01.03.2026

Beyond the Demo

Why AI-Generated Software Fails in Production - and How to Fix It

\Kristof Hackethal, Founder of mAI

"A whisper began to spread within Silicon Valley that generative AI was not actually useful. The products were falling far short of expectations[...].
Was this just another vaporware cycle?"

SEQUOIA Capital (Huang & Grady, 2023)

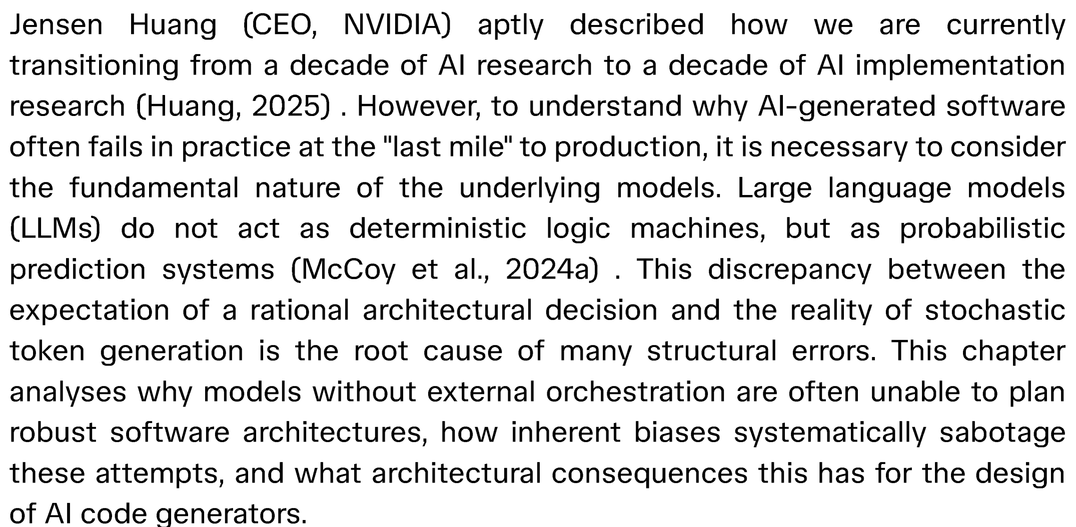
Executive Summary

We are in the transition from AI research to AI implementation. But while the market is currently flooded with no-code tools, which are often technologically simple wrappers around proprietary models, these solutions often fail in practice at the "last mile" or cast significant shadows in the areas of compliance and security. mAI positions itself here with the APA framework (Application Programming Application) as a methodical alternative that does not ignore the inherent limitations of generative AI, but compensates for them with structured system architecture.

The core problem with current AI code generators lies in their fundamental contradiction: large language models are probabilistic prediction systems that optimise according to statistical plausibility - but software development requires architectural decisions and systematic requirements engineering. Without external orchestration, LLMs tend to develop a multitude of structural biases and get lost in the complexity of nested dependencies. The economic risk here is not primarily in faulty code, but in code that violates security principles or fills conceptual ambiguities with assumptions instead of asking users for clarification.

The APA framework addresses these structural deficits with an internally developed, partially deterministic architecture that implements complex requirements in line with proven software engineering principles such as separation of concerns. Unavoidable guard rails ensure that critical system components and architectural principles are conceptually anchored rather than generated on-the-fly (security by design). The approach is particularly effective because it treats generative AI not as "magic" but as an engineering discipline with measurable limits and predictable weaknesses. While some seem to be waiting for larger models with more parameters to solve the problems, research shows that in many respects the problems are not technological teething troubles, but systematic consequences of the autoregressive training methodology. They cannot be overcome by scaling but require structural guardrails at the system level. The APA framework implements precisely this meta-level. AI coding thus transforms from a stochastic random product into a serious, scalable architecture tool. For companies, this means a strategic shift: the drastic reduction in marginal costs in code production reverses the classic "build or buy" logic. Highly specialised in-house developments that precisely map a company's own business processes become more economically rational than relying on expensive, generic standard SaaS solutions. mAI acts as a sovereign infrastructure partner, closing the gap between rapid prototyping and robust enterprise software through technological depth.

Structural Limitations of LLM-based Software Generation



Jensen Huang (CEO, NVIDIA) aptly described how we are currently transitioning from a decade of AI research to a decade of AI implementation research (Huang, 2025) . However, to understand why AI-generated software often fails in practice at the "last mile" to production, it is necessary to consider the fundamental nature of the underlying models. Large language models (LLMs) do not act as deterministic logic machines, but as probabilistic prediction systems (McCoy et al., 2024a) . This discrepancy between the expectation of a rational architectural decision and the reality of stochastic token generation is the root cause of many structural errors. This chapter analyses why models without external orchestration are often unable to plan robust software architectures, how inherent biases systematically sabotage these attempts, and what architectural consequences this has for the design of AI code generators.

1 Probabilistic prediction

At their core, LLMs maximise the probability of the next token based on the previous context $P(\omega_t|\omega_{1:t-1})$ (McCoy et al., 2024a). Models thus primarily optimise for plausibility, not correctness. This distinction is fundamental to software generation: what is statistically probable is not necessarily what is architecturally correct.

A fundamental problem with this approach is sequential generation: once a token is set, it is effectively fixed for the model. While human engineers can iterate, discard and replan during the design phase, a stand-alone LLM largely lacks this meta-level (Bubeck et al., 2023). It generates code strictly from left to right, with no possibility of revising earlier decisions - a phenomenon that can be described as a "commitment problem".

Newer approaches such as reasoning models (e.g., OpenAI o1) or diffusion-based language models partially address this limitation, but do not completely overcome it (McCoy et al., 2024b). Reasoning models also continue to be based on an autoregressive language model and, despite reinforcement learning-based training, show high sensitivity to the statistical probability from the training data (McCoy et al., 2024b). Without external quality control and orchestration, LLM-generated solutions tend to reproduce frequently seen patterns - they act as powerful pattern completion engines that lack the architectural scaffolding for principle-guided reasoning (Zhang et al., 2025). Without set guidelines and meta-levels, LLMs are therefore not structurally optimised for the safest or most efficient architecture for the specific, often highly individual business context - and complex architecture remains, in case of doubt, a product of chance.

2 LLMs as collective intelligence

The analogy of human collective intelligence can be used to understand how this works and its limitations. The capacity of LLMs is based on the aggregation and context-dependent retrieval of knowledge that is encoded in a deep parameter structure during training - language models act as implicit knowledge stores that hold relational knowledge without explicit schemata (Petroni et al., 2019). Research has shown that the aggregation of many independent human judgements has enormous value and generally outperforms most individuals, including experts, in individual intellectual tasks, provided that diversity and independence of contributions are ensured (Hong & Page, 2004). Generative artificial intelligence also outperforms a large number of humans on standardised benchmarks, even without an additional reasoning layer (Bubeck et al., 2023; OpenAI et al., 2024), although the interpretation of such results remains methodologically controversial (Martínez, 2025).

