungshölle". Die Lösung: ein präziser Kontext, Architekturwissen, Domänenwissen und kritisches Review. Oder wie Stefan Toth es in seinem OOP-Vortrag formuliert: Wer KI in der Entwicklung einsetzt, braucht **Context Engineering** (die richtigen Vorgaben und Architekturinformationen als Kontext), **automatisierte Prüfungen** (separate Test- und Review-Agenten) und **Guardrails.** Sowohl präventive Guardrails (die definieren, was die KI tun darf) als auch detektive Guardrails (die Regelverstöße scannen) [Toth 2026].

8.5.5: Unsere Position

Unsere Haltung lässt sich in einem Satz zusammenfassen: **Die KI ist auf der derzeitigen Autonomiestufe 2,5 ein hervorragender Junior-Mitarbeiter–technisch auf hohem Niveau, aber man muss das Ergebnis immer kontrollieren.** Agentic Coding ist ein mächtiges Werkzeug, das unsere Arbeit fundamental verändert, aber die Verantwortung liegt nach wie vor beim Menschen. Peer Reviews–ob durch Menschen oder durch spezialisierte Review-Agenten–bleiben Pflicht. Und die Fähigkeit, Code zu beurteilen, wird wichtiger als die Fähigkeit, ihn zu schreiben.

8.6: Konsequenzen bei der Entwicklung von Software mit KI

8.6.1: Beobachtungen zur Produktivität

Wir haben bei MaibornWolff unsere Projektteams befragt, welche Effizienzgewinne sie durch den KI-Einsatz beobachten. Die Zahlen sind Stand Oktober 2025 und basieren auf Einschätzungen–keine wissenschaftliche Messung, sondern Bauchgefühl erfahrener Projektleiter und Entwickler. Aber genau das macht sie für andere Unternehmen wertvoll als Orientierung (Abbildung 8.6):

- **Requirements Engineering:** ca. 25% Effizienzgewinn–semantische Suche in Anforderungen, automatische Dokumentation, Identifikation von Inkonsistenzen und Lücken
- **Architektur & Design:** ca. 25%–automatische Code-Analyse, Überprüfung auf Einhaltung von Standards, Simulationen und Analysen
- **Coding:** ca. 50%–der größte Hebel. Code Completion, KI-gestützte Refactorings, automatische Erstellung von Tests. Bei Routine-Coding noch höher
- **Test & QA:** ca. 30%–Testfälle automatisch generieren, Berichterstellung, höhere Testabdeckung
- **DevOps / CI-CD:** ca. 10%–hier gibt es bereits hohe Automatisierung, der Zugewinn durch KI ist daher geringer
- **Dokumentation:** ca. 20%–Dokumente automatisch generieren, Übersetzungen, Anpassungen schneller umsetzbar

