Architecture Without Architects: How AI Coding Agents Shape Software Architecture

Phongsakon Mark Konrad
Centre for Industrial Software
University of Southern Denmark
Alsion 2, Sønderborg, 6400, Denmark
phkon23@student.sdu.dk

Tim Lukas Adam
Centre for Industrial Software
University of Southern Denmark
Alsion 2, Sønderborg, 6400, Denmark
tiada23@student.sdu.dk

Riccardo Terrenzi
Centre for Industrial Software
University of Southern Denmark
Alsion 2, Sønderborg, 6400, Denmark
rite@mmmi.sdu.dk

Serkan Ayvaz
Centre for Industrial Software
University of Southern Denmark
Alsion 2, Sønderborg, 6400, Denmark
seay@mmmi.sdu.dk

***Abstract*—AI coding agents select frameworks, scaffold infrastructure, and wire integrations, often in seconds. These are architectural decisions, yet almost no one reviews them as such. We identify five mechanisms by which agents make implicit architectural choices and propose six prompt-architecture coupling patterns that map natural-language prompt features to the infrastructure they require. The patterns range from contingent couplings (structured output validation) that may weaken as models improve to fundamental ones (tool-call orchestration) that persist regardless of model capability. An illustrative demonstration confirms that prompt wording alone produces structurally different systems for the same task. We term the phenomenon *vibe architecting*, architecture shaped by prompts rather than deliberate design, and outline review practices, decision records, and tooling to bring these hidden decisions under governance.**

***Index Terms*—large language models, software architecture, vibe coding, agentic coding, AI-assisted development, prompt-driven development**

I. Introduction

A growing number of developers build software by describing what they want in plain language, a practice Karpathy calls *vibe coding* [1]. What started as line-level autocomplete in 2022 has grown into agents that produce entire systems from a single description.

But these tools do far more than write code. Claude Code [2] scaffolds full projects and delegates subtasks to sub-agents. Cursor [3] runs background agents across parallel worktrees. Devin [4] supports interactive planning. Bolt.new [5] produces full-stack applications in browser containers. Codex [6] runs in sandboxed cloud environments. Windsurf [7] adjusts its output dynamically. In each case, they pick frameworks, configure databases, and set up authentication. Each of those choices is an architectural decision.

Two phenomena are the focus of this paper. The first is *agent-driven architecture*, in which agents choose frameworks, databases, and deployment targets without justifying their choices. The second is *prompt-architecture coupling*, in which Large Language Model (LLM)-integrated code declares tool access, and something must orchestrate those calls. When it injects retrieved documents, a retrieval pipeline follows. In both cases, the prompt dictates the infrastructure. Together, these produce what is here termed *vibe architecting*, architecture shaped by natural-language prompts rather than by deliberate, recorded design.

Architects need ways to catch and guide these decisions. This position paper combines a tool survey with an illustrative case study and makes three contributions:

1) Five mechanisms are identified by which agents make implicit architectural decisions, illustrated by a demonstration where prompt wording is the sole varying factor.
2) Six prompt-architecture coupling patterns are proposed that map prompt-level features to the infrastructure they require, drawing on LangChain [8], LlamaIndex [9], and provider documentation.
3) Several open questions are raised for the architecture community, including converging stacks, new forms of technical debt, and a widening gap between how fast agents build and how fast teams can review.

II. Background and Related Work

White *et al.* [10] catalog prompt patterns, drawing an analogy to classical design patterns. Chen *et al.* [11] go further and treat prompts as first-class software artifacts ("promptware") with lifecycle concerns including versioning, testing, and deployment. Neither work investigated what happens when those prompt choices start dictating the architecture of the surrounding system.

LangChain [8] and LlamaIndex [9] organize their APIs around architectural abstractions (chains, agents, retrievers), but that organization lives inside library code rather than in reusable design knowledge a team could reason about on its own. Wang *et al.* [12] survey LLM-based autonomous agents; Wu *et al.* [13] propose multi-agent conversation frameworks. Neither inspects how a prompt's structure implies particular components. In a systematic literature review of software

architecture and LLMs, Schmid *et al.* [14] found growing work applying LLMs to architecture tasks but very little asking how AI tools shape the architecture of the systems they help build.

