Technology Radar

An opinionated guide to
today's technology landscape

Volume 34
April 2026

/thoughtworks
Design. Engineering. AI.

About the Radar

Thoughtworkers are passionate about technology. We build it, research it, test it, open source it, write about it and constantly aim to improve it — for everyone. Our mission is to champion software excellence and revolutionize IT. We create and share the Thoughtworks Technology Radar in support of that mission. The Thoughtworks Technology Advisory Board, a group of senior technology leaders at Thoughtworks, creates the Radar. They meet regularly to discuss the global technology strategy for Thoughtworks and the technology trends that significantly impact our industry.

The Radar captures the output of the Technology Advisory Board's discussions in a format that provides value to a wide range of stakeholders, from developers to CTOs. The content is intended as a concise summary.

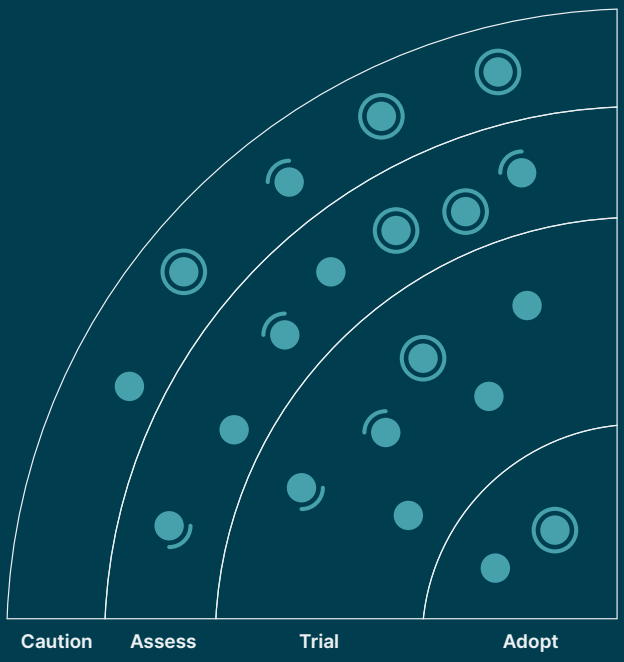
We encourage you to explore these technologies. The Radar is graphical in nature, grouping items into techniques, tools, platforms and languages and frameworks. When Radar items could appear in multiple quadrants, we chose the one that seemed most appropriate. We further group these items in four rings to reflect our current position on them.

For more background on the Radar, see thoughtworks.com/radar/faq.

Radar at a glance

The Radar is all about tracking interesting things, which we refer to as blips. We organize the blips in the Radar using two categorizing elements: quadrants and rings. The quadrants represent different kinds of blips. The rings indicate our recommendation for using that technology.

A blip is a technology or technique that plays a role in software development. Blips are “in motion” — their position in the Radar often changes — usually indicating our increasing confidence in recommending them as they move through the rings.



Adopt: We feel strongly that the industry should be adopting these items. We use them when appropriate in our projects.

Trial: Worth pursuing. It’s important to understand how to build up this capability. Enterprises can try this technology on a project that can handle the risk.

Assess: Worth exploring with the goal of understanding how it will affect your enterprise.

Caution: Proceed with care. These items involve significant concerns that require careful evaluation for your specific context.

New **Moved in/out** **No change**

Our Radar is forward-looking. To make room for new items, we fade items that haven’t moved recently, which isn’t a reflection on their value but rather on our limited Radar real estate.

Contributors

The Technology Advisory Board (TAB) is a group of 22 senior technologists at Thoughtworks. The TAB meets twice a year face-to-face and biweekly virtually. Its primary role is to be an advisory group for Thoughtworks CTO Rachel Laycock.

The TAB acts as a broad body that can look at topics that affect technology and technologists at Thoughtworks. This edition of the Thoughtworks Technology Radar is based on a meeting of the TAB in Bengaluru in March 2026.


Rachel Laycock (CTO)


Abdul Jeelani

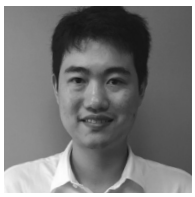

Alessio Ferri


Bharani Subramaniam


Birgitta Böckeler


Brandon Cook


Cecilia Geraldo


Chris Chakrit Riddhagni


Jagdsh LK Chand


Jim Gumbley


Effy Elden


Kief Morris


Ken Mugrage


May Xu


Nati Rivera


Ni Wang


Renan Martins


Nimisha Asthagiri


Selvakumar Natesan


Shangqi Liu


Sarang Sanjay Kulkarni


Vanya Seth

Production credits

Editorial Team

- **Ken Mugrage**
  Technical Editor
- **Nati Rivera**
  Product Owner
- **Preeti Mishra**
  Project and Campaign Manager
- **Richard Gall**
  Content Editor
- **Michael Koch**
  Copy Editor
- **Gareth Morgan**
  Head of Content and Thought Leadership

Design and Multimedia

- **Leticia Nunes**
  Lead Designer
- **Sruba Deb**
  Visual Designer
- **Ryan Cambage**
  Multimedia Specialist
- **Anish Thomas**
  Multimedia Designer

Digital and Web Experience

- **Rashmi Naganur**
  Business Analyst
- **Brigitte Britten-Kelly**
  Digital Content Strategist
- **Vandita Kamboj**
  UX Designer
- **Lohith Amruthappa**
  Analytics Specialist
- **Neeti Thakur**
  Marketing Automation Specialist

Communications

- **Shalini Jagadish**
  Internal Communications Specialist
- **Yasmin Brussulo**
  Internal Communications Specialist
- **Hiral Shah**
  Social Media Specialist
- **Abhishek Kasegaonkar**
  Social Media Specialist
- **Michelle Surendran**
  Public Relations Specialist
- **Anushree Tapuriah**
  Campaigns and Advertisement Specialist
- **Prakhar Nigam**
  Campaigns and Advertisement Specialist

Production credits

Thoughtworkers contributions

India

- Krishnaswamy Subramaniam
- Rahul Garg
- OTTO account team
- Sayeed Hussain
- Srihari Sridharan
- Vijay Raghavan
- Jigar Jani

Europe

- David Rubio
- Jose Pech
- Florian Sellmayr
- Chris Ford
- Ricardo Piccoli
- Ben O'Mahony
- Dan Mutton
- Kai Hendry
- Karsten Lettow
- Alexandra Lovin
- Rieke Heinze
- Jose Maria Segui Gomez de Olea
- Tom Coggrave
- Muhammedsameer Azazi
- Kushwant Kumar
- Sivaprakash Kumar

APAC

- Hongbin Zhang
- Wei Feng
- Mi He
- Zhe Zhao
- Andrew Watson
- Zhe Zhao
- Daval Doshi
- Patiphon Puntusin
- Srini Raguraman
- Bosco Ho
- Dat Tran
- Arthur Nguyen

Americas

- Luiz Miccieli
- Carla Lima
- Gustavo Neves
- Ricardo Cavalcanti
- María Laura Jaramillo
- Gabriel Furini
- Guilherme Silveira
- Thiago Gomes
- Enderson Menezes
- Guilherme Marthe
- Daniel Romero
- Daniel Mansilla
- Chris Kramer
- Kurtis Angell
- Nikola Savic
- Jose Duron
- Cameron Casher
- Lauren O'Neal
- Brooks Bollich

Internal

- Johann Gomes
- Fabiano Damasceno
- Freddy Escobar
- Muhilvarnan V
- Rathinakumar Ponnusamy
- Da Cheng
- Sakthe Karthika T
- Vinodhini V

Themes

The challenge of evaluating technology in an agentic world

AI is changing not only technology but also how we assess and evaluate it. While assembling this volume of the Technology Radar, we noticed that evaluating technology is becoming harder as the industry adopts AI. One contributing factor is semantic diffusion: the rapid emergence of new terms for evolving practices, often before their meanings have stabilized. For example, terms such as spec-driven development and harness engineering are sometimes used inconsistently or overlap in meaning. Without shared definitions, it's difficult to determine whether we're seeing distinct techniques or simply different labels for similar ideas. Distinguishing between a mature, standalone engineering methodology with clear guardrails from the everyday use of AI tools such as coding assistants is an ongoing difficulty.

The challenge extends beyond semantics. The pace of change compounds this uncertainty. We encountered several tools that were less than a month old — some promising, but ultimately too young to assess. In many cases, these tools were maintained by a single contributor working with a coding agent. AI has lowered the barrier to building developer tooling, creating a constant stream of new tools. This stretches the traditional rhythm of the Radar: if we allow tools time to mature, our guidance risks becoming outdated; if we move too quickly, we risk highlighting trends that disappear just as fast. It also raises questions about sustainability: when something can be created quickly and with relatively little effort, what ensures continued investment in maintaining and evolving it?

This environment also risks creating codebase cognitive debt. As more code is generated by AI, it's easier to adopt solutions without developing the mental models needed to understand how they work. Over time, this gap in understanding accumulates, making systems harder to reason about, debug and evolve.

Retaining principles, relinquishing patterns

An interesting consequence of AI in software development is that it's not only forcing us to look to the future; it's also pushing us to revisit the foundations of our craft. While assembling this edition, we found ourselves returning to many established techniques, from pair programming to zero trust architecture, and from mutation testing to DORA metrics. We also revisited core principles of software craftsmanship, such as clean code, deliberate design, testability and accessibility as a first-class concern. This is not nostalgia, but a necessary counterweight to the speed at which AI tools can generate complexity. We also observed a resurgence of the command line: After years of abstracting it away in the name of usability, agentic tools are bringing developers back to the terminal as a primary interface.

Themes

That said, this is not a story of continuity alone. AI-assisted software development represents a fundamental shift in engineering practice, requiring us to rethink — and in some cases discard — long-held assumptions. In particular, how we collaborate and structure teams will need to evolve as we better understand what is possible with agentic systems. We may need to consider agent topologies alongside team topologies, and rethink feedback cycles accordingly. Several techniques in this edition reflect this shift. For example, measuring collaboration quality with coding agents points toward a broader redefinition of what it means to be a software developer. Ultimately, we're likely only at the beginning of a deeper transformation in how we build, understand and interact with software systems.

At the heart of this is the challenge of managing cognitive debt in an AI-driven environment. We must retain the principles of good software engineering without allowing AI tools to amplify complexity or accelerate decision fatigue. Our craftsmanship tradition has long held that speed without discipline compounds cost — a principle worth holding onto as agentic systems make it easier than ever to move quickly and understand what we've built.

Securing permission-hungry agents

"Permission hungry" describes the bind at the heart of the current agent moment: the agents worth building are the ones that need access to everything. OpenClaw and Claude Cowork supervise real work tasks; Gas Town coordinates agent swarms across entire codebases. These agents require broad access to private data, external communication and real systems — each arguing that the payoff justifies it.

However, like a skier who's just learned to turn and confidently points themselves at the hardest black run, the safeguards haven't caught up with that ambition. The appetite for access collides with unsolved problems. Prompt injection means models still can't reliably distinguish trusted instructions from untrusted input. Simon Willison's "lethal trifecta" definition of an unsafe agent — private data, untrusted content, external action — now describes most useful agents by default, not by misconfiguration. Injection isn't the only threat. Model behavior is still inconsistent: a task completed successfully once offers no guarantee it will succeed the next time, let alone at scale. Agents find creative exfiltration paths, push to branches they shouldn't touch and erode approve/deny chokepoints without any malicious intent or explicit prompting.

What can we do today? Zero trust, least privilege, model improvements and defense in depth are now table stakes, but there is no silver bullet. We expect safe agent systems to be composed not of monolithic agents, but of pipelines of more constrained agents, with strong monitoring and control. Emerging practices such as Agent Skills as a controlled alternative to MCP, durable agents and techniques to prevent agent instruction bloat all point in this direction. The space is evolving rapidly, which is exciting — but for now, caution is essential to avoid a costly misstep.

Themes

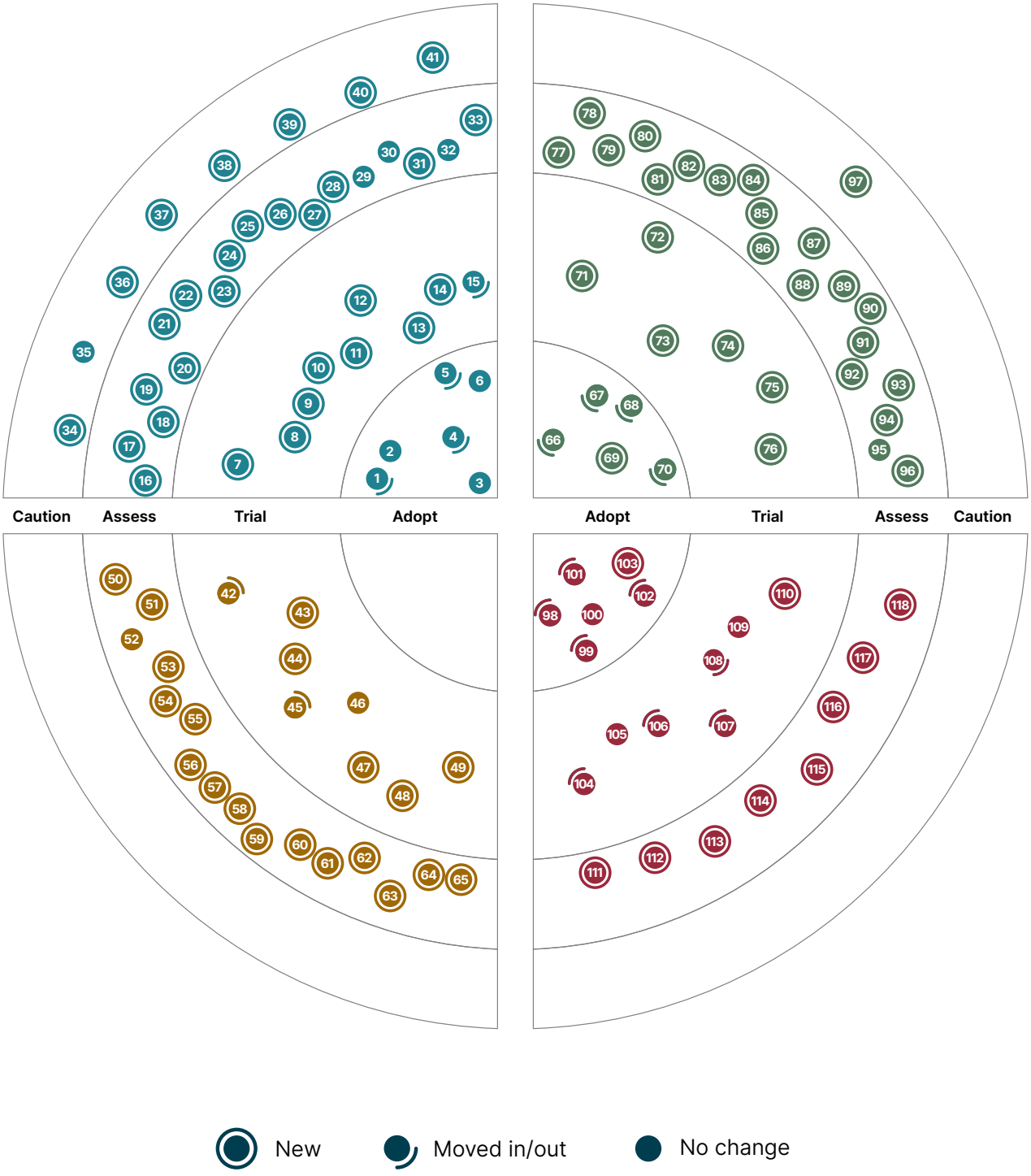
Putting coding agents on a leash

As coding agents become more capable and produce better results, humans are increasingly being tempted to step out of the loop. To mitigate the risks of reduced human supervision, teams are beginning to invest in coding agent harnesses: controls that guide agents' behavior before code is generated and provide feedback afterwards to enable self-correction.

Feedforward controls anticipate what the agent needs to increase the likelihood of a correct first attempt. Agent Skills have been a major recent advancement, helping modularize instructions and conventions, and load them just in time. Superpowers is an example of a catalog of useful skills for software teams, and the emerging concept of plugin marketplaces is making it easier to distribute skills and other context configurations. Our teams have also seen value in spec-driven development frameworks such as GitHub Spec-Kit and OpenSpec, which structure workflows and guide agents through planning, design and implementation.

