The software engineer's field guide to the AI stack

A field guide to the new AI stack for modern engineering

glean

Table of Contents

p.01 How AI is changing software engineering

p.03 The emerging AI tooling landscape for engineers

p.06 The two-layer model: coding + context surfaces

p.08 How the two-layer model powers AI tools

p.10 Key questions to ask when evaluating AI tooling

p.12 Glean: A secure, open platform providing deep context

p.16 How it all comes together

01

How AI is changing software engineering

In many organizations—especially the AI-native ones—senior engineers have stopped writing code line-by-line. GitHub Copilot, for example, is now responsible for writing nearly half (46%) of the average developer's code. Engineers now define intent, set constraints, and review diffs while systems handle much of the mechanical work. That means the real bottleneck is no longer how fast someone can produce code; it's how quickly they can assemble the right context, guardrails, and workflows around that code so it's correct, safe, and aligned with the right architecture.

At the same time, product expectations keep rising, systems are more distributed, and incident and security risk grow with every new surface. Teams are asked to ship more, operate more reliably, and adopt AI faster—often without proportional headcount or budget.

Here's how those pressures are changing day-to-day engineering work:

01

LLMs as the new baseline for daily work.

From code completion to documentation and test generation, tools like GitHub Copilot, Cursor, Claude Code, and others are now assumed. They deliver clear productivity gains at the individual level—but mostly inside a single repo or workspace.

02

From "speed" to "systems."

AI has made it trivial to produce baseline code. What's scarce is the ability to connect that code to the right designs, tickets, runbooks, and constraints. AI acts as an amplifier: it makes good architectures, practices, and review processes more effective—and exposes weak ones quickly.

03

Friction, not features, is the real drag.

Engineers are less blocked by writing code and more by chasing context across GitHub, Jira, incident tools, wikis, logs, and Slack. Every new AI surface that doesn't share context with the others risks becoming yet another tab to manage rather than a net productivity gain.

04

Open and composable over all-in-one.

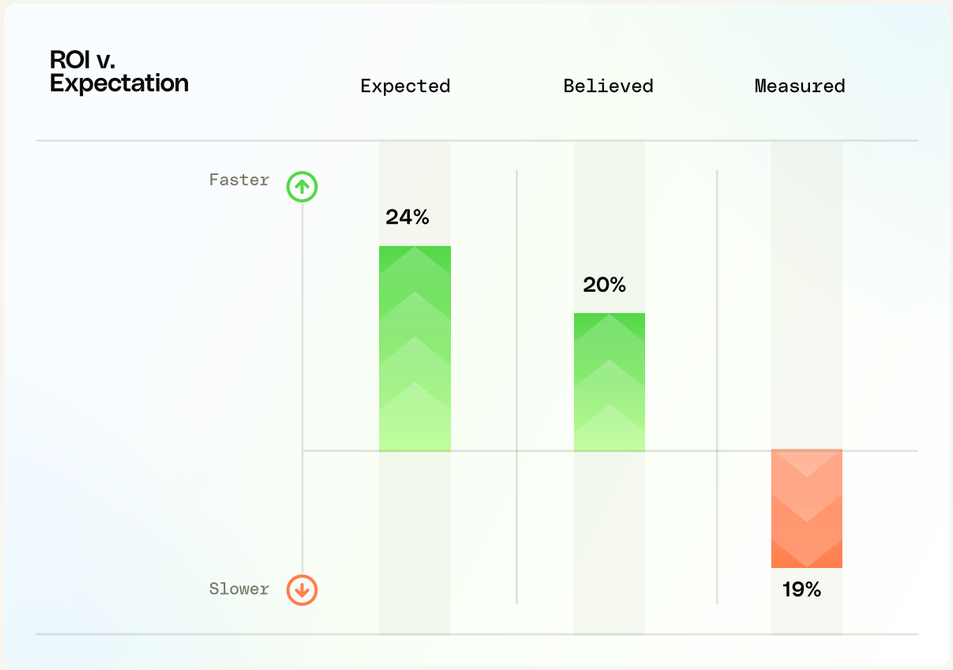
Teams increasingly expect an open, model-agnostic platform they can plug multiple assistants and agents into, not a single vendor that owns every surface. They want a governed context layer plus the freedom to mix and match IDE copilots, incident assistants, and agent frameworks as needs evolve.

05

Code creation to context assembly.

The quality and safety of engineering work now depend less on what an individual can remember and more on how well they can assemble context from code, tickets, docs, logs, incidents, chats, and people. Leaders are starting to evaluate AI tools not as isolated copilots, but as parts of a broader context and governance system that fits their SDLC and stack.