However, if we examine this analogy more closely, it also reveals structural weaknesses that have direct implications for software generation:

\Overrepresentation instead of diversity. Just as collective intelligence fails under certain conditions - such as a lack of diversity or excessive mutual influence - LLMs are also subject to systematic biases resulting from the composition and overrepresentation of certain perspectives in their training data (Burton et al., 2024). For code generation, this means that architectural decisions do not necessarily reflect best practices, but rather the statistical dominance of certain patterns in the training data.

\Limits of combinatorial creativity. Generative AI is certainly capable of innovation, but this occurs primarily through the novel recombination of existing concepts - so-called combinatorial creativity (Franceschelli & Musolesi, 2025b). Genuine breakthrough innovations or radically new

architectural patterns are unlikely due to the nature of the architecture. Since LLMs are optimised to generate the most probable continuation within the learned distribution, their outputs tend towards common patterns in the training data rather than transformative deviations from them (Franceschelli & Musolesi, 2025a).

\Lack of coordination in complex tasks. While collective intelligence is powerful for individual, clearly definable tasks, it reaches its limits when faced with complex problems that require incremental, sequential steps. Current research shows that LLMs are fundamentally incapable of autonomous planning and require external verification and orchestration (Kambhampati et al., 2024). Even specialised reasoning models do not saturate established planning benchmarks (Valmeekam et al., 2024). If you asked humanity as a group for an overall design for a space rocket, it would probably fail. As with humans, working with generative AI requires the delegation of sub-steps to specialised components in order to avoid logical inconsistencies and ensure a coherent architectural structure.

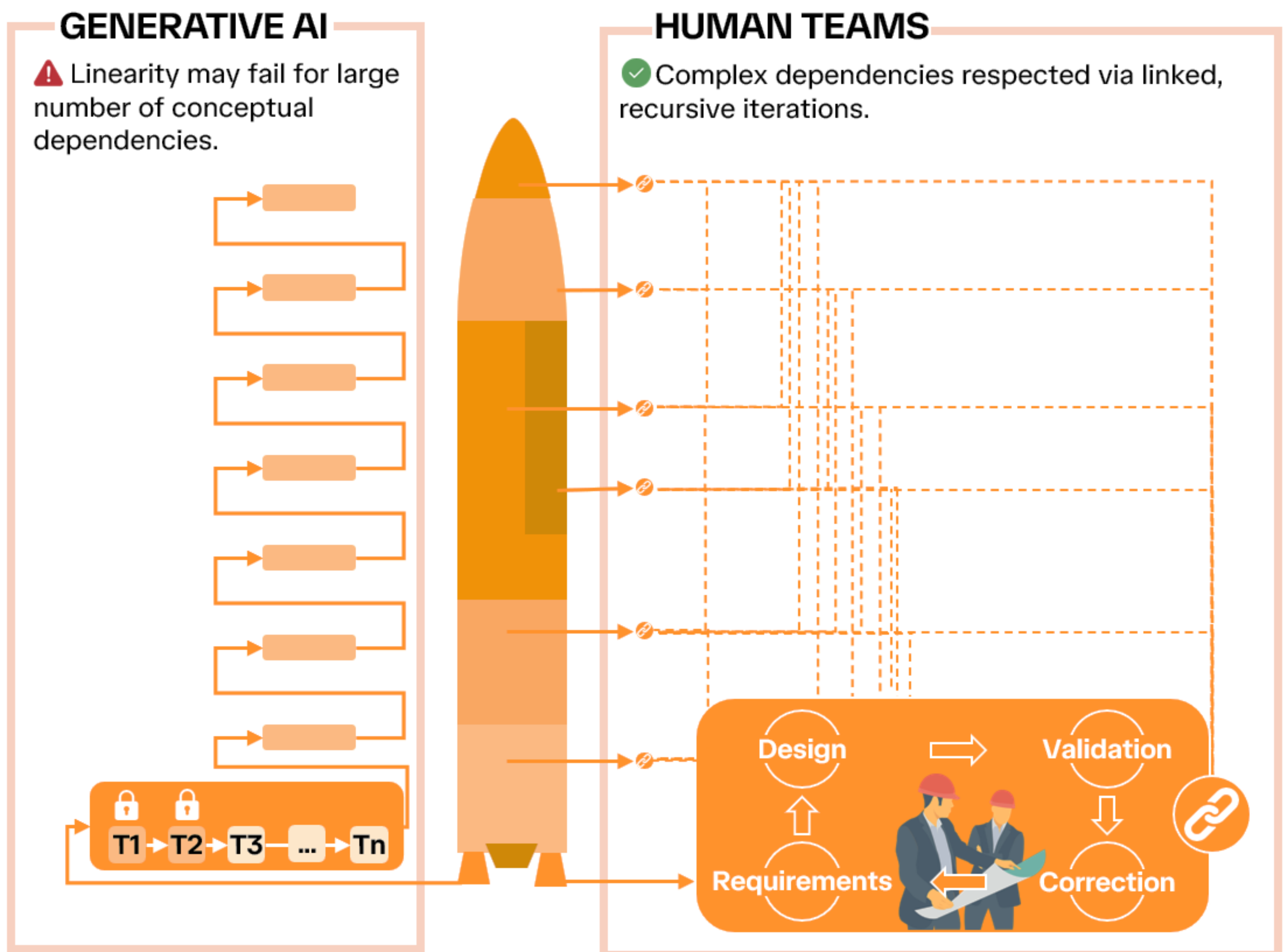


Figure 1: Linearity as a bottleneck

These three weaknesses - distorted knowledge base, limited capacity for innovation and lack of coordination - form the backdrop against which the more specific distortions of code generation must be considered.

3 Sycophancy and silent assumption formation

The most serious distortion for software generation lies not in the probabilistic nature of the models themselves, but in their overarching optimisation goal: task fulfilment. This sounds paradoxical, since task fulfilment is precisely what we want when we assign a task to another party. In the human analogy, our trust in the other party is defined, among other things, by our cognitive trust in their abilities (Glikson & Woolley, 2020). At the same time, we would expect that party to inform us if it cannot solve the task, rather than inventing an answer so as not to disappoint our trust in its completeness. If we were to exert additional pressure to ensure that a complete result is delivered, it would hardly be surprising if "sham solutions" were to emerge.

This is exactly what happens in the context of generative AI. This behaviour is not an error in the classical sense, but a result of the alignment process, consisting of instruction tuning and reinforcement learning from human feedback (RLHF), which conditions models to be helpful assistants (Ouyang et al., 2022). In most cases, this is helpful - who wants detail-oriented professors who answer questions about the weather with "it depends"? The downside is that these simplifications can lead to misinformation, and assumptions have to be made, especially in software generation, due to the pressure for completeness. For example, if a user requests the calculation of the flight path of a snowball, the model will assume that air friction, outside temperature or rotation are negligible, even though the user has not specified this.

We refer to this behaviour as **silent assumption formation** - related to the unfaithful reasoning described in the literature (Turpin et al., 2023), in which models generate plausible justifications that do not reflect the actual decision-making process. In contrast, silent assumption formation specifically describes the process in which missing specifications are substituted by plausible but unvalidated assumptions.