Sculley *et al.* [15] remains the classic reference on hidden debt in ML systems, namely configuration debt, data dependency debt, and feedback loops. Al Mujahid and Imran [16] studied GenAI-induced self-admitted technical debt, documenting cases where developers pull in AI-generated code while openly admitting that they are unsure whether it works. Kravchuk-Kirilyuk *et al.* [17] focused on modularity and showed that LLM-generated code often appears modular on the surface while violating deeper modular principles. The concern here is different. Sculley *et al.*'s [15] debt accumulates during training and deployment. The debt identified in this paper accumulates during code generation, before the system even runs.

Sarkar and Drosos [18] characterize vibe coding as "programming through conversation"; Fawzy *et al.* [19] reviewed practitioner accounts of how developers use these workflows and Falcão *et al.* [20] place AI-driven development on the longer trajectory toward self-coding systems. All three focus on developer interaction and adoption. None examines architectural consequences. The following section identifies the specific mechanisms through which those consequences arise.

III. Agentic Coding as Architectural Decision-Making

AI coding agents are making architectural decisions, and recent adoption data [21] suggests this is a progressive phenomenon rather than a transient one. The scope of what agents decide has widened rapidly. Line-level autocomplete (2022) had almost no architectural footprint [22]. By 2024, multi-file tools [3], [7] were editing across files, indirectly drawing module boundaries. System-level agents arrived in 2025 (Claude Code [2], Devin [4], Codex [6], Kiro [23], Replit Agent [24], among others), and can scaffold entire projects from a single prompt. More recently, coordinated agents split work across sub-agents assigned to specific files, turning decomposition itself into an architectural concern [25]. By 2026, the agent's decision surface covers the full breadth of software architecture as defined by Bass *et al.* [25]. Standardization through MCP [26] and the `AGENTS.md` convention [27] has reinforced this trend, forming a de facto integration architecture. Yet these decisions take seconds, arrive bundled, and *leave no record.*

A. Mechanisms and Evolution

Five mechanisms were identified through which current agents make architectural decisions. They were derived by surveying six tools (Table I) illustrates how they manifest across three prompt variants.

1) Model selection. Different LLMs produce structurally different code. SonarSource found that each model exhibits a distinct "coding personality" [28]. Switching a model selector is an architectural choice, even if nobody frames it that way.

2) Task decomposition. How agents split work shapes architecture. Claude Code delegates subtasks to sub-agents working on individual files; Cursor's background agents operate in parallel worktrees. Because decomposition determines module boundaries [25], the agent designs the system's modular structure.

3) Default configuration. Guardrail mechanisms exist (Claude Code hooks, Cursor's `.cursorrules`, `AGENTS.md` [27]), but all require the developer to set them up. Without explicit rules, agents default to training-data priors.

4) Scaffolding and autonomous generation. Templates preselect framework, database, authentication, and deployment; the architectural choice was made when the template was written. Autonomous agents go further: a single prompt can produce a complete project, folding every choice into one interaction with no visible rationale.

5) Integration protocols. Tools built on MCP [26] produce systems whose integration points follow a standard protocol for tool discovery, invocation, and data exchange. Older tools hard-code service access, binding the system to the generation platform. Either way, the integration architecture is chosen by the tool, not by the team.

To demonstrate these mechanisms, three variants of a customer-service chatbot were built.[1] Each was developed independently with no shared context. The architectural choices that emerged were recorded (Table I). All three variants were generated using Claude Code with default settings; the exact prompts and full generation details are documented in the companion repository. All variants ran on the same LLM at runtime (GPT-4o-mini), so prompt specification was the sole varying factor. The case study illustrates prompt-architecture coupling (Section IV); the five mechanisms themselves are supported independently by the tool survey above. The intent is illustration, not experimental proof.