Given these shifts, in most organizations, the bottleneck is no longer “do we have AI in the IDE?” but “does the rest of the system support it?” AI coding assistants live in many environments, yet the context, workflows, and governance around them lag behind. When we spoke with engineering leaders, they described this stuckness in three recurring ways.

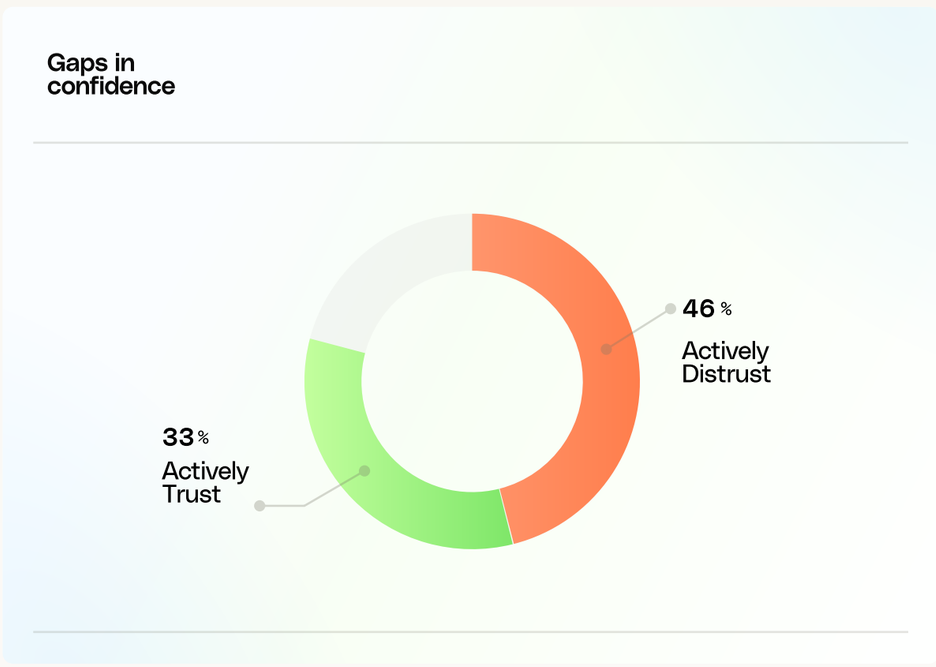


Outcome gap

In a recent randomized controlled trial of experienced open-source developers, issues where developers were allowed to use AI tools took 19% longer to complete—even though they expected to be 24% faster and still believed afterward that AI had sped them up by about 20%. It feels faster because the assistant is always suggesting code, but with only file- or repo-level context it often misses constraints from tickets, designs, and other services, so the “saved” typing time comes back later as rework and integration.

Trust gap

Over time, this creates a confidence problem. More developers actively distrust the accuracy of AI tools than trust them (roughly 46% vs. 33%). 66% of developers say their biggest frustration is AI solutions that are “almost right, but not quite,” and 45% say debugging AI-generated code often takes longer than writing it themselves.





Safety gap

Security reviews are surfacing real risks: in some analyses, close to 45% of AI-generated code contained at least one high-severity vulnerability (such as XSS or SQL injection). Without strong guardrails, real context, and clear review patterns, AI can quietly increase your attack surface even as it helps ship features faster.

To close these gaps, engineering leaders are looking beyond individual tools and starting to think in terms of an overall AI stack—what roles different tools play, what context they can see, and how they fit into the SDLC end-to-end.

02

The emerging AI tooling landscape for engineers

The AI stack for software development is no longer a single model or product choice. Most teams are assembling a portfolio of tools that fall into a few broad categories, each with different strengths and different dependencies on how much real context they can see.

2.1 Coding assistants in the IDE

For most teams, coding assistants (like Cursor, Claude Code, GitHub Copilot, Windsurf, Codeium, Sourcegraph Cody, and others) serve as personal productivity tools for individual developers, rather than as the backbone of an organizational AI strategy.

What they're good at:

- Inline completions and small refactors in the editor
- Scaffolding new endpoints, handlers, or modules
- Generating or filling in unit tests
- Localized debugging inside a single repo or workspace

Where they fall short:

- Little awareness of tickets, incidents, or design rationale
- No unified view across multiple repos, services, and tools
- Security and governance that vary by vendor and need careful setup

41%

41% of developers say their biggest productivity drain is "outdated, inconsistent, or siloed information."