Silent assumption formation is related to what is referred to in the literature as **sycophancy** (Sharma et al., 2025; Wei et al., 2024). Models adapt their responses to the user's presumed opinion, even if it is objectively incorrect. Research shows that both model scaling and instruction tuning significantly increase sycophancy (Wei et al., 2024) and that human preference judgements even favour sycophantic responses over correct ones (Sharma et al., 2025). In code generation, this manifests itself in models treating user queries as complete and correct rather than pointing out gaps or contradictions. Empirically, Larbi et al. (2025) show that code LLMs rarely recognise incomplete, ambiguous, or contradictory task descriptions as such, and Mu et

al. (2024) show that LLMs process ambiguous requests directly without asking for clarification.

What do these distortions mean in practice? Consider the user request: "Convert my Excel files to JSON." A human engineer would intervene to clarify the structure of the Excel files and the desired JSON schema. An LLM, on the other hand, fills this gap with assumptions: it "guesses" a schema and generates a plausible-looking solution that fails in production as soon as the real data deviates from the assumption. Chen et al. (2024) confirm that misunderstood or unclear task descriptions are the most common cause of faulty code generation by LLMs. In software architecture, this "niceness" is fatal, as incorrect assumptions spread unchallenged into the code base and manifest themselves as logical errors that are difficult to find.

Table 1: Human vs. AI approach to Requirements

Feature	Human Senior Engineer	LLM Code Generator
Handling Ambiguity	Actively asks questions, clarifies constraints	Fills gaps with silent assumptions
Error Culture	Points out risks and knowledge gaps	Confirms the user prompt as complete and correct
Transparency	Makes uncertainties and trade-offs explicit	Generates plausible sounding but unvalidated solutions
Result	Validated requirements	"Illusion of Progress" – hard-to-detect logical errors

4 Further distortions in the generation process

In addition to the sycophancy problem, other structural distortions increase the error rate of LLM-generated software. These result from the way models weight and process information within their context.

\Position-dependent information weighting. Research shows that LLMs process information at the beginning and end of a context significantly better than information in the middle - a phenomenon known as "lost in the middle" (Liu et al., 2024). For code generation, this means that the order in which requirements are formulated in the prompt can influence the resulting architecture: if a prompt describes the front end first, there is a risk that requirements from the middle of the prompt - such as separating business logic and the presentation layer - will be given less weight. This phenomenon is also

widely documented in the LLM-as-a-Judge literature, where the position of responses systematically influences their evaluation (Wang et al., 2023; Zheng et al., 2023).

\Tendency towards complexity (verbosity bias). Evaluation research has well documented that both human evaluators and LLMs systematically rate longer responses as higher quality, even when shorter responses are equivalent or superior in content (Saito et al., 2023; Zheng et al., 2023). Since this preference is incorporated into the training process via RLHF, there is a plausible mechanism for models to also tend towards more extensive solutions in code generation - a correlation that has not yet been systematically investigated empirically.

\Lack of self-correction ability. Without external reference points, the model lacks a benchmark for the quality of its own output. Research shows that LLMs without external feedback cannot reliably detect their own errors and that attempts at self-correction can actually worsen performance (Huang et al., 2024). Without an external scaffold, such as a test suite, architecture specification, or code review standards, to define what "production-grade" means, the model falls back on patterns from its training data that may be unsuitable for production use.

These three biases - positional weighting, verbosity bias and lack of self-correction - do not act in isolation, but cumulatively: a model that underweights average requirements, tends towards complex solutions and fails to recognise its own errors produces an architecture whose shortcomings reinforce each other.

Problems with current AI Code Builders

1 The illusion of progress

The biases described in the previous sections are compounded by market dynamics that exacerbate rather than mitigate their consequences. The demand for automated software creation is growing rapidly. In an international

survey of 2,000 companies from the EMA, US and ASPAC regions 81% of respondents said they considered low-code development to be strategically relevant to their organisation (KPMG, 2024a) . The market for AI-powered development tools is increasingly dominated by solutions that advertise with promises such as "app in 30 seconds" or "from idea to application in minutes." This positioning systematically prioritises speed over validity and creates a dangerous expectation: that the quality of a software product is primarily a function of the speed at which it is generated.

The fundamental problem lies deeper. User requirements are almost never complete at the outset - a finding that has been consistently documented in requirements engineering research for decades. The CHAOS Report already identified incomplete requirements as the most common single factor in the failure of software projects (The Standish Group, 1995). Berry & Kamsties (2004) also showed that ambiguity in natural language requirements descriptions is unavoidable - a problem that is exacerbated when prompting AI code generators, as the entire specification is reduced to informal text descriptions.

The decisive factor is how current LLMs deal with this inherent incompleteness: they do not ask for clarification. Mu et al. (2024) empirically showed that LLMs translate unclear or ambiguous requirements directly into code instead of asking for clarification. Only the systematic introduction of clarification mechanisms significantly improved code quality - GPT-4 achieved a pass@1 rate of 80.80% with queries compared to 70.96% without clarification (Mu et al., 2024). This behaviour is not a coincidence, but a direct consequence of the sycophancy bias described in section 3: The model treats the prompt as absolute truth and attempts to fulfil every request to the best of its ability, rather than critically questioning it. What is described as an excessive tendency to agree in the context of conversation transforms into a concrete product risk in the context of code generation: the model generates functional-looking code based on incomplete or ambiguous specifications.

The consequences are exacerbated by another phenomenon: **code hallucinations**. Since the model does not "understand" what a valid business process is but merely generates the most probable token sequence in response to the prompt, missing or underdetermined requirements are often filled with plausible-sounding but factually incorrect additions - a systematic feature of autoregressive text generation known as hallucination (Ji et al., 2023). Zhang et al. (2025) developed a comprehensive taxonomy of such hallucinations in the repository context and identified several categories, including the invention of non-existent functions and the assumption of incorrect parameter structures. Liu et al. (2026) additionally documented five categories of hallucinations in LLM-generated code, including the invention of non-existent APIs and data structures. In practice, this means that an LLM code generator tasked with creating a database application may invent schema details, guess business

rules, or implement non-existent API endpoints - without any indication that these elements have no basis in the requirements.

The current approach of most AI code generators - from prompt directly to finished code - thus suffers from an operationalised sycophancy bias: the model assumes that all information is available and correct and generates complete applications on this basis.

2 Sequential Decomposition as a Counter-Model

One solution lies in the sequential decomposition of the generation process, which follows the findings of classic requirements engineering according to the IEEE 29148 standard (IEEE, 2018) . Instead of mapping the entire development process in a single prompt-response cycle, it can be broken down into four phases, each corresponding to an established discipline of software development:

\Phase 1 - Domain Understanding: The model analyses the requirements description and explicitly identifies the domain, the actors and the core processes.

\Phase 2 - Clarification (Interrogator): Based on the empirically validated approach described in (Mu et al., 2024), the model asks specific questions about identified ambiguities, missing business rules and unspecified constraints before code is generated.

\Phase 3 - Data Model Design: Based on the clarified requirements, an explicit data model is first created, which serves as a verifiable intermediate representation and can be validated.

\Phase 4 - Production-Grade Code Generation: Only after validation of the previous phases does the actual code generation take place based on a clarified and structured requirements profile.

