Developers are moving past AI as just tooling and starting to treat it as a new foundation for how software gets built. Many of the core concepts we’ve taken for granted — version control, templates, documentation, even the idea of a user — are being rethought in light of agent-driven workflows.

As agents become both collaborators and consumers, we expect to see a shift in foundational developer tools. Prompts can be treated like source code, dashboards can become conversational, and docs are written as much for machines as for humans. Model Context Protocol (MCP) and AI-native IDEs point to a deeper redesign of the development loop itself: we’re not just coding differently, we’re designing tools for a world where agents participate fully in the software loop.

Below, we explore nine forward-looking developer patterns that, although early, are grounded in real pain points and give a hint of what could emerge. These range from rethinking version control for AI-generated code, to LLM-driven user interfaces and documentation.

Let’s dive into each pattern, with examples and insights from the dev community.

1. AI-native Git: Rethinking version control for AI agents

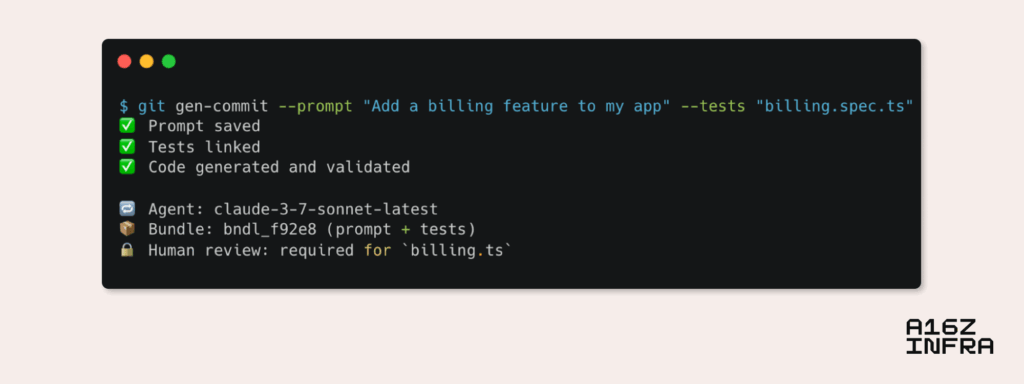
Now that AI agents increasingly write or modify large portions of application code, what developers care about starts to change. We’re no longer fixated on exactly what code was written line-by-line, but rather on whether the output behaves as expected. *Did the change pass the tests? Does the app still work as intended?*

**This flips a long-standing mental model: Git was designed to track the precise history of hand-written code, but, with coding agents, that granularity becomes less meaningful.**Developers often don’t audit every diff — especially if the change is large or auto-generated — they just want to know whether the new behavior aligns with the intended outcome. As a result, the Git SHA — once the canonical reference for “the state of the codebase” — begins to lose some of its semantic value.

A SHA tells you that something changed, but not *why* or *whether it’s valid*. In AI-first workflows, a more useful unit of truth might be a combination of the **prompt that generated the code** and the **tests that verify its behavior**. In this world, the “state” of your app might be better represented by the inputs to generation (prompt, spec, constraints) and a suite of passing assertions, rather than a frozen commit hash. In fact, we might eventually track *prompt+test bundles* as versionable units in their own right, with Git relegated to tracking those bundles, not just raw source code.

**Taking this a step further: In agent-driven workflows, the source of truth may shift upstream toward prompts, data schemas, API contracts, and architectural intent.** Code becomes the byproduct of those inputs, more like a compiled artifact than a manually authored source. Git, in this world, starts to function less as a workspace and more as an artifact log — a place to track not just what changed, but why and by whom. We may begin to layer in richer metadata, such as which agent or model made a change, which sections are protected, and where human oversight is required – or where AI reviewers like [Diamond](http://diamond.graphite.dev/) can step in as part of the loop.

