Redis

Written by Simba Khadder
Head of Engineering,
Context Surfaces

GUIDE

Context architecture for production AI agents

The real bottleneck in AI agent systems is context

Your models are impressive. Your prototypes worked. But somewhere between proof-of-concept and production, something started to crack.

The agent produces confident, irrelevant answers. It has no memory of what a user said three sessions ago. It tries to query your production database in ways no database administrator (DBA) would sanction. Response times are unpredictable. And when something fails—and it will—it is genuinely unclear which layer of the stack is at fault.

You might think the source of the problem is the model, but the more likely source is your context.

> Model quality is rarely the bottleneck in production agent systems. Context architecture usually is.

For grounded tasks, model quality depends heavily on the context, tools, and state the system surfaces at the right moment. The quality, relevance, freshness, and structure of that context determine almost everything about how useful the model actually is in production.

What makes this especially pressing right now: a year ago, an AI agent could work uninterrupted and succeed at a task for roughly two minutes. Research from METR, a model evaluation firm, shows that today, with models like Claude Opus, agents can operate reliably for up to an hour without human intervention. Even more significantly, that number is doubling approximately every nine months. Tool-calling accuracy has also climbed from roughly 50% to over 90% in the same period.

Combine these two growth curves, and you can expect to have agents capable of working autonomously for extended periods, using tools with high reliability. The question is no longer whether the model can do the task. It is: what are you giving it to reason about?

> "You've got a brilliant analyst locked in a room for an hour with nothing to work with. That's not a reasoning problem. That's a data access problem."
>
> — Simba Khadder

This playbook provides a four-pillar framework for building context systems that hold up in production: systems where agents can dynamically navigate to the right data, retrieve it from anywhere, improve with every interaction, and do all of this at the speed users expect.

What context engineering actually means

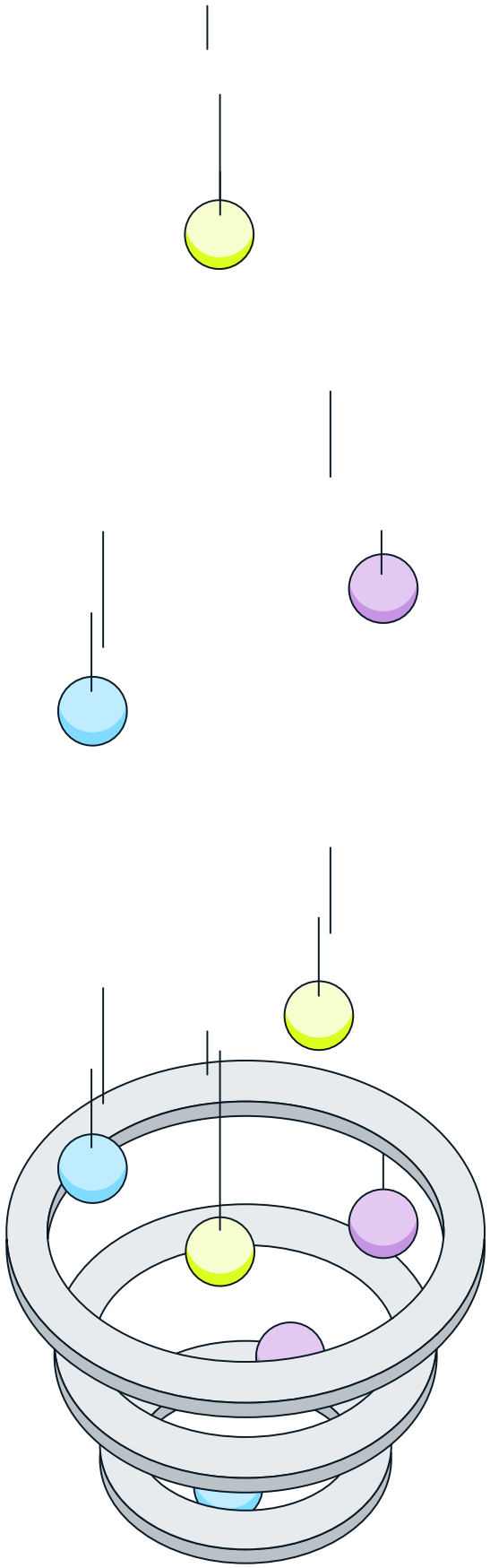
Context engineering is the discipline of deciding what information to put in front of an LLM, in what form, from which sources, and at what time. It sits between your data layer and your model, and it is where the majority of real architectural complexity in agent systems lives.

