ChatGPT

Architecture of Cursor: An AI-Enhanced VS Code Fork

Local Architecture (macOS)

Cursor runs as a standalone desktop application on macOS, built on a fork of **Visual Studio Code (VS Code)** [1]. This means the core editor UI, file system access, Git integration, and extension ecosystem are inherited from VS Code. Cursor adds custom processes and UI components on top of this base, such as an AI **Chat sidebar**, a **Composer** panel for multi-step prompts, and special keybindings (e.g. Tab for code completion, Cmd/Ctrl+K for on-demand code edits) [2]. Because it's a true fork (not just a plugin), these AI features are deeply woven into the editor – for example, inline AI autocompletions are integrated into VS Code's suggestion engine, and the chat assistant can modify files directly in the editor [3].

Under the hood, the Cursor app uses the standard VS Code process architecture (an Electron main process with renderer/extension-host processes). It runs language server processes for languages (Python, TS, Go, etc.) just like VS Code does, providing IntelliSense, diagnostics, "go to definition," and other IDE features [4]. Cursor creatively leverages these for AI: it introduces a concept called a **"shadow workspace."** When the AI needs to validate or test code changes, Cursor spins up a hidden, sandboxed VS Code window in the background (a duplicate of your workspace) [5] [6]. The AI's suggested edits are applied in this invisible window, and the language servers there compile/lint the changes. Any errors or warnings (e.g. undefined variables or type errors) are captured and fed back to the AI [5] [7]. This allows the AI to iteratively refine its suggestions **before** applying changes to your actual files, resulting in more correct code edits. The shadow workspace remains isolated (so it won't interfere with your active session) and is torn down after use, preserving the **independence** of the user's coding experience [6] [8]. In essence, the local Cursor client acts as a smart UI and sandbox: it gathers context (open file content, cursor location, etc.), manages a safe environment for AI-driven changes, and interfaces with the cloud for heavy AI computations.

Aside from the editor UI and sandboxing, most of Cursor's **computation-heavy tasks are not done locally**. The app will perform light preprocessing like scanning your project files and splitting code into chunks for indexing, and it ensures no secrets or ignored files are sent out [9] [10]. But notably, **there is no large AI model running on the Mac**. All the major AI functions (code embeddings, large language model queries, etc.) are handled by backend services (described below). The local app does maintain a small `.cursor` directory in the workspace to cache indexing metadata (like file hashes, index status) and uses VS Code's built-in Git integration to read your repository state. However, *all* AI suggestions (whether a chat answer, a code completion, or a refactoring command) are ultimately generated on Cursor's cloud – the client's role is to collect the prompt and context, and then display or apply the results.

Cloud Infrastructure and Services

Cursor's cloud backend is the AI powerhouse complementing the lightweight client. Whenever you invoke an AI feature from the editor, the Cursor app bundles up the necessary context (your prompt or query, plus relevant code snippets or file context) and sends it to Cursor's cloud API [11] [12]. There, a **server-side orchestration layer** takes over. This backend service (a mostly monolithic service with performance-critical components in Rust [13]) is responsible for constructing the final prompt, routing the request to the appropriate AI model(s), and post-processing the results before returning them to the client [11]. Even if you configure a personal OpenAI/Anthropic API key in Cursor, the requests still **flow through Cursor's servers** – this allows Cursor to insert system instructions, include code context from its indexes, and format the conversation properly around your query [14]. In practice, Cursor requires an internet connection and uses cloud compute for essentially all AI operations [12] [15] (there is currently **no fully-offline mode** for Cursor's core features [16]).

AI Models and Orchestration: Cursor utilizes a mix of third-party large language models and its own specialized models, orchestrated on the backend. For complex natural language conversations about code or very advanced tasks, Cursor will call top-tier "frontier" models like **OpenAI's GPT-4 or Anthropic's Claude** to maximize quality [17]. However, for faster responses (e.g. the real-time Tab completions) and automated code modifications, Cursor uses **custom-trained models** hosted on its servers [17] [18]. Notably, the Cursor team trained a code generation model nicknamed **"Copilot++"** (built on a 70B-parameter Llama2 base) to better predict code completions and edits [19]. They also developed a specialized **"Fast Apply"** model for rapidly applying large diffs or multi-file refactors [18]. This model was fine-tuned on Cursor-specific data (pairs of user edit instructions and the required code changes) and is optimized for high-throughput generation [18]. Both the Copilot++ and Fast Apply models run on Cursor's GPU servers via an inference engine called **Fireworks**, enabling extremely low-latency code generation – up to *1000 tokens per second* with techniques like speculative decoding [20]. An orchestration layer (sometimes referred to as "Anyrun" internally [21]) decides which model to use for a given request, manages prompt templates (e.g. injecting Cursor's custom system prompts [22]), and can even run multiple models in parallel for speed. For example, the backend might use the smaller Copilot++ model to draft a quick completion and concurrently fetch a more thorough suggestion from GPT-4, then return whichever finishes first that meets quality criteria – all transparent to the user, yielding fast yet accurate results [23].

The cloud infrastructure is quite substantial. Cursor's team operates on major cloud providers (AWS for core services and Azure for a large portion of GPU inferencing) and reportedly runs **tens of thousands of NVIDIA H100 GPUs** to serve users [24] [25]. The backend is designed to scale and handle many simultaneous requests (e.g. every keystroke may trigger an autocomplete request). Specific microservices handle different functions: for instance, Cursor has separate API endpoints for real-time **Tab completions** (optimized with HTTP/2 for streaming) and for **codebase indexing** operations [26]. There are also distinct service replicas for **Privacy Mode** users vs. standard users (to avoid any logging – see Security section) [27]. Ancillary infrastructure includes databases and streaming systems: Cursor uses a custom multi-tenant DB (Turbopuffer) to store encrypted file metadata and Merkle trees (for index state tracking) [28], a **vector database** (also backed by Turbopuffer or Pinecone in some cases) to store embeddings [29] [30], and Kafka-like queues (Warpstream) for handling background tasks and telemetry [31]. The web services and account management (billing, auth) are supported by standard tools (Stripe for billing, WorkOS for auth, etc.) [32] [33].

