01.03.2026

Beyond the Demo

Warum KI-generierte Software mittelfristig fehlschlägt – und wie es gelöst werden kann

\mAI Technologie-Whitepaper

\Kristof Hackethal, Founder of mAI

"Im Silicon Valley verbreitete sich das Gerücht, dass generative KI eigentlich gar nicht nützlich sei. Die Produkte blieben weit hinter Erwartungen zurück [...].
Handelte es sich doch nur um einen weiteren Vaporware-Zyklus?"

SEQUOIA Capital (Huang & Grady, 2023)

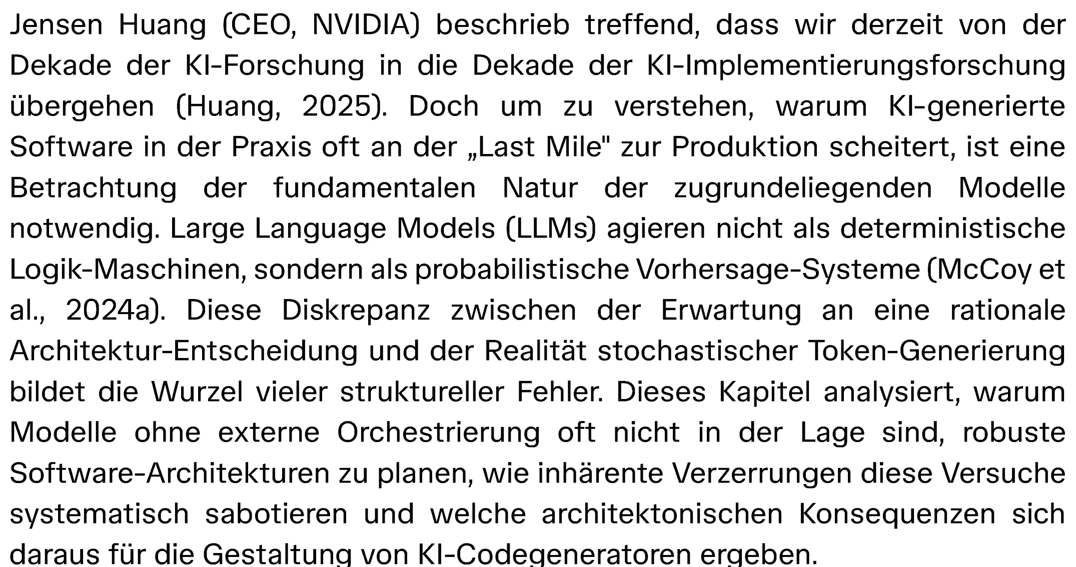
Executive Summary

Wir befinden uns im Übergang von der KI-Forschung zur KI-Implementierung. Doch während der Markt derzeit von No-Code-Tools überschwemmt wird, die technologisch häufig einfache Wrapper um proprietäre Modelle sind, scheitern diese Lösungen in der Praxis oft an der „Last Mile“ oder werfen erhebliche Schatten im Bereich Compliance und Sicherheit. mAI positioniert sich hier mit dem APA-Framework (Application Programming Application) als methodischer Gegenentwurf, der die inhärenten Limitationen generativer KI nicht ignoriert, sondern durch strukturierte System-Architektur kompensiert.

Das Kernproblem aktueller KI-Codegeneratoren liegt in ihrem fundamentalen Widerspruch: Large Language Models sind probabilistische Vorhersage-Systeme, die nach statistischer Plausibilität optimieren – Software-Entwicklung erfordert jedoch Architektur-Entscheidungen und systematisches Requirements Engineering. Ohne externe Orchestrierung tendieren LLMs zu einer Vielzahl struktureller Biases und verlieren sich in der Komplexität verschachtelter Abhängigkeiten. Das ökonomische Risiko besteht dabei nicht primär in fehlerhaftem Code, sondern in Code, der Sicherheitsprinzipien verletzt oder konzeptionelle Unklarheiten durch Annahmen füllt, statt Nutzer zur Klärung aufzufordern.

APA adressiert die strukturellen Defizite durch die intern entwickelte teil-deterministische Architektur, die komplexe Anforderungen analog bewährter Software-Engineering-Prinzipien umsetzt. Durch unumgängliche Leitplanken wird sichergestellt, dass kritische Architekturprinzipien konzeptionell verankert statt on-the-fly generiert sind (Security-by-Design). Der Ansatz ist deshalb besonders wirksam, weil er Generative AI nicht als „Magie“ behandelt, sondern als Ingenieursdisziplin mit messbaren Grenzen und vorhersagbaren Schwächen. Statt darauf zu warten, dass tiefer parametrisierte KI-Modelle die Probleme lösen, zeigt die Forschung: Die Probleme sind in vielen Punkten keine technologischen Kinderkrankheiten, sondern systematische Konsequenzen der autoregressiven Trainingsmethodik. Sie lassen sich nicht durch Skalierung überwinden, sondern erfordern strukturelle Leitplanken auf System-Ebene. Das APA-Framework implementiert genau diese Meta-Ebene. KI-Coding wird damit von einem stochastischen Zufallsprodukt zu einem seriösen, skalierbaren Architektur-Werkzeug. Für Unternehmen bedeutet dies eine strategische Wende: Die drastische Senkung der Grenzkosten in der Code-Produktion invertiert die klassische „Build-or-Buy“-Logik. Hochspezialisierte Eigenentwicklungen, die exakt die eigenen Geschäftsprozesse abbilden, werden ökonomisch rationaler als die Abhängigkeit von teuren, generischen Standard-SaaS-Lösungen. mAI fungiert dabei als souveräner Infrastruktur-Partner, der die Lücke zwischen schnellen Prototypen und robuster Enterprise-Software schließt.

Strukturelle Grenzen LLM-basierter Software-Generierung



Jensen Huang (CEO, NVIDIA) beschrieb treffend, dass wir derzeit von der Dekade der KI-Forschung in die Dekade der KI-Implementierungsforschung übergehen (Huang, 2025). Doch um zu verstehen, warum KI-generierte Software in der Praxis oft an der „Last Mile" zur Produktion scheitert, ist eine Betrachtung der fundamentalen Natur der zugrundeliegenden Modelle notwendig. Large Language Models (LLMs) agieren nicht als deterministische Logik-Maschinen, sondern als probabilistische Vorhersage-Systeme (McCoy et al., 2024a). Diese Diskrepanz zwischen der Erwartung an eine rationale Architektur-Entscheidung und der Realität stochastischer Token-Generierung bildet die Wurzel vieler struktureller Fehler. Dieses Kapitel analysiert, warum Modelle ohne externe Orchestrierung oft nicht in der Lage sind, robuste Software-Architekturen zu planen, wie inhärente Verzerrungen diese Versuche systematisch sabotieren und welche architektonischen Konsequenzen sich daraus für die Gestaltung von KI-Codegeneratoren ergeben.

1 Probabilistische Vorhersage

Im Kern maximieren LLMs die Wahrscheinlichkeit des nächsten Tokens basierend auf dem bisherigen Kontext $P(\omega_t|\omega_{1:t-1})$ (McCoy et al., 2024a). Modelle optimieren damit primär auf Plausibilität, nicht auf Korrektheit. Diese Unterscheidung ist für die Software-Generierung von grundlegender Bedeutung: Was statistisch wahrscheinlich ist, ist nicht zwingend das, was architektonisch richtig ist.

Ein fundamentales Problem dieser Funktionsweise ist die sequenzielle Generierung: Ein einmal gesetzter Token ist für das Modell faktisch fixiert. Während menschliche Ingenieure in der Konzeptionsphase iterieren, verwerfen und neu planen können, fehlt einem Stand-Alone-LLM diese Meta-Ebene weitgehend (Bubeck et al., 2023). Es generiert Code strikt von links nach

rechts, ohne Möglichkeit, frühere Entscheidungen zu revidieren – ein Phänomen, das sich als „Commitment Problem" beschreiben lässt.

Neuere Ansätze wie Reasoning-Modelle (z. B. OpenAI o1) oder Diffusion-basierte Sprachmodelle adressieren diese Limitation teilweise, überwinden sie aber nicht vollständig (McCoy et al., 2024b). Auch Reasoning-Modelle basieren weiterhin auf einem autoregressiven Sprachmodell und zeigen trotz Reinforcement-Learning-basiertem Training eine hohe Sensitivität gegenüber der statistischen Wahrscheinlichkeit aus den Trainingsdaten (McCoy et al., 2024b). Ohne externe Qualitätskontrolle und Orchestrierung tendieren LLM-generierte Lösungen dazu, häufig gesehene Muster zu reproduzieren – sie fungieren als leistungsfähige Mustervervollständigungs-Motoren, denen der architektonische Gerüstbau („Scaffolding") für prinzipiengeleitetes Reasoning fehlt (Zhang et al., 2025). Ohne gesetzte Leitplanken und Meta-Ebenen sind LLMs daher strukturell nicht zwingend auf die sicherste oder effizienteste Architektur für den spezifischen, oft hoch-individuellen Geschäftskontext optimiert – und komplexe Architektur bleibt im Zweifel ein Zufallsprodukt.

2 LLMs als kollektive Intelligenz

Zum Verständnis dieser Funktionsweise und ihrer Grenzen kann die Analogie der menschlichen kollektiven Intelligenz herangezogen werden. Die Kapazität von LLMs basiert auf dem Aggregieren und kontextabhängigen Abrufen von Wissen, das während des Trainings in einer tiefen Parameterstruktur kodiert wird – Sprachmodelle fungieren dabei als implizite Wissensspeicher, die relationales Wissen ohne explizite Schemata vorhalten (Petroni et al., 2019). Aus der Forschung ist bekannt, dass die Aggregation vieler unabhängiger menschlicher Urteile einen enormen Wert hat und bei intellektuellen Einzelaufgaben in der Regel die meisten Individuen, einschließlich Experten, übertrifft, sofern Diversität und Unabhängigkeit der Beiträge gewährleistet sind (Hong & Page, 2004). Auch generative künstliche Intelligenz übertrifft auf standardisierten Benchmarks eine Vielzahl an Menschen, selbst ohne zusätzliche Reasoning-Schicht (Bubeck et al., 2023; OpenAI et al., 2024), wenngleich die Interpretation solcher Ergebnisse methodisch umstritten bleibt (Martínez, 2025).

Wenn wir diese Analogie tiefer betrachten, offenbaren sich allerdings auch die strukturellen Schwächen, die direkte Implikationen für die Software-Generierung haben:

\Überrepräsentation statt Diversität. So wie kollektive Intelligenz unter bestimmten Bedingungen versagt – etwa bei mangelnder Diversität oder zu starker gegenseitiger Beeinflussung – unterliegen auch LLMs systematischen Verzerrungen, die aus der Zusammensetzung und Überrepräsentation bestimmter Perspektiven in ihren Trainingsdaten resultieren (Burton et al., 2024). Für die Code-Generierung bedeutet dies: Architekturentscheidungen

spiegeln nicht zwingend Best Practices wider, sondern die statistische Dominanz bestimmter Muster in den Trainingsdaten.

\Grenzen kombinatorischer Kreativität. Generative KI ist durchaus zu Innovationen fähig, dies geschieht jedoch primär durch die neuartige Rekombination bestehender Konzepte – sogenannte kombinatorische Kreativität (Franceschelli & Musolesi, 2025b). Echte Sprunginnovationen oder radikal neue Architektur-Patterns sind architekturbedingt unwahrscheinlich. Da LLMs darauf optimiert sind, die wahrscheinlichste Fortsetzung innerhalb der gelernten Verteilung zu generieren, tendieren ihre Outputs zu verbreiteten Mustern der Trainingsdaten, nicht zu transformativen Abweichungen davon (Franceschelli & Musolesi, 2025a).