2.2 Enterprise context and knowledge platforms

A second category focuses on connecting the artifacts and people behind the code, rather than just editing the code itself. These systems understand the engineering organization as a whole, not just individual files—and they're critical for turning AI outputs into results the enterprise can actually use.

These tools answer questions like:

- "What broke the last time this alert fired?"
- "Where's the design doc for this service?"
- "Who owns this endpoint now?"

These platforms:

- **Unify code, tickets, logs, docs, designs, incidents, and discussions** from tools like GitHub/GitLab, Jira, observability, wikis, document stores, and Slack/Teams into a single, permission-aware, up-to-date context layer.
- **Build links between artifacts:** PRs — tickets — incidents — designs — chat — so teams can trace from an alert or question to what changed, who owns it, relevant prior work, and past incidents.
- **Expose that graph through search, Q&A, and agents**, so engineers can assemble end-to-end context without hopping across five different tools.

62%

62% of developers say "reconstructing context across multiple tools" slows them down

2.3 AIOps, observability, and incident assistants

Tools in this category usually live inside monitoring and incident response platforms, improving response times and resolution rates. They're most powerful when they can call into a broader context layer rather than operating in isolation.

What they're good at:

- "What broke the last time this alert fired?"
- "Where's the design doc for this service?"
- "Who owns this endpoint now?"

Powerful for reducing MTTR and triage—but typically:

- Scoped to a single substrate (metrics/logs/traces for that tool)
- Unaware of Jira workflows, code ownership, or design decisions stored elsewhere
- Harder to use for questions that span systems (e.g., "what changed, who owns it, and what related incidents have we seen?")

2.4 General-purpose AI platforms and agent frameworks

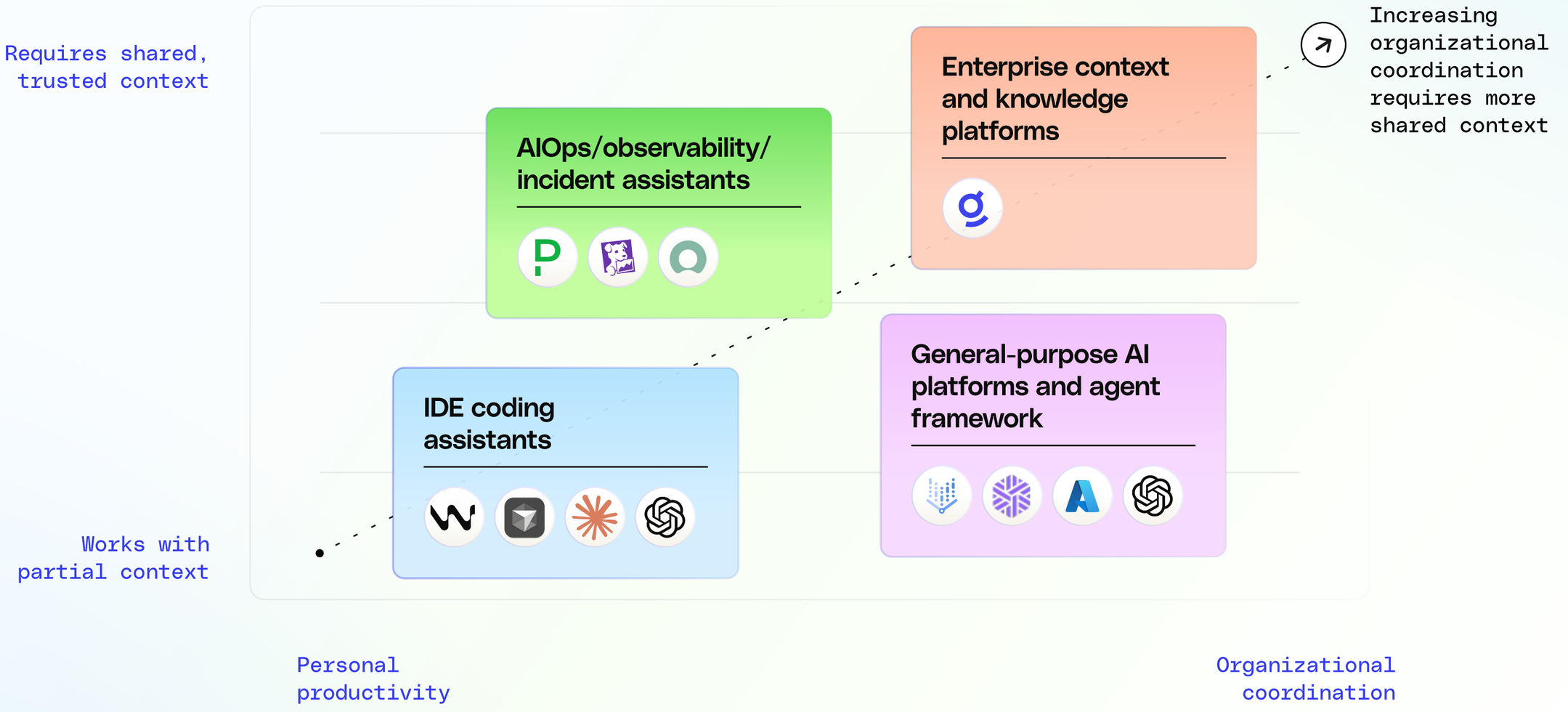
General-purpose AI platforms and agent frameworks. They make it easier to prototype assistants or agents and wire them to tools. They rely on whatever data and structure the organization wires in.