It's worth noting that Cursor's cloud not only runs the **model inference** but also much of the logic for features. For example, when you hit Tab for a completion, the client only sends a small context (the prompt prefix) to the server, which then uses the **Copilot++ model** to generate the next code tokens and streams them back for display [34] [35]. Similarly, a chat query about your project might trigger the backend to perform a semantic code search (using the index) and assemble a prompt with relevant code snippets before calling an LLM [36] [37]. All of these steps happen on the server side. The design choice here is clear: Cursor keeps the heavy lifting on the cloud, allowing the local editor to stay relatively lightweight and not require users to have powerful hardware. The trade-off is that Cursor is inherently a cloud service – it **cannot function without connecting to its backend**. (Enterprise users looking for fully self-hosted or offline solutions must look to other products, as discussed later.) [38]

Code Indexing and Intelligent Context

A key differentiator of Cursor is its ability to **understand and use your entire codebase** in AI-assisted tasks. This is enabled by Cursor's code indexing system, which creates semantic embeddings of your code and uses them for context retrieval (a technique known as **retrieval-augmented generation**). Here's how it works:

- **Initial Indexing:** When you first open a project in Cursor, the client begins **scanning and indexing the whole repository in the background** [39]. The codebase is broken into small *chunks* (on the order of a few hundred tokens each) for embedding. Importantly, Cursor does not just split files arbitrarily – it uses intelligent chunking strategies (leveraging parsers like *Tree-sitter* to split at function or class boundaries) so that each chunk is a coherent code block [40]. This way, a retrieved chunk will contain a logically self-contained snippet (e.g. a full function implementation) which is more useful to the AI than a random fragment [41]. Chunking is done locally on your machine [42], and Cursor also **obfuscates file paths and encrypts code chunks** before sending them out, as an added privacy measure [30]. Each chunk (with metadata like file reference and line numbers) is then sent to Cursor's server, which generates an **embedding vector** for it using either OpenAI's embedding API or Cursor's own embedding model [43] [30]. The numeric embedding is stored in a **vector database** in the cloud (along with minimal metadata like a hash or an encrypted identifier for the file) [44]. **No raw source code or file names are stored** on the server during this process – the original code text exists only on your machine and in transient memory during embedding generation [30]. (If Privacy Mode is on, the server also discards each chunk immediately after embedding it, retaining only the vectors [45].)

- **Semantic Search (RAG):** Once indexing is complete, Cursor's AI features can tap into these embeddings to answer questions about your codebase. For example, if you ask in the chat, *"Where do we configure logging in this app?"*, the backend will take your query, convert it into an embedding vector, and query the vector DB for similar semantic content [46]. It might retrieve several relevant chunks – e.g. a snippet from `logging.py` where a logger is set up, and another from `app.js` where a logging level is defined. These top-matching code snippets are then **fetched from your local files** (the server requests the actual text of those files from the client once it knows which parts it needs [36]) and incorporated into the prompt sent to the LLM [47] [48]. In essence, the AI assistant can "quote" your codebase beyond what fits in the model's fixed token window by dynamically retrieving the most relevant bits. This **semantic code search** allows the Cursor chat to answer codebase-wide questions or implement changes using knowledge of the whole project, not just the open file. It's like having a smart code search engine integrated with the AI: the assistant can reference any part of your repository (functions, classes, comments, etc.) as needed, without you explicitly opening those files [49]. This retrieval-augmented approach is how Cursor achieves "whole-project awareness" in practice [50].

- **Continuous Index Maintenance:** Codebases aren't static – you'll be editing code, adding new files, switching git branches, etc. Cursor's indexer is designed to keep up-to-date efficiently. The client and server both maintain a **Merkle tree** of the project's file structure, where each file and directory is represented by a hash of its contents [51] [52]. Every few minutes (currently, a 3-minute interval), the client computes an updated Merkle root and securely compares it with the server's index state [53] [54]. If differences are found, Cursor can pinpoint exactly which files changed (e.g. file X's hash differs, meaning it was edited) and will re-index only those files. This approach minimizes unnecessary re-embedding work and bandwidth – if you pull a git update that changes 5 files, Cursor will re-upload just those 5 files' chunks for embedding, rather than reprocessing the entire repo [55] [56]. It also means after a overnight coding session or a large merge, Cursor's index catches up with minimal delay, ensuring the AI's answers reflect the latest code. The Merkle tree and other indexing metadata are stored encrypted on the server (via Turbopuffer DB) and locally in the `.cursor` folder [28] [57].

- **Integration with LSP and ASTs:** Aside from pure vector search, Cursor augments its context gathering with traditional static analysis. The client can query language servers or parse ASTs to get definitions, type signatures, or documentation for symbols that appear in your prompt [58]. For instance, if you ask "Why is function `foo()` throwing an error?", Cursor might automatically include the definition of `foo()` (retrieved via *"go to definition"* in the language server) in the prompt it sends to the model [59]. This hybrid approach – combining AI with tooling – improves accuracy, since the model doesn't have to infer what `foo()` is doing; it's explicitly given the relevant code. Similarly, Cursor's **@-commands** (like typing "`@File`" or "`@Symbol`" in a prompt) let you pull in specific file contents or symbols quickly; the client recognizes these and replaces them with the appropriate code text before sending the request.

Overall, Cursor's indexing and context mechanism is akin to what Sourcegraph Cody or similar tools do, but built into the IDE experience. It gives the AI assistant a memory of your entire project. One practical example is **refactoring**: you can ask Cursor in chat "Rename the `AuthService` class to `UserAuthService` everywhere." Cursor will use the index to find all references of `AuthService` across files, and then orchestrate the changes (likely with an AI-generated diff for each occurrence). Because it knows where things are, it can plan multi-file edits reliably. This goes hand-in-hand with Cursor's **agent mode** (Composer), where the AI can autonomously perform multi-step changes: the agent might search the codebase, open files, apply edits, even run your build/tests to verify – acting like a junior developer working on your repo [60] [61]. (By contrast, a simpler tool like Copilot has no knowledge of other files – it only knows the open file, so it can't safely refactor project-wide).