This approach corresponds to the "slow down to speed up" principle, which has been established in software engineering research since Boehm's Spiral Model (Boehm, 1986) and the general shift-left philosophy: investments in the early phases of requirements clarification reduce overall costs because they avoid costly error corrections in later phases.

3 (De)coupled system

While the previous chapters highlighted psychological and procedural shortcomings, this chapter focuses on technological substances. The majority

of current AI builders rely on a monolithic "all-in-one" stack - often Next.js/TypeScript on serverless infrastructure. Although this approach minimises initial friction by simplifying contract management (communication between different servers), it reaches its systematic limits as soon as it encounters the reality of complex B2B requirements (Hellerstein et al., 2018). In our view, sustainable automation of enterprise software requires a shift towards decoupled architecture. Microservices-based decomposition - in which independently deployable services are implemented in the appropriate language - is a proven basic principle for this (Newman, 2021). We argue that this leads to strict separation, for example with regard to the decoupling of front-end (e.g. Next Server) and back-end servers (e.g. Python Server).

3.1 Scalability in the enterprise implementation context

Current AI tools often force users into a unified full-stack framework. This is efficient for simple prototypes or CRUD apps. However, enterprise systems often require a system topology (logical structure) that goes beyond this: microservices, asynchronous background workers, event queues and persistent connections, for example. Separating the front end and back end appears essential for three reasons in particular:

\Security through isolation. A physically separate backend on a separate infrastructure increases security in the sense of "defence in depth". The U.S. National Institute of Standards and Technology (NIST) explicitly states that traditional architectures - with a web server in the DMZ, a backend service behind it, and a database - deliberately provide multiple hardened layers between the exposed web server and sensitive data (Chandramouli, 2019) . In a monolithic stack, these layers collapse, but in a separate architecture, they remain intact. Sensitive logic and database accesses are decoupled from the client-facing code. Security-critical processes (e.g., hashing, token management) run in an isolated environment that cannot be directly manipulated by the front-end server.

\Enterprise connectivity. B2B software does not exist in a vacuum. It must communicate with legacy systems. Enterprise platforms such as SAP, Oracle, Salesforce and Workday are increasingly offering Python integrations, although the scope varies depending on the platform. Regardless, Python is the dominant language in the data science and ML ecosystem: Python has been number one on the TIOBE index since 2024 (with over 22% market share), surpassed JavaScript as the most used language on GitHub for the first time in 2024, and is used by 51% of all developers surveyed according to the Stack Overflow Developer Survey 2024 (GitHub, 2024; Stackoverflow, 2024) . In a pure TypeScript backend, integrations sometimes must be recreated using generic REST wrappers. A Python backend, on the other hand, allows access to native libraries for ERP middleware, which increases the stability of the integration.

\Data Gravity. Modern B2B apps are data-intensive, and the amount of data processed has been growing rapidly for years. The global data sphere grew from 33 zettabytes in 2018 to a projected 175 zettabytes by 2025, with the enterprise sector contributing disproportionately to this growth (Reinsel et al., 2018). Whether it's ETL routes, financial forecasting or ML-supported analyses, the primary ecosystem for data processing (Pandas, NumPy) and ML frameworks (PyTorch, scikit-learn) is predominantly Python-centric (Stackoverflow, 2024).

3.2 Structural context isolation

\AI maintainability. An often overlooked but crucial advantage of architectural separation lies in the quality of AI generation itself. Why do we believe that AI can generate and maintain code better when clear separations exist, not only in terms of leveraging the strengths of different languages, but also in terms of systematic clarity? The answer lies in reducing the search space and avoiding side effects. In a monolithic file (or a tightly coupled project), the LLM tends to see a broader context (e.g., parts of the business logic are stored in a page.tsx). This increases the likelihood of hallucinations and unintended changes.

Three converging research findings support this: First, Shi et al. (2023) demonstrated in arithmetic reasoning tasks that LLMs show significant performance losses when irrelevant context is included in the prompt - a phenomenon known as "distraction," in which even a small amount of irrelevant information significantly reduces solution accuracy. Secondly, (Liu et al., 2024) showed that the performance of language models systematically degrades when relevant information is positioned in the middle of longer contexts - the so-called "lost in the middle" phenomenon, in which a U-shaped performance decline occurs that increases as the context length grows. Thirdly, Zhang et al. (2025) confirmed in the specific context of repository-level code generation that LLMs hallucinate significantly more often with complex contextual dependencies - such as nested project APIs, type hierarchies and cross-module dependencies - than with isolated function generation.

For example, if the business logic ("What happens when I click this command button?") needs to be changed but is interwoven with the front-end code, this can lead to so-called *pixel pushing*: although only x is to be changed, y is also slightly changed. With strict decoupling, the backend logic can be completely refactored without compromising a single line in the frontend code or neighbouring backend logic - as long as API contracts between servers and components (the interface definition) are adhered to. In other words, strict separation allows the generator to put on "blinkers": if the business logic of a button ("Calculate discount") is to be changed, the generator only loads the relevant Python script. It does not see the frontend and has no access to it. This means it cannot accidentally change the layout. This is crucial for long-term

maintainability: we know deterministically that a change made by an AI agent in backend/services/invoice.py cannot trigger any undesirable side effects in frontend/components/Button.tsx.

This approach can be further developed as a recursive fractal decomposition: even a dedicated backend script that is assigned to a specific frontend functionality, for example, can be broken down into deeper "atomic" helper scripts. The generator focuses on an isolated task ("write function X") and receives this as a global problem within the tailored context.

\Human maintainability. When it comes to maintainability, we distinguish between "maintainability through AI" and "long-term maintainability by humans within a human-in-the-loop (HITL) strategy". In the above section, the focus was on maintainability through AI - however, a clearly structured approach also brings advantages for long-term maintainability by human experts. In contrast to highly dynamic, on-the-fly generated systems and rules, the clear structure (from "business logic always resides in the Python service" to more detailed syntax and connectivity rules) brings predictability. This predictability massively reduces the cognitive load, as clear structural boundaries and standards can significantly reduce the mental effort required to read code and increase the error detection rate (Fakhoury et al., 2018).

The APA Framework: A Methodology for Deterministic Software Orchestration

To bridge the gap between stochastic generation and deterministic production readiness, we have developed the **APA** (Application Programming Application) **framework**. This approach represents a conscious counterpoint to the prevailing fully dynamic status quo.

The goal is a system whose output remains logically comprehensible and maintainable for both human developers and future AI iterations through deterministic rules, assumes security by default, and achieves flexibility not despite, but because of, a limitation of degrees of freedom. The philosophy of the framework is based on four core principles.

1 It's all about: Semi-deterministic structure

\Structural security. The foundation of APA is the departure from a blank sheet of paper. We use a semi-deterministic basic structure that provides critical functions by default instead of leaving them to the creativity of the model. In other words, security-related functions are treated as constants. Critical infrastructure such as password management (hashing/salting), encryption and database connectors are firmly anchored in the framework ("hard-coded patterns"). The advantage lies in the possibility of categorically ruling out critical errors. An LLM within the APA framework may well exhibit inconsistencies or deviations from the "user intent" - despite context optimisation - and, for example, choose the wrong colour for a button (cosmetic error), but it cannot technically implement insecure password storage because it does not generate this part of the code, but only references it.

