How Cursor Shipped its Coding Agent to Production

BY TEBY TEGO
JAN 26, 2026

New report: 96% of devs don't fully trust AI code (Sponsored)

REPORT
Sonar
2026 State of Code Developer Survey report
Read the results
State of Code Developer Survey report
2026

AI is accelerating code generation, but it's creating a bottleneck in the verification phase. Based on a survey of 1,100+ developers, Sonar's newest State of Code report analyzes the impact of generative AI on software engineering workflows and how developers are adapting to address it.

Survey findings include:

- 96% of developers don't fully trust that AI-generated code is functionally correct yet only 48% always check it before

committing
- 61% agree that AI often produces code that looks correct but isn't reliable
- 24% of a developer's work week is spent on toil work

Download survey

On October 29, 2025, Cursor shipped Cursor 2.0 and introduced **Composer**, its first agentic coding model. Cursor claims Composer is 4x faster than similarly intelligent models, with most turns completing in under 30 seconds. For more clarity and detail, we worked with Lee Robinson at Cursor on this article.

Shipping a reliable coding agent requires a lot of systems engineering. Cursor's engineering team has shared technical details and challenges from building Composer and shipping their coding agent into production. This article breaks down those engineering challenges and how they solved them.

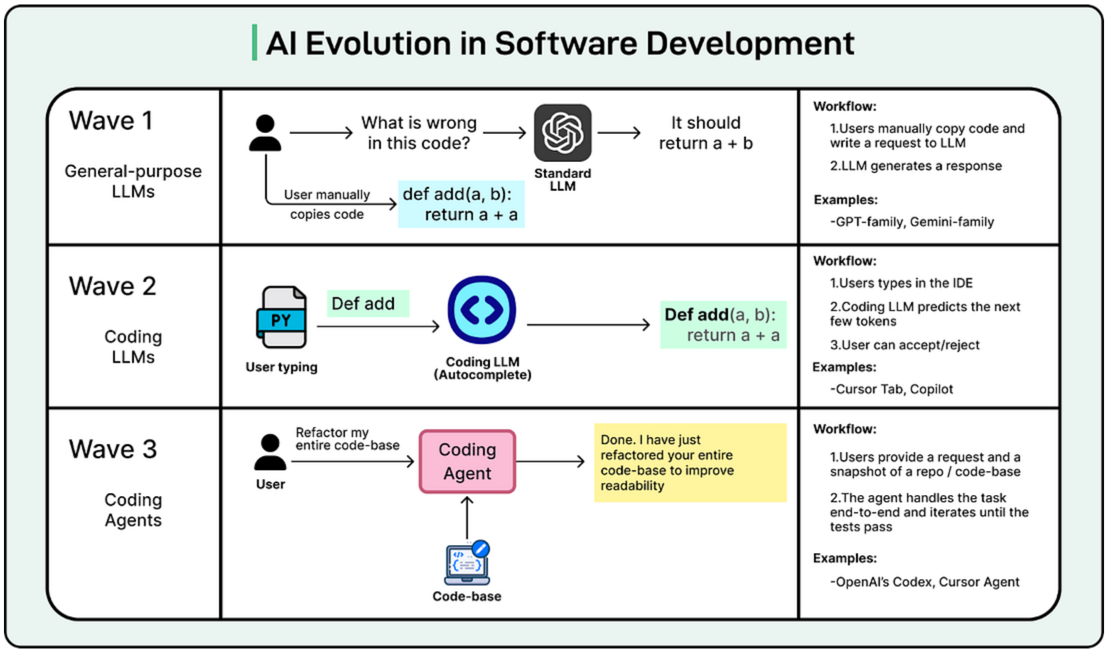
What is a Coding Agent?

To understand coding agents, we first need to look at how AI coding has evolved.

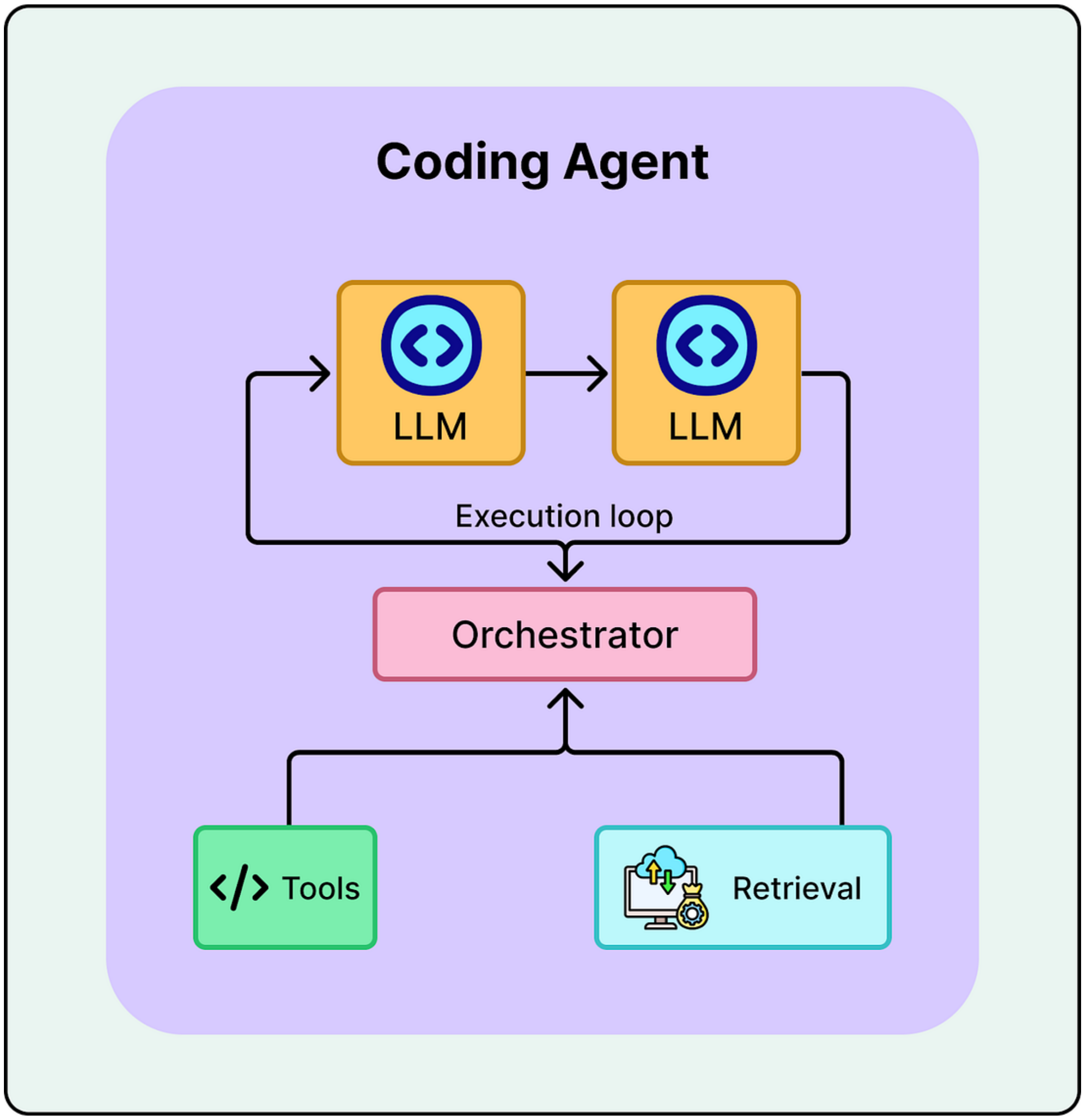
AI in software development has evolved in three waves. First, we treated general-purpose LLMs like a coding partner. You copied code, pasted it into ChatGPT, asked for a fix, and manually applied the changes. It was helpful, but disconnected.