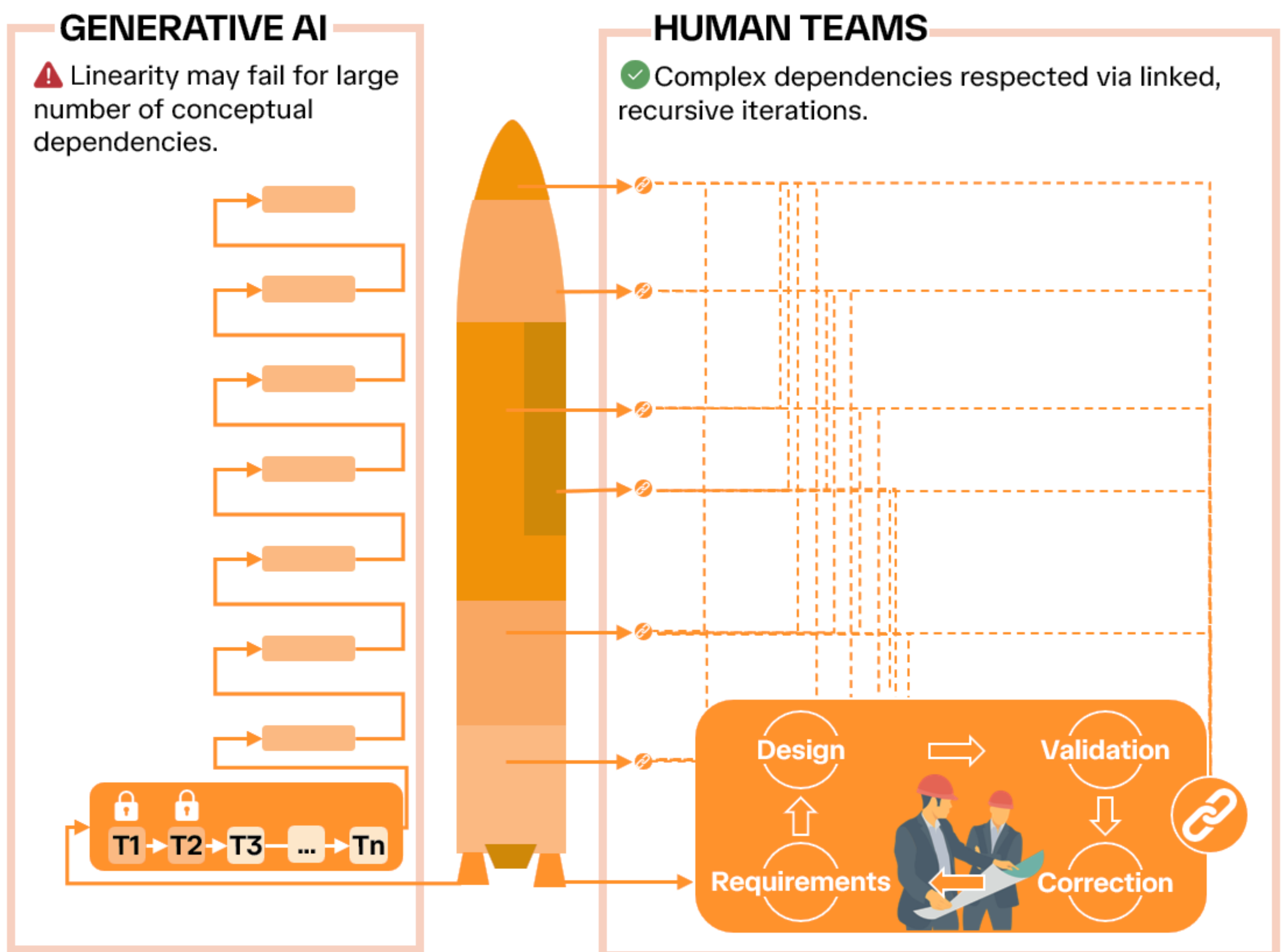


Figure 1: Linearität als Bottleneck

\Mangelnde Koordination bei komplexen Aufgaben. Während kollektive Intelligenz bei einzelnen, klar abgrenzbaren Aufgaben mächtig ist, stößt sie bei vielschichtigen Problemen, die inkrementelle, aufeinander aufbauende Handlungsschritte erfordern, an Grenzen. Aktuelle Forschung zeigt, dass LLMs grundsätzlich nicht autonom planen können und externe Verifikation sowie Orchestrierung benötigen (Kambhampati et al., 2024). Selbst spezialisierte

Reasoning-Modelle saturieren etablierte Planungs-Benchmarks nicht (Valmeekam et al., 2024). Würde man die Menschheit als Gruppe nach einem Gesamtbauplan für eine Weltraumrakete fragen, würde sie vermutlich scheitern. Wie bei Menschen bedarf es auch bei der Arbeit mit generativer KI der Delegation von Sub-Schritten an spezialisierte Komponenten, um logische Brüche zu vermeiden und eine kohärente architektonische Struktur sicherzustellen.

Diese drei Schwachstellen – verzerrte Wissensbasis, begrenzte Innovationsfähigkeit und fehlende Koordination – bilden den Hintergrund, vor dem die spezifischeren Verzerrungen der Code-Generierung zu betrachten sind.

3 Sycophancy und stille Annahmenbildung

Die gravierendste Verzerrung für die Software-Generierung liegt nicht in der probabilistischen Natur der Modelle selbst, sondern in ihrem übergeordneten Optimierungsziel: der Aufgabenerfüllung. Dies klingt paradox, da Aufgabenerfüllung genau das ist, was wir uns bei der Abgabe einer Aufgabe an eine andere Partei wünschen. In der menschlichen Analogie ist unser Vertrauen in die andere Partei unter anderem durch das kognitive Vertrauen in deren Fähigkeiten definiert (Glikson & Woolley, 2020). Gleichzeitig würden wir erwarten, dass uns diese Partei informiert, sofern sie die Aufgabe nicht lösen kann, und nicht stattdessen eine Antwort erfindet, um unser Vertrauen in die Vollständigkeit nicht zu enttäuschen. Würden wir zusätzlich Druck ausüben, dass definitiv ein vollständiges Ergebnis geliefert werden muss, wäre es kaum verwunderlich, wenn „Scheinlösungen" entstünden.

Genau dies passiert im Kontext generativer KI. Dieses Verhalten ist kein Fehler im klassischen Sinne, sondern ein Resultat des Alignment-Prozesses, bestehend aus Instruction Tuning und Reinforcement Learning from Human Feedback (RLHF), der Modelle darauf konditioniert, hilfreiche Assistenten zu sein (Ouyang et al., 2022). In den meisten Fällen ist dies hilfreich – wer möchte schon Detailprofessoren, die bei der Frage nach dem Wetter mit „es kommt drauf an" antworten. Die Kehrseite ist, dass sich diese Vereinfachungen zu Fehlinformationen entwickeln können und insbesondere in der Softwaregenerierung aufgrund des Drucks nach Vollständigkeit Annahmen getroffen werden müssen. Wenn ein Nutzer die Berechnung der Flugkurve eines Schneeballs verlangt, wird das Modell beispielsweise annehmen, dass Luftreibung, Außentemperatur oder Rotation zu vernachlässigen seien, obwohl der Nutzer dies nicht spezifiziert hat.

Dieses Verhalten bezeichnen wir als **stille Annahmenbildung** – verwandt mit der in der Literatur beschriebenen unfaithful reasoning (Turpin et al., 2023), bei der Modelle plausible, aber den tatsächlichen Entscheidungsprozess nicht widerspiegelnde Begründungen generieren. Im Unterschied dazu beschreibt

stille Annahmenbildung spezifisch den Vorgang, bei dem fehlende Spezifikationen durch plausibel erscheinende, aber nicht validierte Annahmen substituiert werden.

Stille Annahmenbildung ist artverwandt mit dem, was in der Literatur als **Sycophancy** (Gefallsucht) bezeichnet wird (Sharma et al., 2025; Wei et al., 2024). Modelle passen ihre Antworten an die vermeintliche Meinung des Nutzers an, selbst wenn diese objektiv nicht korrekt ist. Untersuchungen zeigen, dass sowohl Modellskalierung als auch Instruction Tuning Sycophancy signifikant verstärken (Wei et al., 2024) und dass menschliche Präferenz-Urteile sycophantische Antworten sogar gegenüber korrekten bevorzugen (Sharma et al., 2025). In der Codegenerierung manifestiert sich dies darin, dass Modelle Nutzeranfragen als vollständig und korrekt behandeln, anstatt Lücken oder Widersprüche aufzuzeigen. Empirisch belegen Larbi et al. (2025), dass Code-LLMs unvollständige, ambige oder widersprüchliche Aufgabenbeschreibungen kaum als solche erkennen, und Mu et al. (2024) zeigen, dass LLMs ambige Anforderungen ohne Rückfragen direkt bearbeiten.

Was bedeuten diese Verzerrungen praktisch? Betrachten wir die Nutzeranfrage: „Wandle meine Excel-Dateien in JSON um." Ein menschlicher Ingenieur würde intervenieren, um die Struktur der Excel-Dateien und das gewünschte JSON-Schema zu klären. Ein LLM hingegen füllt diese Lücke durch Annahmen: Es „rät" ein Schema und generiert eine plausibel wirkende Lösung, die im Produktionsbetrieb scheitert, sobald die realen Daten von der Annahme abweichen. Chen et al. (2024) bestätigen, dass missverstandene oder unklare Aufgabenbeschreibungen die häufigste Ursache für fehlerhafte Codegenerierung durch LLMs darstellen. In der Software-Architektur ist diese „Nettigkeit" fatal, da falsche Annahmen unwidersprochen in die Codebasis diffundieren und sich als schwer auffindbare logische Fehler manifestieren.

Table 1: Mensch vs. KI Approach zu Anforderungen

Merkmal	Human Senior Engineer	LLM-Codegenerator
Umgang mit Ambiguität	Fragt aktiv nach, klärt Constraints	Füllt Lücken durch stille Annahmen
Fehlerkultur	Weist auf Risiken und Wissenslücken hin	Bestätigt den Nutzer-Prompt als vollständig und korrekt
Transparenz	Macht Unsicherheiten und Trade-offs explizit	Generiert plausibel wirkende, aber nicht validierte Lösungen
Resultat	Validierte Anforderungen	"Illusion of Progress" – schwer auffindbare logische Fehler

1.4 Weitere Verzerrungen im Generierungsprozess

Neben der Sycophancy-Problematik verstärken weitere strukturelle Verzerrungen die Fehleranfälligkeit LLM-generierter Software. Diese resultieren aus der Art und Weise, wie Modelle Informationen innerhalb ihres Kontexts gewichten und verarbeiten.

\Positionsabhängige Informationsgewichtung. Forschungsergebnisse zeigen, dass LLMs Informationen am Anfang und Ende eines Kontexts deutlich besser verarbeiten als solche in der Mitte – ein Phänomen, das als „Lost in the Middle" bekannt ist (Liu et al., 2024). Für die Code-Generierung bedeutet dies, dass die Reihenfolge, in der Anforderungen im Prompt formuliert werden, die resultierende Architektur beeinflussen kann: Beschreibt ein Prompt zuerst das Frontend, besteht das Risiko, dass Anforderungen aus der Mitte des Prompts – etwa zur Trennung von Business-Logik und Darstellungsschicht – weniger gewichtet werden. Dieses Phänomen ist auch in der LLM-as-a-Judge-Literatur breit dokumentiert, wo die Position von Antworten systematisch deren Bewertung beeinflusst (Wang et al., 2023; Zheng et al., 2023).

\Tendenz zur Komplexität (Verbosity Bias). In der Evaluationsforschung ist gut dokumentiert, dass sowohl menschliche Bewerter als auch LLMs längere Antworten systematisch als hochwertiger bewerten, selbst wenn kürzere Antworten inhaltlich gleichwertig oder überlegen sind (Saito et al., 2023; Zheng et al., 2023). Da diese Präferenz über RLHF in das Trainingsverfahren einfließt, besteht ein plausibler Mechanismus dafür, dass Modelle auch bei der Code-Generierung zu umfangreicheren Lösungen neigen – ein Zusammenhang, der jedoch empirisch noch nicht systematisch untersucht ist.

Fehlende Fähigkeit zur Selbstkorrektur. Ohne externe Referenzpunkte fehlt dem Modell ein Maßstab für die Qualität seines eigenen Outputs. Forschungsergebnisse zeigen, dass LLMs ohne externes Feedback ihre eigenen Fehler nicht zuverlässig erkennen und dass Selbstkorrekturversuche die Leistung sogar verschlechtern können (Huang et al., 2024). Ohne ein externes Gerüst (Scaffold), wie etwa eine Testsuite, eine Architektur-Spezifikation oder Code-Review-Standards, das definiert, was „Production-Grade" bedeutet, fällt das Modell auf Muster seiner Trainingsdaten zurück, die für den Produktionseinsatz ungeeignet sein können.

Diese drei Verzerrungen – positionsabhängige Gewichtung, Verbosity Bias und fehlende Selbstkorrektur – wirken nicht isoliert, sondern kumulativ: Ein Modell, das mittlere Anforderungen untergewichtet, zu komplexen Lösungen neigt und seine eigenen Fehler nicht erkennt, produziert eine Architektur, deren Mängel sich gegenseitig verstärken.