To make this more concrete, below is a mock-up of what an AI-native Git flow could look like in practice:

**[](https://d1lamhf6l6yk6d.cloudfront.net/uploads/2025/05/image3.png)**

2. Dashboards -> Synthesis: Dynamic AI-driven interfaces

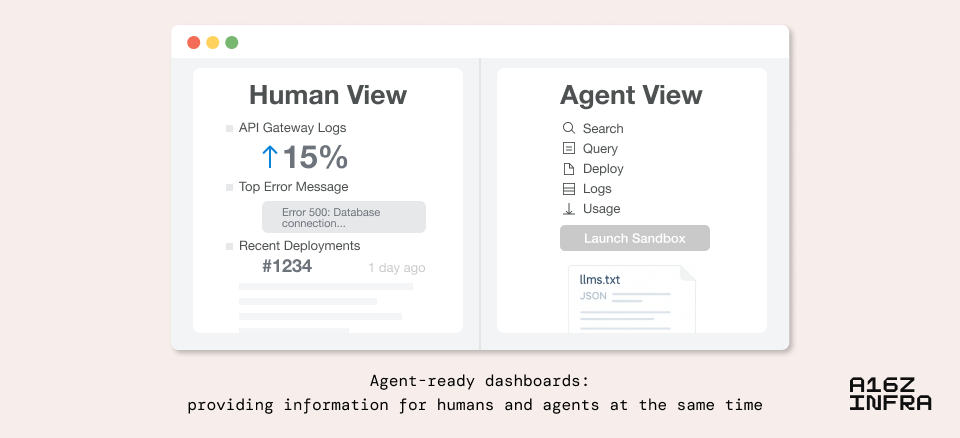
For years, dashboards have served as the primary interface for interacting with complex systems such as observability stacks, analytics, cloud consoles (think AWS), and more. But their design often suffers from an overloaded UX: too many knobs, charts, and tabs that force users to both hunt for information and figure out how to act on it. Especially for non-power users or across teams, these dashboards can become intimidating or inefficient. Users know what they want to achieve, but not where to look or which filters to apply to get there.

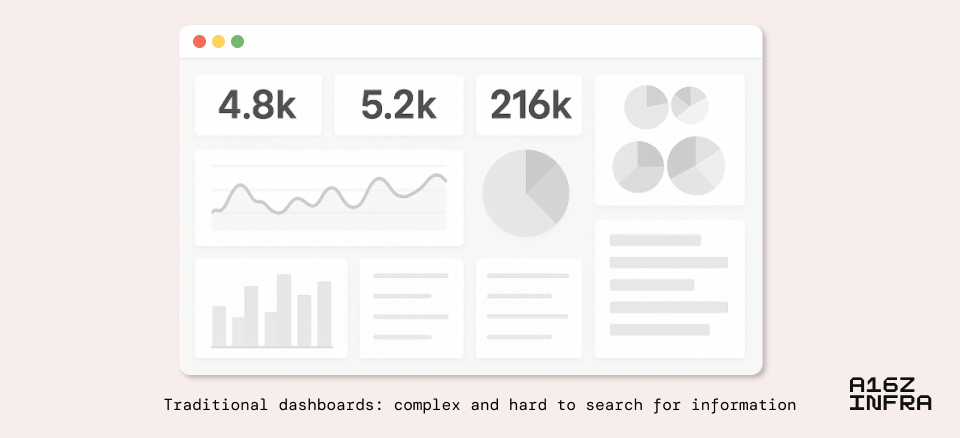
The latest generation of AI models offers a potential shift. **Instead of treating dashboards as rigid canvases, we can layer in search and interaction.** LLMs can now help users find the right control (“Where can I adjust the rate limiter settings for this API?”); synthesize screen-wide data into digestible insights (“Summarize the error trends across all services in staging over the past 24 hours”); and surface unknown/unknowns (“Given what you know about my business, generate a list of metrics I should pay attention to this quarter”).