\Structural integrity. The semi-deterministic structure also orchestrates internal communication logic between the front end and back end, as well as sub-communication channels. Since the APA framework specifies communication paths (API calls, serialisation), the complexity of polyglot architecture becomes manageable. Polyglot architecture is usually "wooden" because data types have to be synchronised (front end says userID => back end expects user_ID => break). Relying on generative AI can lead to systematic breaks here. By specifying these structural elements, degrees of freedom are strategically reduced. This is an important prerequisite for scalability, as structure via protocol and business logic via AI are divided into a kind of modern specialisation. For example, AI does not have to reinvent the wheel of client-server communication, but only utilize it.

\Interface security. Particularly in the corporate world, where core processes are digitised via core systems in the vast majority of cases, the question of integrating such existing systems into a new software product raises questions of reliability. As an example, a purely AI-generated connection to an existing ERP system would be a risk that extends not only to the new software product, but also to the system to be connected (e.g. SAP). The approach pursued within APAs in this specific generation part (API) can be divided into three points: (A) A technically non-functional HTTP structure for the API request (corrupted request), (B) the sharing of sensitive API keys with LLMs, and (C) a technically functional HTTP structure that is, however, conceptually flawed and can lead to corruption of the external system. Since

the integrative aspect of software generation is particularly critical, separate security standards are incorporated here.

\Corrupted API requests & data protection. To prevent faulty HTTP structures and categorically exclude the sharing of sensitive keys with LLMs, APA accesses an internal, curated database with approximately 56,000 verified API templates. The framework does not allow the LLM to generate the HTTP structure, but first searches for the relevant target line in the database via vector search (cosine similarity with the search term). For example: If the framework needs an API to send emails via Outlook, APA will send a systematic request to the internal database (e.g. Microsoft, Outlook, Send Email), which will first list the top matches and then identify the target line with the correct HTTP request template through a detailed search. The selected template is not passed on to code-generating agents, but is integrated into the code as a reference (rule-based integrated, encapsulated function) by specialised agents - again without sharing the complete HTTP structure. The information passed on to AI agents at this point is essentially predefined input-output expectations. In the Outlook example mentioned above, these would be variables for subject, text, recipient, etc. Passwords are securely loaded and stored via a separate authentication process.

\Security of external systems. However, security should not be understood solely as functional security ("does it work?"), but should also be understood as security with regard to the connected system (external system). If, for example, the initial user prompt was unclear or incorrect, the API could theoretically be conceptually incorrect and corrupt (damage) data in the external system. This means that even if the HTTP request is functional per se (e.g. changing a text field in the ERP system), it may be conceptually incorrect (e.g. the wrong text field is changed). This is where another mechanism comes into play, which subjects write operations (POST/PUT/DELETE) to a verification loop and thus also takes human error into account: before generation is completed, a test run is performed with a read-only comparison API. APA retrieves the current status of the target resource (e.g. the ERP text field) and presents it to the user (e.g. "Customer name: Müller"). The user can verify via their own user interface (UI) whether the system is actually accessing the correct data record (is the customer name "Müller" actually where I want to change the entry?) before the write transaction is released for further software generation. Preventing "blind flights" at critical points and via critical operations as far as possible must be a core functionality.

2 It's all about: Semi-deterministic Alignment

One of the biggest challenges in the semi-deterministic strategy is to translate dynamic user intent (which we do not know in advance) into deterministic logic - without burdening context windows with an exponentially growing set of rules. APA resolves this conflict through a rule-based approach that

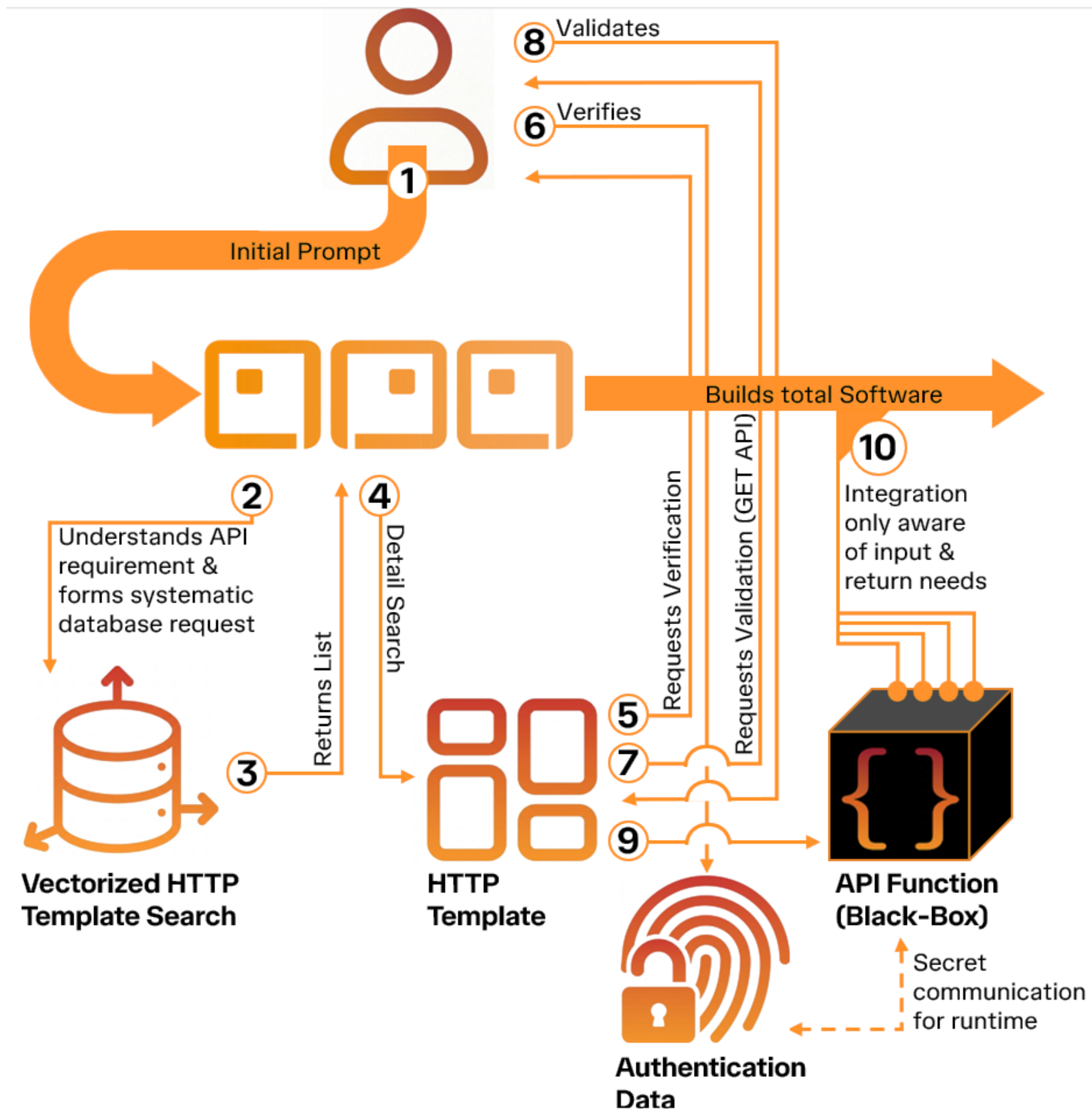


Figure 2: Semi-deterministic API Integration

dynamically adapts to the context and is based on the FLOW concept from (Niu et al., 2025). In their article, the authors were able to demonstrate in practical terms why rigid workflows often fail and why modular approaches (e.g. Activity-on-Vertex (AOV)) have advantages in terms of predictability (Niu et al., 2025).