It encompasses several overlapping concerns that most teams treat as separate problems:

- RAG (Retrieval-Augmented Generation): retrieving relevant unstructured documents at query time via semantic search
- Memory: persisting and retrieving information across interactions and sessions
- State and history: tracking what has happened within and across sessions in real time
- Structured data access: safely surfacing entity-level, transactional data from operational systems
- Prompt engineering: the system instructions that shape agent behavior and reasoning
- Structured outputs: so agents return data in forms that downstream systems can consume

Most agent architectures handle each of these in isolation: a vector store over here, a text-to-SQL hook over there, a session store assembled from scratch. The result is a system with no coherent access layer, no clear failure domain, and no mechanism for improvement over time.

A context engine changes that. It is a unified semantic and access layer that gives an agent consistent, governed, high-speed access to all of the context it needs, regardless of whether that context lives in a document store, a relational database, an external API, or a past conversation.



Why common approaches break in production

Before outlining what good context architecture looks like, it is worth being explicit about why the most common approaches fail at scale. Most teams discover these failure modes through painful production incidents rather than deliberate architectural review.

Failure mode 1: One-shot RAG

RAG was designed for a world where an LLM could reason for roughly thirty seconds. Retrieve the top-k semantically similar passages, inject them into the prompt, generate a response. It works well for narrow, document-lookup use cases.

But it is fundamentally a static, one-shot pattern. It was not designed for agents that can reason for thirty minutes, make dozens of tool calls, and iteratively navigate toward an answer.

The failure mode is easy to reproduce: a user asks, "Why is my order late?" Naive RAG retrieves paragraphs from a help-desk article about common delivery delays. The user gets a generic non-answer. The problem is not the vector database—it is the architectural assumption that a single retrieval pass is sufficient. The actual answer lives in a structured database, not a document corpus.

> RAG as a static, one-shot pattern is losing ground for most agentic use cases. Vector search, however, is more valuable than ever—as a dynamic tool the agent decides when and how to invoke.

Failure mode 2: Unreliable text-to-SQL access

Giving an agent direct SQL access to a production database seems like a simple solution: what more context could you need? But LLMs operate via text, which means you need to convert text to SQL to make native SQL querying possible. Worse is the risk involved in giving agents access to your system of record. It's a seemingly good approach that's ultimately one of the riskiest architectural decisions a team can make. Even assuming query accuracy continues to improve (it will), the structural problems remain:

- Security and access controls are extremely difficult to enforce correctly at the query generation level
- Agents fed large, unfiltered result sets hallucinate at much higher rates—too much irrelevant context is as damaging as too little
- Uncontrolled query patterns can place serious load on production infrastructure
- Iterative, multi-step queries compound all of the above risks

The safer and more scalable alternative is a semantic layer: a defined, agent-readable model of your business entities, attributes, and relationships that maps to a governed set of tools. The agent navigates a model of your domain rather than constructing raw queries against your database.

Failure mode 3: REST-to-MCP conversion

Many teams have tried to auto-convert existing REST APIs into MCP tools using spec converters. In practice, this rarely works well and it makes sense why. REST APIs were designed for developers who can read docs, navigate auth flows, and chain calls manually. An agent handed thousands of tools with complex parameter signatures and multi-step auth tends to get lost.

MCP isn't just a different protocol—it's a different contract. Purpose-built MCP integrations almost always outperform auto-converted ones because the shape of the tool surface matters as much as the data it exposes.

The four-pillar framework

A production-grade context strategy is built on four foundational capabilities. Think of them as the four dimensions your architecture must satisfy—each one distinct, all four necessary for a system that scales.

01 Navigate

Allow agents to dynamically find the context they need

- Agents should actively navigate your data model, not receive a static, pre-assembled prompt
- This requires a semantic layer that defines your business entities, their attributes, and their relationships in natural language
- Tools are generated automatically from that model—coherent, discoverable, and purpose-built for agent reasoning
- Authorization and access control are enforced at the model level, not bolted on at query time
- An agent that can navigate well can solve for user intent even when the answer spans multiple data sources or systems