Feedback controls, by contrast, observe behavior after the agent acts and create a loop for self-correction. This approach is captured by feedback sensors for coding agents: deterministic quality gates — compilers, linters, type checkers and test suites — integrated directly into agent workflows so failures trigger auto-correction before human review. Examples from this Radar include cargo-mutants and other mutation testing tools, fuzz testing tools such as WuppieFuzz and code quality analysis tools such as CodeScene. Beyond in-loop feedback, some of our teams have also reduced architectural drift by combining deterministic structural rules with LLM-based evaluation.

The Radar



The Radar

Techniques

Adopt

1. Context engineering
2. Curated shared instructions for software teams
3. DORA metrics
4. Passkeys
5. Structured output from LLMs
6. Zero trust architecture

Trial

7. Agent Skills
8. Browser-based component testing
9. Feedback sensors for coding agents
10. Mapping code smells to refactoring techniques
11. Mutation testing
12. Progressive context disclosure
13. Sandboxed execution for coding agents
14. Semantic layer
15. Server-driven UI

Assess

16. Agentic reinforcement learning environments
17. Architecture drift reduction with LLMs
18. Code intelligence as agentic tooling
19. Context graph
20. Feedback flywheel
21. HTML Tools
22. LLM evaluation using semantic entropy
23. Measuring collaboration quality with coding agents
24. MITRE ATLAS
25. Ralph loop
26. Reverse engineering for design system
27. Role-based contextual isolation in RAG
28. Skills as executable onboarding documentation
29. Small language models
30. Team of coding agents
31. Temporal fakes
32. Toxic flow analysis for AI
33. Vision language models for end-to-end document parsing

Caution

34. Agent instruction bloat
35. AI-accelerated shadow IT
36. Codebase cognitive debt
37. Coding agent swarms
38. Coding throughput as a measure of productivity
39. Ignoring durability in agent workflows
40. MCP by default
41. Pixel-streamed development environments

Platforms

Adopt

—

Trial

42. AG-UI Protocol
43. Apache APISIX
44. AWS Bedrock AgentCore
45. Graphiti
46. Langfuse
47. Port
48. Replit
49. SigNoz

Assess

50. Agent Trace
51. ClickStack
52. Coder
53. Databricks Agent Bricks
54. DuckLake
55. FalkorDB
56. Google Dialogflow CX
57. MCP Apps
58. Monarch
59. Neutree
60. OptScale
61. Rhesis
62. RunPod
63. Sprites
64. torchforge
65. torchtitan

Caution

—

The Radar

Tools

Adopt

66. Axe-core
67. Claude Code
68. Cursor
69. Kafbat UI
70. mise

Trial

71. cargo-mutants
72. Claude Code plugin marketplace
73. Dev Containers
74. Figma Make
75. OpenAI Codex
76. Typst

Assess

77. Agent Scan
78. Beads
79. Bloom
80. CDK Terrain
81. CodeScene
82. ConfIT
83. Entire CLI
84. Git AI
85. Google Antigravity
86. Google Mainframe Assessment Tool
87. OpenCode
88. OpenSpec
89. PageIndex
90. Pencil
91. Pi
92. Qwen 3 TTS
93. SGLang
94. ty
95. Warp
96. WuppieFuzz

Caution

97. OpenClaw

Languages and Frameworks

Adopt

98. Apache Iceberg
99. Declarative Automation Bundles
100. React JS
101. React Native
102. Svelte
103. Typer

Trial

104. Agent Development Kit (ADK)
105. DeepEval
106. Docling
107. LangExtract
108. LangGraph
109. LiteLLM
110. Modern.js

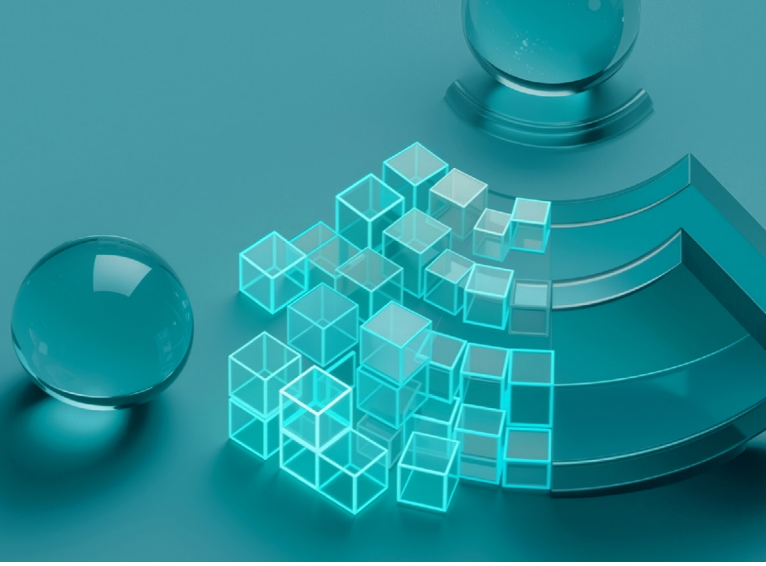
Assess

111. Agent Lightning
112. GitHub Spec Kit
113. Mastra
114. Pipecat
115. Superpowers
116. TanStack Start
117. TOON (Token-Oriented Object Notation)
118. Unsloth

Caution

—

Techniques



Adopt

1. Context engineering
2. Curated shared instructions for software teams
3. DORA metrics
4. Passkeys
5. Structured output from LLMs
6. Zero trust architecture

Trial

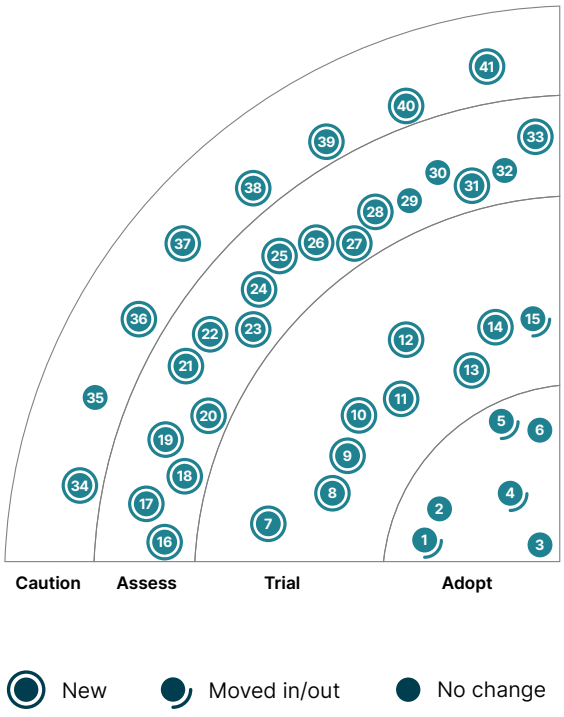
7. Agent Skills
8. Browser-based component testing
9. Feedback sensors for coding agents
10. Mapping code smells to refactoring techniques
11. Mutation testing
12. Progressive context disclosure
13. Sandboxed execution for coding agents
14. Semantic layer
15. Server-driven UI

Assess

16. Agentic reinforcement learning environments
17. Architecture drift reduction with LLMs
18. Code intelligence as agentic tooling
19. Context graph
20. Feedback flywheel
21. HTML Tools
22. LLM evaluation using semantic entropy
23. Measuring collaboration quality with coding agents
24. MITRE ATLAS
25. Ralph loop
26. Reverse engineering for design system
27. Role-based contextual isolation in RAG
28. Skills as executable onboarding documentation
29. Small language models
30. Team of coding agents
31. Temporal fakes
32. Toxic flow analysis for AI
33. Vision language models for end-to-end document parsing

Caution

34. Agent instruction bloat
35. AI-accelerated shadow IT
36. Codebase cognitive debt
37. Coding agent swarms
38. Coding throughput as a measure of productivity
39. Ignoring durability in agent workflows
40. MCP by default
41. Pixel-streamed development environments



1. Context engineering

Adopt

Context engineering has evolved from an optimization tactic into a foundational architectural concern for modern AI systems. Unlike prompt engineering, which focuses on wording, context engineering treats the context window as a design surface and intentionally constructs the AI's information environment.

As agents tackle more complex tasks, dumping raw data into large context windows leads to "context rot" and degraded reasoning. To combat this, teams are shifting from static, monolithic prompts to progressive context disclosure. Instead of front-loading every instruction and reference an agent might need, these systems start with a lightweight index of what's available. The agent determines what prompts or contexts are relevant and pulls in only what's needed, keeping the signal-to-noise ratio sharp at every step.

We're seeing several techniques mature in this space: Context setup leverages prompt caching to front-load static instructions, reducing costs and improving time to first token. Dynamic retrieval goes beyond basic RAG by selecting tools and loading only the necessary MCP servers, avoiding unnecessary context expansion. Context graphs model institutional reasoning — such as policies, exceptions and precedents — as structured, queryable data. Context management techniques use stateful compression and sub-agents to summarize intermediate outputs in long-running workflows.

Treating AI context as a static text box is a fast track to hallucinations. To build resilient enterprise agents, teams must engineer context as a dynamic, precisely managed pipeline.

2. Curated shared instructions for software teams

Adopt

As teams mature in their use of AI, relying on individual developers to write prompts from scratch is emerging as an anti-pattern. We advocate for curated shared instructions for software teams, treating AI guidance as a collaborative engineering asset rather than a personal workflow.

Initially, this practice focused on maintaining general-purpose prompt libraries for common tasks. We're now seeing a more effective evolution specifically for coding environments: anchoring these instructions directly into service templates. By placing instruction files such as `CLAUDE.md`, `AGENTS.md` or `.cursorrules` into the baseline repository used to scaffold new services, the template becomes a powerful distribution mechanism for AI guidance.

During our Radar discussions, we also explored a related practice: anchoring coding agents to a reference application. Here, a live, compilable codebase serves as the source of truth. As architecture and coding standards evolve, both the reference application and embedded instructions can be updated. New repositories then inherit the latest agent workflows and rules by default. This approach ensures consistent, high-quality AI assistance is built into every project from day one, while separating general prompt libraries from repository-specific AI configuration.

3. DORA metrics

Adopt

The metrics defined by the DORA research program have been widely adopted and have proven to be strong leading indicators of how a delivery organization is performing. These include change lead time, deployment frequency, mean time to restore (MTTR), change failure rate and a newer fifth metric, rework rate. Rework rate is a stability metric that measures how much of a team's delivery

pipeline is consumed by unplanned rework to fix work previously considered complete, such as user-facing bugs or defects.

In the era of AI-assisted software development, the DORA metrics are more important than ever. Measuring productivity by lines of code generated by AI is misleading; real improvement must be reflected in delivery flow and stability. If lead times don't decrease and deployment frequency doesn't increase, faster code generation doesn't translate into better outcomes. Conversely, degradation in stability metrics — particularly rework rate — provides an early warning sign of blind spots, technical debt and the risks of unchecked AI-assisted development.

As always, we recommend using these metrics for team reflection and learning rather than just building complex dashboards. Simple mechanisms, such as check-ins during retrospectives, are often more effective than overly detailed tracking tools at improving capabilities.

4. Passkeys

Adopt

Shepherded by the FIDO Alliance and backed by Apple, Google and Microsoft, passkeys have matured into Adopt. They are FIDO2 credentials that can replace passwords using asymmetric public-key cryptography. The private key is stored in a hardware-backed secure enclave on the user's device, protected by biometrics or a PIN, and never leaves it. Each credential is origin-bound to its relying-party domain, making passkeys structurally phishing-resistant: a lookalike site receives nothing, unlike SMS OTP or TOTP codes that a phishing proxy can intercept.

With phishing responsible for more than one third of all data breaches, this structural resistance is increasingly important. The FIDO Alliance Passkey Index 2025 reports there are over 15 billion eligible accounts globally, Google reports a 30% improvement in sign-in success rates across 800 million users and Amazon has seen sign-ins six times faster than using traditional methods. NIST SP 800-63-4 (July 2025) now classifies synced passkeys as AAL2-compliant, reversing earlier guidance, and regulators in the UAE, India and US federal agencies mandate phishing-resistant authentication for financial services and government systems.

The FIDO Credential Exchange Protocol enables secure portability of passkeys between credential managers, addressing earlier vendor lock-in concerns. Major identity providers including Auth0, Okta and Azure AD now support passkeys as a first-class feature, and implementation has been simplified from a multi-month effort to a two-sprint project. We've adopted passkeys internally and treat them as the default starting point for new authentication implementations. Teams should design account recovery carefully and avoid phishable fallback paths such as SMS OTP, which reintroduce the vulnerabilities passkeys eliminate. Device-bound credentials on hardware security keys remain necessary for AAL3 scenarios such as privileged access.

5. Structured output from LLMs

Adopt

Structured output from LLMs is the practice of constraining models to produce responses in a predefined format, such as JSON or a specific programming language class. We continue to see this technique deliver reliable results in production, and we now consider it a sensible default for applications that consume LLM responses programmatically. All major model providers now offer native structured output modes, but implementations differ in the JSON Schema subsets they support, and these APIs continue to evolve rapidly. We recommend using a library such as Instructor or a framework like Pydantic AI to provide a stable abstraction across providers with validation and automatic retries, or Outlines for constrained generation on self-hosted models.

6. Zero trust architecture

Adopt

As we enter the age of agents, many enterprises are grappling with how to build them while addressing the security risks of granting autonomy to unpredictable systems. Zero trust architecture (ZTA) remains a sensible default for securely building and operating agents. Principles such as "never trust, always verify," along with identity-based security and least-privilege access, should be treated as foundational for any agent deployment. Our teams are applying standards like SPIFFE to agents, establishing strong identity foundations and enabling fine-grained authentication in dynamic environments. Continuous monitoring and verification of agent behavior are also critical for proactively managing threats. Beyond agent deployments, our teams are adopting practices such as OIDC impersonation in GCP for different applications, including CI/CD pipelines, replacing long-lived static keys with short-lived tokens issued after identity verification. We recommend teams treat ZTA principles as non-negotiable defaults, regardless of the system being built.

7. Agent Skills

Trial

As AI agents evolve from simple chat interfaces toward autonomous task execution, context engineering has become a critical challenge. Agent Skills provide an open standard for modularizing context by packaging instructions, executable scripts and associated resources such as documentation. Agents load skills only when needed based on their descriptions, which reduces token consumption and mitigates context window exhaustion and problems such as agent instruction bloat.

Skills have been adopted very quickly, not only in coding agents but also in personal assistants such as OpenClaw. They're also one reason teams are becoming more cautious about defaulting to MCP, as many use cases can be addressed just as effectively by pointing an agent at a local CLI or script.

As their popularity has grown, the surrounding ecosystem has expanded as well. Plugin marketplaces are emerging as a way to version and share skills, and multiple efforts are exploring how to evaluate skill effectiveness. We do, however, caution against unreviewed reuse of third-party skills, as they introduce serious supply chain security risks.

8. Browser-based component testing

Trial

In the past, when discussing component testing, we've generally advised against browser-based tools. They were difficult to set up, slow to run and often flaky. This has improved significantly. Today, browser-based component testing, using tools such as Playwright, is a viable and often preferable approach. Running tests in a real browser provides more consistency, as the test matches the environment where the code actually executes. The performance hit is now small enough that the trade-off is worthwhile. Flakiness has also decreased, and we're seeing more value than from emulated environments such as jsdom.

9. Feedback sensors for coding agents

Trial

To make coding agents more effective and reduce the load on human reviewers, teams need feedback loops that agents can directly access. These feedback sensors for coding agents act as a form of feedback backpressure, increasing trust in generated results. Developers have long relied on deterministic quality gates such as compilers, linters, structural tests and test suites; here, they're wired into agentic workflows so that failures trigger timely self-correction.

These checks reduce routine steering work for the human in the loop. Teams can implement them in different ways, such as introducing a reviewer agent responsible for running checks and triggering corrections, or by exposing the checks through a companion process that runs in parallel that agents can query efficiently. Coding agents also make it cheaper to build custom linters and structural tests, further strengthening these feedback loops. Whenever possible, these sensors should run during the coding session and report clean results before a commit is made, rather than relying on post-commit checks.