This includes:

- Hosted LLMs and model hubs
- Agent building frameworks and orchestration runtimes
- Low-code or no-code builders for internal assistants and workflows

By themselves, they don't provide:

- An enterprise engineering graph
- Deep, opinionated integrations with your code, tickets, incidents, and docs
- A consistent, permission-aware view across systems



All four categories matter and are important parts of an engineer's AI toolkit. For most engineering teams, the question is not which one to choose, but how to combine them around a shared, trusted context layer instead of letting each tool recreate its own partial view of your stack.

03

The two-layer model: coding + context surfaces

If you look across the tools in the stack, a pattern shows up:

- One set of tools is where engineers *do the work* – IDEs, code hosts, Jira, chat, AI coding environments like Cursor or Claude Code.
- Another set focuses on *understanding the work* – pulling together code, tickets, incidents, docs, logs, and people into a coherent picture.

For many teams, this division of labor is leading to the formation of a **two-layer model: coding and context surfaces.**

Layer 1 | Enterprise context

This is the layer that answers questions like:

- How do our services relate across teams and repos?
- Which incidents, tickets, and design docs are tied to this change?
- Who owns this system now, and what constraints did we agree on?

To do that, a context layer needs to:

- **Connect to the main systems:** GitHub/GitLab and other hosts, Jira, incident tools, observability, wikis, document stores, Slack/Teams, and more.
- **Index code and documents with hybrid search** (lexical + semantic) so it understands your stack, your naming, and your jargon.
- **Build an enterprise graph** linking services, APIs, incidents, tickets, owners, and design artifacts, instead of treating everything as flat text.
- **Respect security and governance boundaries** end-to-end: source-system ACLs, data residency, auditability.

This is the “source of truth for context” that other tools should be able to call into.