02 Retrieve

Surface context from anywhere, regardless of where it lives

- Production data exists across SQL databases, document stores, time-series systems, external APIs, and conversation history
- Agents must be able to retrieve from all of these through a single, unified access layer
- Unstructured data (documents, knowledge base articles) should be accessible via dynamic, real-time semantic vector search
- Structured, transactional data should flow through a governed semantic layer—not raw database access
- The unified layer enables agents to combine structured and unstructured context in a single reasoning pass

03 Improve

Let context get smarter with every interaction

- Static context is a ceiling. A production system should have a flywheel: the more it's used, the better its context becomes
- This requires both short-term (session-scoped) and long-term (persistent) memory working in concert
- Working memory captures current conversation state; long-term memory retains preferences, facts, and learnings across sessions
- Memory can be populated by the LLM itself, by explicit application logic, or by background extraction agents that run continuously
- Background extraction creates compounding value: context improves automatically without adding latency to the critical path
- A coding agent that accumulates knowledge of your codebase's architecture and conventions is qualitatively more useful than one that starts fresh every session

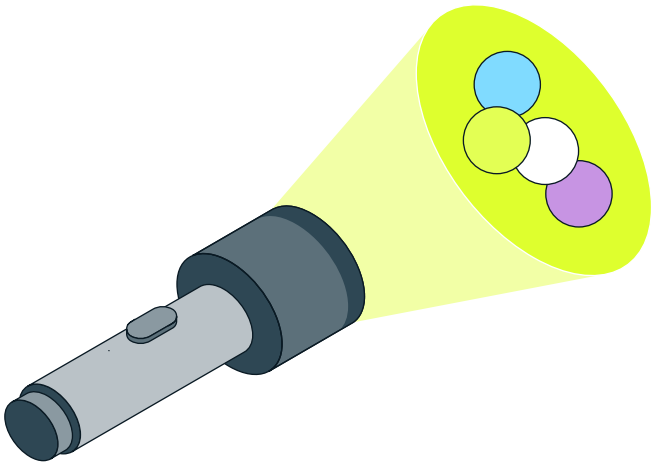
04 Accelerate

Make every context operation fast, regardless of data type

- Agents make many tool calls per task; each must return quickly, especially in interactive use cases like customer support
- Speed is a design constraint, not an afterthought—it must be engineered into the entire context stack
- Semantic caching allows agents to short-circuit expensive LLM calls when the query intent is semantically equivalent to a prior result
- For leaders: semantic caching can reduce inference costs substantially for high-volume workloads with repeated or near-identical intent patterns
- The underlying data infrastructure must support all relevant data types—vector, time-series, geospatial, text—at consistently low latency
- Vector databases in production environments, especially ones with multi-data structure support, can eliminate multi-hop retrieval patterns

What this looks like in practice

These four pillars map directly to the failure modes your team is already encountering. Below is a comparison of how context architecture plays out across common scenarios:



Scenario	Without a context engine	With a context engine
User asks about their order	"Here are common reasons for delays"—pulled from a help-desk article via one-shot RAG	Agent looks up the specific order, surfaces real-time status, retrieves applicable policy, resolves the issue
Identifying fraud transactions	Text-to-SQL query runs against production DB with no access controls; unfiltered results cause hallucination	Agent navigates semantic data model, lists orders with risk flags, cross-references products and user data safely
Returning user onboarding	No memory; agent treats every session as the first one	Long-term memory surfaces user preferences, past decisions, and prior context automatically
High-volume support at peak load	Every query triggers full LLM inference; costs and latency scale linearly	Semantic cache returns near-identical responses; inference costs and latency drop substantially
Coding agent across sessions	No awareness of codebase architecture or prior engineering decisions	Working and long-term memory ensure the agent accumulates knowledge of conventions, patterns, and past mistakes

Reference architecture: the Redis context engine

A context engine is the infrastructure layer that operationalizes all four pillars. It sits between your data sources and your LLM, providing a unified semantic and access layer that agents interact with through MCP.

The following maps each functional area of a production agent system to its Redis component:

The critical architectural property is that all components share a unified semantic layer. Structured data, unstructured documents, short-term state, and long-term memory are all accessible through the same governed interface. The agent does not need to know where data lives or how to retrieve it. It navigates a coherent model of your domain.