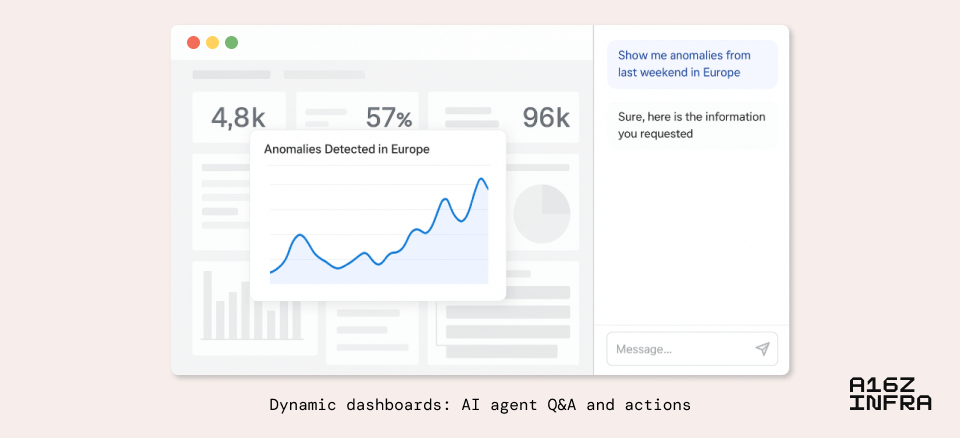
We are already seeing technical solutions like [Assistant UI](https://www.assistant-ui.com/docs/copilots/make-assistant-tool) that make it possible for agents to leverage React components as tools. Just as content has become dynamic and personalized, UI itself can become adaptive and conversational. A purely static dashboard may soon feel outdated next to a natural language-driven interface that reconfigures based on user intent. For example, instead of clicking through five filters to isolate metrics, a user might say, *“Show me anomalies from last weekend in Europe,”* and the dashboard reshapes to show that view, complete with summarized trends and relevant logs. Or, even more powerfully, *“Why did our NPS score drop last week?”,*  and the AI might pull up survey sentiment, correlate it with a product deployment, and generate a short diagnostic narrative.

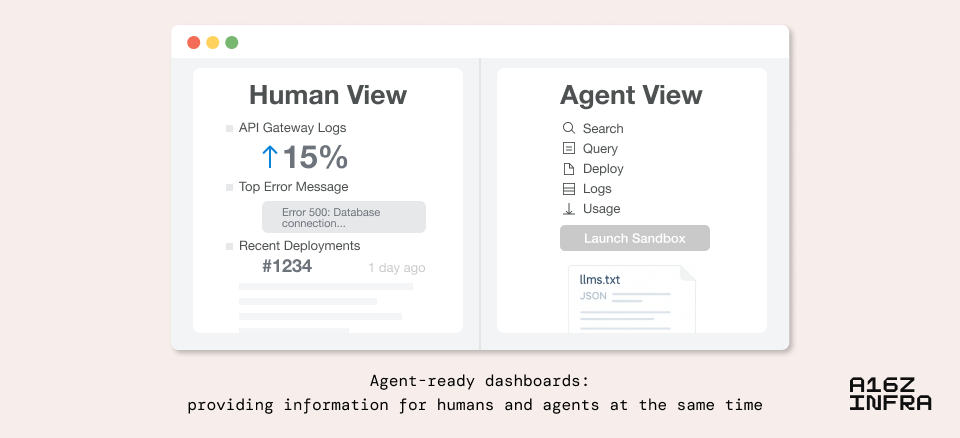
**At a larger scale, if agents are now consumers of software, we may also need to rethink what “dashboards” are or for whom they’re designed.**For example, dashboards could render views optimized for [agent experience](https://agentexperience.ax/) — structured, programmatically accessible surfaces designed to help agents perceive system state, make decisions, and act. This might lead to dual-mode interfaces: one human-facing and one agent-facing, both sharing a common state but tailored to different modes of consumption.