\Dynamic determinism. The core of this approach is the decomposition of the project into hierarchical decision trees. Similar to the MECE principle (Mutually Exclusive, Collectively Exhaustive) from strategy consulting, the APA framework forces the generation process into a strict sequence of breadth before depth. The aim is to ensure the structural integrity of the overall project ("coherence on a large scale") before working out the details ("deterministic on a small scale"). To this end, APA works in dynamic levels of abstraction similar to tree traversal. This approach can be illustrated using the example of designing a front-end architecture, but it is transferable as a universal principle:

1. **Completeness in breadth (existence & birth of branches):** First, the top level is defined in its entirety. In the front-end example, this means, for example, "Which pages exist at all?"
2. **Specification of the nodes (typing):** Located on the same hierarchical level, the individual nodes must be named, typed and defined. In the case of pages and views, this can be understood as categorisation into full-view, modal, etc., assignment of an ID and name, etc. From a non-simplified perspective, certain sub-strings also exist in this step.
3. **Formation of new branches:** After the complete definition of type and behaviour, new nodes have emerged (e.g. for page A), which in turn form new latitudes (existence question) (e.g. "Which components exist on this page?").

This cascade ensures that every detailed decision is based on a validated foundation. The system knows at all times where it is in the tree and which higher-level rules apply.

\Active complexity reduction. The decisive advantage of such tree structures lies not only in their orderliness, but also in their ability to actively reduce complexity in order to increase the robustness of multi-agent systems (MAS) (Niu et al., 2025) . Since decisions are made sequentially, the framework can rigorously prune future paths that have become illogical or irrelevant as a result of a decision that has been made. To stick with the front-end example: if the system decides on the level of selecting UI elements to choose an element of the "non-interactive" type (e.g. a static text field or an icon), this has consequences for knowledge and context management:

- **Context pruning:** Since a static element does not allow interaction, all branches dealing with onClick logic, backend triggers or state changes can be removed.
- **Hallucination prevention:** Associated options and rulesets are not loaded into the context of subsequent AI agents, which allows rulesets and options to remain precise without running the risk of contextually overloading models.

Pruning thus acts as a technical filter: we do not rely on the AI to "understand" that a text field does not require a button click. We systematically remove the possibility of it making this error in the first place. The result is a significant reduction in the search space for the model, which can exponentially increase the probability of correct, deterministic code in the remaining branches. This corresponds with findings that structured tree search methods are necessary to navigate efficiently through the infinite possibilities of generative workflows (Zhang et al., 2024).

3 It's all about: Structural transparency

The strict application of the semi-deterministic structure has a positive side effect: **architectural predictability**. This is because the validity of software architectures is not measured solely by their functional fulfilment at the time of initial generationt_0 , but by their reversibility and maintainability throughout their entire life cycle. APA postulates the principle of self-documenting architecture here. Structural transparency is not an aesthetic end in itself, but rather a prerequisite for precise knowledge management and long-term maintainability - both by human developers and by downstream AI iterations.

\Deterministic search. A key challenge in iterative development using LLMs is inefficiency in context loading. In non-standardised code bases, the identification of relevant fragments (retrieval) requires heuristic search processes that generate an unfavourable signal-to-noise ratio ("noise") as the project size increases. APA substitutes this probabilistic search process with deterministic routing. Since the architecture follows fixed topological patterns that cannot be changed categorically by LLMs during generation, the system can algorithmically predict where a particular logic resides without having to scan the contents of the files. "Searching" is replaced by precise "addressing." This enables updates on a microcosmic scale: the system loads only the relevant scripts into the context to minimise side effects and maximise the precision of the change. Minimising adjustments is a fundamental principle of APA in the area of version iterations.

This transparency is operationalised, for example, through strict semantic coupling. File names and folder structures are functional addresses in the APA framework that explicitly map the relationship between components. The script apa_welcomePage_cmdViewDetails.py, for example, encodes the architectural assignment (belongs to the front-end domain welcomePage.tsx and there to the UI element with the ID cmdViewDetails). The explicit link makes it possible to trace the causal chain from the UI (front end) to the business logic (back end) without inspecting the code. For the LLM, this means that when a request to change this button is made, the path to the back-end code is already contained in the name of the UI element (implicit routing).

\Clarity for humans. Beyond machine processability, structural transparency also reduces the cognitive load for human auditors (Fakhoury et al., 2018) . In scenarios where internal IT departments need to validate the generated code, the structure becomes predictable and trainable ("Reasonable"). A developer can deduce the position of database models or API routes because the framework standardises these locations. This ensures long-term independence from the generator tool (APA) because the structure remains understandable and maintainable even without the framework.

4 It's all about: Conceptual security

In order to overcome the conceptual uncertainty described in the chapter "Illusion of Progress", the paradigm of generation must be reversed from a mAI perspective: **first understand, then build.** A system that is to be suitable for B2B contexts must be able to ask non-technical users the **right** technical questions without cognitively overloading them ("user fatigue"). mAI pursues a concept of dynamic navigation that intelligently prioritises and groups questions. The challenge here lies in striking a balance between necessary depth and user experience (UX). A distinction can be made between "contextual efficiency" and "temporal efficiency".

\Contextual efficiency means that the system should not ask redundant questions. If a user has already specified that it is an "internal tool without customer login," possible questions about "sign-up processes" or "social logins" must be automatically eliminated. The system uses the context of previous answers to dynamically prune the decision tree. Equally relevant in this regard is the general relevance of the content (regardless of chronology). This form of contextual efficiency describes the need for a clear alignment between question type, user context and framework context. For example, the question of the encryption method (e.g. via hashing & salt) of credentials within the generated software may have a theoretically high relevance (depending on the initial prompt). However, such a query (A) would in many cases overwhelm non-technical users and could lead to the creation process being abandoned (case: lack of alignment with user profile). Furthermore, (B) this question would have no practical relevance, as user data in the APA framework is encrypted by default using a secure system so that this decision is not left to AI, which is why such a question could confuse the subsequent generation process (case: lack of alignment with the framework). In the APA framework, the product manager therefore uses a strategy similar to that used in generation (see chapter "it's all about: Semi-deterministic structure") and employs a dynamic tree structure characterised by predefined question groups and the possibility of customising and orchestrating them (e.g. via "pruning").