In the second wave, tools like Copilot and Cursor Tab brought AI directly into the editor. To power these tools, specialized models were developed for fast, inline autocomplete. They helped developers type faster, but they were limited to the specific file being edited.

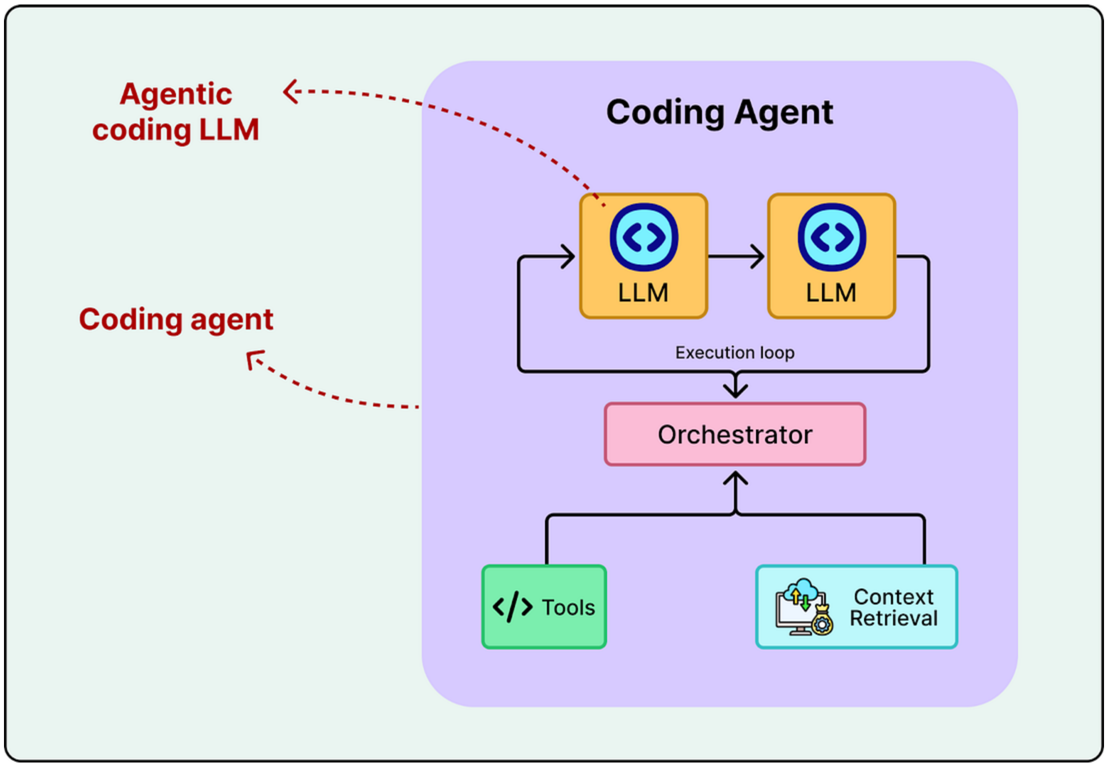
More recently, the focus has shifted to coding agents that handle tasks end-to-end. They don’t just suggest code; they handle coding requests end-to-end. They can search your repo, edit multiple files, run terminal commands, and iterate on errors until the build and tests pass. We are currently living through this third wave.


AI Evolution in Software Development
Wave 1
General-purpose LLMs
What is wrong in this code?
Standard LLM
It should return a + b
User manually copies code
def add(a, b): return a + a
Workflow:
1.Users manually copy code and write a request to LLM
2.LLM generates a response
Examples:
-GPT-family, Gemini-family
Wave 2
Coding LLMs
PY
User typing
Def add
Coding LLM (Autocomplete)
Def add(a, b): return a + a
Workflow:
1.Users types in the IDE
2.Coding LLM predicts the next few tokens
3.User can accept/reject
Examples:
-Cursor Tab, Copilot
Wave 3
Coding Agents
User
Refactor my entire code-base
Coding Agent
Done. I have just refactored your entire code-base to improve readability
Code-base
Workflow:
1.Users provide a request and a snapshot of a repo / code-base
2.The agent handles the task end-to-end and iterates until the tests pass
Examples:
-OpenAI's Codex, Cursor Agent