Indexing Performance: Building the initial index is computationally expensive (it involves running many embedding predictions). Cursor's backend uses GPUs to accelerate this, and in practice indexing is fairly fast: on a mid-sized project it completes in under a minute, though very large monorepos can take many minutes (Cursor will show indexing progress in the UI) [62] [63]. To avoid overwhelming resources, Cursor lets you configure which directories to index or exclude (via a `.cursorignore` file, similar to `.gitignore`) [64]. For extremely large codebases, you might exclude third-party libraries or generated files to focus on the important code. If indexing is disabled entirely (which you can do in settings), the chat fallback is a standard keyword search (BM25) over your files [65] – which is slower and less accurate than embeddings. In normal operation, though, **indexing is central to Cursor's code intelligence**. It's what enables features like "Ask Cursor" to actually fetch answers from your code, rather than just guessing from model training.

Security and Privacy Considerations

Because Cursor processes private source code, security and privacy are critical. The architecture is designed so that **user code is not persistently stored on Cursor's servers**, and users have control over what is shared. Here are the key points:

- **Code Transmission and Storage:** By default, Cursor **does upload code snippets to its cloud** (for indexing and for prompt context), but it tries to minimize what is sent and never stores raw source text at rest. During codebase indexing, file contents are chunked locally and **sent in memory to the server for embedding**, then immediately discarded – only the numeric embeddings and minimal metadata are stored in the vector database [43]. The stored embeddings cannot be reverse-engineered to fully reconstruct the original code (they're essentially high-dimensional abstractions) [66]. Moreover, Cursor avoids even storing file names in plaintext: as noted, filenames may be obfuscated (hashed) before sending to cloud indexing [30]. For on-the-fly AI requests (completions, chat), the relevant code context is included in the prompt sent to the model, but those prompts are not logged or saved on disk by Cursor's servers when privacy protections are in place [67] [29]. In transit, all data is encrypted (Cursor uses HTTPS/TLS for communication, and even encrypts code payloads at the application layer for extra safety in some cases [68]).

- **Privacy Mode:** Cursor provides a **Privacy Mode** (also called "local mode" in early discussions) that users can enable in settings (it's enabled by default for enterprise team accounts) [67]. When Privacy Mode is **ON**, Cursor guarantees that none of your code data is stored or used for training AI models [67]. In practice, requests from privacy-mode users are tagged with a header, and the backend routes them to isolated **"ghost" service replicas** that have logging disabled [27]. In these privacy-dedicated servers, any operation that would log or persist data is either turned off or carefully scrubbed. The architecture even has parallel processing queues for privacy mode vs. normal, to avoid any mix-up [27]. Essentially, if Privacy Mode is on, your code is treated as ephemeral: the servers use it to generate a response and then immediately throw it away. No telemetry will contain your code (and any analytics that is collected – like usage counts – is severed from any code content). This provides assurance against accidental data retention. *More than half of all Cursor users enable privacy mode*, reflecting how important this guarantee is [69]. (One caveat: if privacy mode is off, Cursor may log **anonymized** prompt data or keep embeddings cached to improve the service – but even then they state raw code isn't stored long-term and never used in training without consent [29].)

- **On-Prem and Self-Hosting:** Currently, Cursor does **not offer a self-hosted or fully offline deployment** for end-users [70]. All requests eventually go to Cursor's cloud on AWS. If a company requires that source code never leave their network, Cursor's cloud approach may be a blocker. (Competitors like Codeium/Windsurf and Sourcegraph Cody provide on-prem options where models run on your own servers [71] [72], but Cursor at this time is cloud-only). To mitigate this, Cursor emphasizes its privacy mode and the fact that embeddings are not reversible. Still, organizations handling extremely sensitive code need to weigh this. Cursor's team has obtained SOC 2 Type II certification and undergoes third-party security audits [73], but it's not an air-gapped solution.

- **Data Handling and Scrubbing:** Cursor **respects `.gitignore` files** by default – any file ignored by git (e.g. secrets in `.env` files) will not be indexed or sent to the AI [74]. They also support a `.cursorignore` to explicitly exclude certain files or folders from AI indexing [74]. Furthermore, the client has a built-in **heuristic secrets scanner**: it detects things like API keys, credentials, or tokens in your code, and will **block them from ever being sent** to the cloud [9]. This applies to both code indexing and ad-hoc chat/completion requests. For example, if you accidentally prompt, "Explain what this AWS key does: `<ACCESS_KEY>`," Cursor will scrub or redact the key before formulating the request. This is an important safeguard to prevent leaking sensitive tokens.

- **Use of Third-Party LLMs:** When Cursor uses third-party models (OpenAI, Anthropic, etc.), it routes the requests through its backend and **does not permit those providers to store the data**. For OpenAI, it uses the API policy where data isn't used for training by default. The Privacy Mode header also ensures Cursor's own provider integrations know the request is sensitive. According to Cursor's security page, they guarantee that code data is never retained by their model providers either when privacy mode is on [67]. (In fact, many of Cursor's requests go to their *own* models running on infrastructure via Fireworks, so in those cases data doesn't leave Cursor's control at all [75].)

- **Client Security:** Since Cursor is effectively a VS Code fork, the company tracks upstream VS Code security patches. They merge upstream code regularly (every couple of VS Code releases) and will fast-forward critical security fixes as needed [76]. Some security features of VS Code, like **Workspace Trust**, are turned off by default in Cursor (to avoid confusing users with Cursor's Privacy Mode) [77]. Users can enable Workspace Trust if desired, but by default Cursor assumes you trust your folders (since it's anyway sending code to the cloud). Another difference is that currently Cursor doesn't enforce extension signing – the setting to require signed extensions is off by default (a feature VS Code added for marketplace security) [78]. This is something they plan to support in the future. So, users should be mindful when installing third-party VS Code extensions into Cursor, as those could run arbitrary code (the same risk as vanilla VS Code, except VS Code now verifies publisher signatures).