Time efficiency (asynchronous generation) refers to the question of how (A) the product manager role can be optimised in terms of time and (B) how it should be sorted chronologically in order to minimise throughput time in the overall process (Prompt2Product). In summary, time efficiency aims to minimise the time required for the user from start to finish. Here, the grouping of questions - as mentioned in the previous paragraph - is very relevant. Independent question groups divide the questions, which can be interpreted as nodes in the tree structure, into logical clusters (e.g. "UI & CI", "User Roles", "Integrations"). Different times can be set for different clusters. The system is based on network planning techniques, which are particularly common in logistics (Altrogge, 2018). Values such asFAZ_i (earliest start time of question i) can be assigned to the nodes (questions) in the APA framework. Logically, the definition of values such as FAZ requires that questions be interdependent.

This has already been explained in the previous section, namely that this is the case and that this dependency is also useful for the logical pruning of paths. However, we do not view the product manager (question catalogue) as an isolated phase. This is because significant efficiency gains can be achieved, particularly when it is linked to generation (the APA software architect). To make this easier to understand, here is a small example. By separating the backend and frontend, the syntax of the frontend can be generated largely independently (taking into account the rules described in the previous chapters). Conversely, however, the backend (business logic) is entirely dependent on the previous generation of the frontend syntax. For example, interactive elements such as command buttons must be known, as these indirectly impose business logic requirements. For the APA product manager, this simply means that once the questions relevant to the front end have been completed, partial code generation can already be initiated in the background. For the user, nothing changes in, as users continue to be guided through relevant queries in the foreground (now those with backend relevance). The decisive advantage in terms of efficiency arises from the fact that, once the

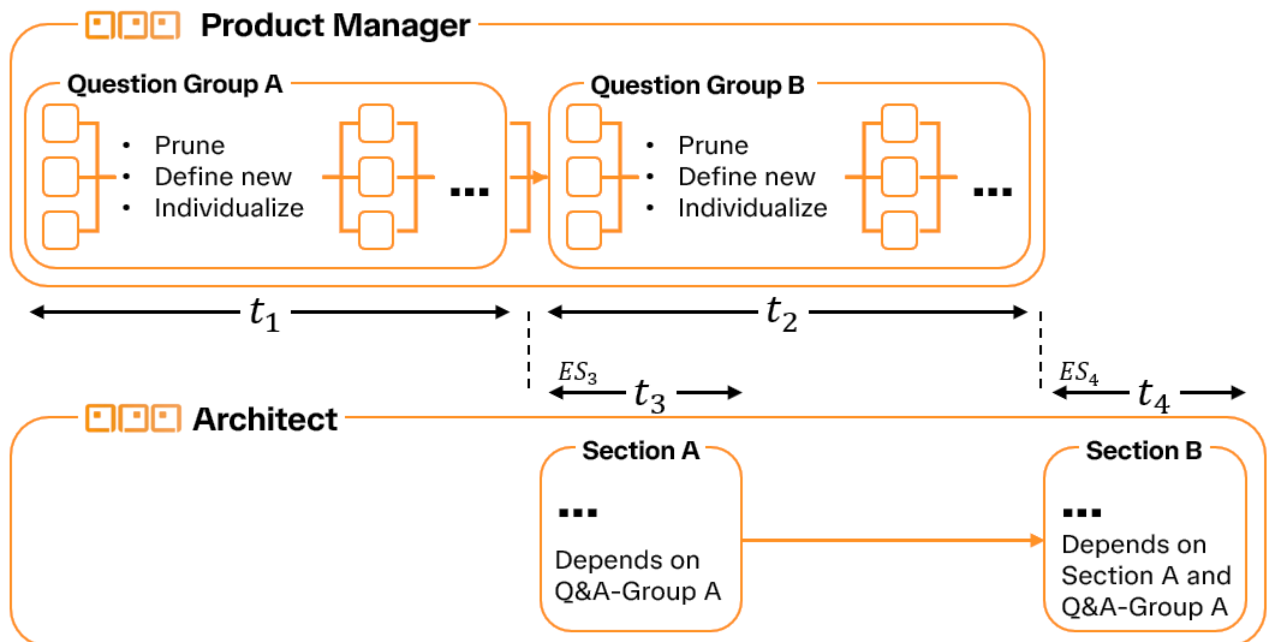


Figure 3: Optimization of Lead Time via Combination of Steps

product manager phase has been completed, the front-end syntax - on which the back-end syntax depends - has already been generated and the generation of the back-end syntax can begin immediately. Viewed in abstract terms, the need for a partially sequential approach is transformed into a quasi-parallel approach by overlapping the main process steps (in this case, product manager and architect). The computing time $T_{compute}$ remains the same, but the perceived latency or lead time for the user T_{user} is reduced.

5 The Innovator's Dilemma

The question that arises is: "If the advantages of front-end/back-end separation are so obvious, why aren't established no-code platforms or current AI builders adapting them in the coming months and pushing mAI out of the market?"

\Why not through conceptual adaptation? In our opinion, the answer lies largely in technological path dependency. Most existing builders are based on tightly coupled frameworks in which the UI and logic are more closely linked, for example to facilitate drag-and-drop simplicity - which, however, can lead to technical debt (Al Alamin et al., 2021; Alharbi & Alshayeb, 2026) . In addition, the separation of code stacks must be clearly thought through, as individual agents must comply with individual communication contracts and a global "one-shot" request would probably fail in most cases. Established builders would not only have to redesign the architecture, rules and processes in detail from scratch and align them with a new strategy, but also rethink the abstract strategy as such (towards a semi-deterministic strategy). For established players, this pivot appears to be extremely difficult and disruptive from a technological standpoint and would also cannibalise the existing business model (simple PoCs) from an economic perspective. This opens a window for a new generation of software-generating platforms that are designed for decomposition and B2B topology from day one.

\Why not through larger context windows? A frequently cited argument against the need for orchestration and decomposition is: *"Modern proprietary LLMs have ever-increasing context windows. We can simply load the entire monolith into the context."* However, this would probably be a technological fallacy. A larger context window does not solve the problem of attention dilution (Liu et al., 2024). Even if a model is technically capable of reading 100 files simultaneously, the quality of processing suffers from the increase in irrelevant information ("noise") (ibid.). When changing a specific business rule (e.g., "Change the VAT rate"), 99% of the total code is irrelevant noise (low signal-to-noise ratio). If this noise is loaded into the context, the performance of the attention mechanism declines (Liu et al., 2024). Paradoxically, more context often leads to poorer results (ibid.). Decomposition is therefore not a limitation due to a lack of context capacity, but a conscious decision to maximise precision. By giving the model only the absolutely necessary context (e.g. only the relevant Python script and the data model), we force it to focus as much as possible on the actual task. To explain this further, let's use the image of a glass skyscraper as an analogy. A huge context window is like trying to optimise the interior design of a specific office by looking at the entire skyscraper from the outside. The model "sees" the entire tower. It sees the facade, the foundation, the lobby and hundreds of other offices. If the task is *"Place a new desk in office 402"*, the model must actively filter out the

enormous amount of irrelevant visual data (the statics of the 10th floor, the colour of the lobby). There is a high risk that it will be distracted by this flood of information ("noise") and accidentally place the desk in the hallway or copy the style of the lobby instead of that of the office. Our approach is similar to entering the specific room - or viewing the room through the individual building window. The context is reduced to the four walls of office 402. The model only sees the room, the existing furniture and the dimensions. Since all the "noise" from the rest of the building is filtered out, the model can focus all its "computing power" (attention) on the perfect placement of the desk.