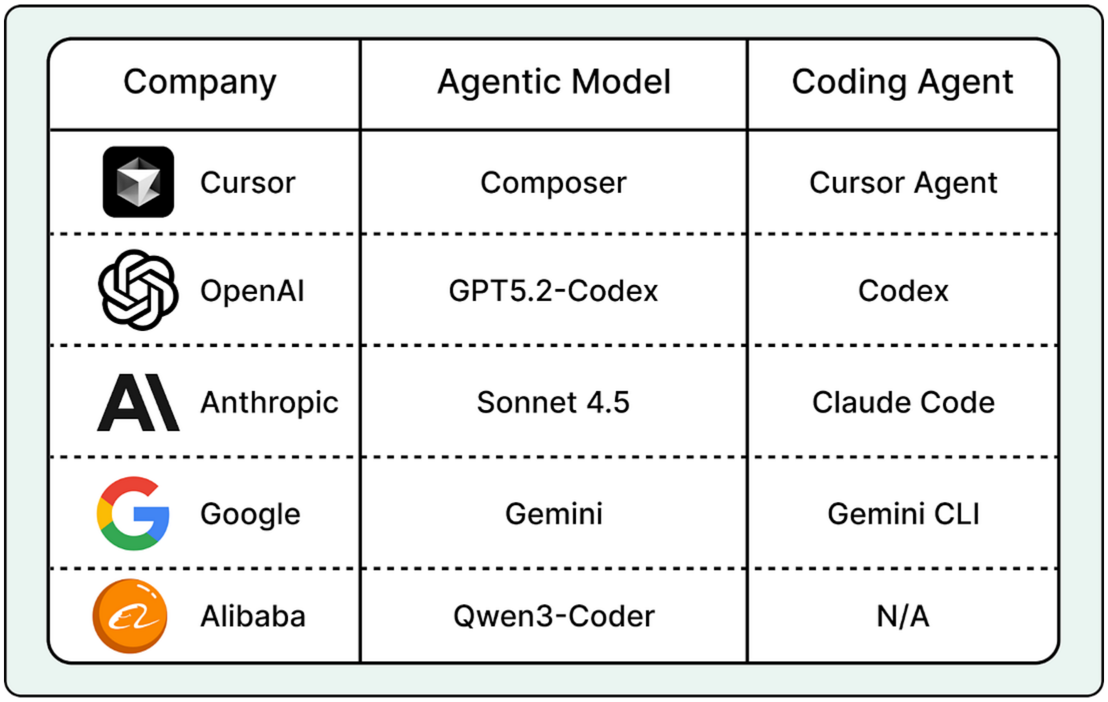
A coding agent is not a single model. It is a system built around a model with tool access, an iterative execution loop, and mechanisms to retrieve relevant code. The model, often referred to as an **agentic coding model**, is a specialized LLM trained to reason over codebases, use tools, and work effectively inside an agentic system.



It is easy to confuse agentic coding models with coding agents. The agentic coding model is like the brain. It has the intelligence to reason, write code, and use tools. The coding agent is the body. It has the “hands” to execute tools, manage context, and ensure it reaches a working solution by iterating until the build and tests pass.



AI labs first train an agentic coding model, then wrap it in an agent system, also known as a harness, to create a coding agent. For example, OpenAI Codex is a coding agent environment powered by the GPT-5.2-Codex model, and Cursor's coding agent can run on multiple frontier models, including its own agentic coding model, Composer. In the next section, we take a closer look at Cursor's coding agent and Composer.



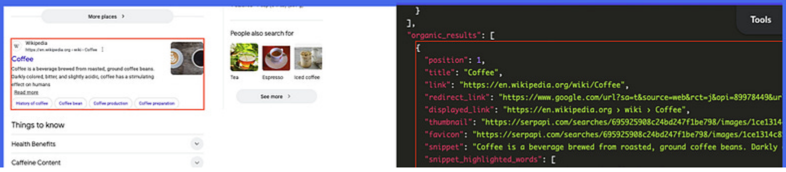
Company	Agentic Model	Coding Agent
Cursor	Composer	Cursor Agent
OpenAI	GPT5.2-Codex	Codex
Anthropic	Sonnet 4.5	Claude Code
Google	Gemini	Gemini CLI
Alibaba	Qwen3-Coder	N/A

[Web Search API for Your AI Applications (Sponsored)]

SerpApi

Web Search API for your AI applications.
Scrape search engines with our fast, easy, and complete API.

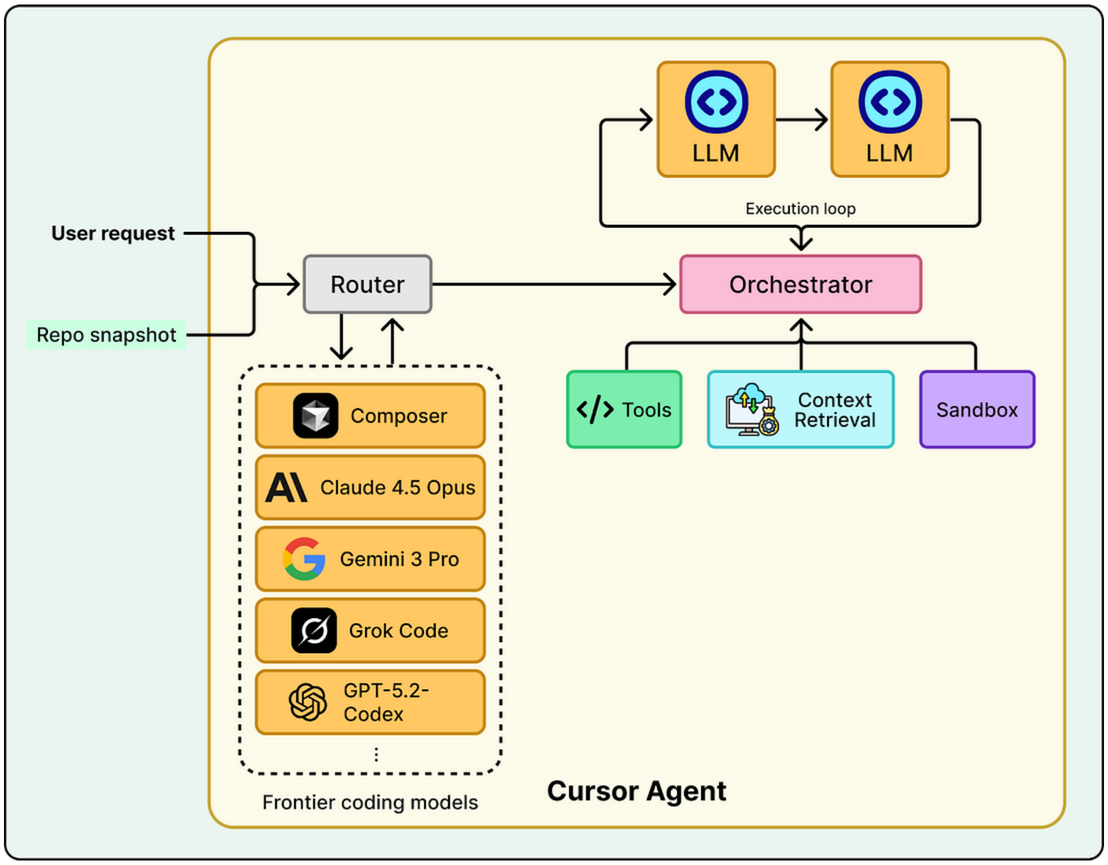
Try For Free



```
  }
],
"organic_results": [
  {
    "position": 1,
    "title": "Coffee",
    "link": "https://en.wikipedia.org/wiki/Coffee",
    "redirect_link": "https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&ur
    "displayed_link": "https://en.wikipedia.org › wiki › Coffee",
    "thumbnail": "https://serpapi.com/searches/695925908c24bd247f1be798/images/1ce1314
    "favicon": "https://serpapi.com/searches/695925908c24bd247f1be798/images/1ce1314c8
    "snippet": "Coffee is a beverage brewed from roasted, ground coffee beans. Darkly
    "snippet_highlighted_words": [
```