TABLE I
Architectural Divergence Across Three Prompt Variants

Variant	Frmwk	Store	Components	Integr.	LoC	Files
A: FAQ	Express	JSON	FAQ loader, chat handler	API call	141	2
B: JSON	Express	—	Zod schema, retry, fallback	JSON mode	472	4
C: Tools	Express	SQLite	Tool registry, agent loop, store	Func. call	827	6

Prompts: A = "answer product questions from a FAQ"; B = "return structured JSON (intent, confidence, entities) with schema validation"; C = "agent with tool access: search_kb(), get_account(), escalate_to_human()". Each variant developed independently with no shared context.

The three variants fall along a spectrum of prompt specificity. Variant A names a task ("answer questions from a FAQ"). Variant B specifies an output contract ("return JSON with fields: intent, confidence, entities"). Variant C declares capabilities ("you have access to: search_kb(), get_account(), escalate()"). As the prompt grows more specific, the line between prompting and architecture blurs. A capability declaration is, in practice, an interface definition. An output

[1] Source code: https://github.com/phomarkon/vibe-architecting-case-study

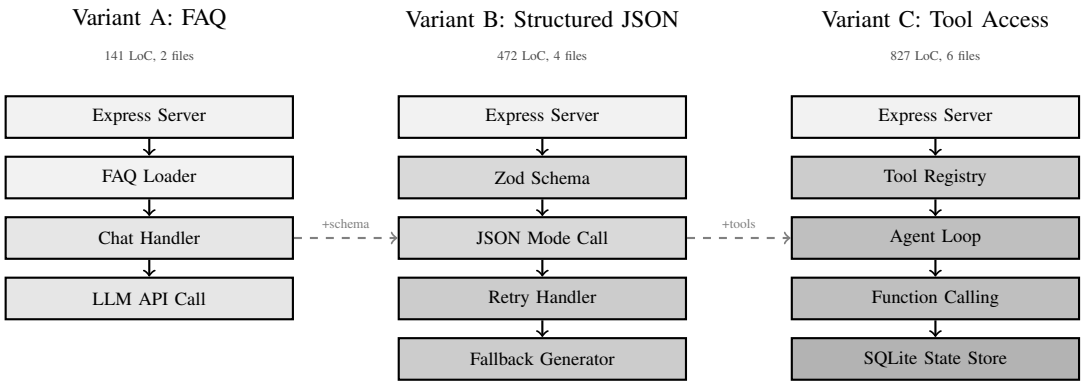

Fig. 1. Architectural components introduced by each prompt variant. Darker shading indicates components added by increasing prompt specificity. Variant A passes FAQ context to the LLM and returns free-form text; Variant B adds Zod validation, JSON Schema mode, and retry logic; Variant C adds a tool registry, agent loop, and SQLite state management.

schema is a data contract. This blurring is not surprising, but the central observation. The prompt becomes the architecture specification, and the generated infrastructure follows from it. When agents compose prompts on their own (as in coordinated sub-agent workflows), this coupling operates without human review.

What distinguishes agent-made from human-made architectural decisions? Three properties stand out. *Scale.* Framework, database, authentication, and deployment are selected in a single interaction, packaged as a unit rather than as separately reviewable choices. *Speed.* Decisions that teams deliberate over for days occur within seconds, outpacing any review process. *Opacity.* Choices remain buried in generated code with no ADRs, no design documents, no recorded rationale. The opacity of these decisions has a further consequence. When the systems that agents build are themselves LLM-integrated, the prompt's influence extends beyond behavior and determines the infrastructure the system requires.

IV. Prompt-Architecture Coupling

Agents make architectural decisions at scale, at speed and without transparency. The problem compounds when the systems they build are themselves LLM-integrated, a common situation in 2026. In such systems, prompt design determines which infrastructure components are needed.

Recent systems research shows that prompt design characteristics, particularly input length and structure, directly determine infrastructure requirements. Longer prompts increase token-processing overhead, forcing architectural choices regarding caching strategy and GPU batching [29]. This coupling between prompt design and infrastructure is particularly acute in multi-agent systems where prompts are generated dynamically.