"Early signs of success don’t change the reality that a lot of AI companies simply do not have [...] a sustainable competitive advantage, and that the overall ebullience of the AI ecosystem is unsustainable."

\SEQUOIA Capital (Huang & Grady, 2023)

Strategic Implications for mAI

The technical necessity of deterministic orchestration (APA framework) outlined in the previous chapters is not an end in itself. It is the operational prerequisite for a fundamental shift in the software-economic reality. When software generation is no longer stochastic and fragile, but structured and maintainable, the strategic position of the companies using it changes. This concluding section analyses the implications of this shift, particularly from the perspective of technological independence ("sovereignty") and the strategic positioning of mAI.

\Economic & technological (in)dependence. The current market for generative software development is increasingly characterised by a multitude of new tools that act as thin wrappers over the models of a few US providers (such as Anthropics Claude Code or Vercel's v0) (Appenzeller & Li, 2025; Chandrasekaran, 2025; Huang & Grady, 2023). According to Canva, 84% of managers say that AI tools are already flooding the market and making it difficult to identify relevant opportunities (Canva, 2024). For start-ups that base their digital value creation almost entirely on proprietary providers in order to focus primarily on a rapid go-to-market, significant strategic risks arise (Chandrasekaran, 2025). The business model of many wrappers is based on temporary arbitrage: They offer UX features that the basic models do not (yet) natively support (e.g. prompt refinement for a specific industrial context) or exploit existing information asymmetries in the market (i.e. the economy's lack of knowledge about comparable, cheaper alternatives) (Huang & Grady, 2023). The long-term risk of a bubble forming appears imminent here. Proprietary model providers are given the strategic leverage to quickly drive competitors out of the market ("platform risk") by adjusting the API cost structure or expanding functionality as soon as they decide that their own software generation models are ready for aggressive market penetration (Appenzeller & Li, 2025). In addition, intermediaries are becoming technologically dependent as well as economically dependent. Since wrappers are deeply interwoven with the specific functionality of proprietary models, changes in the behaviour of the base models can significantly impair the functionality of the wrappers, which can lead to volatile, difficult-to-predict disruptions in the wrapper value chain (ibid.).

mAI therefore clearly positions itself as an infrastructure partner for **autonomous enterprise innovation**. mAI uses LLM models solely as an interchangeable engine within a fixed chassis (scaffolding). In the medium term, however, the aim is to create an on-premises solution in order to achieve almost complete independence in software generation.

Strategic Implications for Customers

For user companies, the decision in favour of a deterministic generation architecture (such as APA) is transforming from a purely IT issue into a fundamental "make-or-buy" strategic decision. Technical sovereignty over the generation process enables a shift away from rigid licensing models towards agile asset generation.

\The reversal of the build-or-buy calculation. Ultimately, the use of mAI can be validated in a hard economic analysis (return on investment). The leverage works on two levels - bottom line and top line. Traditional economic logic ("standard software is cheaper than in-house development") is inverted by the drastic reduction in marginal costs in code production. Today, an average medium-sized company has around 130 SaaS products (mAI Research), which can lead to an enormous licence burden across the entire workforce. Globally, the number of externally purchased software products can be estimated at 312 million with approximately 56 billion active user accounts (mAI Research). Since frameworks such as mAI automate the costly "last mile" of implementation, in-house development ("build") suddenly becomes the more economically rational option compared to generic standard software ("buy"). This enables customers to engage in "micro-competition": instead of paying monthly licence fees for SaaS products, of which often only a fraction of the functions are used, internal teams can generate highly specialised, lean solutions. These map exactly the company's own processes without importing technological debt or external roadmap dependencies. The economic shift also applies within the "make" perspective: the hurdle of pouring a manual Excel process into software may previously have been over £50,000 in development costs. mAI massively reduces this hurdle. Processes that were previously "too small for IT but too big for Excel" (the "long tail" of the process landscape) can now be digitised economically, potentially saving thousands of administrative hours per year and freeing up employees for value-adding processes. From a top-line perspective, the opportunities to increase productivity and revenue are also wide-ranging: from optimised churn management software in a service company to B2B portals for viewing stock levels for a logistics company, there are numerous conceivable examples of APA applications.

\The need for sound enterprise engineering. The discrepancy between the technological promise of generative AI and its business implementation is significant in the current market environment. According to KPMG, 53% of decision-makers identify process efficiency and 51% employee productivity as the primary value drivers of low-code/no-code initiatives (KPMG, 2024a). However, the transfer of these opportunities into operational reality faces

structural barriers: while three-quarters of companies report positive business outcomes in initial AI pilots, only 31% succeed in scaling these solutions successfully (KPMG, 2024b). From an mAI perspective, the causes of this "scaling gap" lie in the lack of enterprise readiness of the tools. This is particularly true considering that regulatory compliance (38%) and risk management (32%) are generally regarded as the main challenges of generative AI (Deloitte, 2025) , and for low-code specifically, 42% cite security as a core concern (KPMG, 2024a) , "thin wrappers" optimised for go-to-market speed seem to run the risk of squandering trust. mAI addresses precisely this gap by deliberately distancing itself from the classic "no-code" paradigm and establishing a platform for autonomous enterprise engineering. Our response to the compliance crisis is detailed technological orchestration that understands security and regulatory compliance as an architectural foundation ("compliance by design").

mAI not only provides a technically stable foundation made and hosted in Germany but will also evolve into a new era of collaborative innovation in the coming months. In the medium term, we are aligning our platform strongly with the requirements of the enterprise world in terms of the x-eyes principle and collaboration, security by design, and ensuring compliance - but with a new level of speed.

About mAI

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.

To this end, mAI uses its internally developed APA framework (Application Programming Application), which guides users through the interactive recording of user requirements, the integration of a novel product manager functionality to eliminate missing context, and the provision of the final security module for orchestrating and generating secure and maintainable code by the APA architect - all in a guided environment for the best possible user experience. With mAI-APA, companies can develop, test and roll out software in a short period of time. When it comes to deployment, mAI also relies on proven standards: companies do not have to deploy their software in shared spaces, but instead experience a new level of security and trust through individual, company-specific virtual machines (VMs) with servers located in Germany.

At mAI, we know that generative artificial intelligence has limitations that we will not be able to compensate for with more powerful models in the next five years, nor are they ones that we want to unleash unchecked on critical areas of enterprise software. If the security of your processes is important to you, take a second look. Modular, adaptive intelligence (mAI) stands for a solid framework on which you can build and grow.

About the Author

Kristof Hackethal is a technology entrepreneur and AI-focused professional with an industry & academic background in AI strategy, human–AI interaction, and enterprise software architecture. His work bridges academic research and real-world application, informed by research experience at leading institutions such as ETH Zurich.

Legal Disclosure

EXIST Funding

We are proud to be funded by the science-driven EXIST program financed by ...

This White Paper is for informational purposes only. While every effort has been made to ensure accuracy, the publisher assumes no responsibility for errors or omissions. The content does not constitute legal, financial, or professional advice.

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

All rights reserved.

Contact Us

Reach out to us via team@mai-platform.de

https://mai-platform.com

References

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