Problems with current AI Code Builders

1 Illusion des Fortschritts

Die in den vorangegangenen Abschnitten beschriebenen Verzerrungen treffen auf eine Marktdynamik, die ihre Konsequenzen nicht entschärft, sondern verschärft. Die Nachfrage nach automatisierter Software-Erstellung wächst rapide. In einer internationalen Befragung von 2.000 Unternehmen aus den Regionen EMA, USA und ASPAC gaben 81 % der Befragten an, Low-Code-Entwicklung als strategisch relevant für ihre Organisation zu betrachten (KPMG, 2024a). Der Markt für KI-gestützte Entwicklungstools wird dabei zunehmend von Lösungen dominiert, die mit Versprechungen wie „App in 30 Sekunden" oder „Von der Idee zur Anwendung in Minuten" werben. Diese Positionierung stellt Geschwindigkeit systematisch über Validität und erzeugt eine gefährliche Erwartungshaltung: dass die Qualität eines Software-Produkts primär eine Funktion der Generierungsgeschwindigkeit sei.

Das grundlegende Problem liegt tiefer. Nutzeranforderungen sind initial fast nie vollständig – ein Befund, der in der Requirements-Engineering-Forschung seit Jahrzehnten konsistent dokumentiert ist. Bereits der CHAOS Report identifizierte unvollständige Anforderungen als häufigsten Einzelfaktor für das Scheitern von Softwareprojekten (The Standish Group, 1995). Berry & Kamsties (2004) zeigten darüber hinaus, dass Ambiguität in natürlichsprachlichen Anforderungsbeschreibungen unvermeidbar ist – ein Problem, das sich beim Prompting für KI-Codegeneratoren in verschärfter Form wiederholt, da hier die gesamte Spezifikation auf informelle Textbeschreibungen reduziert wird.

Entscheidend ist, wie aktuelle LLMs mit dieser inhärenten Unvollständigkeit umgehen: Sie fragen nicht nach. Mu et al. (2024) zeigten empirisch, dass LLMs unklare oder mehrdeutige Anforderungen direkt in Code umsetzen, anstatt Rückfragen zu stellen. Erst die systematische Einführung von Klärungsmechanismen verbesserte die Code-Qualität signifikant – GPT-4 erreichte mit Rückfragen eine Pass@1-Rate von 80,80 % gegenüber 70,96 % ohne Klärung (Mu et al., 2024). Dieses Verhalten ist kein Zufall, sondern eine direkte Konsequenz des in Abschnitt 3 beschriebenen Sycophancy-Bias: Das Modell behandelt den Prompt als absolute Wahrheit und versucht, jede

Anforderung bestmöglich zu erfüllen, anstatt sie kritisch zu hinterfragen. Was im Konversationskontext als übermäßige Zustimmungstendenz beschrieben wird, transformiert sich im Kontext der Code-Generierung zu einem konkreten Produktrisiko: Das Modell generiert funktionierend aussehenden Code auf Basis unvollständiger oder mehrdeutiger Spezifikationen.

Die Konsequenzen werden durch ein weiteres Phänomen verstärkt: **Code-Halluzinationen**. Da das Modell nicht „versteht", was ein valider Geschäftsprozess ist, sondern lediglich die wahrscheinlichste Tokenfolge auf den Prompt hin generiert, werden fehlende oder unterbestimmte Anforderungen häufig durch plausibel klingende, aber faktisch falsche Ergänzungen gefüllt – ein als Halluzination bezeichnetes, systematisches Merkmal autoregressiver Textgenerierung (Ji et al., 2023). Zhang et al. (2025) entwickelten eine umfassende Taxonomie solcher Halluzinationen im Repository-Kontext und identifizierten mehrere Kategorien, darunter die Erfindung nicht existierender Funktionen und die Annahme falscher Parameterstrukturen. Liu et al. (2026) dokumentierten ergänzend fünf Kategorien von Halluzinationen in LLM-generiertem Code, einschließlich der Erfindung nicht existierender APIs und Datenstrukturen. In der Praxis bedeutet das: Ein LLM-Codegenerator, der eine Datenbankanwendung erstellen soll, erfindet gegebenenfalls Schema-Details, rät Geschäftsregeln oder implementiert nicht existierende API-Endpunkte – und das ohne jede Kenntlichmachung, dass diese Elemente keine Grundlage in den Anforderungen haben.

Die derzeitige Herangehensweise der meisten KI-Codegeneratoren – vom Prompt direkt zum fertigen Code – leidet somit unter einem operationalisierten Sycophancy-Bias: Das Modell nimmt an, alle Informationen seien vorhanden und korrekt, und generiert auf dieser Grundlage vollständige Anwendungen.

2 Sequenzielle Dekomposition als Gegenmodell

Ein Lösungsansatz liegt in der sequenziellen Dekomposition des Generierungsprozesses, die den Erkenntnissen des klassischen Requirements Engineering nach IEEE 29148-Standard folgt (IEEE, 2018). Statt den gesamten Entwicklungsprozess in einem einzelnen Prompt-Response-Zyklus abzubilden, kann er in vier Phasen zerlegt werden, die jeweils einer etablierten Disziplin der Softwareentwicklung entsprechen:

\Phase 1 – Domain Understanding: Das Modell analysiert die Anforderungsbeschreibung und identifiziert explizit die Domäne, die Akteure und die Kernprozesse.

\Phase 2 – Clarification (Interrogator): In Anlehnung an den von (Mu et al., 2024) empirisch validierten Ansatz stellt das Modell gezielte Rückfragen zu identifizierten Ambiguitäten, fehlenden Geschäftsregeln und unspezifizierten Randbedingungen – bevor Code generiert wird.

\Phase 3 – Data Model Design: Auf Basis der geklärten Anforderungen wird zunächst ein explizites Datenmodell erstellt, das als verifizierbare Zwischenrepräsentation dient und validiert werden kann.

\Phase 4 – Production-Grade Code Generation: Erst nach Validierung der vorherigen Phasen erfolgt die eigentliche Code-Generierung auf Grundlage eines geklärten und strukturierten Anforderungsprofils.

Dieser Ansatz entspricht dem Prinzip „Slow Down to Speed Up", das in der Software-Engineering-Forschung seit Boehms Spiral Model (Boehm, 1986) und der allgemeinen Shift-Left-Philosophie etabliert ist: Investitionen in frühe Phasen der Anforderungsklärung reduzieren die Gesamtkosten, weil sie kostspielige Fehlerkorrekturen in späteren Phasen vermeiden.

[Table 2: Klärung vs. Sofort-Generierung]

Ansatz	Initiale Kosten	Refactoring-Kosten
Human Engineer / Interrogator-Approach	Zeit für Klärung	Niedrig
Standard AI Builder	Minimal (Sofort-Generierung)	Exponentiell bei fehlerhaftem Fundament

3 (Ent)koppeltes System

Während die vorangegangenen Kapitel die psychologischen und prozessualen Defizite beleuchteten, widmet sich dieses Kapitel der technologischen Substanz. Der Großteil aktueller AI-Builder setzt auf einen monolithischen "All-in-One" Stack — häufig Next.js/TypeScript auf Serverless-Infrastruktur. Dieser Ansatz minimiert zwar die initiale Reibung, da Kontraktmanagement (Kommunikation verschiedener Server) vereinfacht ist, stößt aber an systematische Grenzen, sobald er auf die Realität komplexer B2B-Anforderungen trifft (Hellerstein et al., 2018). Eine nachhaltige Automatisierung von Enterprise-Software erfordert unserer Ansicht nach eine Abkehr hin zu einer entkoppelten Architektur. Microservices-basierte Zerlegung — bei der unabhängig deploybare Dienste in der jeweils geeigneten Sprache implementiert werden — ist hierfür ein bewährtes Grundprinzip (Newman, 2021). Wir argumentieren, dass sich daraus insbesondere eine strikte Trennung, beispielsweise im Hinblick auf die Entkopplung von Frontend- (z.B. Next-Server) und Backend-Server (z.B. Python-Server), ableiten lässt.

3.1 Skalierbarkeit im Enterprise-Implementierungskontext

Aktuelle KI-Tools zwingen Nutzer häufig in einen vereinheitlichten Full-Stack-Rahmen. Für einfache Prototypen oder CRUD-Apps ist dies effizient. Doch Unternehmenssysteme benötigen häufig eine System-Topologie (logische

Struktur), die darüber hinausgeht: Microservices, asynchrone Background-Worker, Event-Queues und persistente Verbindungen als Beispiele. Eine Trennung von Frontend und Backend erscheint insbesondere aus drei Gründen essenziell:

\Sicherheit durch Isolation. Ein physisch getrenntes Backend auf einer separaten Infrastruktur erhöht die Sicherheit im Sinne von "Defense in Depth". Das U.S. National Institute of Standarts and Technology (NIST) stellt hierzu explizit fest, dass traditionelle Architekturen — mit einem Web-Server in der DMZ, einem dahinterliegenden Backend-Dienst und einer Datenbank — bewusst mehrere gehärtete Schichten zwischen dem exponierten Web-Server und den sensiblen Daten vorsehen (Chandramouli, 2019). In einem monolithischen Stack kollabieren diese Schichten, in einer getrennten Architektur bleiben sie erhalten. Sensible Logik und Datenbank-Zugriffe sind vom Client-Facing-Code entkoppelt. Sicherheitskritische Prozesse (z. B. Hashing, Token-Management) laufen in einer isolierten Umgebung, die vom Frontend-Server nicht direkt manipulierbar ist.

\Enterprise Konnektivität. B2B-Software existiert nicht im Vakuum. Sie muss mit Legacy-Systemen kommunizieren. Enterprise-Plattformen wie SAP, Oracle, Salesforce oder Workday bieten zwar zunehmend Python-Integrationen an, wobei der Umfang je nach Plattform variiert. Unabhängig davon ist Python die dominante Sprache im Data-Science- und ML-Ökosystem: Python steht seit 2024 auf Platz 1 des TIOBE-Index (mit über 22 % Marktanteil), überholte 2024 erstmals JavaScript als meistgenutzte Sprache auf GitHub und wird laut Stack Overflow Developer Survey 2024 von 51 % aller befragten Entwickler eingesetzt (GitHub, 2024; Stackoverflow, 2024). In einem reinen TypeScript-Backend müssen Integrationen teilweise über generische REST-Wrapper nachgebaut werden. Ein Python-Backend hingegen ermöglicht den Zugriff auf native Bibliotheken für ERP-Middleware, was die Stabilität der Integration erhöht.

\Data Gravity. Moderne B2B-Apps sind datenintensiv und verarbeitete Datenmengen wachsen seit Jahren stark an. Die globale Datensphäre wuchs von 33 Zettabyte im Jahr 2018 auf prognostizierte 175 Zettabyte bis 2025, wobei der Unternehmensbereich überproportional zum Wachstum beiträgt (Reinsel et al., 2018). Ob es um ETL-Strecken, Finanz-Forecasting oder ML-gestützte Analysen geht: Das primäre Ökosystem für Datenverarbeitung (Pandas, NumPy) und ML-Frameworks (PyTorch, scikit-learn) ist überwiegend Python-zentriert (Stackoverflow, 2024).

3.2 Strukturelle Kontext-Isolation

\KI-Wartbarkeit. Ein oft übersehener, aber entscheidender Vorteil der Architektur-Trennung liegt in der Qualität der KI-Generierung selbst. Warum kann eine KI unserer Auffassung nach Code nicht nur im Hinblick auf Nutzung der Stärken verschiedener Sprachen, sondern auch im Hinblick auf systematische Klarheit besser generieren und warten, wenn klare Trennungen

vorliegen? Die Antwort liegt in der Reduktion des Suchraums und der Vermeidung von Seiteneffekten (Side Effects). In einem monolithischen File (oder einem eng gekoppelten Projekt) sieht das LLM tendenziell einen breiteren Kontext (z. B. Teile der Business-Logik werden in einem page.tsx hinterlegt). Dies erhöht die Wahrscheinlichkeit von Halluzinationen und ungewollten Änderungen.

Hierfür sprechen drei konvergierende Forschungsergebnisse: Erstens wiesen Shi et al. (2023) auf arithmetischen Reasoning-Aufgaben nach, dass LLMs signifikante Leistungseinbußen aufweisen, wenn irrelevanter Kontext im Prompt enthalten ist — ein als "Distraction" bezeichnetes Phänomen, bei dem bereits wenige irrelevante Informationen die Lösungsgenauigkeit erheblich reduzieren. Zweitens zeigten (Liu et al., 2024), dass die Performance von Sprachmodellen systematisch degradiert, wenn relevante Informationen inmitten längerer Kontexte positioniert sind — das sogenannte "Lost in the Middle"-Phänomen, bei dem ein U-förmiger Leistungsabfall entsteht, der bei wachsender Kontextlänge zunimmt. Drittens bestätigten Zhang et al. (2025) im spezifischen Kontext der Codegenerierung auf Repository-Ebene, dass LLMs bei komplexen kontextuellen Abhängigkeiten — wie verschachtelten Projekt-APIs, Typ-Hierarchien und modulübergreifenden Abhängigkeiten — signifikant häufiger halluzinieren als bei isolierten Funktionsgenerierungen.