10. Mapping code smells to refactoring techniques

Trial

Mapping code smells to refactoring techniques means instructing an agent to handle specific issues with a defined approach. The first layer typically points the agent to a generic reference, such as *Refactoring*, for common cases. For more specialized issues, teams can map unique smells to specific techniques using Agent Skills, slash commands or AGENTS.md. When integrated with linting tools, this creates deterministic feedback by triggering the appropriate refactoring approach whenever a smell is detected.

This is particularly effective for legacy stacks like .NET Framework 2.0 or Java 8, where generic training data often falls short. It's also useful for teams with distinctive engineering standards. Without these targeted instructions, an agent will tend to default to generic patterns rather than follow specific requirements.

11. Mutation testing

Trial

Mutation testing remains the most honest signal for evaluating the real fault-detection capability of a test suite. Unlike traditional code coverage, which only tracks line execution, this technique introduces deliberate bugs, or mutations, into source code to verify that tests fail when behavior breaks. If a mutation goes undetected, it reveals a gap in validation rather than just a lack of coverage. This distinction is critical in an era of AI-assisted development, where high coverage percentages can mask logically hollow tests or generated code that has never been meaningfully asserted.

With AI-generated test cases now commonplace, mutation testing acts as a reinforcement layer for catching "perpetually green" tests — those that pass regardless of logic changes due to missing assertions or decoupled mocks. By using tools such as Stryker, Pitest or cargo-mutants, we shift the focus from how much code is executed to how much code is actually verified, particularly in core domain logic. The goal is to ensure that a passing test suite is a reliable signal of functional correctness, rather than simply a report of which lines were executed.

12. Progressive context disclosure

Trial

Progressive context disclosure is a technique within the practice of context engineering. Instead of overwhelming an agent with instructions upfront, you give it a lightweight discovery phase in which it selects what it needs based on the user's prompt, loading detailed information into the context window only when it becomes relevant.

This works great for RAG scenarios, where an agent first identifies the relevant domain from user queries and then retrieves specific instructions and data accordingly. It's also how many agentic coding tools handle Agent Skills by first determining which skills are relevant to a task before loading detailed instructions, rather than providing a single, monolithic instruction set filled with conditions

and caveats. When building agentic systems, it's easy to fall into the trap of bloating instructions with endless "DO" and "DO NOT" rules in an attempt to control behavior, which can ultimately degrade performance. Progressive context disclosure avoids this by ensuring the agent receives the right guidance at the right moment, keeping the context window lean and preventing context rot.

13. Sandboxed execution for coding agents

Trial

Sandboxed execution for coding agents is the practice of running agents inside isolated environments with restricted file system access, controlled network connectivity and bounded resource usage. As coding agents gain autonomy to execute code, run builds and interact with the file system, giving agents unrestricted access to a development environment introduces real risks, from accidental damage to credential exposure. We see sandboxing as a sensible default rather than an optional enhancement.

The landscape of sandboxing options now spans a broad spectrum. At one end, many coding agents offer built-in sandbox modes, and Dev Containers provide familiar container-based isolation. At the other, purpose-built tools take different positions on the ephemeral versus persistent trade-off. Shuru boots disposable microVMs that reset on every run, while Sprites provides stateful environments with checkpoint and restore. For Linux-native isolation, Bubblewrap offers lightweight namespace-based sandboxing, and on macOS, `sandbox-exec` provides similar protection.

Beyond basic isolation, teams should consider the practical requirements of a productive sandbox. This includes everything needed for building and testing, as well as secure, straightforward authentication with services like GitHub and model providers. Developers need port forwarding and sufficient CPU and memory for agent workloads. Whether the sandbox should be ephemeral by default or persistent for session recovery is a design decision that will depend on a team's priorities for security, cost and workflow continuity.

14. Semantic layer

Trial

Semantic layer is a data architecture technique that introduces a shared business logic layer between data stores and consuming applications, including business intelligence (BI) tools, AI agents and APIs. It centralizes metric definitions, joins, access rules and business terminology so consumers have shared definitions. The concept predates the modern data stack but has seen renewed interest with code-first approaches such as metrics stores.

Without a semantic layer, business logic scatters across ad-hoc warehouse tables, dashboards, and downstream applications, while metric definitions quietly diverge—particularly problematic when used to support business decisions. Our teams have seen this become more acute with agentic AI: using LLMs to perform naive text-to-SQL translations will frequently produce incorrect results, especially when business rules, such as revenue recognition, live outside the schema. Cloud platforms are now embedding semantic layers directly: Snowflake calls it Semantic Views and Databricks calls it Metric Views. Standalone tools such as dbt MetricFlow and Cube provide a portable layer across systems. The recent release of Open Semantic Interchange (OSI) v1.0, backed by multiple vendors, signals growing standardization and interoperability across analytics, AI, and BI platforms.

The main cost is upfront data modeling investment. Teams should start with a single domain rather than attempting an enterprise-wide rollout, as broad deployments often leave legacy reports running in parallel with the new layer, reintroducing inconsistent definitions.

15. Server-driven UI

Trial

Server-driven UI is returning to the Trial ring as we see more teams successfully shortening their path to production. By separating rendering into a generic container while providing structure and data via the server, mobile teams can bypass lengthy app store review cycles for every iteration. We've seen this significantly improve time to market, with JSON-based formats enabling real-time updates. While we previously cautioned against the "horrendous, overly configurable messes" that proprietary frameworks can create, the emergence of more stable patterns from companies such as Airbnb and Lyft has helped reduce complexity.

Our experience shows that the substantial investment required for a proprietary framework is now easier to justify for large-scale applications. However, it still requires a strong business case and disciplined engineering to avoid creating a "god-protocol" that becomes difficult to maintain. For teams seeking to reduce client-side complexity, this approach provides a powerful way to scale across teams and synchronize logic across platforms. We recommend applying it to highly dynamic areas of an application rather than as a blanket replacement for all UI development

16. Agentic reinforcement learning environments

Assess

Agentic reinforcement learning environments provide a training ground for LLM-based agents, combining context, tools and feedback to complete multi-step tasks. This approach reframes post-training of LLMs from simple single-turn outputs to agentic behaviors such as reasoning and tool use, with rewards or penalties assigned to each action. Techniques such as RLVR help ensure these rewards are verifiable and resistant to gaming.

AI research labs are currently driving the development of these environments, particularly for coding and computer-use agents. One example outside frontier labs is Cursor's Composer, a specialized coding model trained within their product environment. Organizations building agentic systems should consider whether creating reinforcement learning environments could help produce more capable and domain-specific models.

Setting up the required infrastructure can be complex. However, frameworks and platforms are emerging to simplify the process, including Prime Intellect's Environments Hub, Agent Lightning and NVIDIA NeMo Gym. We recommend exploring this approach where it can deliver more capable and cost-effective models for your domain.

17. Architecture drift reduction with LLMs

Assess

Increased use of AI coding agents can accelerate drift from the intended codebase and architecture designs. Left unchecked, this drift compounds as agents and humans replicate existing patterns, including degraded ones. This creates a feedback loop where poor code begets poorer code. Some of our teams are now addressing architecture drift reduction with LLMs.

This approach combines deterministic analysis tools (such as Spectral, ArchUnit or Spring Modulith) with LLM-powered evaluation to detect both structural and semantic violations. LLMs are then used to help fix these issues. Our teams have applied this to enforce API quality guidelines across services and to define architectural zones that guide agent-generated improvements.

A few lessons learned: like traditional linting, initial scans can surface a large number of violations that require triage and prioritization; LLMs can assist with this process. Keeping agent-produced fixes small and focused makes review easier, and an additional verification loop is essential to confirm changes improve the system rather than introduce regressions.

This technique extends the idea of feedback sensors for coding agents into the later stages of the delivery lifecycle. As one team at OpenAI describes it, drift reduction acts as a form of "garbage collection," reflecting the reality that entropy and decay emerge even in systems with strong early feedback loops.

18. Code intelligence as agentic tooling

Assess

LLMs process code as a stream of tokens; they have no native understanding of call graphs, type hierarchies or symbol relationships. For code navigation, most coding agents today default to text-based search, the most powerful common denominator across all languages. For refactorings that are a quick shortcut in an IDE, agents need to generate multiple textual diffs. As a result, agents end up spending significant tokens reconstructing information that already exists in the abstract syntax tree (AST).

Code intelligence as agentic tooling gives agents access to tools that are aware of the AST, e.g. via the Language Server Protocol (LSP). Through these integrations, agents can perform operations such as "find all references to this symbol" or "rename this type everywhere" as first-class actions rather than relying on fragile text substitutions. Another powerful code intelligence integration are codemod tools like OpenRewrite, which operates on the even richer Lossless Semantic Tree (LST) representation of the code. The result is fewer hallucinated edits and lower token consumption by delegating appropriate tasks to deterministic tools.

Claude Code, OpenCode and others integrate with locally running LSP servers; JetBrains provides an MCP server that exposes IDE navigation and refactoring capabilities to external agents, while the Serena MCP server offers semantic code retrieval and editing.

19. Context graph

Assess

A context graph is a knowledge representation technique where decisions, policies, exceptions, precedents, evidence and outcomes are modeled as first-class connected nodes in a graph, structured for AI consumption. Where systems of record capture what happened, a context graph captures why, turning institutional reasoning buried in Slack threads, approval chains and people's heads into a queryable, machine-readable structure. This is vital for agent effectiveness; an agent handling a discount exception, for example, cannot determine whether it reflects standing policy or a one-time override and may reason incorrectly. A context graph can directly surface that provenance, enabling agents to traverse decision traces, apply relevant precedents and reason across multi-hop causal chains.

Unlike GraphRAG, which builds from static document corpora, a context graph maintains temporal validity on every edge, so superseded facts are invalidated rather than overwritten. Context graphs are worth assessing for agentic applications that require persistent memory across sessions or traceable decision reasoning.

20. Feedback flywheel

Assess

Teams working with coding agents are increasingly adopting spec-driven development workflows. Whether they use a lightweight or opinionated framework, these workflows typically follow a similar flow of spec → plan → implement. The feedback flywheel extends this flow with an additional step focused on continuously improving the coding agent harness.

The approach is similar to retrospectives: teams capture successes and failures during a coding agent session and use them to improve the predictability of future sessions, which compounds over time. It's a meta-technique where a human on the loop focuses on improving the feedforward controls such as curated shared instructions as well as feedback sensors for coding agents. Our teams have found this effective, as it is analogous to code refactoring. The next level looks more like an agentic feedback flywheel, where, based on accumulated feedback, the agent decides what improvements are necessary. For now, however, teams still need a human-in-the-loop to avoid context rot and noisy feedback that could lead the agent astray.

We suggest using this approach to evaluate the entire coding agent harness as your environment evolves, especially when adopting new models; what worked with one model may not be necessary with the next.

21. HTML Tools

Assess

Since agentic tools make it easy to build small, task-specific utilities, the main challenge is often how to deploy and share them. HTML Tools is an approach where a shareable script or utility is packaged as a single HTML file. You can run these directly in a browser, host them anywhere, or simply share the file. This approach avoids the overhead of distributing CLI tools, which require sharing binaries or using package managers. It's also simpler than building a full web application with dedicated hosting. From a security perspective, running untrusted files still carries risk, but the browser sandbox and the ability to inspect source code provide some mitigation. For lightweight utilities, a single HTML file offers a highly accessible and portable way to share tools.

22. LLM evaluation using semantic entropy

Assess

Confabulation, a form of hallucination in LLM QA applications, is difficult to address with traditional evaluation methods. One approach uses information entropy as a measure of uncertainty by analyzing lexical variation in outputs for a given input. LLM evaluation using semantic entropy extends this idea by focusing on differences in meaning rather than surface-level variation.

This approach evaluates meaning rather than word sequences, making it applicable across datasets and tasks without requiring prior knowledge. It generalizes well to unseen tasks, helping identify prompts likely to cause confabulations and encouraging caution when needed. Results show that naive entropy often fails to detect confabulations, while semantic entropy is more effective at filtering false claims.

23. Measuring collaboration quality with coding agents

Assess

We're seeing real productivity gains when using coding agents, but most evaluation metrics still focus too heavily on coding throughput, such as time to first output, lines of code generated and

tasks completed. Measuring collaboration quality with coding agents helps teams avoid falling into "the speed trap" by shifting focus toward how effectively humans and agents work together. Metrics such as first-pass acceptance rate, iteration cycles per task, post-merge rework, failed builds and review burden provide more meaningful signals than speed alone. Teams using Claude Code can use the `/insights` command to generate reports reflecting on successes and challenges from agent sessions. Our teams have also experimented with tracking first-pass acceptance of a customized `/review` command.

In practice, shorter feedback cycles and fewer failed builds indicate more effective interaction with coding agents. When teams find themselves in repeated back-and-forth with their agents, these metrics highlight opportunities to improve the feedback flywheel. We recommend tracking collaboration quality at the team level, rather than the individual level, alongside DORA metrics to build a more complete picture of coding agent adoption.

24. MITRE ATLAS

Assess

Agentic systems and coding tools introduce new architectures and emergent security threats. MITRE ATLAS is a knowledge base of adversarial tactics and techniques targeting AI and ML systems. More focused than the broader MITRE ATT&CK framework and designed to complement it, ATLAS provides a taxonomy of threats for ML pipelines, LLM applications and agentic systems.

We've found that without a shared vocabulary, security risks are often overlooked or reduced to a checkbox exercise. This is where ATLAS can help. Grounded in research on real incidents and technology patterns, teams can use the framework to support threat modeling. Teams may also find it a natural complement to control frameworks such as SAIF, helping describe the evolving threat landscape for AI systems.

25. Ralph loop

Assess

The Ralph loop (also sometimes called the Wiggum loop) is an autonomous coding agent technique where a fixed prompt is fed to an agent in an infinite loop. Each iteration starts with a fresh context window: the agent selects a task from a specification or plan, implements it, and the loop restarts with a fresh context. The core insight is simplicity. Rather than orchestrating teams of coding agents or coding agent swarms, a single agent works autonomously against a specification, with the expectation that the codebase will converge toward the spec over repeated iterations. Using a fresh context window on each iteration avoids the quality degradation that comes from accumulated context, though at significant token cost. Tools such as goose have implemented the pattern, in some cases extending it with cross-model review between iterations.

26. Reverse engineering for design system

Assess

Organizations often struggle with fragmented legacy interfaces where the "design standard" exists only as a loose collection of disjointed webpages, marketing materials and screenshots. Historically, auditing these artifacts to establish a unified foundation has been a manual and time-consuming process. With multimodal LLMs, this extraction can now be automated, effectively reverse-engineering design systems from existing visual assets.

By feeding websites, screenshots and UI fragments into specialized tools or vision-capable AI models, teams can extract core design tokens — such as color palettes, typography scales and spacing rules — and identify recurring component patterns. The AI then synthesizes this unstructured visual data into a structured, semantic representation of a design system. When integrated with tools such as Figma, this output can significantly accelerate the creation of a formalized, maintainable component library.

Beyond reducing effort in visual audits, this technique can serve as a stepping stone toward building "AI-ready" design systems. For enterprises burdened by brownfield design debt, using AI to establish a baseline design system is a practical starting point before a full redesign or front-end standardization.

27. Role-based contextual isolation in RAG

Assess

Role-based contextual isolation in RAG is an architectural technique that moves access control from the application layer down to the retrieval layer. Every data chunk is tagged with role-based permissions at indexing time. At query time, the retrieval engine restricts the search space based on the user's authenticated identity, which is matched against metadata on each chunk. This ensures the AI model cannot access unauthorized context because it's filtered out at the retrieval stage. This provides a zero trust foundation for internal knowledge bases. As many vector databases now support high-performance metadata filtering, such as Milvus or services built on Amazon S3, this technique has become more practical to adopt, even for large knowledge bases.

28. Skills as executable onboarding documentation

Assess

Agent Skills, curated shared instructions and many other context engineering techniques appear throughout this edition of the Radar. One use case we want to highlight in the coding context is the use of skills as executable onboarding documentation. This technique applies at multiple levels. Within a codebase, a `/_setup` skill can take on the role of a go.sh script and a README file, combining scripting with LLM-executed semantics for steps that cannot be scripted. It can also go beyond what a script can do by dynamically taking the current state of the codebase and environment into account. Secondly, library and API creators can provide skills for their consumers as part of their documentation through internal or external skill registries like Tessl. And thirdly, we've found this useful for onboarding teams to internal platforms to lower the barrier to using a key technology or reduce friction when adopting a design system. So far, our experience with this has relied heavily on MCP servers but is now shifting toward skills.