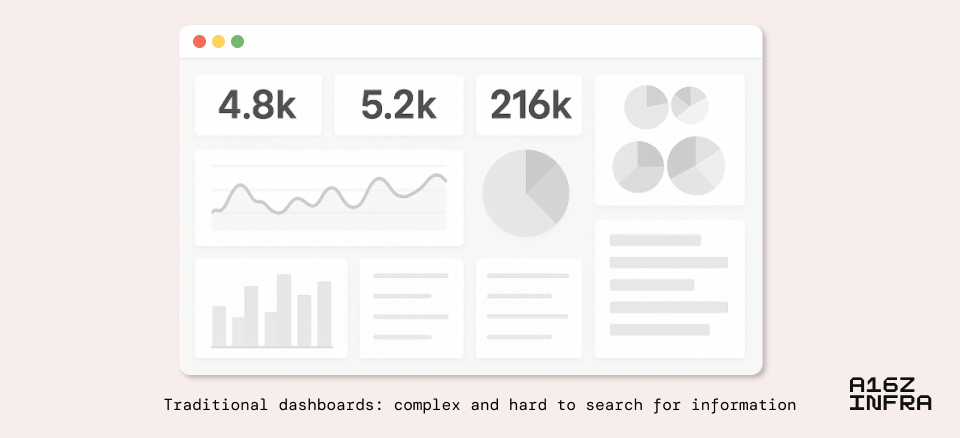
In some ways, agents are stepping into roles once filled by alerts, cron jobs, or condition-based automation, but with far more context and flexibility. Instead of pre-wired logic like *if error rate > threshold, send alert*, an agent might say, *“Error rates are rising. Here’s the likely cause, the impacted services, and a proposed fix.”* In this world, dashboards aren’t just places to observe; they’re places where both humans and agents collaborate, synthesize, and take action.











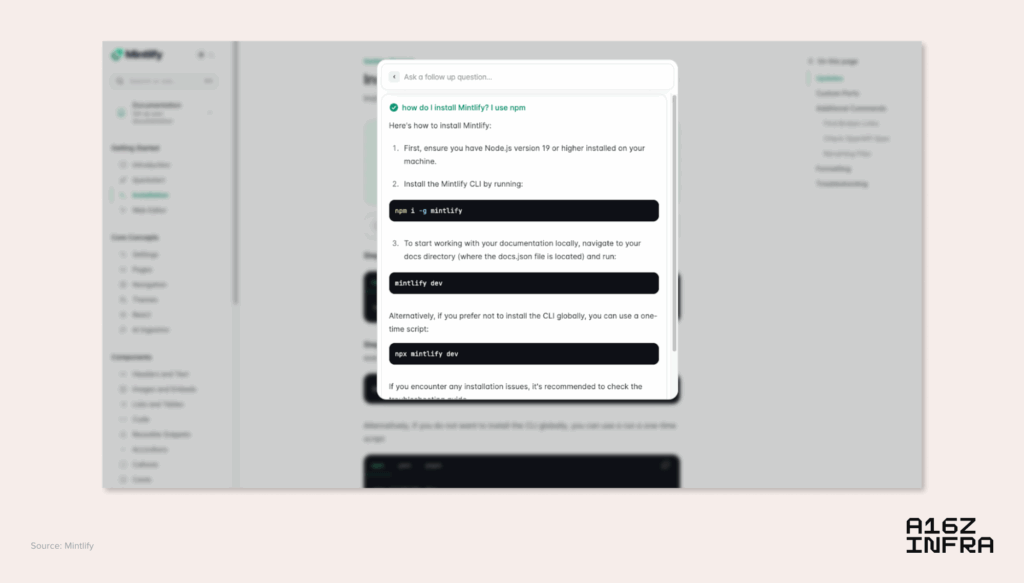
How dashboards might evolve to support both human and AI-agent viewers.