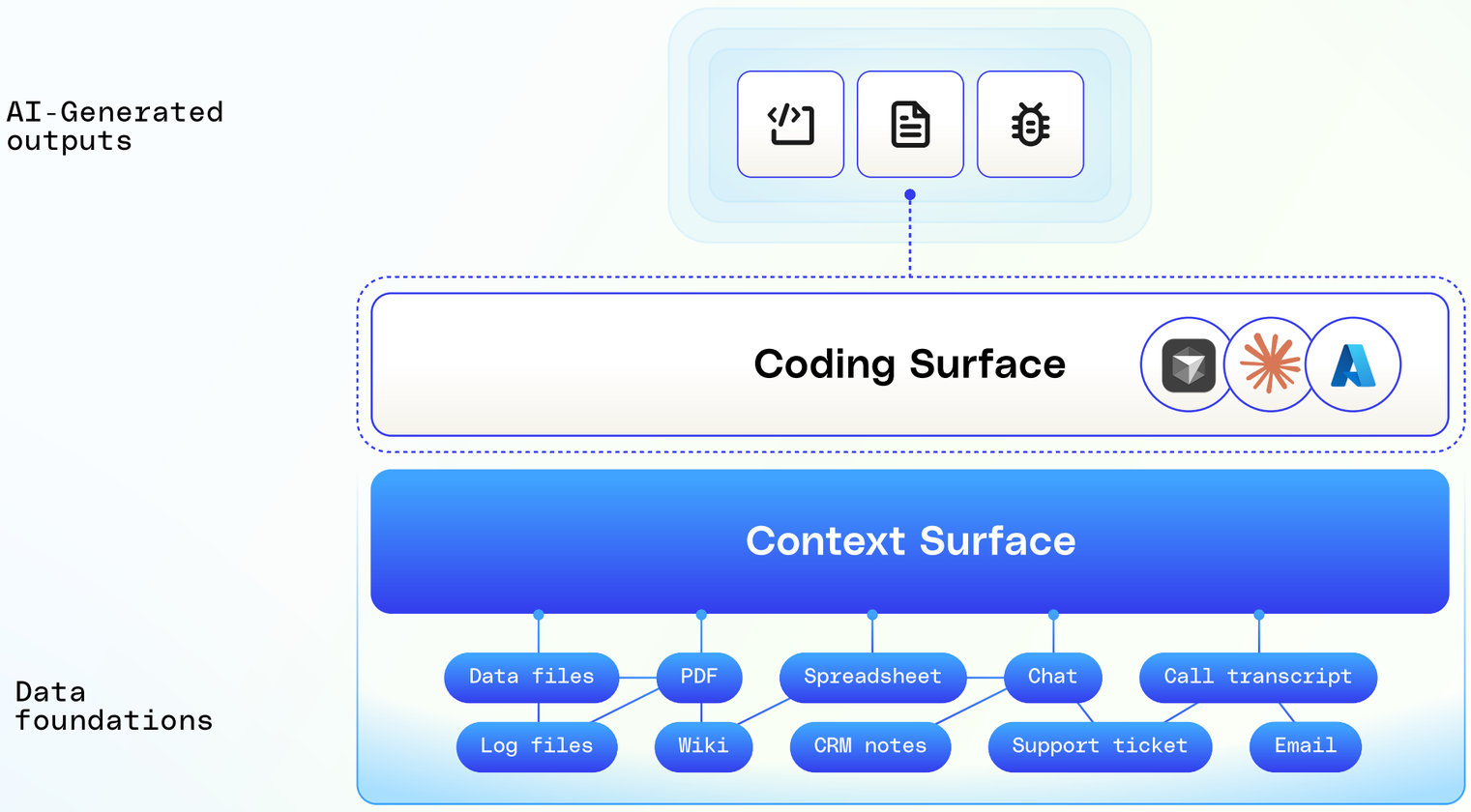
Layer 2 | Coding and interaction surfaces

This is where engineers spend the most time day-to-day:

- IDEs and AI-forward coding environments (Cursor, Claude Code, Copilot, etc.)
- GitHub and other code hosts
- Jira and work tracking
- Slack, Teams, and other collaboration tools

Why adopt a layered model? Coding surfaces are where engineers work every day (IDEs and AI-forward coding environments like Cursor, Claude Code, GitHub, Jira, and more). They provide powerful capabilities and automation, but they can't tap into existing enterprise context unless that context is resupplied and reinvented for every query.

Glean is designed for the context layer: unifying code, tickets, logs, docs, incidents, and discussions into one governed graph, then exposing that context into the tools engineers already rely on—via search, Assistant, agents, and protocols like MCP.



04 How the two-layer model powers AI tools

We're already seeing how AI is showing up in real software development workflows. These are some of the most impactful ways AI is being implemented and leveraged for everyday engineering workflows.

4.1 Onboarding and service understanding

Ownership changes constantly. New hires join, teams reshuffle, and services outlive their original authors. The information you need to get up to speed is usually scattered across old design docs and ADRs, Jira tickets and epics, PRs and code reviews, incident reports and runbooks, Slack threads and email chains, and more.

44% Nearly half (44%) of organizations say onboarding new developers takes more than two months.

AI can make this ramp-up much faster. Teams are already using assistants to:

- explain unfamiliar code paths or components in plain language
- surface the most relevant docs, tickets, and incidents for a given service
- generate onboarding summaries for a service or domain based on real artifacts

When developers use AI for software development, nearly half (43%) say they reduced onboarding time to less than a month.

But this only works if there is a solid context layer underneath. For a query like: "Give me an overview of the payments service and link key design docs," the AI needs more than just access to a codebase. At minimum, the context graph has to know:

- **How "payments service" maps to your systems:** which repos, services, and APIs belong to it.
- **Which artifacts are tied to that service:** design docs, Jira tickets, incidents, runbooks, and major PRs.
- **Who owns it today:** teams and individuals responsible for the service.

With that in place, the answer can be built from code, tickets, incidents, and ownership—not just code comments or whatever happens to match the keyword "payments."

4.2 Day-to-day coding and routine changes

A lot of engineering time disappears into routine work: config tweaks, minor bug fixes, and incremental refactors that keep systems healthy but pull focus from bigger projects.

IDE coding assistants are already strong in this lane. Tools like Cursor and Copilot can handle boilerplate and straightforward logic, suggest small refactors and cleanup, and generate tests while keeping docs in sync as you go.