In summary, Cursor's architecture attempts to **minimize the risks of cloud AI** by not persistently storing code and by giving users a Privacy Mode that routes requests through a "no-log" pipeline [79]. Sensitive data is filtered out locally, and standard network security (TLS) is used for all communication. The main security consideration is that your code *does* transit to remote servers and potentially to model providers – so you are trusting Cursor (and its subprocessors) with ephemeral access to your code. They have taken steps like encryption, obfuscation, SOC2 compliance, and isolation of privacy-mode data flows to earn that trust. Still, for some highly regulated environments, any cloud involvement may be a concern – in those cases, teams might look to on-premises alternatives or limit Cursor's use to non-sensitive portions of the work.

Competitive Analysis

AI-enhanced coding tools have proliferated. Here we compare Cursor's architecture and approach with other notable AI coding assistants: **GitHub Copilot**, **Sourcegraph Cody**, **Codeium/Windsurf**, and **Zed**. The focus is on how they differ in integration, AI backends, indexing capabilities, security, and extensibility.

Cursor vs. GitHub Copilot (in VS Code)

GitHub Copilot is delivered as an extension that plugs into editors (VS Code, JetBrains, etc.), whereas Cursor is a full **standalone IDE**. Philosophically, *"Copilot enhances whatever coding setup you already love, while Cursor wants to become your entire development environment."* [80] In practical terms, Copilot's architecture is simpler: it's essentially a client-side plugin that calls out to OpenAI's Codex/GPT model for suggestions. There is **no project-wide indexing** with Copilot – it operates on immediate context. Copilot's VS Code extension will send the content of your current file (and perhaps a few open tabs or recently edited lines) to the cloud model to generate an autocompletion. It does not scan your whole repository or build a semantic search index. This "suggest-first" approach (as one analysis puts it) works well for boilerplate and single-function tasks [81], but it means Copilot lacks awareness of broader codebase structure [82] [83]. For instance, Copilot cannot answer a question like "find all usages of function X across the codebase" – it wouldn't know where else X is defined or used without the user manually opening those files. In contrast, Cursor's "search-first" architecture indexes the entire codebase and can retrieve relevant snippets before generation [81] [84], enabling it to handle queries about multiple files or perform coordinated changes across a project.

On the AI backend, Copilot uses **OpenAI's models (Codex and now GPT-4)** via GitHub's servers. The Copilot extension sends requests to GitHub's cloud, which then communicates with OpenAI; all inference is done remotely on OpenAI's GPUs. Cursor similarly relies on cloud models, but it uses a *mix*: part OpenAI/ Anthropic and part its **own models** (running on its infrastructure). This gives Cursor more flexibility in tuning for performance – e.g. Cursor's custom models can stream results faster (Copilot's latency is generally tied to OpenAI's speed). Both tools do stream token-by-token to the editor for fast feedback. Copilot recently introduced a Chat mode and an "experimental" code navigation where it can use Microsoft's search (or perhaps the new GPT-4 32k context) to access more files, but generally Copilot's knowledge beyond the current file is limited. Microsoft is integrating Copilot deeply into DevOps (documentation, pull requests, etc.), but within the editor, it's a fairly **lightweight integration** (which some prefer for its simplicity).

Security-wise, Copilot sends your code to the cloud just like Cursor, and there is no privacy toggle – you rely on GitHub's policies. Enterprise users of Copilot can opt for *"Copilot for Business,"* which ensures OpenAI doesn't retain their code for training. But the fundamental model is cloud-based and closed-source. There's no self-host option for Copilot (Microsoft handles everything on Azure/OpenAI). This is one reason some companies opt for Cursor or others – they might prefer a vendor that offers more control or at least explicit no-log modes. An article noted that *Copilot's architecture requires cloud processing* and has prompted concerns by reproducing licensed code or secrets from training data [72], whereas tools like Cody (and Cursor) try to mitigate that by grounding responses in your actual codebase rather than just the training corpus.

In terms of **UX and extensibility**, Copilot is minimal: it offers inline autocomplete and a chat Q&A interface in VS Code. It doesn't expose a plugin system to end-users for adding new AI "tools." Cursor, on the other hand, provides the **Model Context Protocol (MCP)** which lets developers extend the AI's capabilities (e.g. add a plugin so the AI can query your database or an internal API during a session) [85]. This is analogous to OpenAI plugin tools but specific to Cursor's environment. Copilot currently has no such feature in the IDE (though Microsoft is adding "ChatGPT plugins" into Bing and Windows Copilot; those are separate from coding workflow). Another difference: because Cursor is a full VS Code fork, it can integrate things like the "Play" button to apply AI changes, or the shadow workspace mechanism for error checking – these are features beyond Copilot's scope as an add-on. Copilot's advantage, however, is **breadth and familiarity**: it works in many IDEs and is maintained by Microsoft/GitHub, so it's deeply integrated into the VS Code ecosystem (e.g. it respects VS Code settings, remote dev scenarios, etc.). With Cursor, you have to use their custom IDE (which may lag a tad behind official VS Code releases, though they keep up fairly closely). Teams that use a variety of IDEs (some VSCode, some IntelliJ, etc.) might choose Copilot since it's available in multiple editors, whereas Cursor is VSCode-only (and a custom version at that).