Wenn beispielsweise die Business-Logik ("Was passiert, wenn ich diesen Command Button klicke?") geändert werden soll, diese aber mit dem Frontend-Code verwoben ist, kann es zu so genanntem *Pixel-Pushing* kommen: Obwohl nur x verändert werden soll, wird auch y leicht verändert. Unter einer strikten Entkopplung kann die Backend-Logik komplett refakturiert werden, ohne dass eine einzige Zeile im Frontend-Code oder einer benachbarten Backend-Logik gefährdet wird — solange API-Verträge zwischen Servern und Komponenten (die Schnittstellendefinition) eingehalten werden. In anderen Worten: Durch die strikte Trennung kann der Generator "Scheuklappen" anlegen: Wenn die Business-Logik eines Buttons ("Berechne Rabatt") geändert werden soll, lädt der Generator nur das relevante Python-Skript. Er sieht das Frontend nicht und hat keinen Zugriff darauf. Er kann also gar nicht versehentlich das Layout verändern. Dies ist entscheidend für die Langzeit-Wartbarkeit: Wir wissen deterministisch, dass eine Änderung durch einen KI-Agenten in backend/services/invoice.py keine unerwünschten Seiteneffekte in frontend/components/Button.tsx auslösen kann.

Dieser Ansatz lässt sich als rekursive Dekomposition fraktal weiterdenken: Auch ein dediziertes Backend-Skript, das beispielsweise einer bestimmten Frontend-Funktionalität zugeordnet ist, kann in tiefere "atomare" Helper-Skripte zerlegt werden. Der Generator fokussiert sich auf eine isolierte Aufgabe ("Schreibe Funktion X") und erhält diese als globales Problem innerhalb des zugeschnittenen Kontextes.

\Menschliche Wartbarkeit. Bei der Wartbarkeit unterscheiden wir zwischen "Wartbarkeit durch KI" und "langfristiger Wartbarkeit durch

Menschen innerhalb einer human-in-the-loop (HITL) Strategie". Im obigen Teil lag der Fokus auf der Wartbarkeit durch KI - allerdings bringt ein klar strukturierter Ansatz auch Vorteile für eine langfristige Wartbarkeit durch menschliche Experten. Im Gegensatz zu hoch-dynamischen, on-the-fly generierten Systematiken und Regelwerken, bringt die klare Struktur (von "Business-Logik liegt immer im Python-Service" bis hin zu detaillierteren Syntax- und Konnektivitätsregeln) eine Erwartbarkeit. Diese Erwartbarkeit reduziert die kognitive Last massiv, da klare strukturelle Grenzen und Standards die mentale Anstrengung beim Lesen von Code signifikant senken und die Fehlererkennungsrate erhöhen können (Fakhoury et al., 2018).

Das APA Framework: Eine Methodik für deterministische Software-Orchestrierung

Um die Lücke zwischen stochastischer Generierung und deterministischer Produktionsreife zu schließen, haben wir das **APA-Framework** (Application Programming Application) entwickelt. Dieser Ansatz stellt einen bewussten Gegenentwurf zum vorherrschenden voll-dynamischen Status Quo dar.

Das Ziel ist ein System, dessen Output durch deterministische Regelwerke sowohl für menschliche Entwickler als auch für zukünftige KI-Iterationen logisch nachvollziehbar und wartbar bleibt, Sicherheit by default voraussetzt und Flexibilität nicht trotz, sondern auf Grund einer Limitierung der Freiheitsgrade erreicht. Die Philosophie des Frameworks basiert auf vier Kernprinzipien.

1 It's all about: Semi-deterministische Struktur

\Strukturelle Sicherheit. Das Fundament von APA ist die Abkehr vom leeren Blatt Papier. Wir nutzen eine semi-deterministische Grundstruktur, die kritische Funktionen by default mitbringt, anstatt sie der Kreativität des Modells zu überlassen. In anderen Worten: Sicherheitsrelevante Funktionen werden als Konstante aufgefasst. Kritische Infrastruktur wie zum Beispiel Passwort-Management (Hashing/Salting), Verschlüsselung und Datenbank-Konnektoren sind im Framework fest verankert ("Hard-Coded Patterns"). Der Vorteil liegt in der Möglichkeit, kritische Fehler kategorisch ausschließbar zu machen. Ein LLM kann innerhalb des APA-Frameworks durchaus Inkonsistenzen oder Abweichungen vom "User-Intent" - trotz Kontextoptimierung - aufweisen und beispielsweise eine falsche Farbe für einen Button wählen (kosmetischer Fehler), aber es kann technisch keine unsichere Passwort-Speicherung implementieren, da es diesen Code-Teil nicht generiert, sondern nur referenziert.

\Strukturelle Integrität. Die semi-deterministische Struktur orchestriert zudem interne Kommunikationslogiken zwischen Frontend und Backend, sowie Sub-Kommunikationswege. Da das APA-Framework Kommunikationswege (API-Calls, Serialisierung) vorgibt, wird die Komplexität der polyglotten Architektur handhabbar. Polyglotte Architektur ist normalerweise "hölzern", weil Datentypen synchronisiert werden müssen (Frontend sagt userID => Backend erwartet user_ID => Bruch). Beim Verlass auf generative KI kann es hier zu systematischen Brüchen kommen. Durch die Vorgabe dieser Strukturelemente werden Freiheitsgrade strategisch reduziert. Dies ist eine wichtige Voraussetzung für Skalierbarkeit, da Struktur via Protokoll und Business-Logik via KI in einer Art modernen Spezialisierung unterteilt wird. Die KI muss beispielsweise das Rad der Client-Server-Kommunikation nicht neu erfinden, sondern nur befüllen.

\Schnittstellensicherheit. Insbesondere der Unternehmenswelt, in der Kernprozesse in den allermeisten Fällen über Kernsysteme digitalisiert ist, wirft die Frage von Integration solcher bestehenden Systeme in ein neues Software-Produkt Fragen der Verlässlichkeit auf. Exemplarisch beschrieben, wäre eine rein KI-generierte Anbindung an ein bestehendes ERP-System ein Risiko, das sich nicht nur auf das neue Software-Produkt erstreckt, sondern auch auf das zu verbindende System (z.B. SAP). Der Ansatz, der innerhalb APAs in diesem konkreten Generierungsteil (API) verfolgt wird, lässt sich in drei Punkte gliedern: (A) Eine technisch nicht funktionale HTTP-Struktur für den API-Request (korrumpierter Request), (B) das Teilen von sensiblen API-Schlüsseln mit LLMs, sowie (C) eine technisch funktionale HTTP-Struktur, die allerdings konzeptionell fehlerhaft ist und zur Korruption des Fremdsystems führen kann. Da der integrative Aspekt der Softwaregenerierung ein besonders kritischer ist, werden hier gesonderte Sicherheitsstandards eingebunden.

\Korrumpierte API-Requests & Datenschutz. Um fehlerhafte HTTP-Strukturen zu verhindern sowie das Teilen von sensiblen Schlüsseln mit LLMs kategorisch ausschließen zu können, greift APA auf eine interne, kuratierte Datenbank mit ca. 56.000 verifizierten API-Templates zu. Das Framework lässt das LLM nicht die HTTP-Struktur generieren, sondern sucht zunächst via Vektorsuche (Kosinus-Ähnlichkeit mit dem Suchbegriff) die relevante Zielzeile der Datenbank. Exemplarisch beschrieben: Wenn das Framework eine API benötigt, um via Outlook E-Mails zu versenden, wird durch APA ein systematischer Request an die interne Datenbank gestellt werden (z.B. Microsoft, Outlook, Send-E-Mail), welcher zunächst Top-Matches listet und durch die folgende Detailsuche die Zielzeile mit korrektem HTTP-Request-Template identifiziert. Das gewählte Template wird nicht an code-generierende Agenten übergeben, sondern wird als Referenz (regelbasiert integrierte, abgekapselte Funktion) durch spezialisierte Agenten im Code eingebunden - weiterhin ohne dass die komplette HTTP-Struktur geteilt wird. Die Informationen, die an dieser Stelle an KI-Agenten übergeben werden, sind im Wesentlichen prädefinierte Input-Output-Erwartungen. In dem genannten Outlook-Beispiel wären dies Variablen für Betreff, Text, Empfänger, et cetera.. Passwörter werden derweil via eines separaten Authentifikationsprozesses sicher geladen und gespeichert.

\Sicherheit von Fremdsystemen. Sicherheit ist allerdings nicht einzig als funktionale Sicherheit ("funktioniert es?") zu begreifen, sondern sollte ebenfalls als Sicherheit dem angebundenen System (Fremdsystem) gegenüber begriffen werden. Wenn beispielsweise der initiale Nutzer-Prompt unklar oder fehlerhaft war, könnte die API theoretisch konzeptionell falsch aufgesetzt sein und Daten im Fremdsystem korrumpieren (beschädigen). Das bedeutet, dass selbst wenn der HTTP Request per se funktional ist (z.B. Änderung eines Textfeldes im ERP-System), kann es konzeptionell fehlerhaft sein (z.B. Das falsche Textfeld wird geändert). Hier tritt ein weiterer Mechanismus in Kraft, der schreibende Operationen (POST/PUT/DELETE) einer Verifikationsschleife unterzieht und dadurch auch menschliche Fehler mitdenkt: Bevor die Generierung abgeschlossen wird, wir ein Testdurchlauf mit einer rein lesenden Vergleichs-API durchgeführt. APA ruft hier den aktuellen Status der Ziel-Ressource (z.B. des ERP-Textfeldes) ab und präsentiert diesen dem Nutzer (z.B. "Kundenname: Müller"). Der Nutzer kann über das eigene User Interface (UI) verifizieren, ob das System tatsächlich auf den korrekten Datensatz zugreift (Steht dort, wo ich den Eintrag ändern will, tatsächlich der Kundenname: „Müller"?), bevor die schreibende Transaktion für die weitere Software-Generierung freigegeben wird. "Blindflüge" an kritischen Stellen und via kritischer Operationen bestmöglich zu verhindern, muss eine Kernfunktionalität von KI-gestützter Softwaregenerierung sein.