Paired with code search and code-writing agents, they can even propose scoped changes directly from a Jira ticket or Slack thread instead of starting from a blank editor.

The most common AI use case for engineers is writing new code, used by 71% of code writers, followed by modifying existing code (66%) and debugging or explaining concepts (62%).

Teams that want this to be production-ready are taking one more step:

- Defining clear **change classes** where AI is allowed to propose edits (for example: log-level changes, certain config updates, non-behavioral refactors, documentation and tests), and
- Treating comments, design docs, and linked tickets from the context layer as **guardrails**, so suggestions line up with real requirements and constraints—not just generic patterns the model has seen before.

4.3 Incidents and production support

During an incident, engineers are usually juggling alerts, dashboards, logs, runbooks, Jira, and Slack while trying to answer three basic questions: what changed, what's broken, and who needs to be involved. The context-switching alone slows everything down.

Today, 79% of engineering teams are already exploring AI for incident tracking tasks. When those tools sit on top of a solid context layer, they can:

- Summarize alerts, traces, and logs into a coherent narrative
- Pull in related tickets, recent code changes, past incidents, and owners into a single view
- Draft first-pass incident timelines, postmortems, and even customer updates based on real events and artifacts

This is also where the split between layers is easiest to see:

- **Observability tools** provide the raw signals: metrics, traces, logs.
- **Coding assistants** help propose and apply code-level fixes once you know what's wrong.
- A **context platform** like Glean connects the dots across systems—code, tickets, incidents, docs, and people—so the assistant isn't guessing from one tool's data, but working from a joined-up view of your environment.

05

Key questions to ask when evaluating AI tooling

To move from experimentation to something you can standardize on, it's useful to have a common set of questions. The ones below come up repeatedly in conversations with engineering leaders.

?

01 | What context does it actually have?

Most tools claim to be "context-aware." The real question is: **context over what**?

- Does it only see the **current repo or buffer**, or can it also see the **related tickets, incidents, design docs, and runbooks**?
- Can it follow a chain like: Slack alert → incident ticket → relevant code changes → owners → past incidents? Or does someone still have to glue that together by hand?
- How does it behave with **monorepos, multiple code hosts, and polyglot stacks**—have they been tested at that scale, or is that "future work"?

If the tool can't see the same graph your engineers reason over, it will give shallow or misleading answers.

?

02 | How is trust earned, not assumed?

Engineers will not trust a system that behaves like a black box, no matter how slick the UI is.

- Are answers **grounded in your own artifacts** (code, tickets, logs), with links back so people can inspect the source?
- Are outputs **cited and explainable**, or do they look like a wall of text you have to manually re-validate?
- What happens on bad input or edge cases in incidents and change workflows—does it **fail safely and visibly**, or quietly hallucinate?

If people have to re-do the work to check it, you won't get real adoption.

?

03 | Where does your data go, and who can act on it?

Once AI can read code, logs, and tickets—and especially once it can take actions—you're making a security and governance decision, not just a tooling decision.

- Where are the **indexes for code and documents** (lexical + semantic) hosted—your tenant, vendor's multi-tenant, something else?
- Do **code and logs stay inside your VPC / cloud account**, and are there **tight egress controls** on what agents and code writers can send to external models?
- Are **all queries and actions logged** in a way your security team can actually use for review and incident response?

If you can't answer these cleanly, your security team will block—or should.

04 | Does it fit the way your teams already work?

The best tools disappear into existing workflows. The worst ones ask engineers to live in a new tab.

- Does it integrate cleanly with **GitHub/GitLab, Jira, your observability stack, and Slack/Teams,** or is it yet another disconnected interface?
- Can it **coexist with IDE copilots** (via MCP or similar), using them for editing while it provides the shared context layer—or does it require you to rip and replace?
- What's the **realistic rollout path:** can you start with one team, one service, one workflow, and scale out, or does it require a big-bang switch?

If the answer to "where would engineers touch this every day?" is unclear, the tool will end up as a pilot, not part of your SDLC.