Cursor vs. Sourcegraph Cody

Sourcegraph's **Cody** is another AI coding assistant that, like Cursor, emphasizes whole-codebase understanding. Architecturally, Cody is usually accessed via an editor extension (for VS Code or other IDEs) that connects to a Sourcegraph backend. Sourcegraph is essentially a code search engine and indexer; with Cody, it builds embeddings of your repositories similar to Cursor's indexing. However, the big difference is Cody is often run as a **self-hosted service** (especially for enterprises). A company might run a Sourcegraph cluster that indexes all their repos and hosts an LLM either via Sourcegraph's cloud or on-prem. When you ask Cody a question, it performs a **semantic search** across your codebase on the Sourcegraph server, finds relevant snippets, and then calls an LLM (OpenAI's GPT-4 or Anthropic Claude, or even an open-source model) to formulate an answer [82] [86]. The philosophy, as one blog put it, is *"Copilot is suggest-first, while Cody is search-first."* [81] In other words, Cody will fetch explicit context from potentially thousands of files before answering, which makes it very powerful for large, complex codebases (monorepos, multi-repo microservices, etc.). Cursor actually follows a similar approach with its RAG system – the difference is Cursor's index exists for a **single project locally** (scoped to the folder you opened in the Cursor app), whereas Sourcegraph can index your entire organization's code (multiple repos) and Cody can draw upon any of them if asked.

Because Cody's heavy lifting is done by Sourcegraph, one trade-off is that it might require more setup (you need a Sourcegraph deployment or their cloud account). Cursor's setup is simpler (it indexes on the fly when you open a project). Performance-wise, both try to cache and do incremental indexing. Sourcegraph's search index (and associated embeddings) persist on the server, so it can handle even enormous codebases with proper resources. In fact, Sourcegraph's underlying tech is designed for scale – tens of millions of lines – where it does aggressive sharding of search indices. Cursor, being an editor-first tool, may start to **strain on extremely large repos** (some users note memory or performance issues indexing huge monorepos [87]). The Cursor team has addressed this with ignore files and focusing on changed files via Merkle trees, but Sourcegraph might have an edge for truly enterprise-scale code (since it's a dedicated indexing server). On the flip side, Sourcegraph's integration in the editor is via an extension communicating with a remote service, which can introduce latency. Cursor's architecture, by tightly coupling the index with the IDE, can feel a bit more seamless for an individual developer on a single project.

For **AI models**, Cody originally leveraged OpenAI's models (like GPT-4) and especially **Anthropic's Claude** (which has a 100k token context, beneficial for stuffing a lot of code context). They have also introduced their own model ("OpenAI o1" which is a tuned GPT-3.5 variant) [88]. But importantly, Sourcegraph allows you to configure models – you could plug in an open-source LLM or use Azure OpenAI, etc. This configurability, plus the option to self-host, means **Cody can be deployed fully on-premises**. A security-conscious enterprise can run Sourcegraph and the LLM behind their firewall, so no code leaves their environment. **Cursor does not yet offer such an on-prem solution** [38] – you must use Anysphere's cloud. For companies handling very proprietary code that can't even be transiently processed by a third-party, Cody is often a go-to (or an internal tool built on Sourcegraph's APIs).

In terms of **UX**, Cursor and Cody both provide a chat interface in the IDE and the ability to get inline completions. Cody's VS Code extension adds a sidebar for chat, and you can ask it questions or tell it to rewrite code. Cursor's chat is similar but arguably more feature-rich – e.g. Cursor has the "instant apply" button to apply changes from chat with one click, and the separate Composer/agent panel for multi-step workflows. Cody's interface is improving (they recently added "Fix up" to apply changes and a CLI tool), but since Cursor controls the whole IDE, it can offer a more integrated feel (like the **@ references** in prompts, or showing diffs with “Apply” buttons). Both support **multi-file edits**, but approach it differently: Cursor’s agent will plan and edit files sequentially within the IDE, while Cody might leverage Sourcegraph’s knowledge to script changes. For example, to refactor a function used in 10 files, Cursor’s agent will use its index to find those and then edit them one by one (asking for your confirmation). Cody might present you with a diff covering all occurrences, generated in one go by the LLM using Sourcegraph context.

Extensibility: Cursor’s MCP system is somewhat unique; Cody doesn’t have a user-facing plugin mechanism. However, Sourcegraph as a platform can integrate with other data sources (for instance, it can index your internal documentation or wikis, and Cody can then pull that in if you ask it). Cursor’s approach might let the AI call external tools during a session (like run tests, or query an API) via MCP, whereas Cody for now is focused on code and docs Q&A.

In summary, **Cursor vs Cody** often comes down to scope: Cursor is great for an individual repository and developer, giving a very fluid in-IDE experience with full context of that project. Cody is great for large-scale, multi-repo scenarios and companies needing self-hosted control. They share the idea of semantic indexing, but implement it differently (IDE-integrated vs. server-integrated). It’s not uncommon for enterprises to use Sourcegraph Cody for organization-wide code intelligence, and developers to use Cursor for day-to-day work on certain projects – the tools can complement each other, though they do overlap heavily.

Cursor vs. Codeium / Windsurf

Codeium is another code completion and chat tool originally delivered as a free plugin for various IDEs. Recently, Codeium introduced **Windsurf**, which is an AI-first IDE very much like Cursor (in fact, the rivalry is notable – Windsurf is often compared to Cursor as both are “AI-powered VS Code-esque editors”). Windsurf was reportedly built by the Codeium team and shares a lot of its backend. In architecture, Codeium’s plugin and Windsurf’s app mirror the Copilot vs Cursor pattern: Codeium’s IDE plugins are analogous to Copilot (providing completions within your existing editor by calling cloud APIs), whereas Windsurf is a full editor (Electron-based) that can integrate AI more deeply.