As with other forms of documentation, the challenge of keeping this up to date doesn't go away. However, unlike static documentation, executable documentation can help you notice staleness much earlier.

29. Small language models

Assess

Small language models (SLMs) continue to improve and are beginning to offer better intelligence per dollar than LLMs for certain use cases. We've seen teams evaluate SLMs to reduce inference costs and speed up agentic workflows. Recent progress shows steady gains in intelligence density, making SLMs competitive with older LLMs for tasks such as summarization and basic coding. This shift reflects a move away from "bigger is better" toward higher-quality data, model distillation and quantization. Models such as Phi-4-mini and Ministral 3 3B demonstrate how distilled models can

retain many capabilities of larger teacher models. Even ultra-compact models such as Qwen3-0.6B and Gemma-3-270M are becoming viable for running models on edge devices. For agentic use cases where older LLMs have been sufficient, teams should consider SLMs as a lower-cost, lower-latency alternative with reduced resource requirements.

30. Team of coding agents

Assess

In the previous Radar, we described a team of coding agents as a technique where a developer orchestrates a small set of role-specific agents to collaborate on a coding task. Since then, the barrier to adoption has dropped. Subagent support has become more of a table stakes feature across established coding agent tools, and Claude Code now includes an agent teams feature that provides built-in orchestration. In a team of agents, a primary orchestrator typically coordinates task sequencing and parallelization. Agents should be able to communicate not only with the orchestrator but also with one another. Common use cases include teams of reviewers or groups of implementers responsible for different parts of the application, such as backend and frontend.

Although some in the industry are using the terms “agent teams” and “agent swarms” interchangeably (for example, Claude Code describes its agent teams feature as “our implementation of swarms”), we see value in distinguishing between them. A small, deliberate team of agents collaborating on a task differs significantly from a large swarm in terms of entry barriers, complexity and use cases.

31. Temporal fakes

Assess

Temporal fakes extend the idea of simulating real-world systems for development and testing, a practice long used in IoT and industrial platforms. With AI coding agents reducing the effort required to build such simulators, teams can now create high-fidelity replicas of external dependencies much more easily. Unlike traditional mocks that return static request–response pairs, temporal fakes maintain internal state machines and model the temporal evolution of real systems.

One of our teams used this technique while developing an observability stack for large GPU data centers, avoiding the need to procure physical hardware. Testing alert rules, dashboards and anomaly detection against real systems can be impractical — for example, intentionally overheating a GPU to validate a thermal throttle alert. Instead, the team built fakes for hardware domains such as NVIDIA DCGM and InfiniBand fabric using Go. These simulators enabled failure scenarios such as thermal throttling, XID error storms, link flaps and PSU failures with configurable intensity and duration, orchestrated via a process-compose stack.

A central registry defined valid failure scenarios, while an MCP server exposed scenario injection to the agent. The agent could trigger faults, for example, injecting a thermal throttle on a specific GPU and verify that metrics changed, alerts fired and dashboards updated as expected. This temporal fidelity makes the technique valuable for testing complex systems where failures cascade. However, teams must ensure the fakes remain faithful to real-world behaviour; otherwise, they risk creating false confidence in automated pipelines.

32. Toxic flow analysis for AI

Assess

Agent capabilities are outpacing security practices. With the rise of permission-hungry agents like OpenClaw, teams are increasingly deploying agents in environments that expose them to the lethal trifecta: access to private data, exposure to untrusted content and the ability to communicate

externally. As capabilities grow, so too does the attack surface, exposing systems to risks such as prompt injection and tool poisoning. We continue to see toxic flow analysis as a primary technique for examining agentic systems to identify unsafe data paths and potential attack vectors. These risks are no longer limited to MCP integrations; our teams have observed similar patterns in Agent Skills, where a malicious actor can package a seemingly useful skill that embeds hidden instructions to exfiltrate sensitive data. We strongly encourage teams working with agents to perform toxic flow analysis and use tools such as Agent Scan to identify unsafe data paths before they're exploited.

33. Vision language models for end-to-end document parsing

Assess

Document parsing often relies on multi-stage pipelines combining layout detection, traditional OCR and post-processing scripts. These approaches often struggle with complex layouts and mathematical formulas. Vision language models (VLMs) for end-to-end document parsing simplify this architecture by treating the document image as a single input modality, preserving natural reading order and structured content. Open-source models specifically trained for this purpose — such as olmOCR-2, the token-efficient DeepSeek-OCR (3B) and the ultra-compact PaddleOCR-VL — have yielded highly efficient results. While VLMs reduce architectural complexity by replacing multi-stage pipelines, their generative nature makes them prone to hallucinations. Use cases with a low tolerance for error may still require a hybrid approach or deterministic OCR. Teams dealing with high-volume document ingestion should evaluate these unified approaches to determine whether they can replace complex legacy pipelines while maintaining accuracy and reducing long-term maintenance overhead.

34. Agent instruction bloat

Caution

Context files such as AGENTS.md and CLAUDE.md tend to accumulate over time as teams add codebase overviews, architectural explanations, conventions and rules. While each addition is useful in isolation, this often leads to agent instruction bloat. Instructions become long and sometimes conflict with each other. Models tend to attend less to content buried in the middle of long contexts, so guidance deep in a long conversation history can be missed. As instructions grow, the likelihood increases that important rules are ignored. We also see many teams using AI to generate AGENTS.md files, but research suggests hand-written versions are often more effective than LLM-generated ones. When using agentic tools, be deliberate and selective with instructions, adding them as needed and continuously refine toward a minimal, coherent set. Consider leveraging progressive context disclosure to surface only the instructions and capabilities an agent needs for its current task.

35. AI-accelerated shadow IT

Caution

AI continues to lower the barriers for noncoders to build complex systems. While this enables experimentation and early validation of requirements, it also introduces the risk of AI-accelerated shadow IT. In addition to no-code workflow platforms integrating AI APIs (e.g., OpenAI or Anthropic), more agentic tools are becoming available to noncoders, such as Claude Cowork.

When the spreadsheet that quietly runs the business evolves into customized agentic workflows that lack governance, it introduces significant security risks and a proliferation of competing solutions to similar problems. Distinguishing between disposable, one-off workflows and critical processes that require durable, production-ready implementation is key to balancing experimentation with control.

Organizations should prioritize governance as part of their AI adoption strategy by facilitating experimentation within controlled environments. Appropriately instrumented Internal sandboxes give noncoders a place to deploy prototypes where usage can be tracked. Pairing these with a shared catalogue of existing workflows helps teams discover what's already been built before duplicating effort. Workflows that gain traction can then signal where to invest in more robust, production-grade applications.

36. Codebase cognitive debt

Caution

Codebase cognitive debt is the growing gap between a system's implementation and a team's shared understanding of how and why it works. As AI increases change velocity, especially with multiple contributors or Coding Agent Swarms, teams can lose track of design intent and hidden coupling. This, combined with rising technical debt, creates a reinforcing loop that makes systems progressively harder to reason about.

Weaker system understanding also reduces developers' ability to guide AI effectively, making it harder to anticipate edge cases and steer agents away from architectural pitfalls. Left unmanaged, teams reach a tipping point where small changes trigger unexpected failures, fixes introduce regressions and cleanup efforts increase risk instead of reducing it.

Teams should avoid complacency with AI-generated code and adopt explicit countermeasures: feedback sensors for coding agents, tracking team cognitive load and architectural fitness functions to continuously enforce key constraints as AI accelerates output.

37. Coding agent swarms

Caution

Where a team of coding agents is a small, deliberate group, a coding agent swarm applies dozens to hundreds of agents to a problem, with AI determining composition and size dynamically. Projects such as Gas Town and Ruflo (formerly Claude Flow) are good examples of this approach. Early patterns for swarm implementations are emerging: hierarchical role separation (orchestrators, supervisors and ephemeral workers), a durable work ledger that helps agents divide and coordinate work (Gas Town uses beads for this) and a merging mechanism to handle conflicts from parallel work.

Two swarm experiments have drawn particular attention: Anthropic's C compiler generation and Cursor's agent scaling experiment that created a browser over a week. It's worth noting that both teams chose use cases that could rely on existing detailed specifications, and in the case of the C compiler, comprehensive test suites that provide clear, measurable feedback. Those conditions are not representative of typical product development, where requirements are less defined and verification is harder. Nevertheless, these experiments contribute to emerging patterns for making long-running swarms technically viable. They remain costly and are still far from mature, which is why we advise caution when adopting this technique.

38. Coding throughput as a measure of productivity

Caution

AI coding assistants are delivering real productivity gains and are rapidly becoming standard developer tooling. However, we're increasingly seeing organizations measure success using superficial indicators such as lines of code generated or the number of pull requests (PRs). When these coding throughput metrics are used in isolation, they can negatively shape employee behavior. The result is often a flood of poorly aligned code that slows reviews, harms delivery throughput and introduces

security risks. Cycle times increase as engineers raise PRs filled with insufficiently reviewed AI output, leading to repeated back-and-forth with reviewers. These metrics fail to capture the residual effort required to adapt AI-generated code to a team's architecture, conventions and patterns.

More meaningful leading indicators exist, such as first-pass acceptance rate — how often AI output can be used with minimal rework. Measuring this exposes hidden effort and makes improvement actionable: teams can refine prompts, improve priming documents and strengthen design conversations to progressively increase acceptance over time. This creates a virtuous cycle in which AI output requires less correction. First-pass acceptance also connects naturally with DORA metrics: lower acceptance rates tend to increase change failure rates, while repeated iteration cycles extend lead time for changes. As AI assistants become ubiquitous, organizations should shift focus away from coding throughput alone toward metrics that reflect real impact and delivery outcomes.

39. Ignoring durability in agent workflows

Caution

Ignoring durability in agent workflows is an anti-pattern we've seen across many teams, resulting in systems that work in development but fail in production. The challenges facing distributed systems are even more pronounced when building with agents. A mindset that expects failures and recovers gracefully outpaces a reactive approach.

LLM and tool calls can fail due to network interruptions and server crashes, halting an agent's progress and leading to poor user experience and increased operational costs. Some systems can tolerate this when tasks are short-lived, but complex workflows that run for days or weeks require durability.

Fortunately, durable execution is being integrated into agent frameworks such as LangGraph and Pydantic AI. It provides stateful persistence of progress and tool calls, enabling agents to resume tasks after failures. For workflows that involve a human in the loop, durable execution can suspend progress while awaiting input. Durable computing platforms such as Temporal, Restate and Golem also provide support for agents. Built-in observability of tool execution and decision tracking makes debugging easier and improves understanding of systems in production. Teams should start with native durable execution support in their agent framework and reach for standalone platforms as workflows become more critical or complex.

40. MCP by default

Caution

As the Model Context Protocol (MCP) gains traction, we're seeing teams and vendors reach for it as the default integration layer between AI agents and external systems, even when simpler alternatives exist. We caution against using MCP by default. MCP adds real value for structured tool contracts, OAuth-based authentication boundaries and governed multi-tenant access. It also introduces what Justin Poehnelt calls an "abstraction tax": every protocol layer between an agent and an API loses fidelity, and for complex APIs those losses compound.

In practice, a well-designed CLI with good `--help` output, structured JSON responses and predictable error handling often gives agents everything they need without the protocol overhead. As Simon Willison notes, "almost everything I might achieve with an MCP can be handled by a CLI tool instead."

This isn't a rejection of MCP. Teams should avoid adopting it by default and first ask whether their system actually requires protocol-level interoperability. MCP makes sense when its governance and integration benefits outweigh the added complexity and potential fidelity loss.

41. Pixel-streamed development environments

Caution

Pixel-streamed development environments use VDI-style remote desktops or workstations for software development, with editing, builds and debugging performed through a streamed desktop rather than on a local machine or a code-centric remote environment. We continue to see organizations adopt them to meet security, standardization and onboarding goals, especially for offshore teams and lift-and-shift cloud programs. In practice, however, the trade-off is often poor: latency, input lag and inconsistent screen responsiveness create constant cognitive friction that slows delivery and makes everyday development work more tiring. Unlike development environments in the cloud, such as Google Cloud Workstations, or tools like Coder and VS Code Remote Development, which move compute closer to the code without streaming the entire desktop, pixel-streamed setups prioritize centralized control over developer flow and are often imposed with too little input from the engineers who use them. We advise against pixel-streamed development environments as a default choice for software delivery unless a compelling security or regulatory constraint clearly outweighs the productivity cost.

Platforms

Adopt

—

Trial

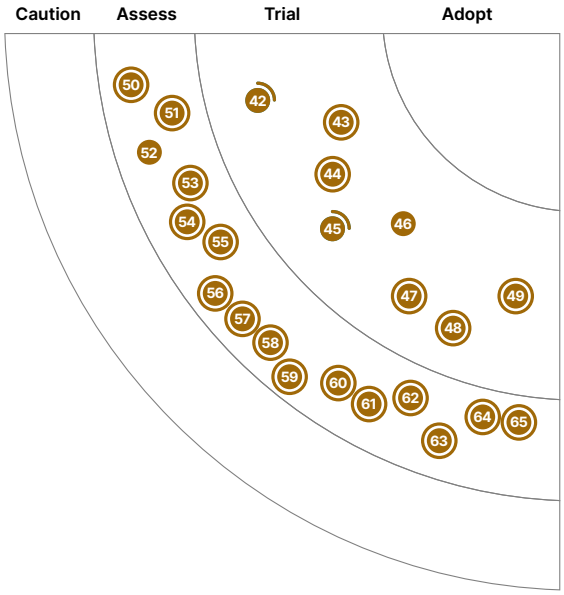
42. AG-UI Protocol
43. Apache APISIX
44. AWS Bedrock AgentCore
45. Graphiti
46. Langfuse
47. Port
48. Replit
49. SigNoz

Assess

50. Agent Trace
51. ClickStack
52. Coder
53. Databricks Agent Bricks
54. DuckLake
55. FalkorDB
56. Google Dialogflow CX
57. MCP Apps
58. Monarch
59. Neutree
60. OptScale
61. Rhesis
62. RunPod
63. Sprites
64. torchforge
65. torchtitan

Caution

—



New Moved in/out No change

42. AG-UI Protocol

Trial

AG-UI is an open protocol and library designed to standardize communication between rich user interfaces and back-end AI agents. Historically, building agentic UIs required bespoke plumbing for bidirectional, stateful collaboration. AG-UI addresses this by providing a consistent, event-driven architecture — supporting transports such as server-sent events (SSE) and WebSockets — for streaming reasoning steps, synchronizing state and rendering dynamic UI components.

However, the architectural landscape for agent interfaces is shifting rapidly. AG-UI intentionally sits outside MCP, functioning as an interface layer between the frontend and the agent backend. We're now seeing a different approach emerge, where newer MCP-based applications package HTML and UI widgets directly within MCP servers or skills.

Because UI components can now be embedded and served alongside the tools themselves — a pattern related to emerging adjacent standards such as MCP-UI — the need for a separate UI protocol layer such as AG-UI is being questioned. While AG-UI remains a solid choice for decoupling front-end UX from back-end orchestration, teams should assess its role in light of the growing trend toward consolidating tool logic and UI within the MCP ecosystem.

43. Apache APISIX

Trial

Apache APISIX is an open-source, high-performance, cloud-native gateway that addresses the limitations of legacy Nginx-based solutions. Built on Nginx and LuaJIT via OpenResty, it uses etcd for configuration storage to eliminate reload-induced latency, making it well-suited for dynamic microservices and serverless architectures. Its primary strength is its fully dynamic, pluggable architecture, which allows teams to customize traffic management, security and observability through APIs and a multi-language plugin ecosystem, including WASM. By supporting the Kubernetes Gateway API, Apache APISIX can be used as a Kubernetes gateway and is a strong candidate for replacing legacy Nginx ingress controllers. Some of our teams are adopting Apache APISIX and find its performance and feature set compelling.