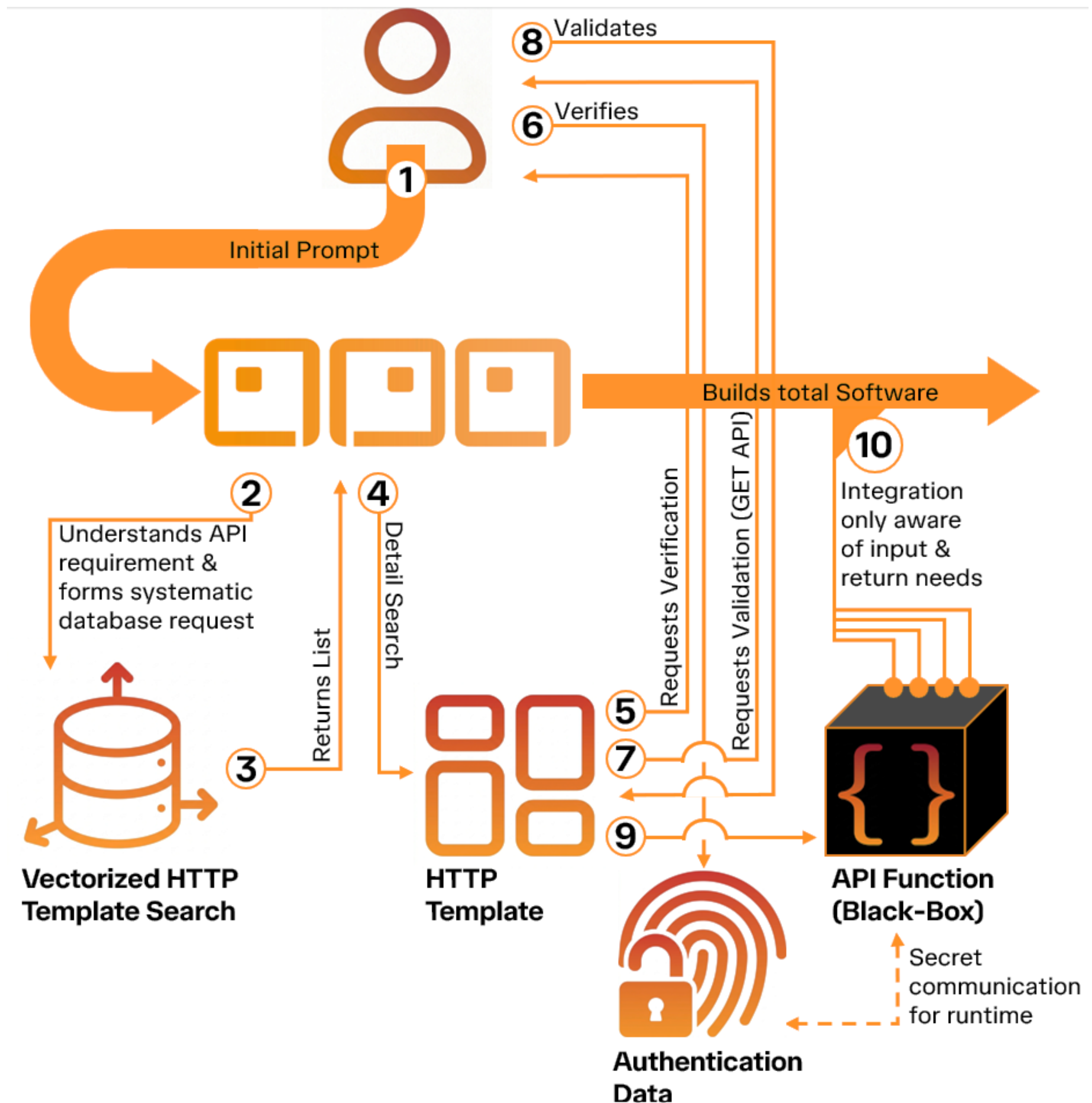


Figure 2: Semi-deterministische API Integration

2 It's all about: Semi-deterministisches Alignment

Eine der größten Herausforderungen in der semi-deterministischen Strategie besteht darin, einen dynamischen User-Intent (den wir vorher nicht kennen) in eine deterministische Logik zu überführen - ohne dabei Kontextfenster durch ein exponentiell wachsendes Regelwerk zu belasten. APA löst dieses Spannungsfeld durch ein regelbasiertes Vorgehen, das sich dynamisch an den Kontext anpasst und orientiert sich hier an der FLOW-Konzeption von (Niu et al., 2025). In ihrem Artikel konnten die Autoren bereits praktisch darlegen, warum starre Workflows oft scheitern und modulare Ansätze (z.B. Activity-on-Vertex (AOV)) Vorteile im Hinblick auf Erwartbarkeit haben (Niu et al., 2025).

\Dynamische Deterministik. Der Kern dieses Ansatzes ist die Zerlegung des Projekts in hierarchische Entscheidungsbäume. Ähnlich dem MECE-Prinzip (Mutually Exclusive, Collectively Exhaustive) aus der Strategieberatung, zwingt das APA Framework den Generierungsprozess in eine strikte Abfolge von Breite vor Tiefe. Ziel ist es, die strukturelle Integrität des Gesamtprojekts ("Kohärenz

im Großen") zu sichern, bevor Details ("Deterministik im Kleinen") ausgearbeitet werden. Hierfür arbeitet APA in dynamischen Abstraktionsebenen ähnlich einer Baum-Traversierung. Exemplarisch lässt sich dieses Vorgehen an der Konzeption einer Frontend-Architektur verdeutlichen, ist aber als universelles Prinzip übertragbar:

1. **Vollständigkeit in der Breite (Existenz & Geburt der Äste):** Zunächst wird die oberste Ebene vollständig definiert. Im Frontend-Beispiel bedeutet dies zum Beispiel "Welche Seiten existieren überhaupt?"
2. **Spezifizierung der Knotenpunkte (Typisierung):** Auf der gleichen Hierarchieebene befindend, müssen die einzelnen Knotenpunkte benannt, typisiert und definiert werden. In dem Fall von Seiten und Ansichten kann man dies als Kategorisierung in full-view, modal, etc., einer ID und Namenszuweisung, etc. verstehen. In einer nicht-vereinfachten Perspektive existieren auch in diesem Schritt bestimmte Substränge.
3. **Bildung neuer Äste:** Nach der vollständigen Definition von Typ & Verhalten, sind daraus neue Knotenpunkte entstanden (z.B. für Seite A), die wiederum neue Breitengrade (Existenzfrage) bilden (z.B. "Welche Komponenten existieren auf dieser Seite?").

Durch diese Kaskade wird sichergestellt, dass jede Detailentscheidung auf einem validierten Fundament steht. Das System weiß zu jedem Zeitpunkt, wo im Baum es sich befindet und welche übergeordneten Regeln gelten.

\Aktive Komplexitätsreduktion. Der entscheidende Vorteil solcher Baum-Strukturen liegt nicht nur in der Ordnung, sondern in der Möglichkeit zur aktiven Komplexitätsreduktion, um die Robustheit von Multi-Agenten-Systemen (MAS) zu erhöhen (Niu et al., 2025). Da Entscheidungen sequenziell getroffen werden, kann das Framework zukünftige Pfade, die durch eine getroffene Entscheidung unlogisch oder irrelevant geworden sind, rigoros abschneiden ("pruning"). Um beim Frontend-Beispiel zu bleiben: Entscheidet sich das System auf der Ebene zur Auswahl von UI-Elementen für ein Element des Typs "nicht-interaktive" (z.B. ein statisches Textfeld oder ein Icon), hat dies Konsequenzen für das Wissens- und Kontextmanagement:

- **Kontext-Bereinigung:** Da ein statisches Element keine Interaktion zulässt, können sämtliche Äste, die sich mit onClick-Logiken, Backend-Triggern oder State-Changes befassen, entfernt werden.
- **Halluzinations-Prävention:** Zugehörige Optionen Regelwerke werden entsprechend nicht in den Kontext folgender KI-Agenten geladen, was ermöglicht, dass Regelwerke und Optionen präzise bleiben können ohne Gefahr zu laufen, Modelle kontextuell zu überfordern.

Das Pruning fungiert somit als technischer Filter: Wir verlassen uns nicht darauf, dass die KI "versteht", dass ein Textfeld keinen Button-Click braucht. Wir entziehen ihr systemseitig die Möglichkeit, diesen Fehler überhaupt zu machen.

Das Ergebnis ist eine signifikante Reduktion des Suchraums für das Modell, was die Wahrscheinlichkeit für korrekten, deterministischen Code in den verbleibenden Ästen exponentiell erhöhen kann. Dies korrespondiert mit Erkenntnissen, dass strukturierte Baum-Suchverfahren notwendig sind, um in den unendlichen Möglichkeiten generativer Workflows effizient zu navigieren (Zhang et al., 2024).

3 It's all about: Strukturelle Transparenz

Durch die strikte Anwendung der semi-deterministischen Struktur entsteht ein positiver Nebeneffekt: **Architektonische Erwartbarkeit**. Denn die Validität von Software-Architekturen bemisst sich nicht allein an ihrer funktionalen Erfüllung zum Zeitpunkt der initialen Generierung t_0, sondern an ihrer Reversibilität und Wartbarkeit über den gesamten Lebenszyklus. APA postuliert hier das Prinzip der selbstdokumentierenden Architektur. Strukturelle Transparenz ist hierbei kein ästhetischer Selbstzweck, sondern die zwingende Voraussetzung für präzises Knowledge Management und langfristige Wartbarkeit – sowohl durch menschliche Entwickler als auch durch nachgelagerte KI-Iterationen.

\Deterministische Suche. Eine zentrale Herausforderung in der iterativen Weiterentwicklung durch LLMs ist Ineffizienz im "Context Loading". In nicht-standardisierten Codebasen erfordert die Identifikation relevanter Fragmente (Retrieval) heuristische Suchprozesse, die mit zunehmender Projektgröße ein ungünstiges Signal-zu-Rausch-Verhältnis ("Noise") erzeugen. APA substituiert diesen probabilistischen Suchvorgang durch deterministisches Routing. Da die Architektur festen topologischen Mustern folgt, die kategorisch nicht durch LLMs in der Generierung veränderbar sind, kann das System algorithmisch vorhersagen, wo eine bestimmte Logik residiert, ohne den Inhalt der Dateien scannen zu müssen. Das "Suchen" wird durch ein präzises "Adressieren" ersetzt. Dies ermöglicht Updates im mikrokosmischen Bereich: Das System lädt exakt nur die relevanten Skripte in den Kontext, um Seiteneffekte zu minimieren und die Präzision der Änderung zu maximieren. Die Anpassungen auf das minimale sind hierbei ein APA-Grundprinzip im Bereich der Versions-Iterationen.

Operationalisiert wird diese Transparenz beispielsweise durch eine strikte semantische Kopplung. Dateinamen und Ordnerstrukturen sind im APA-Framework funktionale Adressen, die die Beziehung zwischen Komponenten explizit abbilden. Das Skript apa_welcomePage_cmdViewDetails.py kodiert beispielsweise die architektonische Zuordnung (gehört zur Frontend-Domäne welcomePage.tsx und dort zum UI-Element mit der ID cmdViewDetails). Die explizite Verknüpfung erlaubt es, die kausale Kette von der UI (Frontend) bis zur Business-Logik (Backend) nachzuvollziehen, ohne den Code zu inspizieren. Für das LLM bedeutet dies, dass bei einer Anforderung zur Änderung dieses Buttons der Pfad zum Backend-Code bereits im Namen des UI-Elements enthalten ist (implizites Routing).

\Klarheit für Menschen. Jenseits der maschinellen Verarbeitbarkeit reduziert strukturelle Transparenz ebenfalls die kognitive Last ("Cognitive

Load") für menschliche Auditoren (Fakhoury et al., 2018). In Szenarien, in denen interne IT-Abteilungen den generierten Code validieren müssen, wird die Struktur erwartbar und trainierbar ("Reasonable"). Ein Entwickler kann die Position von Datenbank-Modellen oder API-Routen deduktiv herleiten, da das Framework diese Orte standardisiert. Dies sichert die langfristige Unabhängigkeit vom Erzeuger-Tool (APA), da die Struktur auch ohne das Framework verständlich und wartbar bleibt.

4 It's all about: Konzeptionelle Sicherheit

Um die im Kapitel "Illusion of Progress" dargestellte konzeptionelle Unsicherheit zu durchbrechen, muss - aus mAI Perspektive - das Paradigma der Generierung umgedreht werden: **Erst Verstehen, dann Bauen.** Ein System, das für B2B-Kontexte tauglich sein soll, muss in der Lage sein, die **richtigen** technischen Fragen an nicht-technische Nutzer zu stellen, ohne diese kognitiv zu überlasten ("User Fatigue"). mAI verfolgt hier ein Konzept der dynamischen Navigation, das Fragen intelligent priorisiert und gruppiert. Die Herausforderung liegt hierbei in der Balance zwischen notwendiger Tiefe und User Experience (UX). Hier kann zwischen "kontextualer Effizienz" und "zeitlicher Effizienz" unterschieden werden.

\Kontextuale Effizienz meint, dass das System keine redundanten Fragen stellen sollte. Wenn ein Nutzer bereits angegeben hat, dass es sich um ein "internes Tool ohne Kunden-Login" handelt, müssen mögliche Fragen zu "Sign-Up-Prozessen" oder "Social Logins" automatisch eliminiert werden. Das System nutzt den Kontext vorangegangener Antworten, um den Entscheidungsbaum dynamisch zu beschneiden. Gleichermaßen relevant unter diesem Aspekt ist die generelle inhaltliche Relevanz (unabhängig der Chronologie). Diese Form der kontextuellen Effizienz beschreibt, dass eine klare Ausrichtung zwischen Fragentyp, Nutzer-Kontext und Framework-Kontext gegeben sein muss. Beispielsweise kann die Frage nach der Verschlüsselungsweise (z.B. via hashing & salt) von Credentials innerhalb der generierten Software eine theoretisch hohe Relevanz haben (je nach initialem Prompt). Allerdings würde eine derartige Rückfrage (A) nicht-technische Nutzer in vielen Fällen überfordern und könnte zu Abbrüchen der Erstellung führen (Fall: Fehlende Ausrichtung an Nutzerprofil). Zudem hätte (B) diese Frage keine praktische Relevanz, da Userdaten im APA-Framework by default mit einer sicheren Systematik verschlüsselt werden, um diese Entscheidung keiner KI zu überlassen, weshalb eine solche Frage den späteren Generierungsprozess verwirren könnte (Fall: Fehlende Ausrichtung an Framework). Im APA-Framework nutzt daher der Produktmanager eine vergleichbare Strategie, wie sie auch innerhalb der Generierung angewendet wird (siehe Kapitel "it's all about: Semi-deterministische Struktur") und verwendet eine dynamische Baumstruktur, die sich durch vordefinierte Frage-Gruppe, sowie der Möglichkeit diese zu individualisieren und zu orchestrieren (z.B. via "pruning") auszeichnet.