A few key differentiators emerged: - **Local vs Cloud Indexing:** By default, Cursor’s code indexing uploads code chunks to the cloud for embedding (though it only stores embeddings, not raw code) [66] . Windsurf, in contrast, performs **local embedding/indexing by default** – your machine computes the embeddings of the codebase locally, keeping code entirely on your machine for that step [89] . (It likely uses a smaller embedding model or some efficient local inference for this purpose.) This means out-of the box Windsurf might appeal to those worried about code leaving their machine. However, Windsurf/Codeium also offers a cloud indexing mode for enterprises: an org can run a private indexing server to centrally index repos and share embeddings team-wide [89] . Cursor currently does not have a notion of team-shared indexes – each dev indexes their repo in their client, and nothing is shared unless code is in a remote git host that Cursor could theoretically access (which it doesn’t, it only indexes what you open locally).

- **On-Prem Deployment:** Codeium is very focused on enterprise options. They offer a fully on-prem solution where a company can deploy Codeium’s models on their own servers (or a VPC) so that code never leaves their network [71] . Windsurf, as an editor, can be configured to use that on-prem endpoint. This is a big selling point for Codeium in enterprise. Cursor **does not offer on-prem** – its cloud is SaaS-only, as mentioned [38] . So architecture-wise, Windsurf can operate in a cloud mode (default, using Codeium’s cloud service similar to Cursor’s cloud) or an offline mode (pointed at a self-hosted Codeium server). If an organization requires an air-gapped environment, Windsurf/ Codeium clearly has the advantage.

- **AI Model Backend:** Codeium has its own in-house **AI code models** (trained on large code corpora). These models power their completions and chat. Windsurf uses these models for things like Tab completions – Codeium's models are optimized in-house for low-latency suggestion generation [90]. For more complex tasks, Codeium can also call OpenAI or other models, but a lot of focus is on their proprietary model (which is how they offer a free tier – they're not paying OpenAI for every request). Cursor, as detailed, uses a combination of licensed models and its own. One interesting note: OpenAI recently announced intent to acquire Codeium's parent (Windsurf) for $3B [91], showing the strategic value of having an in-house model. From a user perspective, both Cursor and Codeium/Windsurf aim to give fast completions. Codeium's models are competitive but some evaluations show they might be slightly behind GPT-4 in quality. Cursor's custom model claims to exceed GPT-4 in certain edit tasks by being domain-focused [92]. Both companies are rapidly iterating their models.

- **Features and UX:** Cursor and Windsurf, being direct competitors, have very similar feature sets: chat assistant, inline completions, the ability to apply multi-file edits via natural language, and even "agent" modes. Windsurf's AI assistant is called **Cascade**, which is described as having an "agentic IDE" philosophy (it can perform multi-step flows autonomously, similar to Cursor's agent in Composer) [93] [94]. One difference noted in a comparison is that Windsurf doesn't make the user explicitly toggle between chat vs agent mode – you ask Cascade in chat and it will take autonomous actions as needed, whereas Cursor often lets you decide when to invoke the more autonomous Agent mode (Cursor's Composer) vs. just Q&A [95] [96]. This is a subtle UX choice. Windsurf tries to **streamline the agent behavior**, while Cursor gives a bit more manual control (the user can choose to run an agent to perform a refactor, or just ask for help and then apply changes themselves). Neither is necessarily better; it's about user preference.

- **Extensibility:** Codeium/Windsurf doesn't have an equivalent to Cursor's MCP at the moment. However, Codeium has stated they integrate with certain internal data sources for enterprise (e.g. they can ingest internal documentation or connect to issue trackers on the backend). Cursor's MCP is a more general plugin framework. Also, because Windsurf is newer, its ecosystem (extensions, etc.) is not as large. Cursor can use VS Code extensions (except those requiring MS's proprietary stuff), whereas Windsurf likely can too if it's a fork – but it might not have full marketplace access. (Details are sparse, but since Cursor is explicitly a VS Code fork, they enable extension installs; Windsurf's basis is presumably VS Code as well given it's an IDE, but it's not confirmed publicly how it handles extensions).

In summary, **Cursor vs Codeium/Windsurf** is a close matchup. Cursor's strengths are a polished integration and its use of multiple top-tier models (often yielding very high-quality results with GPT-4 when needed). Windsurf's strengths are flexibility in deployment (local vs cloud, on-prem) and the fact that it's backed by Codeium's free usage model (for individual devs, Codeium is currently free, whereas Cursor's advanced features are paid). From an architecture standpoint, if you require that embeddings be done locally and no code ever touches a third-party cloud, Windsurf can fulfill that (with some setup) [97] [38], while Cursor cannot – it always sends to its cloud for embedding and LLM calls [12] [98]. If you want the absolute best AI model available for reasoning (GPT-4, as of 2025) tightly integrated, Cursor's approach of orchestration might yield better answers, since Codeium's own model, while good, might occasionally lag in quality for very complex code understanding tasks.

Finally, it's worth noting that both Cursor and Windsurf are evolving rapidly, and given the competitive pressure, we are seeing convergence of features (each inspiring the other). Developers and teams might try both to see which fits their needs and preferences; many report that Cursor feels slightly more "intelligent" with whole-project Q&A, while Codeium/Windsurf feels very fast and more private by default [90] [97].

Cursor vs. Zed AI

Zed is a modern code editor (written in Rust) known for its performance and collaborative features. Zed has an AI assistant feature, but its design philosophy is quite different from Cursor's. Zed's approach to AI is to treat the AI assistant as a collaborator that's fully integrated into the editing experience, but with maximum transparency and user control. In practice, Zed does **not pre-index your entire codebase** for you. Instead, it provides commands to pull in context as needed. Zed's assistant runs on a hosted service powered by **Anthropic's Claude 3.5** model [99]. When you open the **AI Assistant panel** in Zed, it essentially gives you a blank text area where you construct a prompt. Zed exposes the entire prompt that will be sent to the LLM – including system instructions and any included code – for you to review and edit [100]. You can insert content via **slash commands**: e.g. `/file` to include the contents of a file (or even a whole directory tree of files), `/tab` to include whatever is open in your current tab, `/diagnostics` to pull in error messages, etc. [101]. These commands allow a sort of manual retrieval: *you* choose which files or outputs to feed to the AI. Zed will fold the inserted content in the prompt (so you can collapse large sections) but it's all visible to you [102] [103]. This means **Zed doesn't autonomously decide context**; the developer curates it (with helpful shortcuts). It's a very transparent workflow, favored by developers who want fine-grained control over what the AI sees.