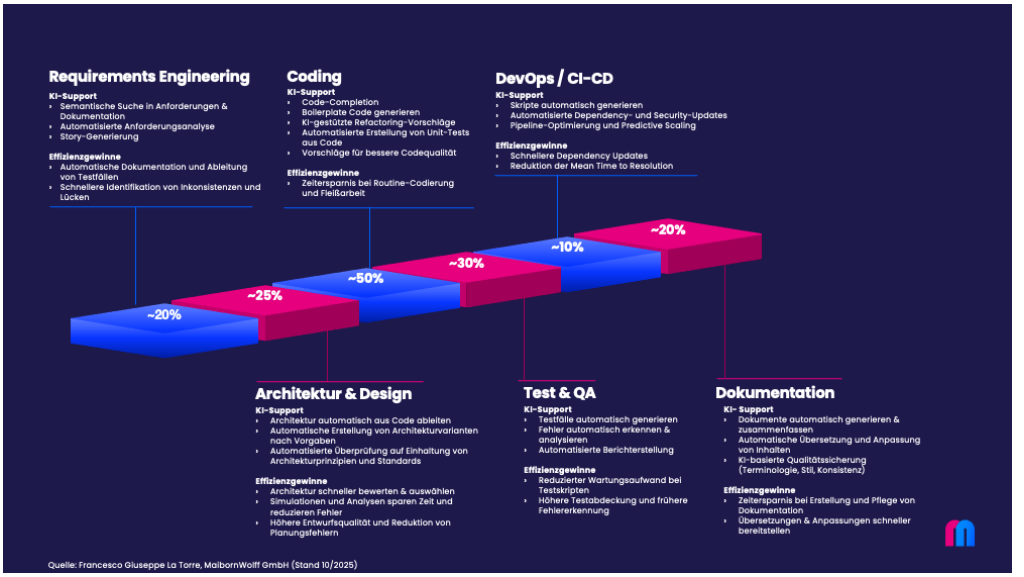

Abbildung 8.6. Geschätzte Produktivitätsgewinne durch KI-Einsatz

Diese Zahlen variieren stark von Projekt zu Projekt. In einem Greenfield-Projekt mit klarer Architektur und modernem Techstack sind die Gewinne höher als in einem Legacy-System. Aber der Trend ist eindeutig: KI-gestütztes Coding bringt signifikante Produktivitätsgewinne, und der größte Hebel liegt beim Coding selbst.

8.6.2: Arbeit der Entwickler verdichtet sich weiter

Es gibt eine Kehrseite der Produktivitätsgewinne, über die noch zu wenig gesprochen wird: **Die Arbeit verdichtet sich.**

Was früher ein zweiwöchiger Sprint war, schrumpft auf wenige Tage. Features, die man vorher in Stunden geschätzt hat, entstehen in Minuten. Das klingt erst einmal fantastisch–aber es bedeutet auch, dass die Zyklen kürzer werden, die Kontextwechsel häufiger und die kognitive Last höher.

Aus den USA gibt es bereits Berichte von Entwicklern, die von Burnout-Symptomen berichten–nicht obwohl, sondern *weil* sie KI einsetzen. Sie sind hypereffizient, aber am Abend fühlen sie sich leer. Fünf Features gleichzeitig ausgecheckt, zwischen ihnen hin- und herspringend wie jemand, der fünf Gerichte in seiner Küche gleichzeitig in der Zubereitung hat. Schon vor zweieinhalb Jahren, als wir bei MaibornWolff intern einen Kaminabend zum Thema KI in der Entwicklung veranstaltet haben, brachte einer unserer Entwickler es auf den Punkt: „Mein Entwickler-Hamsterrad dreht sich doch jetzt schon schnell. Und jetzt soll es sich noch schneller drehen?“

Genau das passiert. Die agile Welt hat uns bereits daran gewöhnt, alle zwei Wochen zu sprinten. Kein Sportler sprintet ausschließlich. Und jetzt verkürzen wir die Sprints von zwei Wochen auf wenige Tage bzw. auf Features. Die Frage, wie wir mit dieser Verdichtung umgehen–wie wir verhindern, dass Effizienz zur Ausbeutung wird, wie wir bestehende Prozesse auf diese höhere Entwicklungsgeschwindigkeit anpassen, das sind die drängendsten Fragen, die uns in den nächsten Jahren beschäftigen werden.

8.6.3: Kosten des KI-Einsatzes

Ein Blick auf unsere Kostendaten zeigt ein interessantes Bild (Abbildung 8.7). Im Bereich Software Engineering haben wir folgende Kosten für LLM-Nutzung:

- **November 2025:** 6.640 € für 180 Nutzer
- **Dezember 2025:** 8.040 € für 220 Nutzer
- **Januar 2026:** 13.000 € für 220 Nutzer

Das entspricht bei uns im Januar 2026 etwa **60 € pro Entwickler und Monat**–weniger als ein einzelner Claude-Max-Account. Im Vergleich zu den Tagessätzen von Softwareentwicklern sind das aktuell moderate Kosten,

jedoch erwarten wir einen massiven Kostenanstieg, wenn Entwickler die Werkzeuge intensiver nutzen. Der Produktivitätsgewinn wird jedoch die Kosten weiter stark überwiegen.