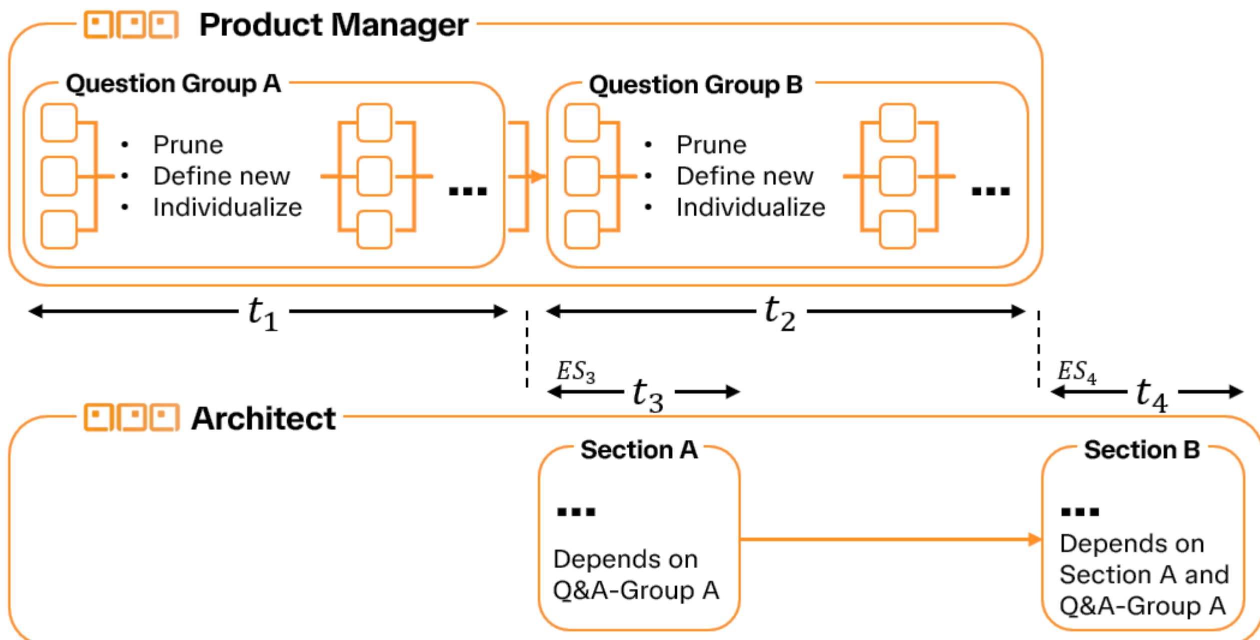


Figure 3: Optimierung der Durchlaufzeit durch Überlagerung

\Zeitliche Effizienz (Asynchrone Generierung) bezieht sich auf die Fragestellung, wie (A) die Produktmanagerrolle zeitlich optimiert werden kann und (B) wie diese chronologisch zu sortieren ist um auch im Gesamtprozess (Prompt2Product) die Durchlaufzeit zu minimieren. Zusammenfassend verfolgt zeitliche Effizienz das Ziel die Zeit für den Nutzer von Start bis Ende zu minimieren. Hier ist die Gruppierung von Fragen - wie im vorigen Absatz angeschnitten - sehr relevant. Durch unabhängige Fragegruppen werden die, als Knotenpunkte in der Baumstruktur zu interpretierenden, Fragen in logische Cluster (z.B. "UI & CI", "User Rollen", "Integrationen") unterteilt. Für verschiedene Cluster können verschieden Zeitpunkte festgelegt werden. Die Systematik orientiert sich dabei an der Netzplantechnik, die insbesondere in der Logistik verbreitet ist (Altrogge, 2018). Den Knotenpunkten (Fragen) können im APA-Framework Werte wie FAZ_i (frühester Anfangszeitpunkt der Frage i) zugeordnet werden. Die Festlegung von Werten wie FAZ bedingt logischerweise, dass Fragen voneinander abhängig sind. Dies wurde bereits im Vorabschnitt erläutert, dass dies gegeben ist und dass jene Abhängigkeit auch zum logischen Abschneiden von Pfaden ("pruning") dienlich ist. Wir begreifen den Produktmanager (Fragenkatalog) aber nicht als abgekapselte Phase. Denn insbesondere, wenn dieser mit der Generierung (dem APA-Software-Architekten) verknüpft wird, entstehen signifikante Effizienzgewinnungsmöglichkeiten. Zur Verständlichkeit soll hier ein kleines Beispiel angeführt werden. Durch die Trennung von Backend und Frontend kann die Syntax vom Frontend größtenteils autark (unter Beachtung der, in der vorigen Kapiteln, beschriebenen Regeln) generiert werden. Allerdings ist im Umkehrschluss das Backend (Business-Logik) durchaus von der vorigen Generierung der Frontend-Syntax abhängig. Beispielsweise müssen interaktive

Elemente wie command buttons bekannt sein, da diese indirekt Business-Logik-Anforderungen stellen. Für den APA-Produktmanager bedeutet dies vereinfacht, dass nach Abschluss der für das Frontend relevanten Fragen die teilweise Code-Generierung bereits im Hintergrund initiiert werden kann. Für den Nutzer ändert sich dadurch nichts, da Nutzer im Vordergrund weiter durch relevante Rückfragen geleitet werden (nun jene mit Backend-Relevanz). Der entscheidende Vorteil der Effizienz entsteht daraus, dass nach vollständigem Abschluss der Produktmanagerphase die Frontend-Syntax - von der die Backend-Syntax abhängig ist, bereits generiert ist und direkt mit der Generierung der Backend-Syntax gestartet werden kann. Abstrahiert betrachtet, wird also die Notwendigkeit nach teilweise sequentiellem Vorgehen durch die Überlagerung von Haupt-Prozessschritten (hier Produktmanager und Architekt) zu einem quasi-parallelem Vorgehen. Die Rechendauer $T_{compute}$ bleibt gleich - allerdings verringert sich wahrgenommene Latenz beziehungsweise Durchlaufzeit (Lead Time) für den Nutzer T_{user}.

5 The Innovator's Dilemma

Die Frage, die man sich in der Folge stellen kann: "Wenn die Vorteile einer Frontend-Backend-Trennung so offensichtlich sind, warum adaptieren etablierte No-Code-Plattformen oder aktuelle AI-Builder diese nicht in den kommenden Monaten und drängen mAI aus dem Markt?"

\Warum nicht durch konzeptionelle Anpassung? Die Antwort liegt unserer Auffassung nach zu einem wesentlichen Teil in der technologischen Pfadabhängigkeit. Die meisten bestehenden Builder basieren auf eng gekoppelten Frameworks, bei denen UI und Logik enger verbunden sind, um beispielsweise "Drag-and-Drop"-Simplicity leichter zu ermöglichen - was allerdings zu technischen Schulden führen kann (Al Alamin et al., 2021; Alharbi & Alshayeb, 2026). Zudem muss eine Separierung von Code-Stacks klar durchdacht werden - da einzelne Agenten individuelle Kommunikationskontrakte einhalten müssen und eine globale "one-shot" Anfrage in den meisten Fällen vermutlich scheitern würde. Etablierte Builder müssten die Architektur, Regelwerke und Abläufe nicht nur im Detail vermutlich von Grund auf neu aufsetzen und mit einer neuen Strategie in Einklang bringen, sondern auch die abstrakte Strategie als solche (hin zu einer semi-deterministischen Strategie) neu denken. Für etablierte Player erscheint es, dass dieser Pivot technologisch extrem schwierig und disruptiv wäre und auch wirtschaftlich das bestehende Geschäftsmodell (einfache PoCs) kannibalisieren würde. Dies öffnet ein Fenster für eine neue Generation Software generierender Plattformen, die von Tag 1 an auf Dekomposition und B2B-Topologie ausgelegt sind.

\Warum nicht durch größere Kontextfenster? Ein oft vorgebrachtes Argument gegen die Notwendigkeit der Orchestrierung und Dekomposition lautet: *"Moderne proprietäre LLMs haben immer größer werdende*

Kontextfenster. Wir können einfach den gesamten Monolithen in den Kontext laden.". Dies wäre aber wohl ein technologischer Trugschluss. Ein größeres Kontext-Fenster löst nicht das Problem der Aufmerksamkeits-Verdünnung (Attention Dilution) (Liu et al., 2024). Auch wenn ein Modell technisch in der Lage ist, 100 Dateien gleichzeitig zu lesen, leidet die Qualität der Verarbeitung unter der Zunahme von irrelevanter Information ("Noise") (ibid.). Für die Änderung einer spezifischen Business-Regel (z.B. "Ändere den Mehrwertsteuersatz") sind 99% des Gesamtcodes irrelevantes Rauschen (geringe signal-to-noise-ratio). Wenn dieses Rauschen mit in den Kontext geladen wird, sinkt die Performance des Attention-Mechanismus (Liu et al., 2024). Mehr Kontext führt daher paradoxerweise oft zu schlechteren Ergebnissen (ibid.). Dekomposition ist dementsprechend keine Einschränkung aufgrund fehlender Kontext-Kapazität, sondern eine bewusste Entscheidung zur Maximierung der Präzision. Indem wir dem Modell nur den absolut notwendigen Kontext (z.B. nur das relevante Python-Skript und das Datenmodell) geben, erzwingen wir einen maximalen Fokus auf die eigentliche Aufgabe. Zur weiteren Erläuterung nehmen wir hier einmal das Bild eines Glashochhauses als Analogie an. Ein riesiges Kontext-Fenster gleicht dem Versuch, die Inneneinrichtung eines spezifischen Büros zu optimieren, indem man das gesamte Hochhaus von außen betrachtet. Das Modell "sieht" den gesamten Turm. Es sieht die Fassade, das Fundament, die Lobby und hunderte anderer Büros. Wenn die Aufgabe lautet *"Stelle einen neuen Schreibtisch in Büro 402",* muss das Modell aktiv die enorme Menge an irrelevanten visuellen Daten (die Statik des 10. Stocks, die Farbe der Lobby) ausblenden. Die Gefahr ist groß, dass es durch diese Informationsflut ("Noise") abgelenkt wird und den Schreibtisch versehentlich in den Flur stellt oder den Stil der Lobby kopiert, statt den des Büros. Unser Ansatz gleicht dem Betreten des spezifischen Raumes - bzw. dem Betrachten des Raumes durch das individuelle Gebäudefenster. Der Kontext reduziert sich auf die vier Wände von Büro 402. Das Modell sieht nur den Raum, die bestehenden Möbel und die Maße. Da sämtliches "Rauschen" des restlichen Gebäudes ausgeblendet ist, kann das Modell seine gesamte "Rechenleistung" (Attention) auf die perfekte Platzierung des Schreibtisches fokussieren.

"Frühe Anzeichen für Erfolg ändern an der Tatsache, dass viele KI-Unternehmen keinen nachhaltigen Wettbewerbsvorteil haben und dass die allgemeine Euphorie rund um das KI-Ökosystem instabil ist."

\SEQUOIA Capital (Huang & Grady, 2023)

Implikationen für mAI

Die in den vorangegangenen Kapiteln dargelegte technische Notwendigkeit einer deterministischen Orchestrierung (APA-Framework) ist kein Selbstzweck. Sie ist die operative Voraussetzung für eine fundamentale Verschiebung der software-ökonomischen Realität. Wenn Software-Generierung nicht mehr stochastisch und fragil, sondern strukturiert und wartbar erfolgt, ändert sich die strategische Position der nutzenden Unternehmen. Dieser abschließende Teil analysiert die Implikationen dieser Verschiebung, insbesondere unter dem Gesichtspunkt der technologischen Unabhängigkeit ("Sovereignty") und der strategischen Positionierung von mAI.