System Architecture

A production-ready coding agent is a complex system composed of several critical components working in unison. While the model provides the intelligence, the surrounding infrastructure is what enables it to interact with files, run commands, and maintain safety. The next Figure shows the key components of a Cursor’s agent system.



Router

Cursor integrates multiple agentic models, including its own specialized Composer model. For efficiency, the system offers an “Auto” mode that acts as a router. It dynamically analyzes the complexity of each request to choose the best model for the job.

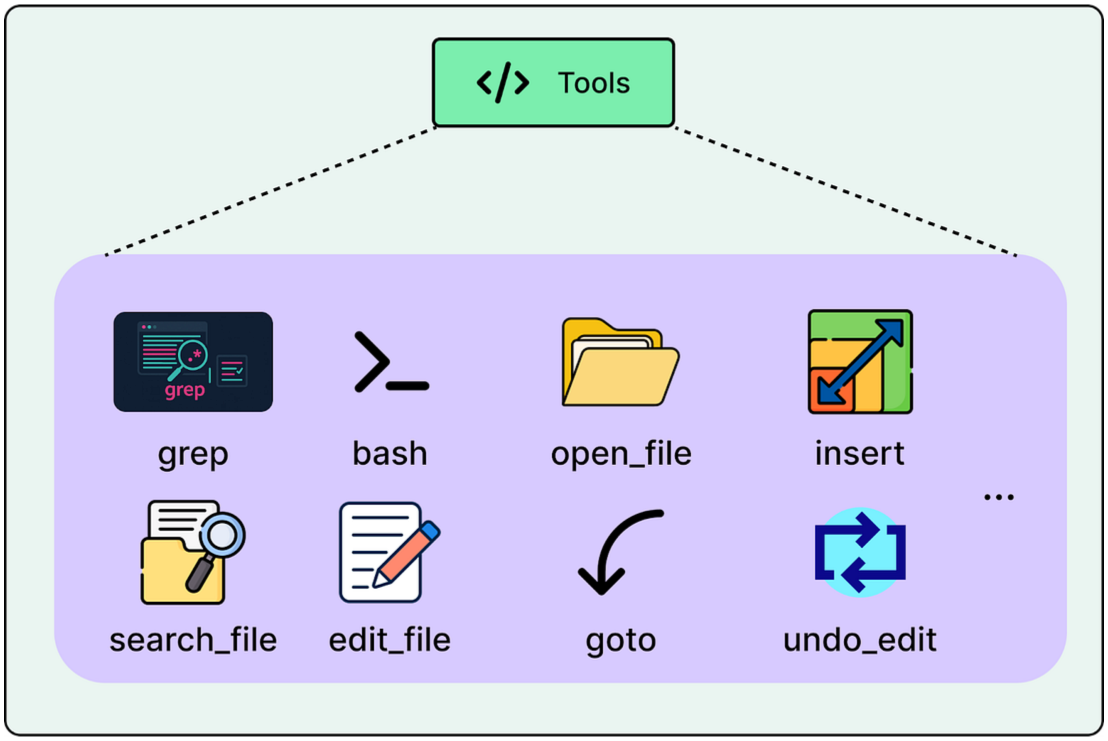
LLM (agentic coding model)

The heart of the system is the agentic coding model. In Cursor’s agent, that model can be Composer, or any other frontier coding models picked by the router. Unlike a standard LLM trained just to predict the next token of text, this model is trained on trajectories, sequences of actions that show the model how and when to use available tools to solve a problem.

Creating this model is often the heaviest lift in building a coding agent. It requires massive data preparation, training, and testing to ensure the model doesn't just write code, but understands the process of coding (e.g., "search first, then edit, then verify"). Once this model is ready and capable of reasoning, the rest of the work shifts to system engineering to provide the environment it needs to operate.