3. Docs are becoming a combination of tools, indices, and interactive knowledge bases

Developer behavior is shifting when it comes to documentation. Instead of reading through a table of contents or scanning top-down, users now start with a question. The mental model is no longer *“Let me study this spec”*, but *“Rework this information for me, in a way I like to consume.”* This subtle shift — from passive reading to active querying — is changing what docs need to be. Rather than just static HTML or markdown pages, they’re becoming interactive knowledge systems, backed by indices, embeddings, and tool-aware agents.

As a result, we’re seeing the rise of products like Mintlify, which not only structure documentation as semantically searchable databases, but also serve as context sources for coding agents across platforms. Mintlify pages are now frequently cited by AI coding agents — whether in AI IDEs, VS Code extensions, or terminal agents — because coding agents use up-to-date documentation as grounding context for generation.

**This changes the purpose of docs: they’re no longer just for human readers, but also for agent consumers.** In this new dynamic, the documentation interface becomes something like instructions for AI agents. It doesn’t just expose raw content, but explains how to use a system correctly.

[](https://d1lamhf6l6yk6d.cloudfront.net/uploads/2025/05/image5.png)

A screenshot from the Mintlify where users can bring up the AI chat window to do Q&A over Mintlify documentations using the cmd+k shortcut

4. Templates to generation: Vibe coding replaces create-react-app

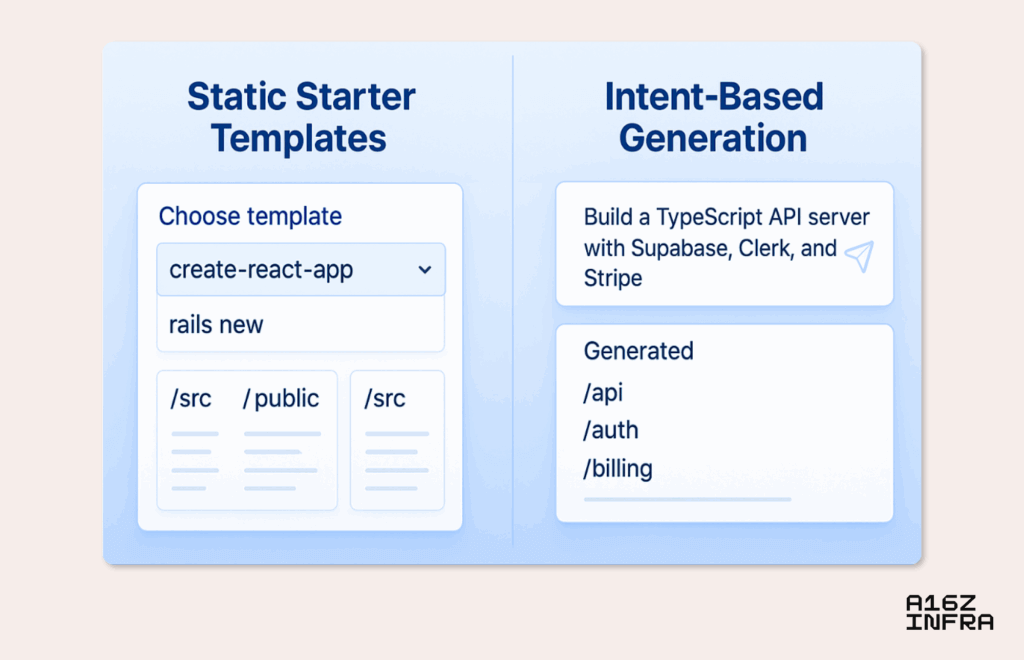
In the past, getting started on a project meant choosing a static template such as a boilerplate GitHub repo or a CLI like *create-react-app*, *next init*, or *rails new*. These templates served as the scaffolding for new apps, offering consistency but little customization. Developers conformed to whatever defaults the framework provided or risked significant manual refactoring.