A. The Problem of Implicit Coupling

The illustrative case study makes the point concretely. Variant A uses a free-form prompt and results in a flat architecture with 2 files and 141 LoC. Adding a structured output requirement (Variant B) forces the system to acquire a Zod schema, a retry handler, and a fallback generator, bringing the total to 4 files and 472 LoC. Declaring tool access (Variant C) pushes further by introducing a tool registry, an agent loop, and an SQLite state store, yielding 6 files and 827 LoC. The same task, expressed through three different prompts, produces systems with different component structures, dependency graphs and failure modes (Fig. 1). What appears to be a minor prompt adjustment can create entirely new, unplanned infrastructure. When an AI agent makes prompt choices on the developer's behalf, the architectural consequences can accumulate unnoticed.

B. Six Recurring Couplings Patterns

Abstractions in LangChain [8], LlamaIndex [9], and provider documentation [30], [31] were traced, following each prompt-level feature to the infrastructure it requires. Six patterns emerged across three categories (Table II). Each is classified as either *contingent* (likely to weaken as models gain native capabilities, e.g. native JSON output) or *fundamental* (logically necessary regardless of model capability, e.g., tool execution always requires orchestration). The catalog is not exhaustive but covers the most frequently observed couplings.

Constraining the form of output introduces the first class of coupling. A prompt requiring structured responses ("return JSON with fields: intent, confidence, entities") forces a parser, schema validator, retry handler, and fallback generator into the system. LangChain's `with_structured_output()` [8] and OpenAI's strict mode [30] wrap exactly this pipeline. In the case study, Variant B acquired a Zod schema, exponential-backoff retry, and fallback, three components absent from Variant A, solely because the prompt demanded structured output. Dynamic few-shot selection creates a parallel dependency. When a prompt selects examples at query time rather than hard-coding them, the system needs an embedding model, a vector store for the example bank, and an example curator [9]. Both couplings are contingent. As models gain native JSON guarantees or larger context windows, the validation and retrieval stacks may shrink.

Declaring what a model can do introduces a different, more durable kind of coupling. Typed tool signatures (`search_kb(query: string)`) create a service-oriented

TABLE II
PROMPT-ARCHITECTURE COUPLING PATTERNS

Pattern	Trigger	Infrastructure Required	Quality Impact	Type
Constraint patterns				
1A: Structured Output [8], [30]	JSON schema	Parser, validator, retry, fallback	Interoperability, robustness	Contingent
1B: Few-Shot [9]	Dynamic examples	Embedding, vector store, curator	Capability, grounding	Contingent
Capability patterns				
2A: Function Calling [30], [31]	Tool signatures	Router, validator, error handler, agent loop	Extensibility, attack surface	Fundamental
2B: ReAct Reasoning [8], [32]	CoT with tools	State machine, validator, timeout handler	Autonomy, testability	Fundamental
Context patterns				
3A: RAG [8], [9]	Bounded context	Ingest, chunk, embed, vector store, ranker	Accuracy, infrastructure cost	Contingent
3B: Context Reduction [9]	Token budget	Summarizer, filter, extractor	Cost, information loss	Contingent

Contingent couplings may weaken as models gain native capabilities; fundamental couplings are logically necessary regardless of model capability. Composition multiplies effects super-linearly.

architecture with a function router, argument validator, error handler, and response loop [30], [31]. Variant C's three tool declarations produced a tool registry, an agent loop (ten iterations), and an SQLite state store, none of which exist in A or B. Each additional tool widens the attack surface [33]. ReAct-style reasoning [32] layers a state machine on top, tracking reasoning steps with per-step validation, partial failure handling, and timeout enforcement (LangGraph's `StateGraph` [8] is a direct implementation). Both couplings are fundamental. Tool execution will always require orchestration, and interleaved reasoning steps resist unit testing [12] regardless of how capable models become.