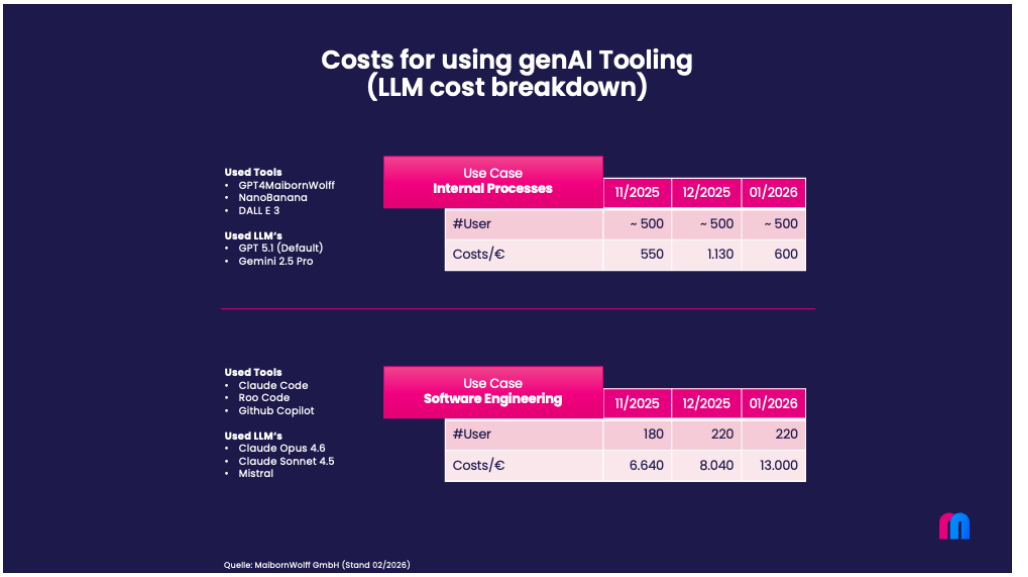

Abbildung 8.7. Kostenentwicklung KI-Einsatz

Aber die Kurve zeigt steil nach oben. Die Kosten haben sich in zwei Monaten nahezu verdoppelt. Nicht weil einzelne Entwickler mehr zahlen, sondern weil mehr Entwickler die Tools intensiver nutzen und mehr Tokens verbrauchen. Wenn man hier die Kosten nicht beobachtet und steuert und ohne Monitoring und Budgetierung arbeitet, kommt irgendwann ein böses Erwachen.

Deshalb ist das Kostenmonitoring über den zentralen AI-Proxy (LiteLLM) ein wesentlicher Bestandteil unserer AI-Trusted-Zone. Aktuell budgetieren wir Limits pro Entwickler. Man kann das ähnlich sehen wie eine Prepaid-Karte für ein Smartphone. Wenn das Budget aufgebraucht ist und noch viel Monat übrig ist, muss man sich mit seinem Teamleiter abstimmen. Das schafft Bewusstsein für den Ressourcenverbrauch und verhindert, dass die Kosten unkontrolliert davonlaufen. Diese Budgetierung bringt uns aber auch in eine Zwickmühle. Auf der einen Seite sagen wir allen Mitarbeitern, sie sollen AI einsetzen, um ihre Produktivität zu erhöhen. Auf der anderen Seite ist das Limit von 60 Euro für AI Agenten pro Monat zu niedrig, um sie mehr als ein paar Tage zu nutzen. Das hält viele davon ab, die Tools in den Arbeitsalltag zu integrieren.

Auch hier gilt die Botschaft an andere Unternehmen: KI-Coding-Kosten

sind aktuell noch sehr moderat im Vergleich zum Nutzen. Aber sie steigen schnell, wenn KI umfangreicher eingesetzt wird, und ohne zentrales Monitoring und Budgetierung kann es zu einer Kostenexplosion kommen. Wir müssen lernen, wie ein sinnvoller Umgang mit KI-Kosten in Zukunft aussehen kann.

8.7: Über die Autoren

Alex Hofmann (CTO): Verantwortet die technologische Zukunft und Innovationsführung der MaibornWolff GmbH. Als Informatiker und leidenschaftlicher Softwarearchitekt arbeitet er intensiv mit Kunden und Technologiepartnern zusammen, treibt KI als Innovation voran und schafft die technologische Grundlage für neue Geschäftsmodelle. Seine Mission ist es, die Kunden von MaibornWolff mit exzellenter Software zu begeistern und KI-Technologie so einzusetzen, dass sie Nutzen stiftet.