44. AWS Bedrock AgentCore

Trial

AWS Bedrock AgentCore is the agentic platform for building, running and operating agents at scale, securely, without the overhead of infrastructure management, similar to GCP Vertex AI Agent Builder and Azure AI Foundry Agent Service. While it's tempting to adopt the platform as a monolithic black box, we've seen more success with a nuanced, decoupled architecture: using the AgentCore runtime for production concerns such as session isolation, security and observability, while retaining orchestration logic in external frameworks like LangGraph. This separation of concerns allows teams to benefit from managed infrastructure while maintaining the flexibility to adapt as the LLM landscape evolves. By focusing on the runtime first, organizations can incrementally move agentic workloads into production without ceding control of their core logic to a vendor-specific orchestration layer.

45. Graphiti

Trial

We're moving Graphiti to Trial as this open-source temporal knowledge graph engine from Zep has demonstrated its production viability for addressing the LLM memory problem. While flat vector stores in RAG pipelines fail to track how facts change over time, Graphiti ingests data as discrete episodes and maintains bi-temporal validity windows on graph edges, so outdated facts are invalidated rather than overwritten. Unlike batch-oriented GraphRAG, it updates the graph incrementally and delivers sub-second retrieval via hybrid retrieval combining semantic search, BM25 and graph traversal, without query-time LLM calls. Two factors drove this move: peer-reviewed benchmarks reporting 18.5% accuracy improvements and 90% latency reductions, and the release of a first-class MCP server enabling Model Context Protocol–compliant agents to attach persistent temporal memory with minimal integration effort. Strong community adoption further signals production readiness. We're using Graphiti to build context-aware agents with stateful, temporally-aware knowledge graphs and recommend evaluating it for agentic applications. Neo4j is the primary backend, with FalkorDB as a lighter alternative. Teams should also account for per-write LLM extraction costs and pin dependencies given its pre-1.0 release status.

46. Langfuse

Trial

Langfuse is an open-source LLM engineering platform covering observability, prompt management, evaluations and dataset management. The project has matured significantly since we last assessed it. The v3 architecture introduces ClickHouse, Redis and S3 as back-end components, making it more scalable but also more complex to self-host.

Both the Python and TypeScript SDKs are now built natively on OpenTelemetry, making Langfuse a natural fit for teams that already use OTEL-based observability. New capabilities such as the experiment runner SDK and structured output support for prompt experiments move Langfuse beyond pure tracing into systematic evaluation workflows. This makes it worth considering in an increasingly crowded space that includes Arize Phoenix, Helicone and LangSmith.

Teams building primarily on Pydantic AI may also consider Pydantic Logfire, which takes a broader approach as a full-stack OTEL observability platform rather than an LLM-specific tooling suite. Langfuse remains in Assess as it is a credible choice for teams that need integrated tracing, evaluations and prompt management in one self-hostable platform. However, teams should evaluate whether the infrastructure commitment is justified for their scale and whether a narrower tool like Helicone may suffice if the primary need is model-layer cost and latency visibility.

47. Port

Trial

Port is a commercial internal developer portal designed to improve developer experience by centralizing software assets, automating workflows and enforcing engineering standards, giving platform teams a single source of truth for self-service workflows. We're seeing it matter more as organizations look to standardize engineering workflows while exposing templates, APIs, automations and agents in a form developers can actually use, including directly in the IDE through Port's API and MCP layers rather than only through a standalone portal.

In our experience, Port works well for organizations that want productized portal capabilities without investing heavily in platform engineering. In client engagements, it has supported thousands of developers while enabling relatively small platform teams to deliver effective self-service quickly. We think Port is worth assessing for organizations that need internal developer portal capabilities quickly and can accept the constraints of a commercial platform and vendor dependence.

48. Replit

Trial

Replit is a cloud-native collaborative development platform that provides instant dev environments, real-time coding and integrated AI assistance right in the browser. It combines an editor, runtime, deployment and AI coding workflows into one unified platform, allowing developers to start coding immediately without any local setup. We found that this AI-based collaborative IDE is really helpful for reducing onboarding friction, making it a great fit for prototyping together as a team. We also find it very effective for training sessions, knowledge sharing and bootcamps. While some might see Replit as a place for AI-assisted hobby projects, we think it stands out because the environment is powerful enough to compete with traditional local IDEs, making iteration and collaboration much easier.

49. SigNoz

Trial

SigNoz is an open-source, OpenTelemetry-native observability platform that provides unified support for logs, metrics and traces. It addresses the APM and instrumentation needs of modern microservices and distributed architectures while avoiding vendor lock-in. By leveraging ClickHouse as its underlying columnar database, SigNoz provides scalable, high-performance and cost-effective storage with fast querying, positioning it as a strong self-hosted alternative to platforms such as Datadog. It supports flexible querying through PromQL and ClickHouse SQL, along with alerting across multiple notification channels. In practice, we've seen SigNoz reduce infrastructure resource consumption and overall observability costs without compromising performance. While a managed cloud service is available, ready-to-use Docker images and Helm charts make it a practical choice for organizations that prefer to retain control over their data and infrastructure.

50. Agent Trace

Assess

Agent Trace is an open specification proposed by Cursor looking to standardize AI code attribution. As adoption of coding agents increases, understanding who has modified code now extends beyond human developers to include AI-generated changes. We're seeing early interest from teams that need better traceability around these changes. Existing tools such as `git blame` can show a line of code has been modified, but fail to capture whether that change was made by a human, an AI or both. Agent Trace takes a vendor-neutral approach to defining how code changes are traced and is unopinionated about how those traces are stored. It's compatible with multiple version control systems, including Git, Mercurial and Jujutsu. The specification defines contributor types such as human, AI, mixed and unknown, along with a trace record describing the origin of each contribution. There are early signals of adoption, with support from tools such as Cline and OpenCode as well as implementations like Git AI. Teams adopting coding agents should assess tooling that implements the Agent Trace specification to improve code attribution.

51. ClickStack

Assess

ClickStack is an OpenTelemetry-compatible, open-source observability platform that unifies logs, traces, metrics and sessions in a single high-performance data store built on ClickHouse. As infrastructure grows and observability costs increase, many teams struggle with fragmented telemetry toolchains and expensive vendor platforms. ClickStack addresses this challenge by leveraging ClickHouse's columnar storage to enable sub-second, high-cardinality queries across large volumes of telemetry data, offering a simpler and more cost-effective foundation for observability.

52. Coder

Assess

Coder presents a good alternative to pixel-streamed development environments by separating where code runs from how developers interact with it. Instead of streaming full desktop interfaces, developers connect to remote environments using local IDEs such as VS Code or a browser, resulting in a more responsive experience without compromising usability.

In this model, code executes on remote, scalable infrastructure while environments are defined and managed as code. This enables teams to standardize development setups and simplify the onboarding of new developers. It also makes it easier to provide controlled access to internal systems and streamline access to pre-approved AI coding agents.

We see Coder as a middle ground between local development and fully virtualized desktops: it provides centralized control and governance without the usability limitations of pixel-streamed VDIs. This makes it a good option for organizations that require remote or controlled execution environments, particularly where higher compute or secure access is needed. As with similar approaches, teams should evaluate the operational overhead and security responsibilities that come with managing these environments.

53. Databricks Agent Bricks

Assess

As agent-based approaches become more mainstream, we're seeing data platforms evolve to support these workloads natively rather than as a bolt-on. Databricks Agent Bricks provides prebuilt, auto-optimizing components for common AI patterns such as knowledge assistants and data analysts. It follows a declarative approach: developers define the goal and underlying data, while the framework handles the execution and optimization. By simplifying LLMOps and reducing the effort required for data curation, teams can focus more on business outcomes than on boilerplate. For example, our teams have used it alongside custom agents to evaluate and build complex RAG solutions for preclinical R&D. If you're already invested in the Databricks ecosystem and exploring agent-based approaches for common use cases such as chatbots and document extraction, consider assessing Agent Bricks.

54. DuckLake

Assess

DuckLake is an integrated data lake and catalog format that simplifies the lakehouse architecture by using standard SQL databases for catalog and metadata management. While traditional open table formats like Iceberg or Delta Lake rely on complex, file-based metadata structures, DuckLake stores metadata in a catalog database (for example, SQLite, PostgreSQL or DuckDB) while persisting

data as Parquet files on local disk or S3-compatible object storage. This hybrid approach improves query planning latency and transactional reliability during concurrent updates. DuckDB serves as the query engine via its `ducklake` extension, providing a familiar SQL interface for standard DDL and DML operations. DuckLake also retains lakehouse characteristics, such as partitioning, while omitting indexes and primary or foreign keys. With support for time travel, schema evolution and ACID compliance, DuckLake offers a low-complexity option for teams seeking a standalone analytical stack. Although still early in maturity, DuckLake is a promising, lightweight alternative to traditional lakehouse architectures. It avoids the operational overhead associated with Spark or Trino-based ecosystems, making it a good fit for streamlined data environments.

55. FalkorDB

Assess

FalkorDB is a Redis-based graph database that supports Cypher and suits teams that want graph capabilities without adopting a heavyweight graph platform. We see it as a practical option for organizations building relationship-rich AI and application workloads where low operational friction matters, and where a server-based graph service is preferable to embedded storage. We're placing it in Assess because the architecture is promising and the developer model is approachable, but teams should validate production behavior around scaling, operational tooling and long-term ecosystem maturity before committing to broad adoption.

56. Google Dialogflow CX

Assess

Google Dialogflow CX is Google Cloud's managed conversational AI platform, combining a graph-based state machine build from Flows and Pages with Vertex AI Gemini–powered generative capabilities. We previously tracked its predecessor, Dialogflow, in the Radar. CX represents a significant redesign that gained traction after Google integrated Vertex AI Gemini models in 2024, introducing Generative Playbooks for instruction-driven agents and Data Store RAG for grounding responses in indexed content. We used it to build a natural language data discovery agent, choosing it over a custom SDK approach for its low-code environment and Generative Playbooks. We configured these with few-shot prompting to translate natural language queries into SQL. Teams on Google Cloud building natural language interfaces over structured internal data will find Dialogflow CX accelerates delivery compared to a custom agent stack. However, the platform has no free tier; its deep Google Cloud dependency introduces significant vendor lock-in, and teams should plan for context engineering effort.

57. MCP Apps

Assess

MCP Apps is the first official extension to the Model Context Protocol, letting MCP servers return interactive HTML interfaces as dashboards, forms and visualizations that render directly in the conversation. Co-developed by Anthropic, OpenAI and open-source contributors, the extension standardizes a `ui://` resource scheme where tools declare UI templates rendered in sandboxed iframes with graceful degradation to text when the host lacks UI support. Unlike AG-UI, which operates as a separate library layer, MCP Apps packages UI directly inside MCP servers. The bidirectional design lets models observe user actions while the interface handles live data and direct manipulation that text cannot. Clients including Claude, ChatGPT, VS Code and Goose already ship support. Teams exploring richer agent interactions should assess whether the added complexity over plain text responses is warranted for their use case.

58. Monarch

Assess

Monarch is an open-source distributed programming framework that brings the simplicity of single-machine PyTorch workloads to large GPU clusters. It provides a Python API for spawning remote processes and actors, grouping them into collections called meshes that support broadcast messaging. It also offers fault tolerance through supervision trees, where failures propagate up a hierarchy to enable clean error handling and fine-grained recovery. Additional features include support for point-to-point RDMA transfers for efficient GPU and CPU memory movement and a distributed tensor abstraction that allows actors to work with tensors sharded across processes while maintaining an imperative programming model. Monarch is built on a high-performance Rust backend. Although still in the early stages of development, its abstraction — making distributed tensors behave like local ones — is powerful and can greatly reduce the complexity of large-scale distributed AI training.

59. Neutree

Assess

Neutree is an open-source platform for managing and serving LLMs on private infrastructure, positioning itself as a model-as-a-service layer for enterprise AI. It provides a unified control plane for model lifecycle management, inference serving and compute scheduling across heterogeneous hardware such as NVIDIA, AMD and Intel accelerators. As organizations move away from hosted APIs toward self-hosted, governed deployments, Neutree addresses a clear gap: operating LLM workloads with enterprise-grade capabilities such as multi-tenancy, access control, usage accounting and infrastructure abstraction. By separating model serving from application logic, it enables teams to deploy, scale and route models across environments — including bare metal, VMs and containers — without tightly coupling to a specific cloud provider. However, Neutree is still relatively new, and teams should approach adoption with caution. Its ecosystem, operational maturity and integration capabilities are still evolving compared to more established ML platforms. While promising, it's best suited for teams willing to invest in evaluating and shaping emerging enterprise AI infrastructure.

60. OptScale

Assess

OptScale is an open-source, multi-cloud FinOps platform with support for AI/ML-heavy workloads where GPU and experimentation costs can spike quickly. It ingests billing and usage data from cloud APIs, combining cost visibility, optimization recommendations, budget tracking and anomaly detection in one system with policy-based alerts aligned to teams or business structures.

Compared with OpenCost, OptScale covers broader non-Kubernetes FinOps use cases while still providing Kubernetes-level analysis. It also offers more control and less vendor lock-in than enterprise suites such as IBM Cloudability, CloudZero, CloudHealth, IBM Kubecost and Flexera One. The trade-off is higher operational overhead, with concerns around deployment complexity, connector edge cases and container image security hygiene. Teams should treat OptScale as a platform capability investment rather than a plug-and-play product.

61. Rhesis

Assess

Rhesis is an open-source testing platform for LLM and agentic applications that lets teams define expected behavior in natural language, generate adversarial test scenarios and evaluate outcomes through both a UI and an SDK or API. It's becoming more relevant as traditional testing approaches assume deterministic behavior, while AI systems fail in more subtle ways, including jailbreaks, multi-turn interactions, policy violations and context-dependent edge cases. In our evaluation, Rhesis is a useful platform for teams that need more than simple prompt evaluations. Features such as the conversation simulator, adversarial testing, OpenTelemetry-based tracing and self-hosting via Docker make it a practical way to bring product, domain and engineering teams into a shared testing workflow. The main benefit is improved pre-production validation for non-deterministic systems. However, teams should consider common trade-offs in this space, including evaluation cost, the limits of LLM-as-judge metrics and the need for well-defined requirements before the platform delivers value. We think Rhesis is worth assessing for teams building LLMs or agentic systems that require collaborative, repeatable testing beyond basic prompt checks.

62. RunPod

Assess

As organizations increasingly experiment with training and fine-tuning LLMs, hyperscalers such as AWS and Google Cloud can introduce high costs and limited hardware availability. RunPod provides a cost-effective alternative for compute-intensive AI workloads. Operating as a globally distributed GPU marketplace, it offers on-demand access to a wide range of hardware, from enterprise-grade H100 clusters to consumer-grade RTX 4090s, often at significantly lower cost than traditional cloud providers. For teams needing flexible, budget-friendly infrastructure to develop, train or deploy AI models without long-term commitments or vendor lock-in, RunPod is a practical option worth evaluating.

63. Sprites

Assess

Sprites is a stateful sandbox environment from Fly.io designed for running AI coding agents in isolation. Where most agent sandboxes are ephemeral, spinning up for a task and disappearing, Sprites provides persistent Linux environments with unlimited checkpoint and restore capabilities. This allows developers to snapshot the entire environment state — including installed dependencies, run-time configuration and file system changes — and roll back when an agent goes off track. This goes beyond what Git alone can recover, capturing system state that version control does not track. As our teams increasingly adopt sandboxed execution for coding agents as a sensible default, Sprites represents one end of the spectrum: a non-ephemeral, stateful approach that trades the simplicity of throwaway containers for richer recovery options. Teams evaluating agent sandboxing should consider Sprites alongside ephemeral alternatives such as Dev Containers based on their needs and workflow.