The remaining two patterns control which information reaches the model. Constraining answers to retrieved context requires an ingestion, chunking, embedding, and ranking pipeline [34], [35]. Variant A's FAQ injection is a minimal static analogue. At scale, the relevance ranking remains valuable even as context windows expand. Token budget constraints add a complementary pre-processing stage, summarization, filtering, extraction [9], before the prompt reaches the model. Privacy filtering, cost control, and batch processing keep this pattern relevant despite growing context limits. LlamaIndex documents several node-level transformations for both retrieval and reduction. The trade-off in each case is infrastructure complexity versus information quality.

Across all six patterns, the direction is the same: the prompt drives the architecture. The couplings raise practical questions, in particular, how teams should review prompt-level decisions and what tooling would make the resulting architectural choice visible. The following section addresses both.

V. DISCUSSION

A. How Agentic Coding Is Changing Architecture

Agent-scaffolded projects are converging on a narrow set of stacks. Bolt.new, Lovable, and v0 [36] all default to React, TypeScript, and Tailwind. This simplifies knowledge transfer but concentrates vulnerability exposure [25]. At the same time, every decision is implicit. A developer typing "a todo app with auth" receives database, authentication, and deployment choices already made, with no rationale on record. Kravchuk-Kirilyuk *et al.* [17] show that such code appears modular on the surface while hiding deeper coupling, and developers knowingly skip validation [16].

The resulting speed-review gap is acute. Agents scaffold systems in minutes; teams need hours or days to audit them. Tool-use patterns (2A, 2B) widen the attack surface through indirect prompt injection [33]. MCP [26] standardizes tool integration but also standardizes attack vectors. The `AGENTS.md` convention [27] is a first guardrail, but securing composed patterns remains open when multiple tool-use patterns combine (Fig. 2).

B. Implications for Practice

Prompt specifications are architectural artifacts and belong in architectural review. An impact statement (e.g., "Structured JSON output adds validator, retry handler, fallback; +330 LoC, two new failure modes") would make the coupling explicit before code generation begins. Prompt choices and agent decisions also belong in ADRs. Existing tools (Claude Code hooks, `.cursorrules`, `AGENTS.md`) constrain reactively. Proactive guidance that flags consequences before generation is still missing. The 5.9× code growth (141→827 LoC) and 3× file increase in the case study (Table I) suggest that complexity thresholds, baselines that trigger review when exceeded, would help teams catch uncontrolled scaffolding early.

C. Toward Architecture-Aware AI-Assisted Development

A common gap connects both phenomena. Agents make architectural decisions, but no feedback loop ties those decisions to established architectural knowledge. This paper sketches a three-layer framework (Fig. 3) mapping existing tool mechanisms to software architecture concepts.

The first layer, *constraints*, specifies what the agent may and may not do. Instruction files (`AGENTS.md`, `.cursorrules`)