Component	What it does	Redis capability
Vector database	Real-time semantic search over unstructured documents and knowledge bases	Redis
Agent memory	Dual-tiered memory: session-scoped working memory + persistent long-term memory with semantic retrieval	Redis Agent Memory (open source)
Context engine	Unified semantic layer; translates business entity models into agent-navigable tools with built-in authorization	Redis for AI
Semantic cache	Fuzzy-match response caching; reduces LLM inference cost and latency for semantically equivalent queries	RedisVL or Redis LangCache
Agent framework durability	Checkpointing for LangGraph and other frameworks; ensures agent state survives interruptions	Redis for LangGraph: checkpointing and storage
Structured data access	Governed, high-speed access to transactional and operational data via defined entity models	Redis Search

A closer look: memory architecture

Memory is the mechanism through which context improves over time. It's also one of the most underspecified areas in most agent architectures. Redis Agent Memory is a dual-tiered system:

Working memory (session-scoped)

Working memory captures the state of the current session: conversation history, intermediate reasoning, and partial results. It is durable by default, supports optional time-to-live, and automatically summarizes when sessions grow long to prevent context window overflow. This is what allows an interactive agent to maintain coherent multi-turn reasoning.

Best for: interactive chatbots, multi-turn customer support agents, coding assistants during an active work session.

Long-term memory (persistent)

Long-term memory persists user preferences, extracted facts, learnings, and important decisions across sessions. It uses vector embeddings for semantic retrieval—so an agent can surface "what did this user prefer last time they did X?" without exact keyword matches. Advanced filtering by time, topics, entities, and users enables precise memory retrieval in complex, multi-user systems.

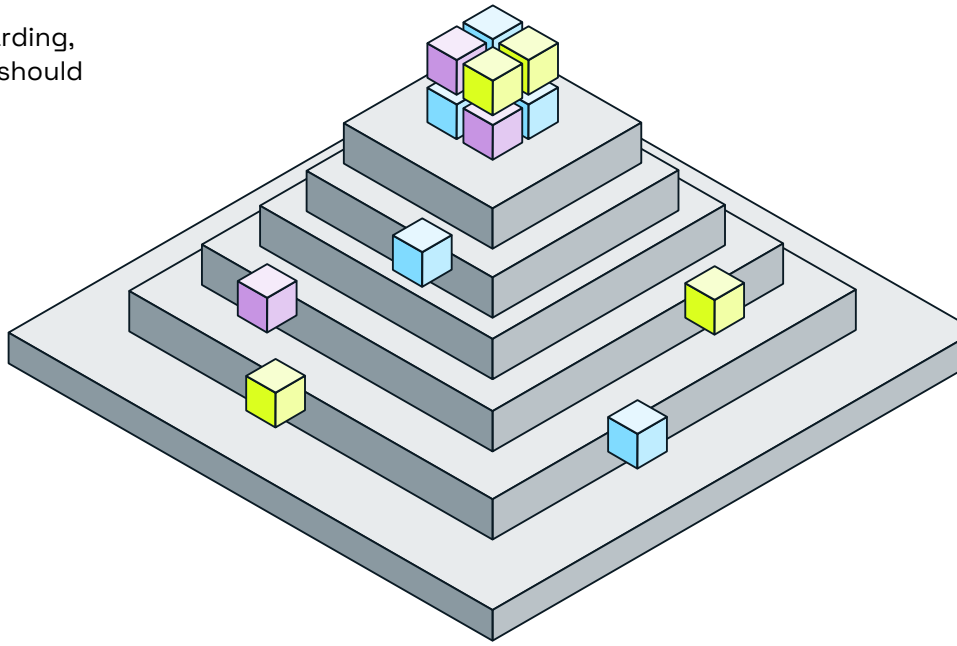
Best for: personalization, progressive onboarding, recommendation systems, and agents that should measurably improve with usage.

Memory extraction patterns

How memories are written is as important as how they are stored. Redis Agent Memory supports three integration patterns:

- LLM-driven: The model decides what to remember and writes memories via tool calls. Best for conversational agents where the LLM has natural visibility into what matters.
- Code-driven: Application logic explicitly writes memories via SDK. Best for structured workflows where the information to persist is deterministic.
- Background extraction: An asynchronous process reads conversation history and automatically extracts context according to a configurable strategy. Enables continuous learning without adding latency to the critical path.

Background extraction deserves special attention. It enables a powerful architectural pattern: a secondary agent that continuously grooms your context surface, enriching it based on actual usage. This is the mechanism that makes context compound over time without ongoing engineering effort.