Now, that dynamic is shifting with the emergence of text-to-app platforms like Replit, Same.dev, Loveable, Chef by Convex, and Bolt, as well as AI IDEs like Cursor. Developers can describe what they want (e.g., “a TypeScript API server with Supabase, Clerk and Stripe”) and have a custom project scaffolded in seconds. The result is a starter that’s not generic, but personalized and purposeful, reflecting both the developer’s intent and their chosen stack.

This unlocks a new distribution model in the ecosystem. **Instead of a few frameworks sitting at the head of the long tail, we may see a wider spread of composable, stack-specific generations** where tools and architectures are mixed and matched dynamically. It’s less about picking a framework and more about describing an outcome around which the AI can build a stack. One engineer might create an app with Next.js and tRPC, while another starts with Vite and React, but both get working scaffolds instantly.

Of course, there are tradeoffs. Standard stacks bring real advantages, including making teams more productive, improving onboarding, and making troubleshooting easier across orgs. Refactoring across frameworks isn’t just a technical lift; it’s often entangled with product decisions, infrastructure constraints, and team expertise. But what’s shifting is *the cost of switching frameworks or starting without one.* With AI agents that understand project intent and can execute large refactors semi-autonomously, it becomes much more feasible to experiment — and to reverse course, if needed.

**This means** **framework decisions are becoming much more reversible**. A developer might start with Next.js, but later decide to migrate to Remix and Vite, and ask the agent to handle the bulk of the refactor. This reduces the lock-in that frameworks used to impose and encourages more experimentation, especially at early stages of a project. It also lowers the bar for trying opinionated stacks, because switching later is no longer a massive investment.

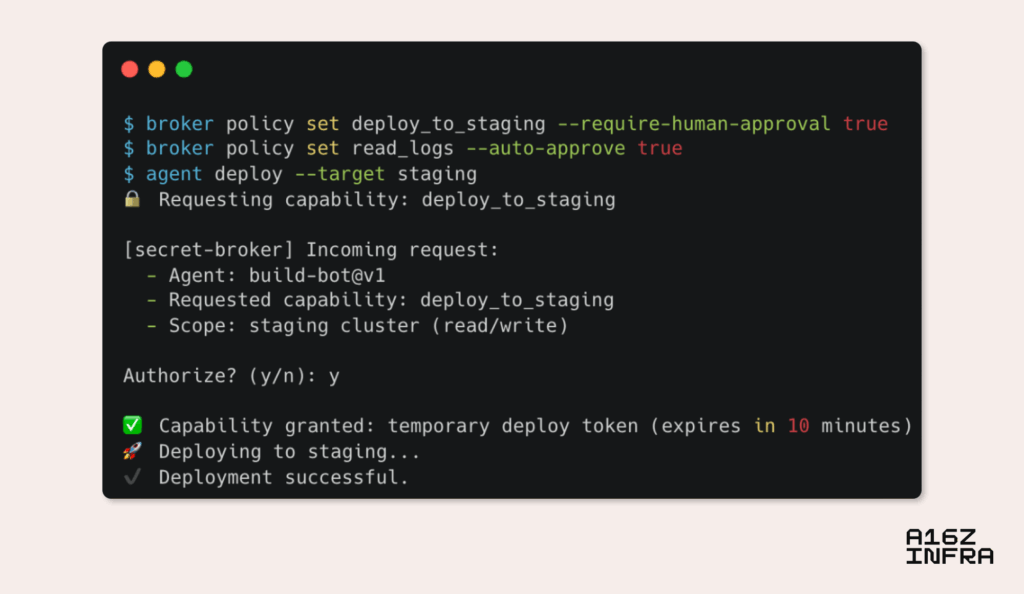
**[](https://d1lamhf6l6yk6d.cloudfront.net/uploads/2025/05/image2.png)**