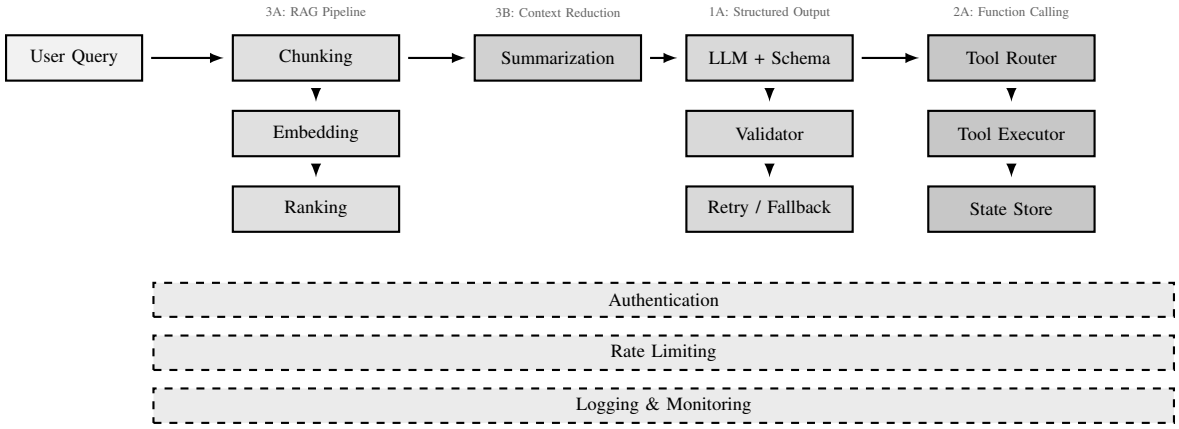

Fig. 2. Composition of coupling patterns in a single system (RAG chatbot with tool access). Pattern IDs match the catalog in Section IV. Cross-cutting concerns (dashed) span multiple patterns and add infrastructure no single pattern accounts for on its own.

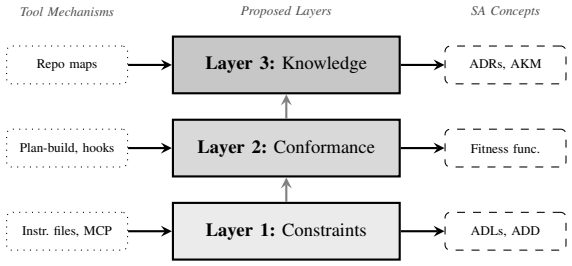

Fig. 3. Three-layer framework for architecture-aware AI-assisted development. Each layer maps tool mechanisms (left) to SA concepts (right).

and MCP server configurations [26] already play this role informally. Architecture Description Languages (ADLs) and Attribute-Driven Design (ADD) [25] formalize the same concern. The second layer, *conformance*, checks generated code against those constraints. Plan-build workflows (where the agent proposes before executing) and post-generation hooks are analogous to fitness functions in evolutionary architecture. The third layer, *knowledge*, feeds architectural context back to the agent. Repository maps and context files serve this purpose today; ADRs and Architectural Knowledge Management (AKM) do so in the SA literature. Concretely, a conformance layer could compare the dependency graph of generated code against declared constraints (e.g., "no new database dependencies without review") and flag violations before they propagate. The knowledge layer could extract architectural decisions from agent reasoning traces and persist them as ADRs, closing the documentation gap between generation speed and review capacity.

Falcão *et al.* [20] envision a trajectory toward self-coding systems with runtime autonomy. The framework here is complementary, targeting development-time governance with a human in the loop rather than runtime self-adaptation. Whether the two converge as agents grow more autonomous is an open question.

D. Threats to Validity

Three validity concerns bound the claims made in this paper and shape the next steps. For *construct validity*, "architectural decision" is defined following Bass *et al.* [25], and "vibe architecting" is framed as a deliberate extension of Karpathy's "vibe coding" [1]. The six coupling patterns are analytical constructs proposed for community discussion, not yet empirically validated.

For *internal validity*, the prompts encode architectural expectations at varying specificity; future work should test with minimally prescriptive prompts to isolate what the agent decides unprompted. Quality impact assessments in Table II are analytical, not empirical.

For *external validity*, the tool survey reflects agentic coding as of early 2026, a space where new tools appear regularly. The six coupling patterns draw on three frameworks and additional patterns almost certainly exist beyond this set. Each prompt in the illustrative example was run once, with one tool and one LLM, so replicating across agents, models, and developer populations would test whether the observed architectural divergence generalizes.

E. Research Agenda

Five directions follow from this analysis. First, cross-agent replication. The case study varied prompts while holding the agent constant. Do Claude Code [2], Cursor [3], and Devin [4] produce substantially different architectures for the same specification? If so, agent selection is itself an architectural decision requiring governance.

Second, architectural footprint metrics. LoC and file count (Table I) are coarse. Richer metrics capturing component count, dependency depth, and new failure modes would let teams flag prompt changes exceeding complexity thresholds.