Because of this, Zed's architecture doesn't need to embed every file upfront or run a vector search. If you want to ask about a function, you might literally do `/file src/util.py` to include it in the prompt, or use their fuzzy finder to quickly pull relevant code. Zed's creators (which include folks who made Tree-sitter and Atom) have focused on **low-latency and streaming**. When you send a prompt, Claude generates answers which Zed streams back token-by-token. Zed implemented a custom **streaming diff** display for code edits: if you ask it to transform code, it can show you a live diff that updates as the AI streams out changes [104]. This is enabled by Zed's underlying CRDT-based document model, which is very fast at applying edits. Cursor also streams responses, but Zed's UX might feel even more real-time due to optimizations like Anthropic's *prompt caching* (Zed worked with Anthropic to cache large contexts so you don't resend unchanged code on each request) [105].

Integration: Zed's AI panel is effectively an advanced prompt console. It doesn't have a chat history the way Cursor's does (you manage the text in the panel as you see fit). Zed also allows **inline transformations**: you can select code and press a shortcut to have the AI transform it in place (similar to Cursor's Cmd+K, but Zed streams the transformation live) [106]. Zed's emphasis on being **lightweight and local-first** shows in that the editor remains extremely fast and you pull in AI assistance as needed. However, Zed's AI cannot on its own figure out where in a large project something is defined unless you explicitly provide that context. It's less of an all-knowing assistant and more of a powerful text manipulation tool under your guidance.

In terms of **security**, Zed's model (Claude) still requires sending code to Anthropic's cloud. There is no offline model option at the moment for Zed AI either. But since you explicitly choose what to send, you might feel a bit more in control – e.g. you wouldn't accidentally send the entirety of a proprietary codebase, only what you intentionally include. Of course, that also means more work for the user to gather context, whereas Cursor automates that for you.

Extensibility: Interestingly, Zed allows customization of those slash commands via extensions (you could create new commands that fetch context from arbitrary sources, possibly similar in spirit to Cursor's MCP) [107]. And Zed being open-source means its community might extend its AI capabilities over time. Right now, though, Zed's AI is relatively young (introduced in mid-2024) and not as full-featured as Cursor's or even Copilot's. It doesn't have an "agent" that will write code across multiple files autonomously – the user drives each step (though one could manually paste multiple files into the prompt).

Summary: Cursor is a more **hands-off, AI-driven experience** – it tries to automatically use all relevant context and even take actions (with your approval) to edit code. Zed is a more **hands-on, user-driven AI experience** – it gives you the tools to bring context to the AI and apply its output, but you orchestrate the process. Zed's architecture (Rust native app + Claude API) also prioritizes performance and minimal UI latency, while Cursor (Electron app + heavy cloud) might use more resources and occasionally feel heavier when indexing. For developers who enjoy maximal insight into what the AI sees and does, Zed provides that. For those who want the AI to manage context and just give high-level instructions ("find the bug in this project and fix it"), Cursor's approach is far more capable.

To wrap up, **Cursor's architecture** represents a blend of a traditional IDE with cloud AI superpowers. By forking VS Code, it inherits a robust local development platform; by integrating cloud services, it achieves an "AI pair programmer" that is deeply aware of your code. The local component handles what it does best (editing, parsing, LSP features, sandboxing changes), and the cloud component handles what scale demands (large language models and big-data search over your code). Each competitor takes a slightly different approach on this spectrum of local vs cloud, integrated vs plugin, manual vs automatic context, etc. The table below highlights some of these differences:

Aspect	**Cursor** (VS Code Fork)	**GitHub Copilot** (VS Code Ext.)	**Sourcegraph Cody** (Ext. +Server)	**Codeium/ Windsurf**	**Zed AI** (Rust Editor)
Integration	Standalone editor (VS Code-based) with AI deeply integrated into workflow [80].	Extension in your existing IDE (VS Code, etc.) – enhances your current setup [80].	Editor plugin + separate server (Sourcegraph) provides AI; less native but broad repo support.	Standalone editor (Windsurf) or extensions (Codeium) – VS Code-like fork for tight integration.	Standalone editor (built from scratch in Rust) with optional AI panel; very fast and minimal.

Aspect	**Cursor** (VS Code Fork)	**GitHub Copilot** (VS Code Ext.)	**Sourcegraph Cody** (Ext. +Server)	**Codeium/ Windsurf**	**Zed AI** (Rust Editor)
Code Indexing	Yes – indexes entire project into semantic embeddings (cloud-based vector DB) [66]. Enables full-project semantic search in prompts.	No automatic indexing – limited to open files and local context. No semantic search of entire codebase [82].	Yes – indexes all repositories via Sourcegraph; semantic search across org's code (requires Sourcegraph infra).	Yes – in Windsurf, indexes code; can do **local** embedding by default [97] [89] (or use server for team indexes).	No automatic indexing – user includes context via commands. (Relies on user to provide relevant code to AI).
AI Models	Mixed cloud models. Uses OpenAI GPT-4/3.5 or Anthropic for chat, plus **custom models (Copilot++, Fast Apply)** on Cursor's servers for completions/ refactors [17] [18]. Orchestration picks model per task.	OpenAI models via GitHub (Codex, GPT-4). No custom model; one-size-fits-all (with some model upgrades in 2023–24).	Primarily OpenAI (GPT-4, etc.) and Anthropic Claude (100k context) – configurable. Can integrate open-source models for self-host.	Codeium's **own ML models** for autocomplete (fast, specialized) [90]; plus optional OpenAI/ Claude for some features. Windsurf's "Cascade" agent uses these.	Anthropic Claude 2 (Claude v3.5) via Zed's cloud service [99]. No indexing model – just the main LLM answering with provided context.