5. Beyond .env: Managing secrets in an agent-driven world

For decades, .env files have been the default way for developers to manage secrets (e.g., API keys, database URLs, and service tokens) locally. They’re simple, portable, and developer-friendly. But in an agent-driven world, this paradigm begins to break down. **It’s no longer clear who owns the .env when an AI IDE or agent is writing code, deploying services, and orchestrating environments on our behalf.**

We’re seeing hints of what this could look like. The latest MCP spec, for example, includes an authorization [framework](https://modelcontextprotocol.io/specification/draft/basic/authorization#2-authorization-flow) based on OAuth 2.1​, signaling a possibility to move toward giving AI agents scoped, revocable tokens instead of raw secrets. We can imagine a scenario where an AI agent doesn’t get your actual AWS keys, but instead obtains a short-lived credential or a capability token that lets it perform a narrowly defined action.

Another way this could shake out is through the rise of local secret brokers — services running on your machine or alongside your app that act as intermediaries between agents and sensitive credentials. Rather than injecting secrets into .env files or hardcoding them into scaffolds, the agent could request access to a capability (*“deploy to staging*” or *“send logs to Sentry”*), and the broker determines whether to grant it — just-in-time, and with full auditability. This decouples secret access from the static filesystem, and makes secret management feel more like API authorization than environment configuration.

[](https://d1lamhf6l6yk6d.cloudfront.net/uploads/2025/05/image1.png)

A CLI mock-up of what the agent-centric secret broker flow could look like.

6. Accessibility as the universal interface: Apps through the eyes of an LLM

We’re starting to see a new class of apps (e.g., [Granola](https://www.granola.ai/) and [Highlight](https://highlightai.com/)) that request access to accessibility settings on macOS not for traditional accessibility use cases, but to enable AI agents to observe and interact with interfaces. However, this isn’t a hack: It’s a glimpse into a deeper shift.

Accessibility APIs were built to help users with vision or motor impairments navigate digital systems. But those same APIs, when extended thoughtfully, may become the universal interface layer for agents.**Instead of clicking pixel positions or scraping DOMs, agents could observe applications the way assistive tech does — semantically.** The accessibility tree already exposes structured elements like buttons, headings, and inputs. If extended with metadata (e.g., intent, role, and affordance), this could become a first-class interface for agents, letting them perceive and act on apps with purpose and precision.

There are a couple potential directions:

* **Context extraction**: A standard way for an LLM agent using accessibility or semantic APIs to query what’s on screen, what it can interact with, and what the user is doing.
* **Intentful execution:** Rather than expecting an agent to chain multiple API calls manually, expose a high-level endpoint where it can declare goals (“add an item to the cart, choose fastest shipping”), and let the backend figure out the steps.
* **Fallback UI for LLMs:** Accessibility features provide a fallback UI for LLMs. Any app that exposes a screen becomes agent-usable, even if it doesn’t have a public API. For developers, it suggests a new “render surface” — not just visual or DOM layers, but agent-accessible context, possibly defined via structured annotations or accessibility-first components.

7. The rise of asynchronous agent work

As developers begin to work alongside coding agents more fluidly, we’re seeing a natural shift toward asynchronous workflows where agents operate in the background, pursue parallel threads of work, and report back when they’ve made progress. This mode of interaction is starting to look less like pair programming and more like task orchestration: you delegate a goal, let the agent run, and check in later.

**Crucially, this isn’t just about offloading effort; it also compresses coordination.** Instead of pinging another team to update a config file, triage an error, or refactor a component, developers can increasingly assign that task directly to an agent that acts on their intent and executes in the background. What once required sync meetings, cross-functional handoffs, or long review cycles could become an ambient loop of *request, generate,*and*validate*.

The surfaces for agent interaction are expanding, too. Instead of always prompting via IDE or CLI, devs could begin to interact with agents by, for example:

* Sending messages to Slack.
* Commenting on Figma mocks.
* Creating inline annotations on code diffs or PRs (e.g. Graphite’s review assistant).
* Adding feedback based on deployed app previews.
* Utilizing voice or call-based interfaces, where devs can describe changes verbally.