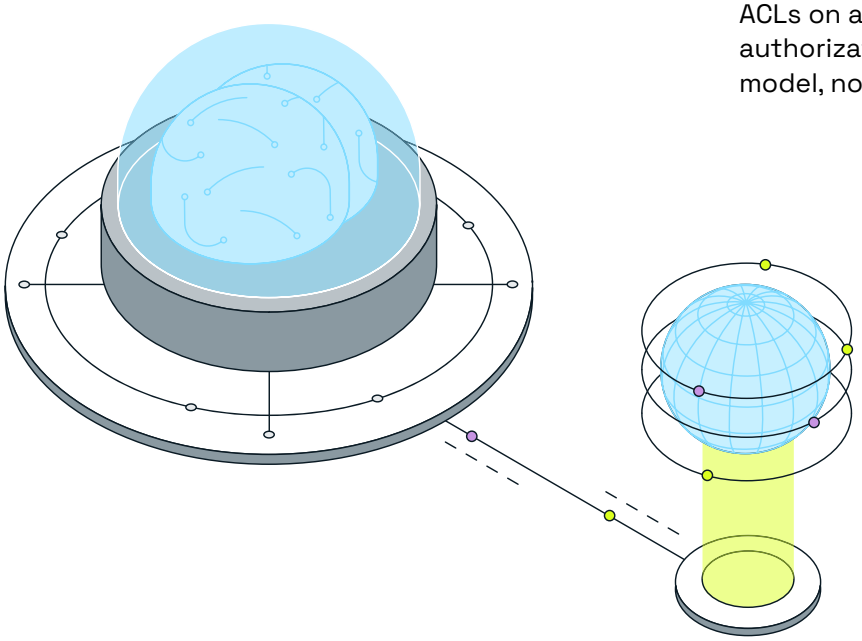


A closer look: structured data access

Structured, transactional data—orders, users, accounts, events, transactions—is the most informationally dense data your organization holds. A single row in an orders table contains more actionable context about a user's situation than many paragraphs of help-desk content. Yet most agent architectures treat unstructured data as the primary context source and tack on structured access as an afterthought.

The correct pattern is a semantic layer: you define your business entities as a typed model with attributes, relationships, and natural language descriptions. That model generates a governed set of tools the agent can call. The agent navigates your domain model rather than constructing raw queries.

> Think of what a human support agent has in front of them: a portal with search, tabbed views of orders and conversations, account summaries. That is what you need to give your AI agent—a coherent interface to your data, not a wall of API documentation or unrestricted database access.



This approach provides three properties that raw database access or REST-to-MCP conversion cannot:

- Security by design: the agent can only access what the model exposes. Row-level and entity-level access controls are built into the model, not retrofitted onto query outputs.
- Reliability: the agent operates on semantically coherent tools—get_order_by_id, list_orders_with_risk_flags—rather than constructing SQL or chaining underdocumented API calls. Predictable tool surfaces produce predictable agent behavior.
- Speed: when the access layer is backed by an in-memory system designed for low-latency reads, tool calls can remain fast enough to support multi-step reasoning loops. Agents can make dozens of calls in a reasoning loop without accumulating latency that degrades the user experience.

It is also worth noting that tenant separation—a critical concern for any multi-user or enterprise deployment—is far easier to implement correctly through a semantic layer than through row-level ACLs on a database the agent queries directly. The authorization logic is a first-class property of the model, not a downstream filter.

The business case: cost, reliability, & speed

For technical leaders, context architecture is not just an engineering concern. It has direct, measurable implications across the three dimensions that typically drive infrastructure investment decisions.

Reliability

The primary driver of agent failure in production is not model quality—it is the wrong context, or no context, reaching the model. Hallucination rates drop significantly when agents have access to accurate, structured, entity-specific data. A context engine that ensures agents receive the right information at the right time is the most direct path to improving production reliability. It is also the most direct path to shortening the feedback loop when things go wrong: with a unified context engine, the failure domain is identifiable.

Speed

For interactive use cases—customer support, co-pilots, real-time recommendation agents—latency is a product quality metric that users notice directly. Redis has been independently benchmarked as the fastest vector database across multiple datasets and query types. Its native support for time-series, geospatial, text, and vector data eliminates the multi-hop retrieval patterns that accumulate latency in architectures that distribute these concerns across multiple specialized systems.

There is also a compounding efficiency argument for investing in context architecture early: the better the context engine, the less manual engineering intervention the agent system requires over time. Context that improves automatically with usage reduces the curation burden on your team. That compounding effect is the most compelling long-term case for treating context architecture as a strategic infrastructure investment rather than a tactical fix.