Dr. Claas Busemann (Principal IT Architect): Leitet bei MaibornWolff die Agentic Modernization Gilde und verantwortet die Einführung von Agentic Coding als neue Arbeitsweise in Kundenprojekten. Mit seinem Team hat er gezeigt, dass sich durch den gezielten Einsatz von Agentic Coding Projekte schneller und mit kleineren Teams umsetzen lassen. Sein Ziel ist es, diese Methodik vom Experiment zum Standard zu machen und Entwicklungsteams in die Lage zu versetzen, sie produktiv einzusetzen.

Dr. Josef Reislhuber (IT-Architekt): Gestaltet seit 2017 bei MaibornWolff Software- und Cloud-Architekturen mit Fokus auf modernen, kundenspezifischen Microservice-Lösungen. Sein Schwerpunkt liegt im Aufbau leistungsfähiger Teams, der Weitergabe von Wissen und der Schaffung nachhaltiger, zukunftsfähiger Softwarelandschaften.

Francesco La Torre (IT-Berater & Product Owner): Informatiker mit über 20 Jahren Projekterfahrung in Qualitätssicherung, Test- und Projektmanagement. Er arbeitet mit Leidenschaft daran, dass gute Ideen nicht an technischer Komplexität scheitern. Bei MaibornWolff unterstützt er Unternehmen dabei, agentische KI so einzusetzen, dass Teams schneller zu tragfähigen Lösungen kommen.

Tobias Wagner (Senior Lead IT-Architekt): Als technischer Experte der Agentic Modernization Gilde verfolgt er aktiv die neuesten Entwicklungen im Bereich Agentic Coding und AI. Mit über acht Jahren Erfahrung in

der Entwicklung komplexer, skalierbarer Webanwendungen bringt er tiefes technisches Verständnis mit, um KI-gestützte Ansätze praxisnah in Projekten einzusetzen. Sein Anspruch ist es, durch Agentic Coding die Entwicklungsgeschwindigkeit spürbar zu steigern, ohne dabei Kompromisse bei Architektur und Codequalität einzugehen. Dieses Wissen trägt er aktiv in die Gilde und das Unternehmen weiter.

MaibornWolff GmbH: Einer der führenden IT-Dienstleister in Deutschland; 800 Mitarbeitende; Pionier im Einsatz von KI basiertem Software Engineering. 10 Standorte in Deutschland, Spanien und Afrika. Gegründet 1989 von Volker Maiborn und Holger Wolff. Ihre Vision: ein Unternehmen zu schaffen, das heute die Technologie von morgen anbietet und den Menschen in den Mittelpunkt stellt. Umsetzung innovativer Digitalisierungslösungen für Kunden, wie: Bundesagentur für Arbeit, BMW, Mercedes Benz, MAN, VW, Deutsche Bahn, IKEA Deutschland, R+V, Dräger, Miele, Stihl und Siemens.

8.8: Literatur

[Borg et al. 2025]
Markus Borg, Dave Hewett, Nadim Hagatulah, Noric Couderc, Emma Söderberg, Donald Graham, Uttam Kini, Dave Farley: Echoes of AI: Investigating the Downstream Effects of AI Assistants on Software Maintainability. 2025. Online: https://arxiv.org/abs/2507.00788v2

[Duyunov 2026]
Dmitry Duyunov: Vibe Coding: Hype oder Zukunft? LinkedIn, 2026.

[Karpathy 2025]
Andrej Karpathy: „There's a new kind of coding I call 'vibe coding'". X (Twitter), Februar 2025.

[Stal 2026]
Michael Stal: Editorial JavaSPEKTRUM 1/26: Back to the Future. Januar 2026. Online: https://www.sigs.de/artikel/editorial-javaspektrum-1-26-back-to-the-future/

[Toth 2026]
Stefan Toth: Kill the Vibe? Architektur im KI-Zeitalter. Vortrag auf der OOP-Konferenz, Februar 2026.