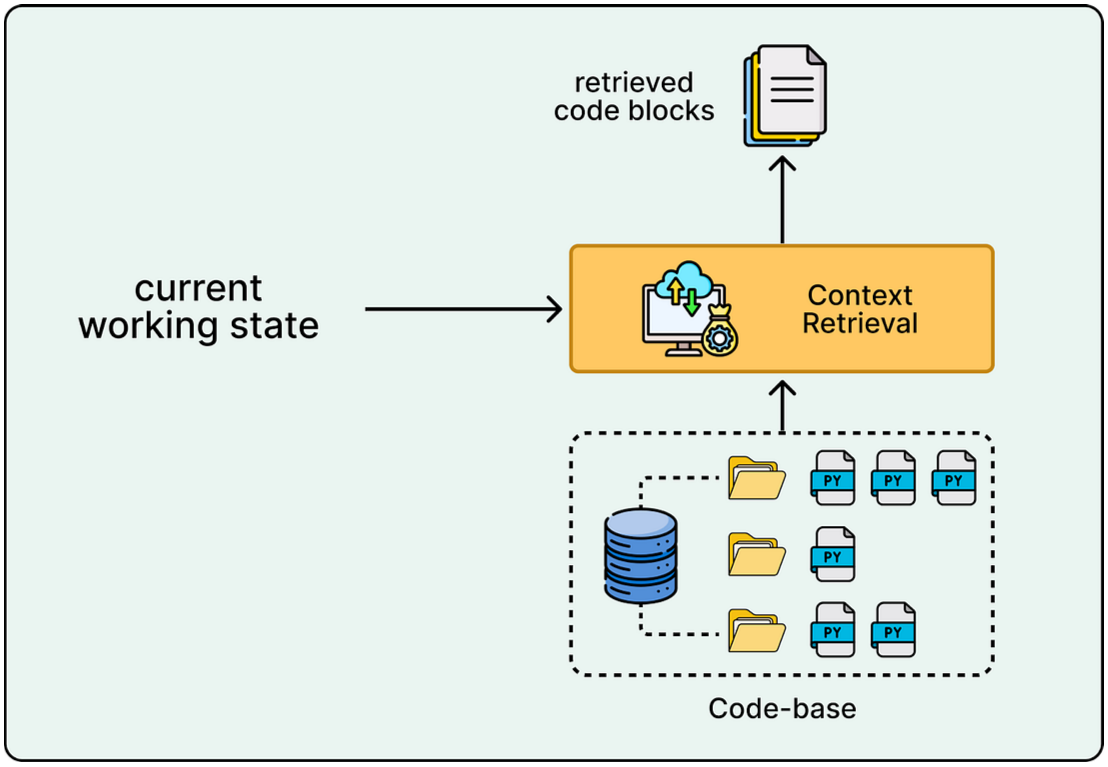
Tools

Composer is connected to a tool harness inside Cursor's agent system, with more than ten tools available. These tools cover the core operations needed for coding such as searching the codebase, reading and writing files, applying edits, and running terminal commands.



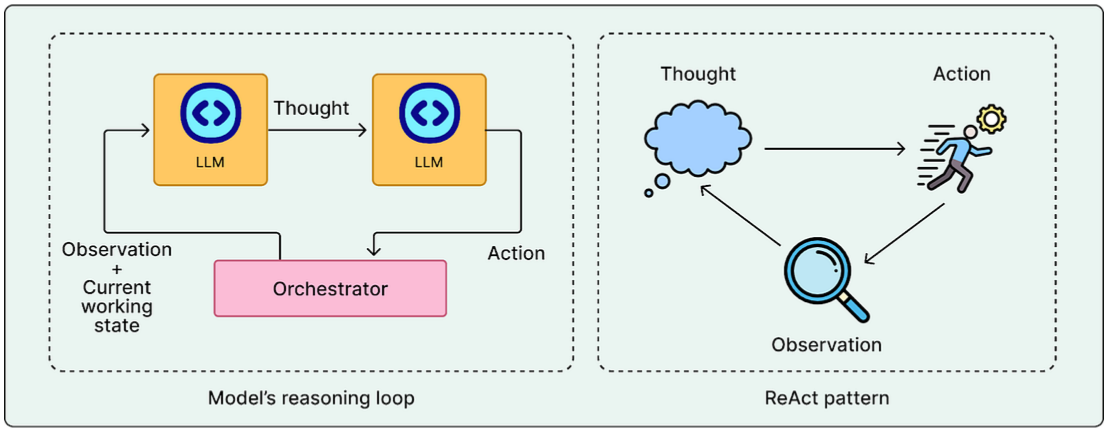
Context Retrieval

Real codebases are too large to fit into a single prompt. The context retrieval system searches the codebase to pull in the most relevant code snippets, documentation, and definitions for the current step, so the model has what it needs without overflowing the context window.



Orchestrator

The orchestrator is the control loop that runs the agent. The model decides what to do next and which tool to use, and the orchestrator executes that tool call, collects the result such as search hits, file contents, or test output, rebuilds the working context with the new information, and sends it back to the model for the next step. This iterative loop is what turns the system from a chatbot into an agent.



One common way to implement this loop is the ReAct pattern, where the model alternates between reasoning steps and tool actions based on the observations it receives.

Sandbox (execution environment)

Agents need to run builds, tests, linters, and scripts to verify their work. However, giving an AI unrestricted access to your terminal is a security risk. To solve this, tool calls are executed in a Sandbox. This secure and isolated environment uses strict guardrails to ensure that the user's host machine remains safe even if the agent attempts to run a destructive command. Cursor offers the flexibility to run these sandboxes either locally or remotely on a cloud virtual machine.

Note that these are the core building blocks you will see in most coding agents. Different labs may add more components on top, such as long-term memory, policy and safety layers, specialized planning modules, or collaboration features, depending on the capabilities they want to support.

Production Challenges

On paper, tools, memory, orchestration, routing, and sandboxing look like a straightforward blueprint. In production, the constraints are harsher. A model that can write good code is still useless if edits do not apply cleanly, if the system is too slow to iterate, or if verification is unsafe or too expensive to run frequently.

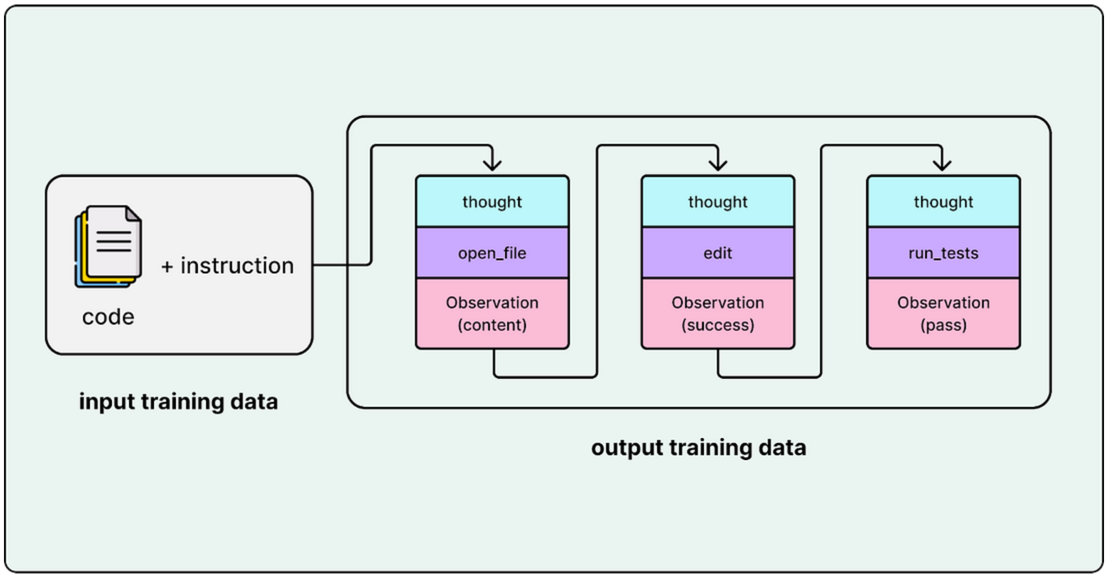
Cursor’s experience highlights three engineering hurdles that general-purpose models do not solve out of the box: reliable editing, compounded latency, and sandboxing at scale.