Cost

As agents take on longer and more complex tasks, inference costs scale. Two mechanisms in the context engine architecture directly reduce this:

- Semantic caching Redis LangCache returns semantically equivalent responses without triggering a new LLM inference call. For workloads with high query repetition—customer support, FAQ handling, batch scoring—this can reduce inference costs substantially. The fuzzy-match approach means it handles natural language variation without requiring exact query matches.
- Precise context delivery: overfeeding an agent with irrelevant context wastes tokens and degrades answer quality. A well-structured context engine surfaces exactly the right data, reducing prompt length and improving output reliability simultaneously.

Architectural self-audit: four-pillar checklist

Use the following to evaluate your current agent architecture against each pillar. Gaps in this audit represent your highest-priority architectural decisions. A gap in navigate typically manifests as agents that perform well on generic queries but fail on anything requiring knowledge of a specific user, order, or account. A gap in improve manifests as agents that feel stateless—intelligent but amnesiac. A gap in accelerate manifests as latency spikes that are hard to attribute to a single cause.

Navigate

- Do agents access a semantic model of your business entities, or are they querying raw databases / REST APIs directly? ☐ YES ☐ NO
- Are authorization and access controls defined at the data model level, not at query time? ☐ YES ☐ NO
- Can an agent traverse entity relationships without chaining undocumented API calls? ☐ YES ☐ NO

Retrieve

- Is there a single, unified access layer for both structured (transactional) and unstructured (document) data? ☐ YES ☐ NO
- Can an agent retrieve context from external APIs, document stores, and databases through the same interface? ☐ YES ☐ NO
- Is vector search a dynamic tool the agent calls when needed, not a static pre-retrieval step? ☐ YES ☐ NO

Improve

- Does your agent system have session-scoped working memory that persists across a task? ☐ YES ☐ NO
- Is there persistent long-term memory that retains preferences and learnings across sessions? ☐ YES ☐ NO
- Is any memory extraction automated, or does every new use case require explicit engineering effort? ☐ YES ☐ NO

Accelerate

- Is there a semantic cache in place to short-circuit repeated or near-identical queries? ☐ YES ☐ NO
- Are all tool calls backed by infrastructure capable of sub-10ms response times under production load? ☐ YES ☐ NO
- Does your context stack support all relevant data types without multi-hop retrieval patterns? ☐ YES ☐ NO

Where to start

For most teams, the highest-leverage first move is establishing a semantic layer for structured data access. This is the pillar with the highest failure risk in typical architectures—text-to-SQL unreliability and REST-to-MCP conversion being the two most common anti-patterns—and the highest upside in terms of unlocking the full reasoning capability of long-running agents.

The second-highest-leverage move is standing up dual-tiered agent memory. Most teams have some form of session state; very few have persistent long-term memory with semantic retrieval. Redis Agent Memory is open-source, exposes a RESTful API and MCP interface, and integrates directly with LangGraph and other common agent frameworks.

From there, the pillars build on each other: a semantic layer enables better navigation and retrieval; memory enables continuous improvement; Redis' performance characteristics and Redis LangCache ensure none of these capabilities come at a latency or cost penalty users will notice.

> Context engineering is where most of your engineering team's effort will increasingly flow—building and maintaining the systems that give agents the right information at the right time to solve for what users actually need.

The Redis context engine—comprising the Redis Search (RedisVL), Redis Agent Memory, and Redis LangCache—provides the production-ready components for all four pillars. The architectural work is in assembling them coherently, defining your semantic layer with intention, and building the feedback loops that let context improve with use.

That is the work of context engineering. And it's the work that will determine whether your agent investments compound in value over time—or become the source of exactly the kind of technical debt you are trying to avoid.

About Redis for AI

Redis is the real-time context engine for AI. We provide the infrastructure that gives agents the right context, at the right time, at the speed production demands—through a unified semantic and access layer spanning structured data, unstructured documents, and agent memory.

Production-ready components:

- Redis Search: the fastest benchmarked vector database for real-time semantic search
- Redis Agent Memory open-source dual-tiered memory with MCP and REST interfaces
- Redis for AI: real-time context engine for structured transactional data with built-in authorization
- Redis LangCache: semantic caching to reduce LLM inference cost and latency
- Redis checkpointers: agent framework durability for LangGraph and similar orchestration systems