Third, proactive governance tooling. Existing guardrails (hooks, `.cursorrules`, `AGENTS.md` [27]) constrain reactively. A tool previewing architectural impact ("this adds vector database, embedding pipeline, async job queue") before code generation would address the review-speed gap.

Fourth, automated ADR generation. The rationale behind architectural choices exists implicitly in the agent's reasoning trace; extracting it into structured ADRs would close the documentation gap.

Fifth, pattern composition analysis. Fig. 2 shows that coupling patterns compose. A RAG chatbot with tool access pulls in retrieval, function calling, and structured output simultaneously. Whether cross-cutting concerns (authentication, rate limiting, logging) scale additively or super-linearly with pattern composition remains an open question.

VI. Conclusion

AI coding agents make architectural decisions through five identifiable mechanisms, and prompt-architecture coupling produces six recurring patterns where natural-language instructions determine infrastructure. Varying the prompt wording alone yielded structurally different systems in the illustrative example, from 141 to 827 lines of code and two to six files for the same task.

Vibe coding becomes vibe architecting the moment a prompt determines system structure. The question for the architecture community is whether these prompt-level decisions deserve the same governance as design-level ones. This paper argues that they do. The review practices, decision records, and tooling to make that governance possible do not yet exist. Building them is the next step.

Vibe architecting is a research program, not a single contribution. The five directions in Section V-E each connect to existing work in software architecture while raising questions specific to the AI-assisted setting. As agents grow more capable and autonomous, the governance gap will only widen. These starting points are intended to invite both empirical follow-ups and tool-building efforts from the community.

References

[1] A. Karpathy, "There's a new kind of coding I call 'vibe coding'," X/Twitter, Feb. 2025, [Online]. Available: https://x.com/karpathy/status/1886192184808149383.

[2] Anthropic, "Claude Code documentation," [Online]. Available: https://code.claude.com/docs/en/overview, 2025, accessed: Feb. 8, 2026.

[3] Cursor, "Cursor documentation," [Online]. Available: https://cursor.com/docs/agent, 2025, accessed: Feb. 8, 2026.

[4] Cognition, "Devin 2.0," [Online]. Available: https://cognition.ai/blog/devin-2, 2025, accessed: Feb. 8, 2026.

[5] StackBlitz, "Bolt.new," [Online]. Available: https://github.com/stackblitz/bolt.new, 2025, accessed: Feb. 8, 2026.

[6] OpenAI, "Codex," [Online]. Available: https://openai.com/index/introducing-codex/, 2025, accessed: Feb. 8, 2026.

[7] Windsurf, "Windsurf editor," [Online]. Available: https://windsurf.com/, 2025, accessed: Feb. 8, 2026.

[8] LangChain, "LangChain framework, v1.2," [Online]. Available: https://github.com/langchain-ai/langchain, 2025, accessed: Feb. 8, 2026.

[9] LlamaIndex, "LlamaIndex framework, v0.14," [Online]. Available: https://github.com/run-llama/llama_index, 2025, accessed: Feb. 8, 2026.

[10] J. White *et al.*, "A prompt pattern catalog to enhance prompt engineering with ChatGPT," *arXiv preprint arXiv:2302.11382*, 2023.

[11] Z. Chen, C. Wang, W. Sun, X. Liu, J. M. Zhang, and Y. Liu, "Promptware engineering: Software engineering for prompt-enabled systems," *ACM Trans. Softw. Eng. Methodol.*, 2026, to appear. Preprint: arXiv:2503.02400.

[12] L. Wang *et al.*, "A survey on large language model based autonomous agents," *Frontiers of Computer Science*, vol. 18, no. 6, 2024.

[13] Q. Wu *et al.*, "AutoGen: Enabling next-gen LLM applications via multi-agent conversation," *arXiv preprint arXiv:2308.08155*, 2023.

[14] L. Schmid *et al.*, "Software architecture meets LLMs: A systematic literature review," *arXiv preprint arXiv:2505.16697*, 2025.