Challenge 1: The “Diff Problem”

General-purpose models are trained primarily to generate text. They struggle significantly when asked to perform edits on existing files.

This is known as the “Diff Problem.” When a model is asked to edit code, it has to locate the right lines, preserve indentation, and output a rigid diff format. If it hallucinates line numbers or drifts in formatting, the patch fails even when the underlying logic is correct. Worse, a patch can apply incorrectly, which is harder to detect and more expensive to clean up. In production, incorrect edits are often worse than no edits because they reduce trust and increase cleanup time.

A common way to mitigate the diff problem is to train on edit trajectories. For example, you can structure training data as triples like (original_code, edit_command, final_code), which teaches the model how an edit instruction should change the file while preserving everything else.



Another critical step is teaching the model to use specific editing tools such as search and replace. Cursor emphasized that these two tools were significantly harder to teach than other tools. To solve this, they ensured their training data contained a high volume of trajectories specifically focused on search and replace tool usage, forcing the model to over-learn the mechanical constraints of these operations. Cursor utilized a cluster of tens of thousands of GPUs to train the Composer model, ensuring these precise editing behaviors were fundamentally baked into the weights.

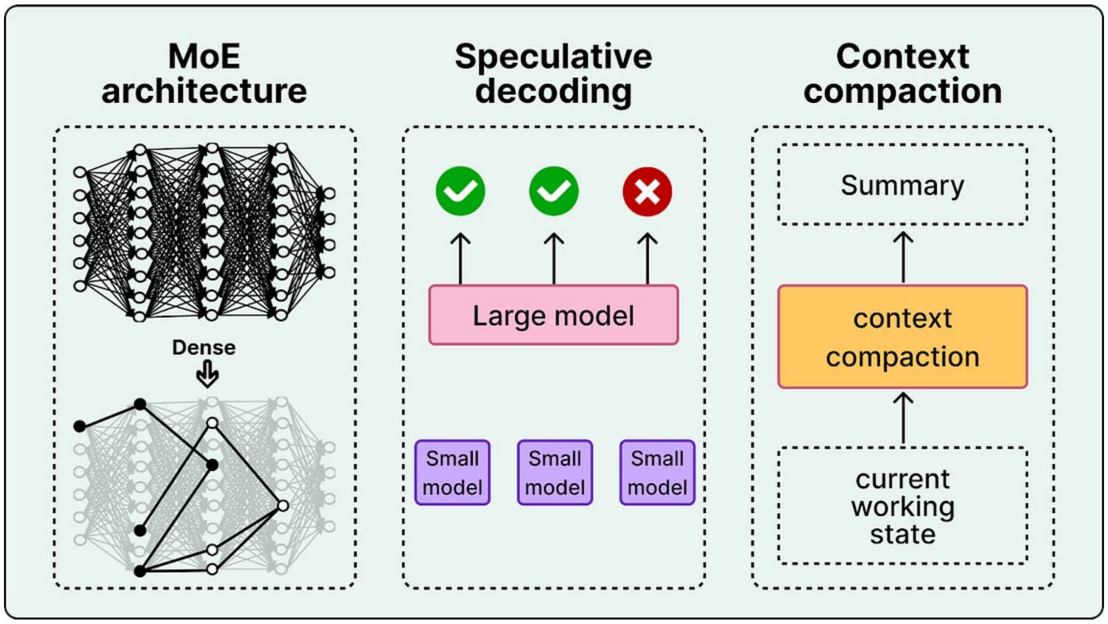
Challenge 2: Latency Compounds

In a chat interface, a user might tolerate a short pause. In an agent loop, latency compounds. A single task might require the agent to plan, search, edit, and test across many iterations. If each step takes a few seconds, the end-to-end time quickly becomes frustrating.

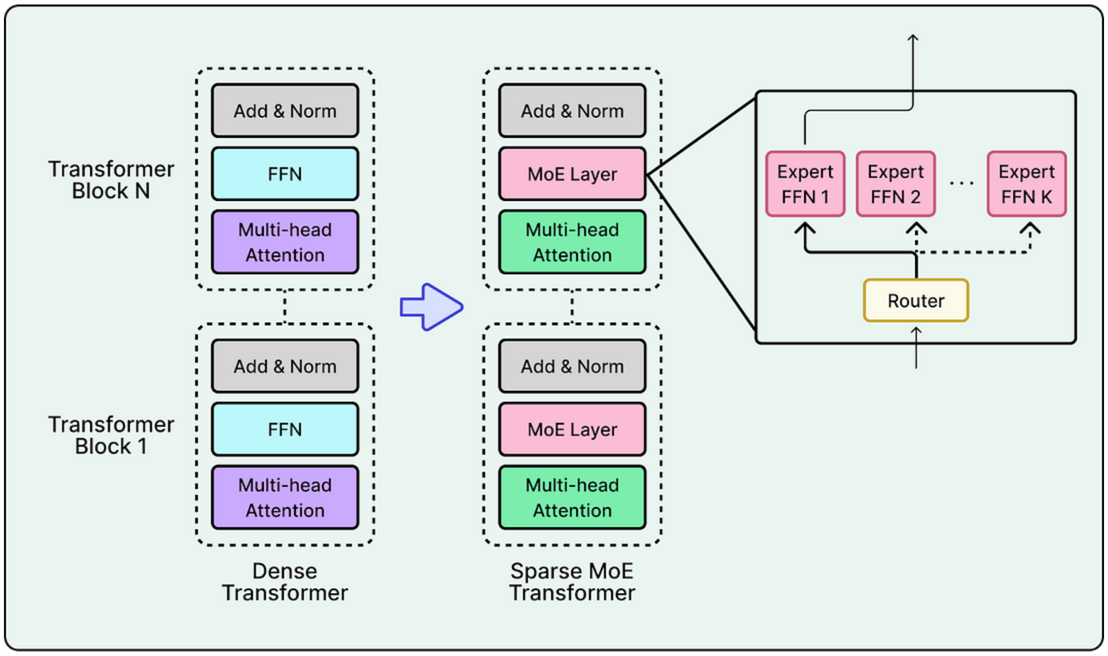
Cursor treats speed as a core product strategy. The make the coding agent fast, they have employed three key techniques:

- Mixture of Experts (MoE) architecture
- Speculative decoding

- Context compaction



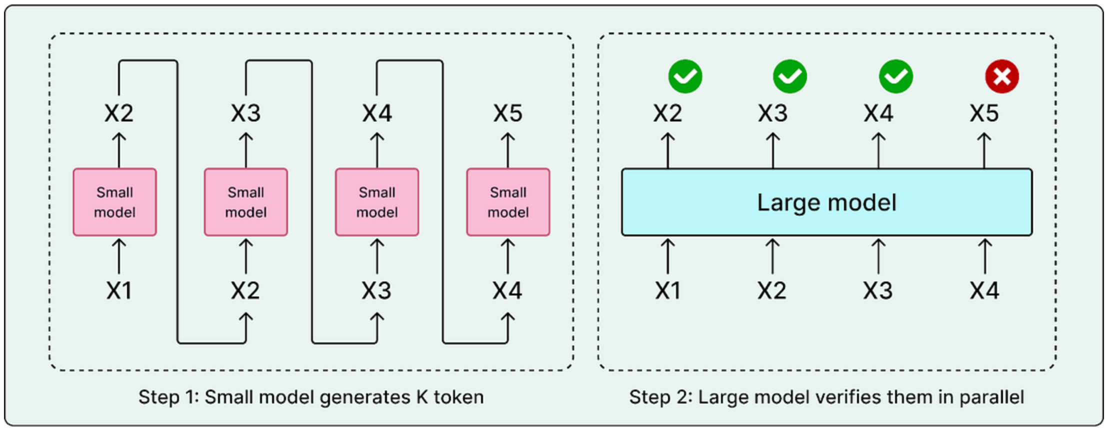
MoE architecture: Composer is a MoE language model. MoE modifies the Transformer by making some feed-forward computation conditional. Instead of sending every token through the same dense MLP, the model routes each token to a small number of specialized MLP experts.



MoE can improve both capacity and efficiency by activating only a few experts per token, which can yield better quality at similar latency, or similar quality at lower latency, especially at deployment scale. However, MoE often introduces additional engineering challenges and complexity. If every token goes to the same expert, that expert becomes a bottleneck while others sit idle. This causes high tail latency.

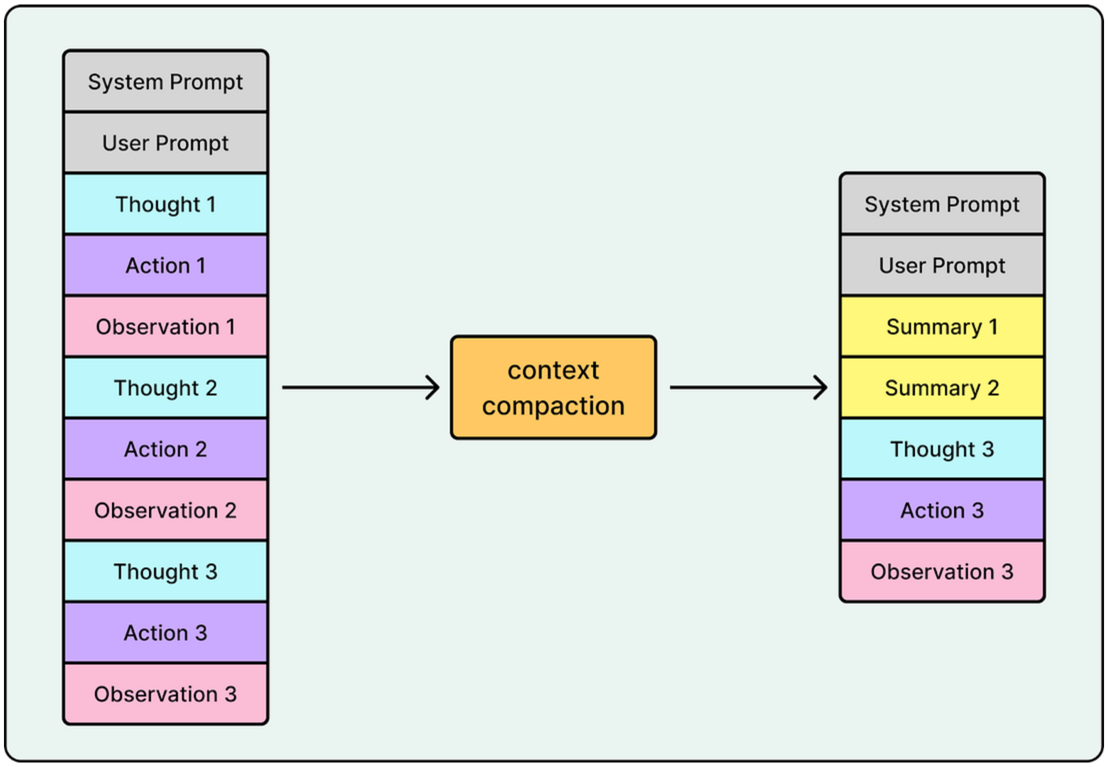
Teams typically address this with a combination of techniques. During training they add load-balancing losses to encourage the router to spread traffic across experts. During serving, they enforce capacity limits and reroute overflow. At the infrastructure level, they reduce cross-GPU communication overhead by batching and routing work to keep data movement predictable.

Speculative Decoding: Generation is sequential. Agents spend a lot of time producing plans, tool arguments, diffs, and explanations, and generating these token by token is slow. Speculative decoding reduces latency by using a smaller draft model to propose tokens that a larger model can verify quickly. When the draft is correct, the system accepts multiple tokens at once, reducing the number of expensive decoding steps.



Since code has a very predictable structure, such as imports, brackets, and standard syntax, waiting for a large model like Composer to generate every single character is inefficient. Cursor confirms they use speculative decoding and trained specialized “draft” models that predict the next few tokens rapidly. This allows Composer to generate code much faster than the standard token-by-token generation rate.