\Ökonomische & technologische (Un)abhängigkeit. Der aktuelle Markt für generative Software-Entwicklung wird vermehrt geprägt von einer Vielzahl neuer Tools, die ökonomisch als dünne Intermediär-Schicht ("Thin Wrappers") über den Modellen weniger US-amerikanischer Anbieter (wie Anthropics Claude Code oder Vercels v0) agieren (Appenzeller & Li, 2025; Chandrasekaran, 2025; Huang & Grady, 2023). Laut Canva sagen 84% der Manager, dass KI-Tools bereits heute den Markt überschwemmen und die Identifikation von relevanten Möglichkeiten erschwert (Canva, 2024). Für Start-Ups, die ihre digitale Wertschöpfung nahezu vollständig auf proprietären Anbietern aufbauen, um primär einen schnellen Go-To-Market zu fokussieren, entstehen signifikante strategische Risiken (Chandrasekaran, 2025). Das Geschäftsmodell vieler Wrapper basiert dabei auf einer temporären Arbitrage: Sie bieten UX-Features, die die Basis-Modelle (noch) nicht nativ abbilden (z.B. Prompt-Refinement für einen bestimmten Industriekontext), oder nutzen bestehende Informationsasymmetrien im Markt aus (d.h. die Unkenntnis der Wirtschaft über vergleichbare, günstigere Alternativen) (Huang & Grady, 2023). Das langfristige Risiko einer Blasenbildung erscheint hier imminent. Proprietären Modell-Anbietern wird der strategische Hebel überlassen, durch Anpassung der API-Kostenstruktur oder Funktionserweiterung Konkurrenz schnell aus dem Markt zu verdrängen ("Platform Risk"), sobald diese entscheiden, dass die eigenen Software-Generierungs-Modelle reif für eine aggressive Marktdurchdringung sind (Appenzeller & Li, 2025). Zusätzlich begeben sich Intermediäre neben einer ökonomischen auch in eine technologische Abhängigkeit. Da Wrapper tief mit der spezifischen Funktionalität der proprietären Modelle verwoben sind, können Änderungen des Verhaltens der Basis-Modelle die Funktionalität der Wrapper signifikant beeinträchtigen, was zu volatilen, schwer prognostizierbaren Disruptionen in der Wertschöpfungskette der Wrapper führen kann (ibid.).

mAI positioniert sich daher klar als Infrastruktur-Partner für **Autonomous Enterprise Innovation**. mAI nutzt LLM Modelle lediglich als austauschbaren Motor innerhalb eines festen Chassis (Scaffolding). Mittelfristig soll aber auch

hier eine onPremises Lösung geschaffen werden, um eine nahezu vollständige Unabhängigkeit in der Software-Generierung zu erreichen.

Strategische Implikation für Unternehmen

Für anwendende Unternehmen transformiert sich die Entscheidung für eine deterministische Generierungs-Architektur (wie APA) von einer reinen IT-Frage zu einer fundamentalen "Make-or-Buy"-Strategieentscheidung. Die technische Hoheit über den Generierungsprozess ermöglicht eine Abkehr von starren Lizenzmodellen hin zu einer agilen Asset-Generierung.

\Die Umkehr der Build-or-Buy Rechnung. Final lässt sich der Einsatz von mAI in einer harten ökonomischen Betrachtung (Return on Invest) validieren. Der Hebel wirkt dabei auf zwei Ebenen - Bottom-Line und Top-Line. Die traditionelle ökonomische Logik ("Standardsoftware ist günstiger als Eigenentwicklung") wird durch die drastische Senkung der Grenzkosten in der Code-Produktion invertiert. Heute hat ein durchschnittliches, mittelständisches Unternehmen circa 130 SaaS Produkte (mAI Research), das über die Breite der Belegschaft zu einer enormen Lizenzlast führen kann. Global kann die Zahl extern gekaufter Software auf 312 Millionen mit circa 56 Milliarden aktiver User-Accounts geschätzt werden (mAI Research). Da Frameworks wie mAI die kostspielige "Last Mile" der Implementierung automatisieren, wird Eigenentwicklung ("Build") plötzlich zur ökonomisch rationaleren Option gegenüber generischer Standardsoftware ("Buy"). Dies befähigt Kunden zur "Micro-Competition": Anstatt monatliche Lizenzgebühren für SaaS-Produkte zu entrichten, von denen oft nur ein Bruchteil der Funktionen genutzt wird, können interne Teams hochspezialisierte, schlanke Lösungen generieren. Diese bilden exakt die ureigenen Prozesse ab, ohne technologische Schulden oder externe Roadmap-Abhängigkeiten zu importieren. Die ökonomische Wende gilt ebenfalls innerhalb der "Make"-Perspektive: Die Hürde, einen manuellen Excel-Prozess in eine Software zu gießen, kann vorher bei Entwicklungskosten von über 50.000 € gelegen haben. Durch mAI sinkt diese Hürde massiv. Prozesse, die bisher "zu klein für IT, aber zu groß für Excel" waren (der "Long Tail" der Prozesslandschaft), können nun wirtschaftlich digitalisiert werden, was potenziell tausende, administrativer Arbeitsstunden pro Jahr einsparen kann und Mitarbeiter für wertschöpfende Prozesse freisetzt. Aus Top-Line-Perspektive sind Möglichkeiten die Produktivität und Umsätze zu steigern ebenfalls breit gestreut: Von optimierter churn-management-software in einem Dienstleistungsunternehmen über B2B-Portale zur Einsicht in Lagerstände für einen Logistiker sind zahlreiche Beispiele zum Einsatz von APA denkbar.

\Notwendigkeit von fundiertem Enterprise-Engineering. Die Diskrepanz zwischen dem technologischen Versprechen generativer KI und ihrer betriebswirtschaftlichen Realisierung ist im aktuellen Marktumfeld signifikant. Zwar identifizieren laut KPMG 53% der Entscheidungsträger Prozesseffizienz und 51% Mitarbeiterproduktivität als die primären Werttreiber von Low-Code/No-Code-Initiativen (KPMG, 2024a). Doch der Transfer dieser Chancen in die operative Realität stößt auf strukturelle Barrieren: Während drei Viertel der Unternehmen positive Business Outcomes in initialen KI-Piloten verzeichnen, gelingt es lediglich 31%, diese Lösungen erfolgreich zu skalieren (KPMG, 2024b). Die Ursachen dieses "Scaling Gap" liegen - aus mAI Perspektive - in der fehlenden Enterprise-Readiness der Werkzeuge. Insbesondere unter Betrachtung, dass regulatorische Compliance (38%) und Risikomanagement (32%) als Haupt-Herausforderungen von generativer KI allgemein angesehen werden (Deloitte, 2025), und für low-code spezifisch 42% Sicherheit als Kernbedenken tragen (KPMG, 2024a), scheinen "Thin Wrappers", die auf Go-To-Market-Geschwindigkeit optimiert sind, Gefahr zu laufen Vertrauen zu verspielen. mAI adressiert exakt diese Lücke, indem es sich bewusst vom klassischen Paradigma des "No-Code" abgrenzt und eine Plattform für Autonomous Enterprise Engineering etabliert. Unsere Antwort auf die Compliance-Krise ist eine technologische Orchestrierung im Detail, die Sicherheit und regulatorische Konformität als architektonisches Fundament begreift ("Compliance by Design").

mAI liefert hier nicht nur ein technisch stabiles Fundament made & hosted in Germany, sondern entwickelt sich in den kommenden Monaten zu einer neuen Ära der kooperativen Innovation. Wir richten unsere Plattform mittelfristig stark an den Anforderungen der Enterprise-Welt im Hinblick auf x-Augen-Prinzip & Kollaboration, Sicherheit by Design, und Gewährleistung von Compliance aus - allerdings mit einer neuen Stufe der Schnelligkeit.

Über mAI

mAI ist ein prompt-basiertes „Autonomous Enerprise Engineering“ Plattform-Startup aus Hamburg, das KMUs und Enterprise-Kunden zur Entwicklung maßgeschneiderter Business-Software befähigt. mAI liefert in Minuten produktionsreife Softwarelösungen für Unternehmen, die über Proof-of-Concepts hinausgehen. Während der Markt von No-Code-Tools dominiert wird, die als einfache Wrapper um proprietäre Modelle fungieren, adressiert mAI die fundamentalen Herausforderungen KI-generierter Software: Skalierbarkeit, Wartbarkeit und Enterprise-Compliance.

Dafür nutzt mAI das intern entwickelte APA-Framework (Application Programming Application) das Nutzer von der interaktiven Erfassung der Nutzeranforderungen, über die Integration einer neuartige Produktmanager-Funktionalität zum Ausräumen von fehlendem Kontext bis hin zur Bereitstellung des finalen Sicherheitsbausteins zur Orchestrierung und Generierung von sicherem und wartbarem Code durch den APA-Architekten – alles in einer geführten Umgebung für bestmögliche User-Experience. Unternehmen können mit mAI-APA in kurzer Zeit Software entwickeln, testen und ausrollen. Beim Deplyoment setzt mAI dabei ebenfalls auf bewährte Standards: Unternehmen müssen ihre Software nicht in geteilten Spaces deployen, sondern erleben durch individuelle, unternehmensspezifische virtuelle Maschinen (VMs) mit Serverstandort Deutschland ein neues Niveau an Sicherheit & Vertrauen.

Wir bei mAI wissen, dass generative künstliche Intelligenz Limitationen hat, die wir weder in den kommenden 5 Jahren durch leistungsfähigere Modelle kompensieren können, noch sind es welche, die wir ungezügelt auf kritische Bereiche von Unternehmenssoftware loslassen wollen. Wenn Ihnen die Sicherheit Ihrer Prozesse am Herzen liegt: Schauen Sie gegebenenfalls zwei Mal hin. Modulare, adaptive Intelligenz (mAI) steht für ein solides Gerüst, auf dem man bauen & wachsen kann.

mAI is a prompt-based "Autonomous Enterprise Innovation" platform start-up from Hamburg that enables SMEs and enterprise customers to develop tailor-made business software. mAI delivers production-ready software solutions for companies in minutes that go beyond proof-of-concepts. While the market is dominated by no-code tools that act as simple wrappers around proprietary models, mAI addresses the fundamental challenges of AI-generated software: scalability, maintainability, and enterprise compliance.

About den Autor

Kristof Hackethal ist Tech-Gründer und mit Erfahrung in Management-Beratung und akademischem Hintergrund in den Bereichen KI-Strategie, Mensch-KI-Interaktion und Unternehmenssoftware-architektur. Seine Arbeit verbindet akademische Forschung und praktische Anwendung und basiert auf Forschungserfahrungen an führenden Institutionen wie der ETH Zürich.

Rechtliche Hinweise

EXIST Förderung

Wir sind stolz darauf, vom wissenschaftsorientierten EXIST-Programm gefördert zu werden, das finanziert wird von ...

Dieses Whitepaper dient ausschließlich zu Informationszwecken. Obwohl alle Anstrengungen unternommen wurden, um die Richtigkeit der Angaben zu gewährleisten, übernimmt der Herausgeber keine Verantwortung für Fehler oder Auslassungen. Der Inhalt stellt keine rechtliche, finanzielle oder professionelle Beratung dar.

© Modulare Adaptive Intelligenz (mAI) UG (haftungsbeschränkt).

Alle Rechte vorbehalten.