Aspect	**Cursor** (VS Code Fork)	**GitHub Copilot** (VS Code Ext.)	**Sourcegraph Cody** (Ext. +Server)	**Codeium/ Windsurf**	**Zed AI** (Rust Editor)
Offline/On-Prem	**No** – Cloud required (even with user's API key, requests go through Cursor's cloud) [15] [98]. No self-host available.	**No** – Cloud only (GitHub/ Microsoft servers + OpenAI). GitHub offers cloud enterprise plans but not on-prem.	**Yes** – Can self-host Sourcegraph and even run LLM on-prem. Data stays on your servers if configured [72] [108]. Also Sourcegraph Cloud option.	**Yes** – Codeium offers on-prem deployment for enterprises [71]. Also supports **local indexing** to avoid sending code out [97]. (Public cloud service available for individuals).	**No** – Cloud dependent (Anthropic API). No on-prem Claude (Anthropic doesn't offer self-host). Zed gives control over what you send, but requires internet for AI.
Security & Privacy	Privacy Mode (no code or prompts stored server-side; separate no-log processing) [27]. Embeddings stored with no raw code [44]. Client-side secret scrubbing [9]. Relies on cloud trust (SOC 2, etc.).	Code always goes to cloud (Microsoft/ OpenAI). Business tier promises no training on your data, but code is processed on external servers [72]. Some org policies available, but no private hosting.	Self-host option means code need not leave your network at all. If using Sourcegraph's cloud, similar concerns as others (they claim no data retention by models when using enterprise mode).	On-prem mode keeps data internal. Default cloud mode: code goes to Codeium servers (which promise no storage of raw code). Embeddings can be local or only vectors stored remote [66].	Code sent to Anthropic (third-party) with their data policies. User manually controls context to avoid sending sensitive files. No long-term storage of prompts, but reliant on Anthropic's cloud policy.

Aspect	**Cursor** (VS Code Fork)	**GitHub Copilot** (VS Code Ext.)	**Sourcegraph Cody** (Ext. +Server)	**Codeium/ Windsurf**	**Zed AI** (Rust Editor)
Unique Strengths	Tight IDE integration (apply changes with one click), shadow workspace for error-checking AI edits [5] [7], multi-step agent abilities, plugin system (MCP) [85]. Very rich feature set for power users.	Simplicity and ubiquity – works across editors, very easy to use (autocomplete feels native). Leverages vast training data of GitHub. Strong GitHub ecosystem integration (codespaces, PRs, etc.).	Unparalleled code search and cross-repo intelligence (built on Sourcegraph's best-in-class search). Great for large organizations. Highly configurable (choose models, integrate with dev workflows).	Extremely fast code completions due to custom model; more control on-prem. Windsurf agent can autonomously implement high-level tasks. Emphasis on **data privacy options**. Often more affordable (Codeium has a free tier).	Blazing-fast native editor with minimal latency. Full transparency into AI prompts – good for learning and trust. Lets you bring in exactly the context you want. Great for collaborative editing + AI (since Zed is multiplayer by design).

References: The information in this report is drawn from official documentation and technical deep-dives on Cursor [1] [4] [11] [17] [18], insider discussions about its internal architecture [109] [8], as well as comparisons with other tools [66] [97] [81] [80]. We also consulted Cursor's security notes [43] [9] and the Pragmatic Engineer's coverage of building Cursor at scale [24] [30]. Each tool's approach was considered to give a comprehensive view of how Cursor stands out in the landscape of AI-assisted development environments. By combining a local VS Code fork with a powerful cloud AI stack, Cursor truly exemplifies a next-generation IDE – one that **extends the developer's capabilities** by making the entire context of code available to an AI partner, all while striving to maintain the performance, safety, and control expected in professional software development [110] [111].

1 2 3 4 5 7 11 14 17 18 19 20 22 23 29 39 40 41 46 48 49 50 58 59 85 91 110 111 How Cursor Works Internally? – Aditya Rohilla
https://adityarohilla.com/2025/05/08/how-cursor-works-internally/

6 8 109 Iterating with shadow workspaces · Cursor
https://cursor.com/blog/shadow-workspace

9 42 43 44 45 Codebase Indexing - Discussions - Cursor - Community Forum
https://forum.cursor.com/t/codebase-indexing/36

10 13 21 24 25 28 30 31 32 33 34 35 36 37 47 51 52 53 54 55 56 57 62 63 64 68 74 Real-world engineering challenges: building Cursor
https://newsletter.pragmaticengineer.com/p/cursor

12 15 16 38 60 61 66 70 71 89 90 93 94 95 96 97 98 Cursor vs Windsurf - Choose the Right AI Code Editor for Your Team
https://www.devtoolsacademy.com/blog/cursor-vs-windsurf/

26 27 67 69 73 75 76 77 78 79 Security · Cursor
https://cursor.com/security

65 Experience on issues with Codebase indexing. : r/cursor - Reddit
https://www.reddit.com/r/cursor/comments/1fstf8r/experience_on_issues_with_codebase_indexing/

72 81 82 83 84 86 108 GitHub Copilot vs Sourcegraph Cody: Which Gets Your Codebase? - Augment Code
https://www.augmentcode.com/guides/github-copilot-vs-sourcegraph-cody-which-gets-your-codebase

80 Cursor vs. Copilot: Which AI coding tool is best? [2025]
https://zapier.com/blog/cursor-vs-copilot/

87 Why Cursor Freezes on Large Codebases: 5 Alternatives
https://www.augmentcode.com/guides/why-cursor-freezes-on-large-codebases-5-alternatives

88 How Cody understands your codebase | Sourcegraph Blog
https://sourcegraph.com/blog/how-cody-understands-your-codebase

92 How Cursor built Fast Apply using the Speculative Decoding API
https://fireworks.ai/blog/cursor

99 100 101 102 103 104 105 106 107 Introducing Zed AI — Zed's Blog
https://zed.dev/blog/zed-ai