Context Compaction: Agents also generate a lot of text that is useful once but costly to keep around, such as tool outputs, logs, stack traces, intermediate diffs, and repeated snippets. If the system keeps appending everything, prompts bloat and latency increases.



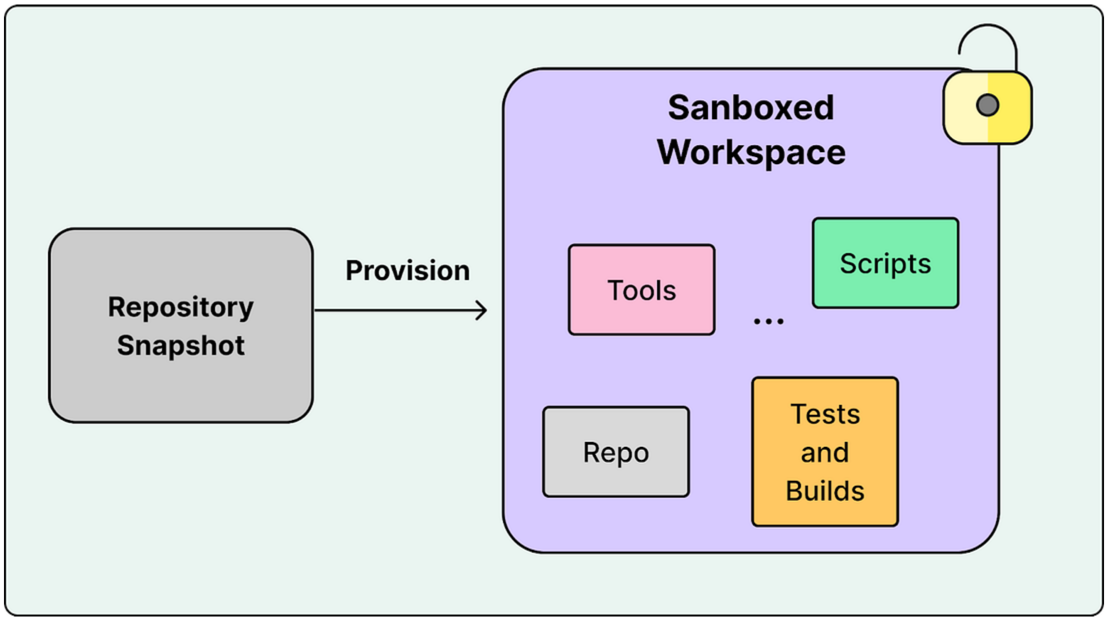
Context compaction addresses this by summarizing the working state and keeping only what is relevant for the next step. Instead of carrying full logs forward, the system retains stable signals like failing test names, error types, and key stack frames. It compresses or drops stale context, deduplicates repeated snippets, and keeps raw artifacts outside the prompt unless they are needed again. Many advanced coding agents like OpenAI's Codex and Cursor rely on context compaction to stay fast and reliable when reaching the context window limit.

Context compaction improves both latency and quality. Fewer tokens reduce compute per call, and less noise reduces the chance the model drifts or latches onto outdated information.

Put together, these three techniques target different sources of compounded latency. MoE reduces per-call serving cost, speculative decoding reduces generation time, and context compaction reduces repeated prompt processing.

Challenge 3: Sandboxing at Scale

Coding agents do not only generate text. They execute code. They run builds, tests, linters, formatters, and scripts as part of the core loop. That requires an execution environment that is isolated, resource-limited, and safe by default.



In Cursor’s flow, the agent provisions a sandboxed workspace from a specific repository snapshot, executes tool calls inside that workspace, and feeds results back into the model. At a small scale, sandboxing is mostly a safety feature. At large scale, it becomes a performance and infrastructure constraint.

Two major issues dominate when training the model:

- **Provisioning time becomes the bottleneck.** The model may generate a solution in milliseconds, but creating a secure, isolated environment can take much longer. If sandbox startup dominates, the system cannot iterate quickly enough to feel usable.
- **Concurrency makes startup overhead a bottleneck at scale.** Spinning up thousands of sandboxes all at once very quickly is

challenging. This becomes even more challenging during training. Teaching the model to call tools at scale requires running hundreds of thousands of concurrent sandboxed coding environments in the cloud.

These challenges pushed the Cursor team to build custom sandboxing infrastructure. They rewrote their VM scheduler to handle bursty demand, like when an agent needs to spin up thousands of sandboxes in a short time. Cursor treats sandboxes as core serving infrastructure, with an emphasis on fast provisioning and aggressive recycling so tool runs can start quickly and sandbox startup time does not dominate the time to a verified fix.

For safety, Cursor defaults to a restricted Sandbox Mode for agent terminal commands. Commands run in an isolated environment with network access blocked by default and filesystem access limited to the workspace and /tmp/. If a command fails because it needs broader access, the UI lets the user skip it or intentionally re-run it outside the sandbox.

The key takeaway is to not treat sandboxes as just containers. Treat them like a system that needs its own scheduler, capacity planning, and performance tuning.

Conclusion

Cursor shows that modern coding agents are not just better text generators. They are systems built to edit real repositories, run tools, and verify results. Cursor paired a specialized MoE model with a tool harness, latency-focused serving, and sandboxed execution so the agent can follow a practical loop: inspect the code, make a change, run checks, and iterate until the fix is verified.

Cursor’s experience shipping Composer to production points to three repeatable lessons that matter for most coding agents:

1. **Tool use must be baked into the model.** Prompting alone is not enough for reliable tool calling inside long loops. The model needs to learn tool usage as a core behavior, especially for editing operations like search and replace where small mistakes can break the edit.
2. **Adoption is the ultimate metric.** Offline benchmarks are useful, but a coding agent lives or dies by user trust. A single risky edit or broken build can stop users from relying on the tool, so evaluation has to reflect real usage and user acceptance.
3. **Speed is part of the product.** Latency shapes daily usage. You do not need a frontier model for every step. Routing smaller steps to fast models while reserving larger models for harder planning turns responsiveness into a core feature, not just an infrastructure metric.

Coding agents are still evolving, but the trend is promising. With rapid advances in model training and system engineering, we are moving toward a future where they become much faster and more effective.