[15] D. Sculley *et al.*, "Hidden technical debt in machine learning systems," in *Advances in Neural Information Processing Systems*, vol. 28, 2015, pp. 2503–2511.

[16] A. Al Mujahid and M. M. Imran, "TODO: Fix the mess Gemini created: Towards understanding GenAI-induced self-admitted technical debt," in *Proc. 9th Int. Conf. Technical Debt (TechDebt)*, 2026, preprint: arXiv:2601.07786.

[17] A. Kravchuk-Kirilyuk, F. Graciolli, and N. Amin, "The modular imperative: Rethinking LLMs for maintainable software," in *Proc. 1st ACM SIGPLAN Int. Workshop on Language Models and Programming Languages (LMPL)*, 2025.

[18] A. Sarkar and I. Drosos, "Vibe coding: Programming through conversation with artificial intelligence," *arXiv preprint arXiv:2506.23253*, 2025.

[19] A. Fawzy, A. Tahir, and K. Blincoe, "Vibe coding in practice: Motivations, challenges, and a future outlook – a grey literature review," in *Proc. 47th IEEE/ACM Int. Conf. Softw. Eng. (ICSE), SEIP Track*, 2026, preprint: arXiv:2510.00328.

[20] R. Falcão, F. Elberzhager, and K. Vaidhyanathan, "Toward self-coding information systems," *arXiv preprint arXiv:2601.14132*, Jan. 2026.

[21] R. Robbes, T. Matricon, T. Degueule, A. Hora, and S. Zacchiroli, "Agentic much? adoption of coding agents on GitHub," *arXiv preprint arXiv:2601.18341*, Jan. 2026.

[22] GitHub, "GitHub Copilot," [Online]. Available: https://docs.github.com/en/copilot, 2025, accessed: Feb. 8, 2026.

[23] AWS, "Kiro," [Online]. Available: https://kiro.dev/, 2025, accessed: Feb. 8, 2026.

[24] Replit, "Replit Agent," [Online]. Available: https://docs.replit.com/replitai/agent, 2025, accessed: Feb. 8, 2026.

[25] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 4th ed. Boston, MA, USA: Addison-Wesley, 2021.

[26] Anthropic, "Model Context Protocol specification," [Online]. Available: https://modelcontextprotocol.io/specification/2025-11-25, 2025, accessed: Feb. 8, 2026.

[27] Linux Foundation, "Agentic AI foundation," [Online]. Available: https://aaif.io/, Dec. 2025, accessed: Feb. 8, 2026.

[28] SonarSource, "The coding personalities of leading LLMs," State of Code Report, Aug. 2025. [Online]. Available: https://www.sonarsource.com/the-coding-personalities-of-leading-llms/, 2025, accessed: Feb. 13, 2026.

[29] A. Agrawal *et al.*, "Taming throughput-latency tradeoff in LLM inference with Sarathi-Serve," in *Proc. 18th USENIX Symp. Operating Syst. Design Implementation (OSDI)*, 2024.

[30] OpenAI, "Function calling," OpenAI Platform Documentation. [Online]. Available: https://platform.openai.com/docs/guides/function-calling, 2025, accessed: Feb. 8, 2026.

[31] Anthropic, "Tool use (function calling)," Anthropic Documentation. [Online]. Available: https://docs.anthropic.com/en/docs/build-with-claude/tool-use, 2025, accessed: Feb. 8, 2026.

[32] S. Yao *et al.*, "ReAct: Synergizing reasoning and acting in language models," in *Proc. Int. Conf. Learn. Representations (ICLR)*, 2023.

[33] K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, and M. Fritz, "Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection," in *Proc. 16th ACM Workshop on Artificial Intelligence and Security (AISec)*, 2023, pp. 79–90.

[34] P. Lewis *et al.*, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 9459–9474.

[35] Y. Gao *et al.*, "Retrieval-augmented generation for large language models: A survey," *arXiv preprint arXiv:2312.10997*, 2024.

[36] Vercel, "v0," [Online]. Available: https://v0.dev/, 2025, accessed: Feb. 8, 2026.