**This creates a model where agents are present across the full lifecycle of development.** They’re not just writing code, but interpreting designs, responding to feedback, and triaging bugs across platforms. The developer becomes the orchestrator who decides which thread to pursue, discard, or merge.

Perhaps this model of branching and delegating to agents becomes the new *Git branch* — not a static fork of code, but a dynamic thread of intent, running asynchronously until it’s ready to land.

8. MCP is one step closer to becoming a universal standard

We recently published [a deep dive on MCP](https://a16z.com/a-deep-dive-into-mcp-and-the-future-of-ai-tooling/). Since then, momentum has accelerated: OpenAI publicly adopted MCP, several new features of the spec were merged, and toolmakers are starting to converge around it as the default interface between agents and the real world.

At its core, MCP solves two big problems:

* It gives an LLM the right set of context to complete tasks it may have never seen.
* It replaces N×M bespoke integrations with a clean, modular model in which tools expose standard interfaces (servers) usable by any agents (clients).

We expect to see broader adoption as remote MCP and a de-facto registry come online. And, over time,**apps may begin shipping with MCP surfaces by default.**

Think of how APIs enabled SaaS products to plug into each other and compose workflows across tools. MCP could do the same for AI agents by turning standalone tools into interoperable building blocks. A platform that ships with an MCP client baked in isn’t just “AI-ready,” but is part of a larger ecosystem, instantly able to tap into a growing network of agent-accessible capabilities.

**Additionally, MCP clients and servers are logical barriers, not physical boundaries.** This means any client can also act as a server, and vice versa. This could theoretically unlock a powerful level of composability via which an agent using an MCP client to consume context can also expose its own capabilities via a server interface. For example, a coding agent could act as a client to fetch GitHub issues, but also register itself as a server that exposes test coverage or code analysis results to other agents.

9. Abstracted primitives: Every AI agent needs auth, billing, and persistent storage

As vibe coding agents get more powerful, one thing becomes clear: Agents can generate a lot of code, but they still need something solid to plug into. Just like human developers lean on Stripe for payments, Clerk for auth, or Supabase for database capabilities, agents need similarly clean and composable service primitives to scaffold reliable applications.

In many ways, these services — APIs with clear boundaries, ergonomic SDKs, and sane defaults that reduce the chance of failure — are increasingly serving as the runtime interface for agents. If you’re building a tool that generates a SaaS app, you don’t want your agent to roll its own auth system or write billing logic from scratch; you want it to use providers like Clerk and Stripe.

As this pattern matures, **we may start to see services optimize themselves for agent consumption by exposing not just APIs, but also schemas, capability metadata, and example flows** that help agents integrate them more reliably.

Some services might even start shipping with MCP servers by default, turning every core primitive into something agents can reason about and use safely out of the box. Imagine Clerk exposing an MCP server that lets an agent query available products, create new billing plans, or update a customer’s subscription — all with permission scopes and constraints defined up front. Instead of hand-authoring API calls or hunting through docs, an agent could say, *“Create a monthly ‘Pro’ plan at $49 with usage-based overages,”* and Clerk’s MCP server would expose that capability, validate the parameters, and handle the orchestration securely.

Just as the early web era needed Rails generators and rails new to move fast, the agent era needs trustworthy primitives — drop-in identity, usage tracking, billing logic, and access control — all abstracted enough to generate against, but expressive enough to grow with the app.

Conclusion

These patterns point to a broader shift in which new developer behaviors are emerging alongside more capable foundation models. And, in response, we’re seeing new toolchains and protocols like MCP take shape. It’s not just AI layered onto old workflows, but is a redefinition of how software gets built with agents, context, and intent at the core. Many developer-tooling layers are fundamentally shifting, and we are excited to build and invest in the next generation of tools.