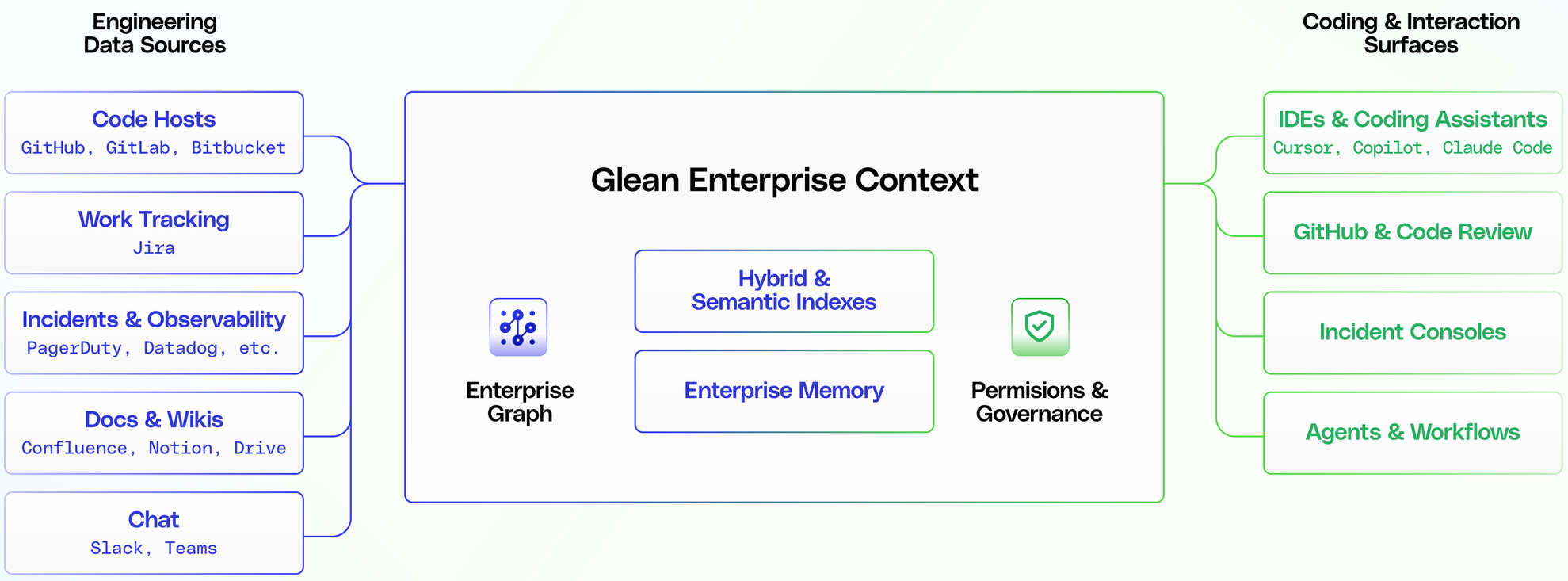
You can apply these same questions to coding assistants, observability add-ons, agent frameworks, and context platforms. They're a way to test whether a tool will actually improve how your systems are built and run—or just add one more layer of complexity.

06

Glean: A secure, open platform providing deep context

In the two-layer model, Glean is the **context layer**: the shared, trusted view of your engineering environment that everything else—IDEs, incident consoles, agents—can plug into. It's designed for teams that want AI in the critical path of their SDLC without giving up control of their systems or data.

Glean as the Context Layer



Single-tenant, customer-controlled deployment (AWS/Azure/GCP)

2 *out of* 3

2 of the top 3 coding-assistant organizations rely on Glean to power the context layer behind their engineering tools.

Four properties matter most for software teams.

6.1 Deep enterprise context

Glean's powerful connectors build and maintain an **enterprise context graph** over your real stack:

- Code across hosts and repos
- Jira tickets and epics
- Incidents and runbooks
- Observability data, where connectors exist
- Design docs, ADRs, wikis, and Drive/SharePoint content
- Slack/Teams conversations and other collaboration trails

For engineers, this enables:

- **Search as the entry point to a system**, not just a file finder: "show me the main docs, tickets, incidents, and owners for service X."
- The ability to move from an alert or question straight to the relevant code, people, and design decisions—without rebuilding the story by hand.
- AI assistants and agents working over **real, linked artifacts** instead of guessing from a single repo or buffer.

The graph becomes the backbone that higher-level tools rely on, rather than each one trying to reconstruct partial context on its own.

Teams using Glean's context layer see measurable impact across their SDLC:

20%	20% faster time to ship code at leading rideshare company
5.3%	5.3% increase in developer output at the world's largest home improvement retailer
$2.4M	$2.4M in annual engineering savings at leading networking platform from a Glean-powered Threat-bot agent

6.2 Open, horizontal platform

The rest of your stack will keep changing: IDE copilots, incident assistants, SaaS apps, models, and agent frameworks will all evolve on different timelines. Glean is deliberately **horizontal** so it can sit underneath that movement instead of competing with it.