Kontaktieren Sie Uns

Kontaktieren Sie uns unter team@mai-platform.de

https://mai-platform.com

Quellenangaben

Al Alamin, M. A., Malakar, S., Uddin, G., Afroz, S., Haider, T. B., & Iqbal, A. (2021). An Empirical Study of Developer Discussions on Low-Code Software Development Challenges. *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 46–57. https://doi.org/10.1109/MSR52588.2021.00018

Alharbi, M., & Alshayeb, M. (2026). Automatic Code Generation Techniques: A Systematic Literature Review. *Automated Software Engineering*, *33*(1), 4. https://doi.org/10.1007/s10515-025-00551-3

Altrogge, G. (2018). *Netzplantechnik* (3. Aufl. Reprint 2018). Oldenbourg Wissenschaftsverlag. https://doi.org/10.1515/9783486790405

Appenzeller, G., & Li, Y. (2025). *The Trillion Dollar AI Software Development Stack*. Andreessen Horowitz. https://a16z.com/the-trillion-dollar-ai-software-development-stack/

Berry, D. M., & Kamsties, E. (2004). Ambiguity in Requirements Specification. In J. C. S. do Prado Leite & J. H. Doorn (Eds.), *Perspectives on Software Requirements* (pp. 7–44). Springer US. https://doi.org/10.1007/978-1-4615-0465-8_2

Boehm, B. (1986). A spiral model of software development and enhancement. *ACM SIGSOFT Software Engineering Notes*, *11*(4), 14–24. https://doi.org/10.1145/12944.12948

Bubeck, S., Chandrasekaran, V., Eldan, R., Gehrke, J., Horvitz, E., Kamar, E., Lee, P., Lee, Y. T., Li, Y., Lundberg, S., Nori, H., Palangi, H., Ribeiro, M. T., & Zhang, Y. (2023). *Sparks of Artificial General Intelligence: Early experiments with GPT-4* (arXiv:2303.12712). arXiv. https://doi.org/10.48550/arXiv.2303.12712

Burton, J. W., Lopez-Lopez, E., Hechtlinger, S., Rahwan, Z., Aeschbach, S., Bakker, M. A., Becker, J. A., Berditchevskaia, A., Berger, J., Brinkmann, L., Flek, L., Herzog, S. M., Huang, S., Kapoor, S., Narayanan, A., Nussberger, A.-M., Yasseri, T., Nickl, P., Almaatouq, A., … Hertwig, R. (2024). How large language models can reshape collective intelligence. *Nature Human Behaviour*, *8*(9), 1643–1655. https://doi.org/10.1038/s41562-024-01959-9

Canva. (2024, January). *CIO AI Report Findings*. https://www.canva.com/newsroom/news/cio-ai-report-findings/

Chandramouli, R. (2019). *Security strategies for microservices-based application systems* (NIST SP 800-204; p. NIST SP 800-204). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-204

Chandrasekaran, A. (2025). *The 2025 Hype Cycle for GenAI Highlights Critical Innovations*. Gartner Inc. https://www.gartner.com/en/articles/hype-cycle-for-genai

Chen, Q., Yu, J., Li, J., Deng, J., Chen, J. T. J., & Ahmed, I. (2024). *A Deep Dive Into Large Language Model Code Generation Mistakes: What and Why?*

Deloitte. (2025, January). https://www2.deloitte.com/content/dam/Deloitte/us/Documents/consulting/us-state-of-gen-ai-q4.pdf

Fakhoury, S., Ma, Y., Arnaoudova, V., & Adesope, O. (2018). The effect of poor source code lexicon and readability on developers' cognitive load. *Proceedings of the 26th Conference on Program Comprehension*, 286–296. https://doi.org/10.1145/3196321.3196347

Franceschelli, G., & Musolesi, M. (2025a). Creativity and Machine Learning: A Survey. *ACM Computing Surveys*, *56*(11), 1–41. https://doi.org/10.1145/3664595

Franceschelli, G., & Musolesi, M. (2025b). On the Creativity of Large Language Models. *AI & SOCIETY*, *40*(5), 3785–3795. https://doi.org/10.1007/s00146-024-02127-3

GitHub. (2024). Octoverse: AI leads Python to top language as the number of global developers surges. *The GitHub Blog*. https://github.blog/news-insights/octoverse/octoverse-2024/

Glikson, E., & Woolley, A. (2020). Human trust in artificial intelligence: Review of empirical research. Academy of Management Annals (in press). *The Academy of Management Annals*.

Hellerstein, J. M., Faleiro, J., Gonzalez, J. E., Schleier-Smith, J., Sreekanti, V., Tumanov, A., & Wu, C. (2018). *Serverless Computing: One Step Forward, Two Steps Back* (Version 1). arXiv. https://doi.org/10.48550/ARXIV.1812.03651

Hong, L., & Page, S. (2004). *Groups of diverse problem solvers can outperform groups of high-ability problem solvers*. https://doi.org/10.1073/pnas.0403723101

Huang, J. (2025, January 27). *NVIDIA CEO Jensen Huang's Vision for the Future (Min 23:08)* [Interview]. https://www.youtube.com/watch?v=7ARBJQn6QkM

Huang, J., Chen, X., Mishra, S., Zheng, H. S., Yu, A. W., Song, X., & Zhou, D. (2024). *Large Language Models Cannot Self-Correct Reasoning Yet* (arXiv:2310.01798). arXiv. https://doi.org/10.48550/arXiv.2310.01798

Huang, S., & Grady, P. (2023). *Generative AI's Act Two* [Industry Report]. Sequoia. https://sequoiacap.com/article/generative-ai-act-two/

IEEE. (2018). ISO/IEC/IEEE International Standard—Systems and software engineering – Life cycle processes – Requirements engineering. *ISO/IEC/IEEE 29148:2018(E)*, 1–104. https://doi.org/10.1109/IEEESTD.2018.8559686

Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., Ishii, E., Bang, Y. J., Madotto, A., & Fung, P. (2023). Survey of Hallucination in Natural Language Generation. *ACM Comput. Surv.*, *55*(12), 248:1-248:38. https://doi.org/10.1145/3571730

Kambhampati, S., Valmeekam, K., Guan, L., Verma, M., Stechly, K., Bhambri, S., Saldyt, L., & Murthy, A. (2024). *LLMs Can't Plan, But Can Help Planning in LLM-Modulo Frameworks* (arXiv:2402.01817). arXiv. https://doi.org/10.48550/arXiv.2402.01817

KPMG. (2024a). *Low-code adoption as a driver of digital transformation*. https://assets.kpmg.com/content/dam/kpmgsites/content/dam/kpmgsites/xx/pdf/2024/02/kpmg-low-code-adoption-as-a-driver-of-digital-transformation.pdf.coredownload.inline.pdf

KPMG. (2024b, September). *Global Tech Report*. https://assets.kpmg.com/content/dam/kpmgsites/xx/pdf/2024/09/kpmg-global-tech-report-2024.pdf

Larbi, M., Akli, A., Papadakis, M., Bouyousfi, R., Cordy, M., Sarro, F., & Traon, Y. L. (2025). *When Prompts Go Wrong: Evaluating Code Model Robustness to Ambiguous, Contradictory, and Incomplete Task Descriptions* (Version 1). arXiv. https://doi.org/10.48550/ARXIV.2507.20439

Liu, F., Liu, Y., Shi, L., Yang, Z., Zhang, L., Lian, X., Li, Z., & Ma, Y. (2026). *Beyond Functional Correctness: Exploring Hallucinations in LLM-Generated Code* (arXiv:2404.00971). arXiv. https://doi.org/10.48550/arXiv.2404.00971

Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., & Liang, P. (2024). Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics*, *12*, 157–173. https://doi.org/10.1162/tacl_a_00638

Martínez, E. (2025). Re-evaluating GPT-4's bar exam performance. *Artificial Intelligence and Law*, *33*(3), 581–604. https://doi.org/10.1007/s10506-024-09396-9

McCoy, R. T., Yao, S., Friedman, D., Hardy, M. D., & Griffiths, T. L. (2024a). Embers of autoregression show how large language models are shaped by the problem they are trained to solve. *Proceedings of the National Academy of Sciences*, *121*(41), e2322420121. https://doi.org/10.1073/pnas.2322420121

McCoy, R. T., Yao, S., Friedman, D., Hardy, M. D., & Griffiths, T. L. (2024b). *When a language model is optimized for reasoning, does it still show embers of autoregression? An analysis of OpenAI o1* (arXiv:2410.01792; Version 1). arXiv. https://doi.org/10.48550/arXiv.2410.01792

Mu, F., Shi, L., Wang, S., Yu, Z., Zhang, B., Wang, C., Liu, S., & Wang, Q. (2024). ClarifyGPT: A Framework for Enhancing LLM-Based Code Generation via Requirements Clarification. *Proceedings of the ACM on Software Engineering*, *1*, 2332–2354. https://doi.org/10.1145/3660810

Newman, S. (2021). *Building microservices: Designing fine-grained systems.* O'Reiley Media, Inc.

Niu, B., Song, Y., Lian, K., Shen, Y., Yao, Y., Zhang, K., & Liu, T. (2025). *Flow: Modularized Agentic Workflow Automation* (Version 2). arXiv. https://doi.org/10.48550/ARXIV.2501.07834

OpenAI, Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., Avila, R., Babuschkin, I., Balaji, S., Balcom, V., Baltescu, P., Bao, H., Bavarian, M., Belgum, J., … Zoph, B. (2024). *GPT-4 Technical Report* (arXiv:2303.08774). arXiv. https://doi.org/10.48550/arXiv.2303.08774

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P., Leike, J., & Lowe, R. (2022). *Training language models to follow instructions with human feedback* (arXiv:2203.02155). arXiv. https://doi.org/10.48550/arXiv.2203.02155

Petroni, F., Rocktäschel, T., Riedel, S., Lewis, P., Bakhtin, A., Wu, Y., & Miller, A. (2019). Language Models as Knowledge Bases? In K. Inui, J. Jiang, V. Ng, & X. Wan (Eds.), *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)* (pp. 2463–2473). Association for Computational Linguistics. https://doi.org/10.18653/v1/D19-1250

Reinsel, D., Gantz, J., & Rydning, J. (2018). *The Digitization of the World from Edge to Core*.

Saito, K., Wachi, A., Wataoka, K., & Akimoto, Y. (2023). *Verbosity Bias in Preference Labeling by Large Language Models* (arXiv:2310.10076). arXiv. https://doi.org/10.48550/arXiv.2310.10076

Sharma, M., Tong, M., Korbak, T., Duvenaud, D., Askell, A., Bowman, S. R., Cheng, N., Durmus, E., Hatfield-Dodds, Z., Johnston, S. R., Kravec, S., Maxwell, T., McCandlish, S., Ndousse, K., Rausch, O., Schiefer, N., Yan, D., Zhang, M., & Perez, E. (2025). *Towards Understanding Sycophancy in Language Models* (arXiv:2310.13548). arXiv. https://doi.org/10.48550/arXiv.2310.13548

Shi, F., Chen, X., & Misra, K. (2023). *Large Language Models Can Be Easily Distracted by Irrelevant Context*. ResearchGate. https://www.researchgate.net/publication/367961918_Large_Language_Models_Can_Be_Easily_Distracted_by_Irrelevant_Context

Stackoverflow. (2024). *Technology / 2024 Stack Overflow Developer Survey.* https://survey.stackoverflow.co/2024/technology

The Standish Group. (1995). *The Chaos Report.* ResearchGate. https://www.researchgate.net/publication/263849222_The_Chaos_Report

Turpin, M., Michael, J., Perez, E., & Bowman, S. R. (2023). *Language Models Don't Always Say What They Think: Unfaithful Explanations in Chain-of-Thought Prompting* (arXiv:2305.04388). arXiv. https://doi.org/10.48550/arXiv.2305.04388

Valmeekam, K., Stechly, K., & Kambhampati, S. (2024). *LLMs Still Can't Plan; Can LRMs? A Preliminary Evaluation of OpenAI's o1 on PlanBench* (arXiv:2409.13373). arXiv. https://doi.org/10.48550/arXiv.2409.13373

Wang, P., Li, L., Chen, L., Cai, Z., Zhu, D., Lin, B., Cao, Y., Liu, Q., Liu, T., & Sui, Z. (2023). *Large Language Models are not Fair Evaluators* (arXiv:2305.17926). arXiv. https://doi.org/10.48550/arXiv.2305.17926

Wei, J., Huang, D., Lu, Y., Zhou, D., & Le, Q. V. (2024). *Simple synthetic data reduces sycophancy in large language models* (arXiv:2308.03958). arXiv. https://doi.org/10.48550/arXiv.2308.03958

Zhang, J., Xiang, J., Yu, Z., Teng, F., Chen, X., Chen, J., Zhuge, M., Cheng, X., Hong, S., Wang, J., Zheng, B., Liu, B., Luo, Y., & Wu, C. (2024). *AFlow: Automating Agentic Workflow Generation* (Version 4). arXiv. https://doi.org/10.48550/ARXIV.2410.10762

Zhang, Y., Li, Y., Cui, L., Cai, D., Liu, L., Fu, T., Huang, X., Zhao, E., Zhang, Yu, Chen, Y., Wang, L., Luu, A. T., Bi, W., Shi, F., & Shi, S. (2025). Siren's Song in the AI Ocean: A Survey on Hallucination in Large Language Models. *Computational Linguistics*, *51*(4), 1373–1418. https://doi.org/10.1162/COLI.a.16

Zhang, Z., Wang, C., Wang, Y., Shi, E., Ma, Y., Zhong, W., Chen, J., Mao, M., & Zheng, Z. (2025). LLM Hallucinations in Practical Code Generation: Phenomena, Mechanism, and Mitigation. *Proc. ACM Softw. Eng.*, *2*(ISSTA), ISSTA022:481-ISSTA022:503. https://doi.org/10.1145/3728894

Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E. P., Zhang, H., Gonzalez, J. E., & Stoica, I. (2023). *Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena* (arXiv:2306.05685). arXiv. https://doi.org/10.48550/arXiv.2306.05685