64. torchforge

Assess

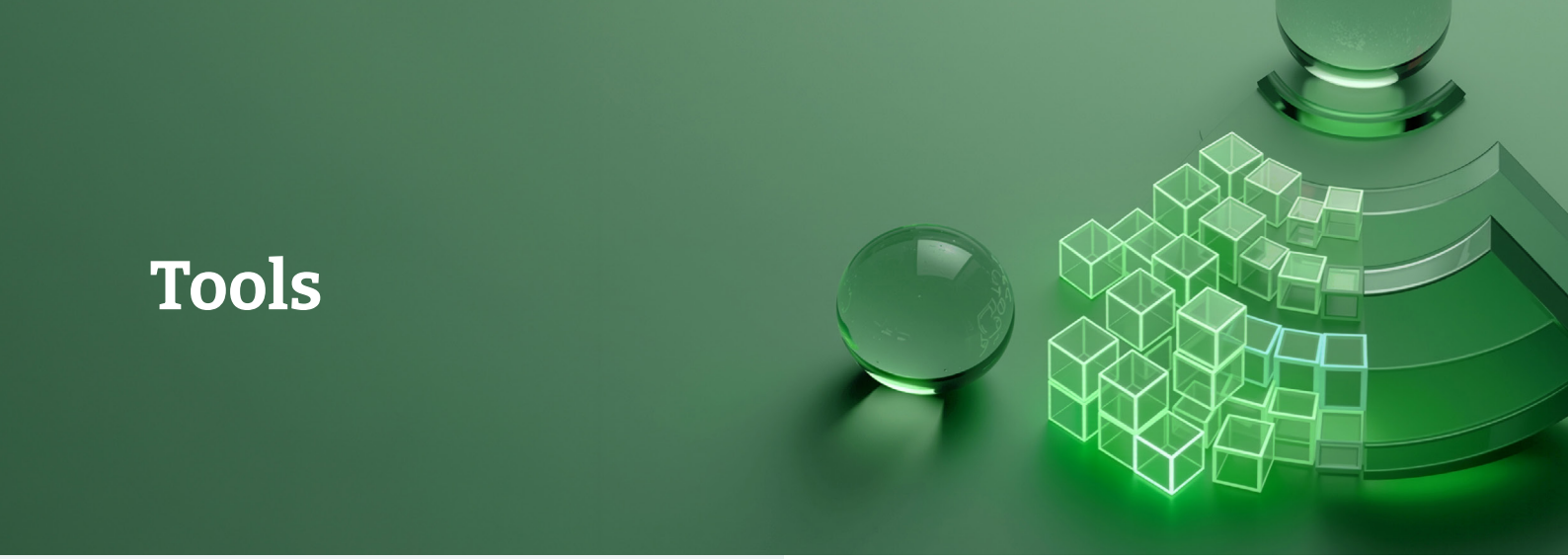
torchforge is a PyTorch-native reinforcement learning library designed for large-scale post-training of language models. It provides a higher-level abstraction that decouples algorithmic logic from infrastructure concerns, orchestrating components such as Monarch for coordination, vLLM for inference and torchtitan for distributed training. This approach allows researchers to express complex reinforcement learning workflows using pseudocode-like APIs, while scaling workloads across thousands of GPUs without managing low-level concerns such as resource synchronization, scheduling or fault tolerance. By separating the "what" (algorithm design) from the "how" (distributed execution), torchforge simplifies experimentation and iteration in large-scale alignment systems. We see this as a useful step toward making advanced post-training techniques more accessible, although teams should evaluate its maturity and fit within their existing ML infrastructure.

65. torchtitan

Assess

torchtitan is a PyTorch-native platform for large-scale pre-training of generative AI models, providing a clean and modular reference implementation for high-performance distributed training. It brings together advanced distributed primitives into a cohesive system, supporting 4D parallelism: data, tensor, pipeline and context parallelism. As training models at the scale of Llama 3.1 405B demands significant scale and efficiency, torchtitan offers a practical foundation for building and operating large training workloads. Its modular design makes it easier for teams to experiment with and evolve parallelism strategies while maintaining production readiness. We see torchtitan as a useful step toward standardizing large-scale model training in the PyTorch ecosystem, particularly for teams building their own pre-training infrastructure.

Tools



Adopt

66. Axe-core
67. Claude Code
68. Cursor
69. Kafbat UI
70. mise

Trial

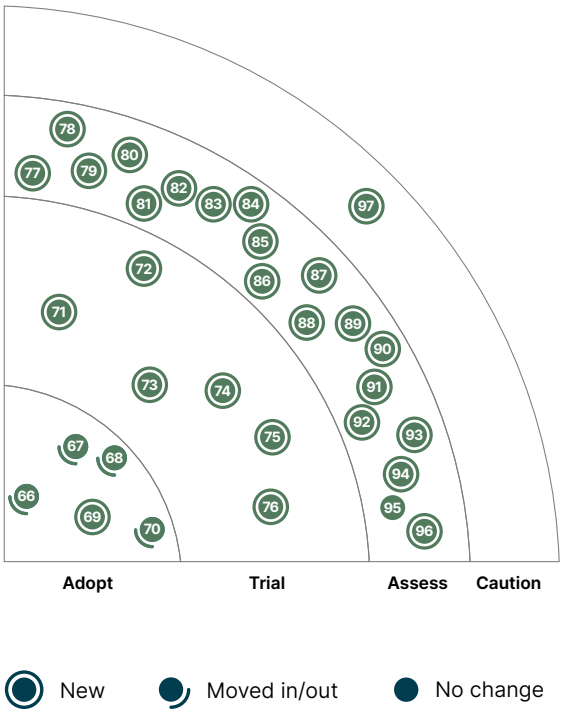
71. cargo-mutants
72. Claude Code plugin marketplace
73. Dev Containers
74. Figma Make
75. OpenAI Codex
76. Typst

Assess

77. Agent Scan
78. Beads
79. Bloom
80. CDK Terrain
81. CodeScene
82. ConfIT
83. Entire CLI
84. Git AI
85. Google Antigravity
86. Google Mainframe Assessment Tool
87. OpenCode
88. OpenSpec
89. PageIndex
90. Pencil
91. Pi
92. Qwen 3 TTS
93. SGLang
94. ty
95. Warp
96. WuppieFuzz

Caution

97. OpenClaw



66. Axe-core

Adopt

Axe-core is an open-source accessibility testing tool that detects issues in websites and other HTML-based applications. It checks pages for compliance with standards such as WCAG — including conformance levels A, AA and AAA — and flags common accessibility best practices. Since first appearing in the Radar in Trial in 2021, several of our teams have adopted Axe-core with clients. Accessibility is increasingly a mandatory quality attribute. In Europe, for example, regulations such as the European Accessibility Act require organizations to ensure their digital services meet accessibility requirements. Axe-core fits well within modern development workflows by enabling automated checks in CI pipelines. This helps teams prevent regressions, maintain compliance and receive early feedback during development, ensuring accessibility is part of the feedback loop, particularly as AI-assisted and agentic coding tools are more widely adopted.

67. Claude Code

Adopt

Anthropic's Claude Code is an agentic AI coding tool for planning and executing complex multi-step workflows. We're moving Claude Code to Adopt because teams inside and outside Thoughtworks now use it day-to-day in production software delivery, where it's widely treated as a benchmark for capability and usability. The CLI agent landscape has expanded quickly with tools such as OpenAI's Codex CLI, Google's Gemini CLI, OpenCode and pi, but Claude Code remains the preferred option for many teams. Its use now extends beyond code authoring into broader workflow execution, including specifications, stories, configuration, infrastructure, documentation and markdown-defined business processes. Claude Code continues to introduce features that other tools follow, such as skills, subagents, remote control and agentic team workflows.

Teams adopting Claude Code should pair it with disciplined operating practices. Agentic coding shifts developer effort from manual implementation toward specifying intent, constraints and review boundaries. This can accelerate delivery, but it also increases the risk of complacency with AI-generated code, which can make systems harder to maintain and evolve for both people and agents. We're seeing growing attention to harness engineering as a way to implement context engineering (topic-aware, scope-driven context selection) and curated shared instructions to make agentic workflows more reliable.

68. Cursor

Adopt

Cursor is among the most widely adopted coding agents we see today, consistently appearing alongside Claude Code as a default choice for our delivery teams. It has matured into a comprehensive agentic environment with features such as plan mode, hooks and subagents. While terminal-based agents remain popular, many of our developers find that supervising an agent within an IDE provides a richer experience for reviewing and refining plans before execution. The adoption of the Agent Client Protocol has further lowered barriers for the large Jetbrains user base, making Cursor's capabilities accessible within those IDEs.

We particularly value the ability to inspect individual agent steps or roll back to earlier stages when a plan deviates. By leveraging Agent Skills, teams can package reusable instructions to help standardize how agents interact with complex codebases. While productivity gains are evident, agentic autonomy still requires rigorous automated testing and human oversight to catch subtle regressions.

69. Kafbat UI

Adopt

Kafbat UI is a free, open-source web UI for monitoring and managing Apache Kafka clusters. We've found it especially useful when teams need to inspect payloads that are otherwise hard to read during day-to-day debugging. In our experience, teams often get stuck when debugging encrypted messages, and Kafbat UI's support for built-in and pluggable SerDes provides a practical way to apply decryption or custom decoding so messages become readable again. Compared with one-off debug scripts, it offers faster feedback and a better operational experience for developers and support teams. We recommend Kafbat UI for Kafka-heavy environments where secure message inspection and efficient troubleshooting should be standard practice.

70. mise

Adopt

Since our last assessment, mise has evolved from a high-performance alternative to asdf into a default frontend for the development environment. We're moving it to Adopt because it consolidates three fragmented concerns — tool and language versioning, environment variable management and task execution — into a single high-performance, Rust-based tool, configured through a declarative mise.toml file. mise is easy to set up and works well with CI/CD pipelines. It also adds a layer of supply chain security through integration with Cosign and GitHub Artifact Attestations, which is often missing from other version managers.

For teams looking to standardize their developer environment setup, mise has become our recommended default. In polyglot environments with multiple microservices, this is especially useful when codebases adopt new language versions at the same time. Best of all, mise also works with existing language-specific tooling, so teams do not need to migrate all at once.

71. cargo-mutants

Trial

cargo-mutants is a mutation testing tool for Rust that helps our teams move beyond simple code coverage metrics. By automatically injecting small, intentional bugs — such as swapping operators or returning default values — it verifies whether existing tests actually catch regressions. We've found its zero-configuration approach particularly effective; unlike previous tools, it requires no changes to the source tree. For our teams new to Rust, it provides a useful feedback loop, identifying missing edge cases and improving the reliability of unit and integration tests.

This tool is a specialized implementation of mutation testing, which we're also trialing in other ecosystems. The primary cost is increased test execution time, as each mutant requires an incremental build. To manage this, we recommend targeting specific modules during local development or running full suites asynchronously in CI. While teams may occasionally need to filter out logically equivalent mutants, the resulting increase in testing confidence outweighs the additional noise.

72. Claude Code plugin marketplace

Trial

Previously, sharing custom commands, specialized agents, MCP servers and skills was a manual process that relied on developers to copy and paste instructions from Confluence or other external sources. This often led to version drift, with team members using outdated project instructions.

Our teams are now leveraging the Claude Code plugin marketplace to distribute shared commands, prompts and skills using a Git-based distribution model. By hosting internal team marketplaces on GitHub or similar platforms, organizations can distribute these artifacts more securely and consistently. Developers can then sync the AI-driven workflows and tools directly into their local environments via the CLI. Other coding agents such as Cursor also support a team plugin marketplace, enabling a more streamlined and governed way to share these artifacts.

73. Dev Containers

Trial

Dev Containers provide a standardized way to define reproducible, containerized development environments using a `devcontainer.json` configuration file. Originally designed to give teams consistent development setups, Dev Containers have found a compelling new use case as sandboxed execution environments for coding agents. Running an AI coding agent inside a Dev Container isolates it from the host file system, credentials and network, allowing teams to grant agents broad permissions without risking the host machine. The open specification is supported natively by VS Code and VS Code–based tools such as Cursor. DevPod extends devcontainer support to any editor or terminal workflow via SSH. Dev Containers take an ephemeral-by-default approach (i.e., the container rebuilds from the configuration on each launch) which provides a clean security boundary at the cost of reinstalling tools and dependencies. For teams that need persistent state or checkpoint and restore capabilities, alternatives such as Sprites take a different approach. Dev Containers also offer supply chain security benefits beyond agent sandboxing. By defining the toolchain in a declarative configuration, teams reduce exposure to compromised packages and unexpected dependencies on developer machines.

74. Figma Make

Trial

We previously blipped self-serve UI prototyping with GenAI and this technique is now widely adopted by development teams, including product managers and designers, to generate user-testable, high-fidelity prototypes. Figma Make is a strong option for building such prototypes because it leverages actual components and layers from a design system, making the results closely resemble production applications. Figma Make uses a bespoke AI model trained on high-quality design patterns. Our teams are using it to create new design screens, enhance existing ones and build shareable prototypes to gather rapid user feedback.

75. OpenAI Codex

Trial

OpenAI Codex has evolved into a standalone agentic coding tool, available via a dedicated macOS app and CLI. It's designed for autonomous task delegation: given a prompt, it plans, implements and iterates across files with minimal intervention. Our teams have found it effective as a high-velocity drafting tool, particularly for greenfield tasks and repetitive implementation work. However, its tendency to suggest logically sound but functionally outdated library patterns means that automated testing and human review are essential. As with other agentic tools in this Radar, the risk of accumulating subtle technical debt is real and proportional to the level of autonomy teams grant it.

76. Typst

Trial

Typst is a markup-based typesetting system positioned as a modern successor to LaTeX for programmatic document generation. It combines high-quality typography with a simpler syntax and a significantly faster compilation pipeline, compiling even very large documents in a fraction of the time required by traditional LaTeX toolchains. Typst offers clearer error messages and built-in scripting capabilities such as conditionals and loops. It can also load structured data from JSON or CSV, making it well suited to automated document generation.

Our teams have used Typst to generate statements and reports for banking and financial services customers where documents must be produced at scale with consistent formatting. The open-source compiler can be self-hosted, and its growing ecosystem includes community-contributed packages. Typst is more approachable than LaTeX while delivering comparable typographic quality.

77. Agent Scan

Assess

Agent Scan is a security scanner for agent ecosystems that discovers local components, including MCP servers and skills, and flags risks such as prompt injection, tool poisoning, toxic flows, hardcoded secrets and unsafe credential handling. It addresses an emerging gap in agent supply chain visibility and provides a practical way to inventory and test rapidly growing agent surfaces. However, adoption should be deliberate. This is for several reasons: Scans require sharing component metadata with Snyk APIs, and signal quality and false-positive rates require validation in your environment. It's important that teams confirm the operational value of Agent Scan before making it part of mandatory delivery gates.

78. Beads

Assess

Beads is a Git-backed issue tracker designed as a persistent memory layer for coding agents. Instead of relying on ad hoc Markdown plans, it gives agents a structured task graph with blocker relationships, ready-work detection and branch-friendly coordination for long-horizon work across sessions. It's built on Dolt, a SQL database with built-in version control that supports branches, merging, diffs and table cloning similar to a Git repository. Beads represents a new category of agent-native project memory and task tracking tools. Other early projects in this space include ticket and tracer. Unlike traditional ticketing systems such as GitHub Issues and Jira, Beads and similar tools enable new workflows for coordinating autonomous multi-agent execution, including agents assigning tasks to one another.

79. Bloom

Assess

Bloom is a tool from Anthropic for AI safety researchers evaluating LLM behavior. It probes for behaviors such as sycophancy and self-preservation. Compared to static benchmarks, it uses a seed configuration that defines target behaviors and evaluation parameters to dynamically generate diverse test conversations and then evaluate the results. This approach to automated behavioral evaluation is necessary to keep up with the pace of model releases, enabling external research teams to conduct evaluations. Petri is a companion tool that identifies which behaviors emerge for a given model, while Bloom identifies in what scenarios and how often those behaviors occur, together forming a

more complete evaluation suite. One concern is that Bloom requires a teacher (or evaluator) model that assesses a given student model. Teacher models may have blind spots and biases, so using multiple evaluators can reduce bias in the results. AI safety research teams should assess Bloom as a complement to static benchmarks for evaluating emergent model behaviors.

80. CDK Terrain

Assess

CDK Terrain (CDKTN) is a community fork of the Cloud Development Kit for Terraform (CDKTF), which HashiCorp deprecated and archived in December 2025. CDK Terrain picks up where CDKTF left off, allowing teams to define infrastructure using TypeScript, Python or Go and provision it through Terraform or OpenTofu. For teams already invested in CDKTF, CDK Terrain offers a migration path that preserves existing code and workflows rather than forcing a move to HCL or Pulumi. The project releases every month and has added OpenTofu support as a first-class target. However, community-maintained forks of vendor-abandoned projects carry inherent risk around long-term support, and the CDKTF approach never achieved broad adoption. HashiCorp cited lack of product-market fit when sunsetting it. Teams currently using CDKTF should assess CDK Terrain as a continuation option, but also weigh whether this is the right moment to migrate to a more widely supported approach.