Concretely:

- It's **model-agnostic**: you can route different tasks (coding, reasoning, multimodal, cheaper batch work) to different models while keeping one consistent context and governance layer.
- Indexed data is exposed through **APIs and MCP-compatible tooling**, so multiple assistants and frameworks can all call into the same graph rather than rebuilding their own embeddings and permissions logic.
- It's built to support **multiple functions**—engineering, GTM, support, operations—on top of the same governed enterprise context, so the organization doesn't fragment again at the AI layer.

You index once; you reuse that context across tools and surfaces.

6.3 Built for enterprise security, governance, and scale

Glean treats AI as an infrastructure decision, with isolation, residency, and auditing that security and compliance teams can approve. For serious engineering organizations, AI is a security and governance decision as much as a tooling decision. Glean's architecture reflects that:

- **Single-tenant by default**: each customer runs in an isolated environment; no shared indexes, no cross-tenant logs.
- Ability to **run it in your own cloud** (AWS, Azure, GCP) or as Glean-hosted single-tenant SaaS; in all cases, code, logs, and data stay inside your controlled environment.
- **Source-system ACLs are enforced end-to-end** across search, Assistant, and every agent, so AI cannot see or act on anything a user couldn't see in the underlying tools.
- Every query and action is **audited**, giving security and compliance teams the visibility they expect for production systems.

This lets you add connectors, ship new agents, and onboard more teams without cloning indexes or re-implementing permission models in every tool.

6.4 Better together with coding assistants

Glean is built to work with the coding assistants your engineers already like—Cursor, Copilot, Claude Code, and others—by giving them a better substrate to stand on.

The division of labor is simple:

- **Coding assistants** stay focused on local editing and generation inside the IDE.
- **Glean** provides the governed, organization-wide context layer, reachable via search, Assistant, agents, and an MCP server those assistants can call from the editor.

In practice, that looks like:

- Glean agents turning Jira bugs or Slack threads into **scoped implementation PRs**, which developers then refine and extend with their preferred coding assistant.
- Debugging a production issue in Cursor or Claude Code, with the assistant calling Glean via MCP to bring in recent incidents, Jira tickets, ownership, and design docs, then proposing a context-aware patch without leaving the editor.

For forward-leaning engineering teams, this combination is what makes AI feel like part of the stack rather than a sidecar: a stable context and security foundation that multiple assistants and agents can build on over time, while leaving developers free to keep the tools they already trust in the editor.

07

How it all comes together

As AI becomes a standard part of software engineering, three priorities matter more than any single feature or model.

AI is moving from experiments to infrastructure
Teams that standardize on a context layer and a small number of well-understood workflows will move faster than those running disconnected pilots in individual tools.

Strong stacks blend tools, not vendors
Coding assistants, observability AI, and enterprise context platforms solve different problems. The winning pattern is a portfolio wired around shared context and governance, not a bet that one product will do everything.

Security, governance, and explainability are now table stakes
Where AI runs, what it can see, how it's audited, and whether engineers can see why it gave a particular answer will matter as much as raw model quality.

Taken together, these make the two-layer model less of a mental framework and more of an architectural requirement. A solid context layer lets different coding assistants and agent frameworks coexist, reduces context-chasing by avoiding the addition of more surfaces, and keeps one trusted place where organizational knowledge, permissions, and history live.

The race is no longer "how quickly can we add another AI tool?" It's "can we design an AI stack that matches how our engineers actually work, keeps us in control of our systems and data, and gives every tool the same, trusted view of our environment?" For software engineering leaders looking to get a head start, we'll leave you with the five priorities to focus on during AI implementation:

- [ ] Start measuring impact in terms of cycle time and quality, not just how many developers have a coding assistant installed.
- [ ] Focus your AI investments on reducing handoffs and rework across systems, not just on faster code generation in a single tool.
- [ ] Prioritize security, governance, and transparency before implementation and adoption. Trust and safety are prerequisites for AI, not an afterthought to be patched in.
- [ ] SaaS sprawl came first, AI sprawl will come second. Keep scalability in mind whenever you're hunting for solutions. Look for open platforms that allow you to apply context, maintain visibility, and enforce policies across all your current and future AI tooling.
- [ ] Link broader context and your organization as a whole to the work you do. Bringing together the right data is key to better code creation—find context systems that fit your SDLC and stack.

Glean provides the context layer for modern engineering stacks—trusted, secure, and shared across your tools—so teams can ship faster with AI while keeping systems reliable.

Get a demo