81. CodeScene

Assess

We blipped social code analysis back in 2017, but with the increased adoption of coding agents we're seeing renewed interest in it, particularly in tools such as CodeScene. CodeScene is a behavioral code analysis tool that identifies technical debt by combining code complexity metrics with version control history. Unlike traditional static analysis, it highlights "hotspots" to help teams prioritize refactoring based on actual development activity and business impact. CodeScene now provides guidance for AI-friendly code design. Our teams are finding code quality has become even more important as coding agents can modify code far more rapidly than human developers. CodeScene's CodeHealth metric provides a useful guardrail by identifying areas where code is too complex for LLMs to safely refactor without a high risk of hallucinations. We recommend teams assess CodeScene as a guardrail for coding agent adoption. Its CodeHealth metric highlights safe refactoring targets and flags areas that require remediation before agents are applied.

82. ConfIT

Assess

ConfIT is a library for defining integration and component-style API tests declaratively in JSON rather than writing test flows imperatively in code. We're seeing increased interest in this approach, as large test suites often accumulate boilerplate around HTTP clients, request setup and assertions. AI-assisted development further reinforces this trend, making structured test definitions easier to generate and maintain than verbose procedural code.

Based on client experience and our evaluation, a declarative layer can reduce duplication between component and integration tests, improve readability and make test intent easier to evolve across teams. However, ConfIT itself appears to have limited community adoption and a small ecosystem, making it harder to recommend broadly despite these benefits. We think ConfIT is worth assessing for .NET teams exploring specification-driven API testing. Teams should, however, validate its long-term maintainability, ecosystem fit and operational trade-offs before adopting it more widely.

83. Entire CLI

Assess

Entire CLI hooks into Git workflows to capture AI coding agent sessions — including transcripts, prompts, tool calls, files touched and token usage — as searchable metadata stored on a dedicated repository branch. It supports Claude Code, Gemini CLI, OpenCode, Cursor, Factory AI Droid and GitHub Copilot CLI. As AI agents become primary contributors to codebases, teams face a growing gap between what Git tracks and what actually happens during a coding session. Entire CLI addresses this by recording full sessions alongside commits, creating an audit trail of agent activity without polluting the main branch history. Its checkpoint system also enables practical recovery: teams can rewind to a known-good state when an agent deviates and resume from any checkpoint. While the tool is still very new and the ecosystem for agent session traceability is still forming, teams with compliance or audit requirements around AI-generated code may find Git-native session capture a natural fit.

84. Git AI

Assess

Git AI is an open-source Git extension that tracks AI-generated code in your repositories, linking every AI-written line to the agent, model, and prompts that generated it. Git AI uses checkpoints and hooks to track incremental code changes between the start and end of a commit. Each checkpoint contains a diff between the current state and the previous checkpoint, marked as either AI or human authored. This approach is more accurate than those approaches focusing on tracking AI code authorship by counting lines of code at the moment of insertion.

Git AI uses an open standard for tracking AI-generated code with Git Notes. While the ecosystem of supported agents is still maturing, we believe Git AI is worth assessing for teams looking to maintain long-term accountability and maintainability in an agentic workflow. Both humans and AI agents can query the original intent and architectural decisions behind a specific block of code via the /ask skill, by referencing the archived agent session.

85. Google Antigravity

Assess

Google Antigravity is a standalone VS Code fork, built on technology licensed from Windsurf and launched in public preview alongside Gemini 3 in November 2025. Antigravity centers the IDE around multi-agent orchestration: an Agent Manager runs multiple agents in parallel across tasks, a built-in Chromium browser allows agents to interact with live UIs directly and a skills system stores reusable agent instructions in the repository.

The Agent Manager acts as a “Mission Control” dashboard rather than a standard chat sidebar, fundamentally shifting the developer’s role from writing code line by line to orchestrating multiple autonomous workstreams. Developers can still drop into the editor for human-in-the-loop (HITL) control when needed. Antigravity integrates with Google Cloud and Firebase through the Model Context Protocol and supports agent development via the Agent Development Kit. We’re placing Antigravity in Assess because it remains in public preview with no GA date, and its security posture and enterprise readiness are still evolving. The multi-agent execution model and autonomous browser access signal where agentic IDEs are heading.

86. Google Mainframe Assessment Tool

Assess

Google's Mainframe Assessment Tool helps organizations reverse-engineer applications running on mainframes by analyzing either an entire portfolio or individual systems. At its core, it relies on deterministic language parsers to map call flows and data dependencies across codebases, creating a structural view of how applications interact. On top of this foundation, generative AI features provide summaries, documentation, test case generation and modernization suggestions. This approach aligns with a broader pattern of using GenAI to understand legacy codebases, where strong insights about the system form the foundation for the effective use of AI. While the tool does not yet support all major mainframe technology stacks, it's evolving rapidly. Our teams have found it helpful for client engagements focused on mainframe application discovery and modernization.

87. OpenCode

Assess

OpenCode has quickly become one of the most prominent open-source coding agents, with a strong terminal-first experience. A major strength is model flexibility: it supports hosted frontier models, self-hosted endpoints and local models. This makes OpenCode attractive for cost control, customization and restricted environments including air-gapped setups. It also means users need to be explicit about licensing and provider terms when using subscriptions or APIs.

OpenCode's extension model is another key part of its appeal, supporting both plugins and MCP integrations for team-specific workflows, tools and guardrails. Many users leverage Oh My OpenCode, an optional but popular harness that provides a more opinionated, batteries-included setup with coordinated agent teams and richer orchestration patterns.

88. OpenSpec

Assess

As the capabilities of AI coding agents evolve, developers increasingly face challenges with predictability and maintainability when requirements and context live only in ephemeral chat histories. To address this, we're seeing the emergence of spec-driven development (SDD) tools. OpenSpec is an open-source SDD framework that introduces a lightweight specification layer, ensuring human developers and AI agents align on what to build before code is generated.

What sets OpenSpec apart is its fluid, minimal workflow, which is often reduced to three steps: propose → apply → archive. Many SDD frameworks (e.g., GitHub Spec Kit) or Agentic Skills workflows (e.g., Superpowers) are better suited to greenfield projects than brownfield ones. We particularly like OpenSpec's focus on spec deltas rather than defining a complete specification upfront, making it well-suited for existing systems.

Unlike heavier alternatives that enforce more rigid workflows (e.g., BMAD) or require vendor-specific IDE integrations (e.g., Kiro), OpenSpec is iterative and tool agnostic. For teams looking to introduce structure and predictability into AI-assisted development without adopting a heavyweight process, OpenSpec is a developer-friendly framework worth assessing. Meanwhile, as models and coding agents continue to grow more powerful, we also recommend teams continue to monitor and revisit native capabilities and re-evaluate the need for SDD tooling.

89. PageIndex

Assess

PageIndex is a tool that builds a hierarchical index of a document for vectorless, reasoning-based RAG pipelines, rather than relying on traditional embedding-based retrieval. Instead of chunking a document into vectors, which can lose structural information and provide limited visibility into why results were retrieved, PageIndex builds a table of contents index that an LLM traverses step-by-step to retrieve relevant content. This produces an explicit reasoning trace that explains why a particular section was selected, similar to how a human scans headings and drills down into specific sections. Some of our teams have found this approach works well for documents where meaning depends heavily on structure rather than semantics, such as financial reports with numerical data, legal documents with cross-referenced articles and complex clinical or scientific documents. However, this approach comes with trade-offs. For instance, because LLM inference is part of the retrieval process, it can introduce significant latency and cost, especially for large documents.

90. Pencil

Assess

Pencil is a design canvas tool that integrates with IDEs and coding agents such as Cursor and Claude Code. Unlike Figma, which currently only offers read access, Pencil runs a bidirectional local MCP server that gives both read and write access to directly manipulate the canvas. It also provides design-to-code capabilities similarly to tools such as Figma Make and Builder.io, but takes a more developer-centric approach, with design files stored in the repository in an open JSON format called .pen, making design assets versionable alongside code. This approach can help close the design-to-development hand-off gap by integrating with tools familiar to developers. For large-scale and complex design systems, Figma remains the standard for collaboration across roles. However, Pencil is worth considering for teams without dedicated designers or for teams with developers who have strong design skills.

91. Pi

Assess

Pi is a minimalist, open-source terminal coding agent written in TypeScript. We see it as an appealing option for tinkerers and experimenters rather than a mainstream enterprise default. Pi is a bare-bones harness that is more customizable than a full agent such as OpenCode. It's also easier to adapt than building a new agent with an agentic framework such as ADK, LangGraph or Mastra.

The project is still early and primarily maintainer-led, despite strong traction and active releases. Treat pi as an engineer-facing building block, not a complete enterprise platform with full guardrails and support.

92. Qwen 3 TTS

Assess

Qwen 3 TTS is an open-source text-to-speech model that closes much of the quality gap with commercial offerings while providing greater developer control than many paid APIs. It supports multiple languages, can clone voices from short samples (roughly 10–15 seconds) and allows post-training fine-tuning for domain- or character-specific voices, making it a compelling option for teams that need brand-specific speech or on-prem control. It's still a recent release, and teams should validate stability, safety controls, licensing fit and operational maturity before adopting it for production-critical voice workloads.

93. SGLang

Assess

SGLang is a high-performance serving framework that reduces the compute overhead of LLM inference through a co-design of its front-end programming language and back-end runtime. It introduces RadixAttention, a memory management technique that aggressively caches and reuses the KV (key-value) states across prompts. This approach delivers significant performance improvements over standard serving engines such as vLLM in scenarios with high prefix overlap. For teams building complex autonomous agents, relying on long system prompts or using extensive few-shot prompting with shared examples, SGLang can provide substantial gains in latency and efficiency.

94. ty

Assess

As Python continues to grow in popularity, especially in the AI and data science space, having a strong type system becomes increasingly valuable. Ty is an extremely fast Python type checker and language server written in Rust. It's part of the Astral ecosystem, which also includes tools like uv and ruff. Ty provides fast feedback and integrates well with common editors such as Visual Studio Code and others. In our experience, using ty alongside other Astral tools simplifies Python development at scale in large organizations. As agentic coding becomes more common, having a deterministic type checker with a fast feedback loop helps catch mistakes early and reduces code review effort on simple errors.

95. Warp

Assess

Since we last included Warp in the Radar, it has evolved well beyond its "terminal with AI features" description. Its core strengths remain — block-based command output, AI-driven suggestions and notebook features — but Warp has expanded into territory traditionally occupied by IDEs. It now can render Markdown, display a file tree and open files directly in the terminal, supporting a full agentic development workflow across panes: a coding agent such as Claude Code in one, a shell in another and a workspace file view in a third pane.

One practical advantage we've observed is that Warp handles the high-throughput text output produced by modern coding agents better than traditional terminals, where rendering speed and readability can become bottlenecks. Warp has also added a built-in coding assistant, though this hasn't been widely evaluated in our teams. Warp recently launched Oz, an orchestration platform for cloud agents that integrates with the terminal. This blip focuses on the terminal itself. Teams that prefer a lightweight, composable terminal and want to bring their own AI tooling may find Ghostty a better fit; it takes a deliberately minimalist approach in contrast to Warp's batteries-included philosophy. The pace of new features and Warp's broader platform ambitions make a move to Trial premature until the product stabilizes and we gain more field experience with its newer capabilities.

96. WuppieFuzz

Assess

WuppieFuzz is an open-source fuzzer for REST APIs that uses an OpenAPI definition to generate valid requests, mutates them to explore edge cases and relies on server-side coverage feedback to prioritize inputs that reach new execution paths. This matters because most teams still rely on example-based integration and contract tests, which rarely probe unexpected inputs, unusual request sequences or failure-heavy paths, even though APIs are often the main integration surface of modern

systems. Based on our early evaluation, WuppieFuzz looks like a promising complement to these tests, because it can uncover issues such as unhandled exceptions, authorization gaps, sensitive data leaks, server-side errors and logic flaws that scripted tests may miss. Teams still need to evaluate how it fits into CI, the run-time overhead it introduces and how useful its results are in practice. For that reason, we think WuppieFuzz is worth assessing for teams building critical or externally exposed REST APIs.

97. OpenClaw

Caution

OpenClaw is an open-source project in what its creator calls the "hyper-personal AI assistant" category. Users host their own instance, keep it continuously available through messaging channels such as WhatsApp or iMessage and then let it execute tasks through connected tools. With persistent memory of conversations, preferences and habits, OpenClaw creates a persistent personal experience that feels materially different from GenAI chat interfaces or typical coding agents. The model is clearly compelling and has already inspired followers such as Claude Cowork.

We have placed OpenClaw on Caution because the model requires substantial security trade-offs. The more access you grant it — to calendar, email, files and communications — the more useful it becomes, and the more it concentrates permissions in exactly the pattern we warned about in toxic flow analysis for AI. This risk is not unique to OpenClaw; it applies to other implementations of the same pattern, including offerings from established vendors. We've published advice for teams considering OpenClaw and sandboxed execution environments, and alternatives such as NanoClaw or ZeroClaw can reduce blast radius. However, the hyper-personal assistant pattern itself remains permission-hungry and high risk.

Languages and Frameworks

Adopt

98. Apache Iceberg
99. Declarative Automation Bundles
100. React JS
101. React Native
102. Svelte
103. Typer

Trial

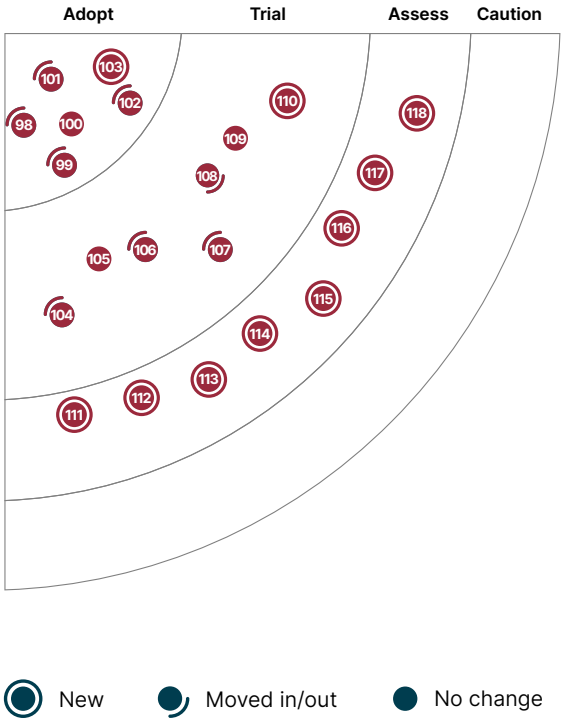
104. Agent Development Kit (ADK)
105. DeepEval
106. Docling
107. LangExtract
108. LangGraph
109. LiteLLM
110. Modern.js

Assess

111. Agent Lightning
112. GitHub Spec Kit
113. Mastra
114. Pipecat
115. Superpowers
116. TanStack Start
117. TOON (Token-Oriented Object Notation)
118. Unsloth

Caution

—



98. Apache Iceberg

Adopt

Apache Iceberg is an open table format for large-scale analytical datasets that defines how data files, metadata and schemas are organized on storage systems such as S3. Having evolved significantly in recent years, it has become a foundational building block for technology-agnostic lakehouse architectures.

Iceberg is now supported by all major data platform providers — including AWS (Athena, EMR, Redshift), Snowflake, Databricks and Google BigQuery — making it a strong option for avoiding vendor lock-in. What distinguishes Iceberg from other open table formats is its openness across features and governance, unlike alternatives whose capabilities are limited or controlled by a single vendor.

From a reliability perspective, Iceberg's snapshot-based design provides serializable isolation, safe concurrent writes through optimistic concurrency and version history with rollback. These capabilities deliver strong correctness guarantees while avoiding performance bottlenecks.

While Apache Spark remains the most common engine used with Iceberg, it's also well supported by Trino, Flink, DuckDB and others, making it suitable for a wide range of use cases, from enterprise data platforms to lightweight local analytics. Across many of our teams, Iceberg has earned strong trust as a stable, open data format; we recommend it as a default choice for organizations building modern data platforms.

99. Declarative Automation Bundles

Adopt

Declarative Automation Bundles (formerly known as Databricks Asset Bundles) has evolved into a primary tool for bringing software engineering and CI/CD practices to the Databricks ecosystem. The tool has matured significantly and now enables our teams to manage most platform resources as code, including clusters, ETL pipelines, jobs, machine-learning models and dashboards. Using the databricks `bundle plan` command, teams can preview changes and apply repeatable deployment practices to Databricks artifacts, similar to how infrastructure is managed with tools such as Terraform. Treating traditionally mutable assets, such as dashboards and ML pipelines, as code allows them to be version-controlled, tested and deployed with the same rigor as traditional microservices. Based on our experience in production environments, Declarative Automation Bundles has become a reliable approach for managing data and ML workflows on Databricks. Teams working extensively in the Databricks ecosystem should consider adopting it to standardize infrastructure management practices.

100. React JS

Adopt

React has been our default choice for JavaScript UI development since 2016, but with the stable release of React Compiler coming last October (as part of React 19), it's worth revisiting. There are a number of reasons this feature is notable. By handling memoization at build time, for example, it makes manual `useMemo` and `useCallback` largely unnecessary, though the team recommends keeping them as escape hatches when precise control over effect dependencies is required. Battle-tested at Meta and supported by Expo SDK 54, Vite and Next.js, the compiler also removes a category of performance boilerplate that has long been a cost of working at scale with React. React 19 also introduces Actions and hooks such as `useActionState` and `useOptimistic`, which simplify form

handling and data mutations without relying on external libraries. The launch of the React Foundation under the Linux Foundation in 2025 — with Amazon, Expo, Callstack, Microsoft, Software Mansion and Vercel joining Meta — further strengthens the library's long-term stability and addresses a concern cautious teams have historically cited when considering adoption.

101. React Native

Adopt

React Native has moved to Adopt as our default choice for cross-platform mobile development. While previously in Trial, the rollout of the New Architecture — specifically `JSI` and `Fabric` — has addressed long-standing concerns regarding bridge bottlenecks and initialization speed. Our teams have observed significant performance gains in complex UI transitions and data-intensive workloads. By moving away from the asynchronous bridge, React Native now delivers responsiveness that rivals native implementations while maintaining a single codebase.

We've used it successfully across multiple production projects and found the ecosystem around Expo and React to be mature and stable. While state management still requires careful planning, the productivity benefits of `fast refresh` workflows and shared skill sets outweigh these costs. For most hybrid mobile use cases, React Native is now our primary recommendation for teams seeking performance, consistency and speed.

102. Svelte

Adopt

Svelte is a JavaScript UI framework that compiles components into optimized JavaScript at build time rather than relying on a large browser-side runtime or virtual DOM. Since we last featured it in Trial, we've seen more teams use it successfully in production. SvelteKit has also made it a more robust choice for SSR and full-stack web applications, increasing our confidence in moving it to Adopt.

In our experience, the original reasons to choose Svelte still hold: it produces small bundles, delivers strong run-time performance and offers a simpler component model. Newer capabilities in Svelte 5, such as runes and snippets, make reactivity and UI composition more explicit and flexible. Compared with heavier front-end frameworks, Svelte provides a cleaner development experience with less code. Feedback from teams increasingly suggests it's a credible alternative to React or Vue rather than a niche option. Teams should still consider ecosystem familiarity, hiring and platform fit, but we now recommend Svelte as a sensible default for building modern web applications where performance and delivery simplicity matter.

103. Typer

Adopt

Typer is a Python library for building command-line interfaces from standard type-annotated functions, providing automatic help text, shell completion and a clear path from small scripts to larger CLI applications. We're seeing increased relevance as teams turn internal tooling, automation and AI-adjacent developer workflows into first-class CLIs. In our experience, Typer is easy to introduce on real projects, and teams appreciate how quickly it produces clear, readable commands. Its strengths include type-hint-driven APIs, automatic help and completion and a low-friction path from simple scripts to multi-command CLIs. However, it's a Python-specific solution and may not be the best fit when highly customized CLI behavior or cross-language consistency is required. We recommend Typer for teams building CLIs for delivery, operations and developer experience workflows.

104. Agent Development Kit (ADK)

Trial

Agent Development Kit (ADK) is Google's framework for building and operating AI agents with software-engineering-oriented abstractions for orchestration, tools, evaluation and deployment. Its ecosystem and operational capabilities have matured significantly since we included it in Assess, with active multi-language development and stronger observability and run-time features. Vendor-native agent frameworks are now a crowded field, with Microsoft Agent Framework, Amazon Bedrock AgentCore, the OpenAI Agents SDK and the Claude Agent SDK advancing competing options. Open-source alternatives such as LangGraph and CrewAI remain strong choices, especially where teams prioritize framework portability and broader ecosystems. ADK remains pre-GA in parts, with occasional rough edges and upgrade friction, but we've seen more projects using it successfully, particularly those already invested in Google's platform.

105. DeepEval

Trial

DeepEval is an open-source, Python-based framework for assessing LLM performance. It can be used to evaluate retrieval-augmented generation (RAG) systems and applications built with frameworks such as LlamaIndex or LangChain, as well as to baseline and benchmark models. DeepEval goes beyond simple word-matching metrics, assessing accuracy, relevance and consistency to provide more reliable evaluation in real-world scenarios. It includes capabilities such as hallucination detection, answer relevance scoring and hyperparameter optimization. One feature our teams have found particularly helpful is that it allows teams to define custom, use-case–specific metrics.

Recently, DeepEval has expanded to support complex agentic workflows and multi-turn conversational systems. Beyond evaluating final outputs, it provides built-in metrics for tool correctness, step efficiency and task completion, including evaluation of interactions with MCP servers. It also introduces conversation simulation to automatically generate test cases and stress-test multi-turn applications at scale.

106. Docling

Trial

Docling is an open-source Python and TypeScript library for converting unstructured documents into clean, machine-readable outputs. Using a computer vision–based approach to layout and semantic understanding, it processes complex inputs — including PDFs and scanned documents — into structured formats such as JSON and Markdown. That makes it a strong fit for retrieval-augmented generation (RAG) pipelines and for producing structured outputs from LLMs, in contrast to vision-first retrieval approaches such as ColPali.

Docling provides an open-source, self-hostable alternative to proprietary cloud-managed services such as Azure Document Intelligence, Amazon Textract and Google Document AI, while integrating well with frameworks such as LangGraph. In our experience, it performs well in production-scale extraction workloads across digital and scanned PDFs, including very large files containing text, tables and images. It delivers a strong quality-to-cost balance for downstream agentic RAG workflows. Based on these results, we're moving Docling to Trial.

107. LangExtract

Trial

LangExtract is a Python library that uses LLMs to extract structured information from unstructured text based on user-defined instructions, with precise source grounding that links each extracted entity to its location in the original document. It processes domain-specific materials such as clinical notes and reports. A key strength is source traceability, which ensures each extracted data point can be traced back to its source. The extracted entities can be exported as a JSONL file, a standard format for language model data, and visualized through an interactive HTML interface for contextual review. Teams considering structured output from LLMs for document processing should evaluate LangExtract alongside schema-enforcement approaches such as Pydantic AI. LangExtract is better suited to long-form, unstructured source material, while Pydantic AI excels at constraining output formats for shorter, more predictable inputs.

108. LangGraph

Trial

Since the previous Radar, we've observed that the LangGraph architecture — which treats every multi-agent system as stateful graphs with a global shared state — is not always the best approach for building agentic systems. We've also seen an alternative approach, used in frameworks such as Pydantic AI, that also works well.

Instead of starting with a rigid graph and a massive shared state, this approach favors simple agents communicating through code execution, with graph structures added later when needed. It often results in leaner and more effective systems for many use cases. Because each agent only has access to the state it needs, reasoning, testing and debugging become easier. As a result, we've moved LangGraph out of Adopt. While it remains a powerful tool, we no longer see it as the default choice for building every agentic system.

109. LiteLLM

Trial

LiteLLM started as a thin abstraction layer over multiple LLM providers but has since expanded into a full-fledged AI gateway. Beyond simplifying API integration, it addresses common cross-cutting concerns in GenAI systems such as retries and failover, load balancing across providers and cost tracking with budget controls.

Our teams are increasingly adopting LiteLLM as a sensible default for AI-powered applications. The gateway provides a consistent place to address governance concerns, including request tracing, access control, API key management and edge-level guardrails such as content filtering and data redaction or masking. However, teams that rely on differentiating provider features will find these often require provider-specific parameters, reintroducing the coupling the gateway is meant to eliminate. It's also worth noting that `drop_params` mode silently discards unsupported parameters, meaning capabilities may be lost across routing decisions without visibility. LiteLLM remains a pragmatic choice for operational control, but teams should understand that leaning into provider-specific capabilities means maintaining both a gateway dependency and provider-coupled code.

110. Modern.js

Trial

Modern.js is a React meta-framework from ByteDance that we're placing in Trial for teams with micro-frontend requirements built on Module Federation. The trigger is practical: `nextjs-mf` is heading toward end-of-life. The Pages Router will receive only small backported fixes, no new development is planned, and CI tests are expected to be removed by mid-to-late 2026. With Next.js lacking official Module Federation support and the community plugin being phased out, the Module Federation core team now recommends Modern.js as the primary supported framework for federation-based architectures. The `@module-federation/modern-js-v3` plugin provides automatic build wiring out of the box, with streaming SSR and Bridge APIs available as separate capabilities. However, combining them has limitations: `@module-federation/bridge-react` is not yet compatible with Node environments, making Bridge unusable in SSR scenarios.

Our early experience is positive, and the migration path is well defined for teams already using Module Federation. The ecosystem outside ByteDance is still maturing, so teams should plan for thinner documentation and closer engagement with upstream. This remains a Trial recommendation, because investment is justified for Module Federation use cases where no better-supported alternative currently exists.

111. Agent Lightning

Assess

Agent Lightning is an agent optimization and training framework that enables automatic prompt optimization, supervised fine-tuning and agentic reinforcement learning. Most agent frameworks focus on building agents, not improving them over time. Agent Lightning addresses this by enabling teams to continuously improve existing agents without changing their underlying implementation, as it supports frameworks such as AutoGen and CrewAI. It achieves this through an approach called Training-Agent Disaggregation, which introduces a layer between the training and agent frameworks. This layer consists of two core components: the Lightning Server and the Lightning Client. The Lightning Server manages the training process and exposes an API for updated models, while the Lightning Client acts as a runtime that collects traces and sends them back to the server to support training. We recommend teams with established agent deployments explore Agent Lightning as a way to continuously improve agent performance.

112. GitHub Spec Kit

Assess

Spec-driven development featured prominently in our discussion this cycle. Two broad camps are emerging: teams that rely on the continually improving capabilities of coding agents with minimal structure, and those that favor defined workflows and detailed specifications.

Several of our teams are experimenting with spec-driven practices using GitHub Spec Kit, mostly in brownfield environments. A key concept in Spec Kit is the constitution, a foundational rulebook that aligns the software development lifecycle. In practice, teams reported that a useful constitution typically captures project scope, domain context, technology versions, coding standards and repository structure (for example, hexagonal architecture or layered modules). This shared context helps agents operate within the intended architectural boundaries.

Teams also encountered challenges such as instruction bloat, where continually adding project context led to a growing agent instruction set and eventually context rot. One team addressed this by extracting reusable guidance into skills, keeping agent instructions lean and loading detailed context only when needed.

In brownfield systems, much rework stems from unclear intent, hidden assumptions and late discovery of constraints. One team adopted a lifecycle of spec → plan → tasks → coding → review, which helped surface issues earlier. Over time they also moved repeatable context into files such as `.github/prompts/speckit.<command>.prompt.md`, making prompts shorter and agent behavior more consistent. Teams did report rough edges, including unnecessary defensive checks and overly verbose markdown outputs that increased cognitive load. Customizing Spec Kit templates and instructions — for example limiting the number of generated markdown files and reducing console verbosity — helped address some of these issues. Ultimately, experienced engineers, particularly those with strong clean coding and architectural practices, tend to extract the most value from spec-driven workflows.

113. Mastra

Assess

Mastra is an open-source, TypeScript-native framework for building AI applications and agents. It offers a graph-based workflow engine, unified access to a variety of LLM providers, human-in-the-loop suspension and resumption, as well as RAG and memory primitives. It also includes MCP server authoring and built-in tooling for evaluation and observability, supported by clear developer documentation. Mastra provides an alternative to Python-heavy stacks, allowing teams to build feature-rich AI capabilities directly within existing web ecosystems such as Node.js or Next.js. It's worth evaluating for teams already invested in the TypeScript ecosystem who want to avoid switching to Python for their AI layer.

114. Pipecat

Assess

Pipecat is an open-source framework for building real-time voice and multimodal agents with a modular pipeline model for STT, LLM, TTS and transport orchestration. It's attracting strong interest because teams can iterate quickly on conversational behavior and swap providers with relatively low friction.

Compared to LiveKit Agents, Pipecat offers greater framework flexibility but a less integrated production path, especially for self-hosted deployment, transport reliability and low-latency turn handling at scale. We've placed Pipecat in Assess because it provides a strong engineering-facing foundation, but significant platform engineering work is required before it can be relied on for business-critical production workloads.

115. Superpowers

Assess

With the growing use of coding agents, there is no single prescribed workflow for every team. Instead, teams are evolving bespoke workflows based on their context and constraints. Our teams across the globe are actively defining and experimenting with such AI-assisted coding workflows. Superpowers is one such workflow built from composable skills. It wraps a coding agent in a structured workflow

of skills that encourage brainstorming before coding, detailed planning before implementation, test-driven development with enforced red-green-refactor cycles, systematic root-cause-first debugging and post-implementation code review. Superpowers is distributed as a plugin via the Claude Code plugin marketplace as well as the Cursor plugin marketplace.

116. TanStack Start

Assess

TanStack Start is a full-stack framework built on TanStack Router for React and Solid. It's comparable to Next.js, supporting SSR, caching and many of the same features. TanStack Start provides end-to-end compile-time safety across server functions, loaders and routing, reducing the risk of broken links or mismatched data shapes on the frontend.

The framework favors explicit configuration over convention, making the experience closer to working with plain React. You can also progressively add SSR capabilities as needed. Compared to Next.js, which has more opinionated defaults that can lead to unexpected behavior if teams are unfamiliar with its inner workings, TanStack Start is more explicit and predictable. The TanStack ecosystem has also matured significantly, offering a strong set of tools for building modern web applications.

117. TOON (Token-Oriented Object Notation)

Assess

TOON (Token-Oriented Object Notation) is a human-readable encoding for JSON data designed to reduce token usage when structured data is passed to LLMs. It allows teams to retain JSON in existing systems and transform it only at the point of interaction with the model. This matters because token cost, latency and context-window constraints are becoming real design considerations in RAG pipelines, agent workflows and other AI-heavy applications. Raw JSON often spends tokens on repeated keys and structural overhead rather than useful content.

In our early evaluation, TOON is an interesting last-mile optimization for prompt inputs, particularly for large, regular datasets where a more schema-aware format can be both more efficient and easier for models to process than JSON. It's not a replacement for JSON in APIs, databases or model outputs, and is often the wrong choice for deeply nested or non-uniform structures, semi-uniform arrays or flat tabular data where CSV is more compact. It may also be less suitable for latency-critical paths where compact JSON performs well. For these reasons, we think TOON is worth assessing for teams building LLM applications where structured input size is a meaningful cost or quality concern. Teams should benchmark it against JSON or CSV using their own data and model stack.

118. Unsloth

Assess

Unsloth is an open-source framework for LLM fine-tuning and reinforcement learning that focuses on making training significantly faster and more memory efficient. Fine-tuning LLMs involves billions of matrix multiplications, workloads that benefit from GPU acceleration. Unsloth optimizes these operations by translating them into highly efficient custom kernels for NVIDIA GPUs, dramatically reducing cost and memory usage. This makes it possible to fine-tune models on consumer GPUs such as T4 and above rather than requiring expensive H100 clusters. Unsloth supports LoRA, full fine-tuning, multi-GPU training and long-context fine-tuning (up to 500K tokens) for popular models including Llama, Mistral, DeepSeek-R1, Qwen and Gemma. As domain-specific AI applications increasingly rely on fine-tuning, Unsloth significantly lowers the barrier.

Stay up to date with all Radar-related news and insights

Subscribe to the Technology Radar to receive emails every other month for tech insights from Thoughtworks and future Technology Radar releases.

Subscribe now