A TECHNICAL WHITEPAPER

Building with LLMs: The Framework Atlas

A Decision Guide for Architects and Engineers Shipping AI in 2026

Author	Ajay Walia
Version	4.0
Date	April 2026
Audience	Technical architects, engineers, AI practitioners
Scope	End-to-end framework landscape for LLM-powered systems

Abstract

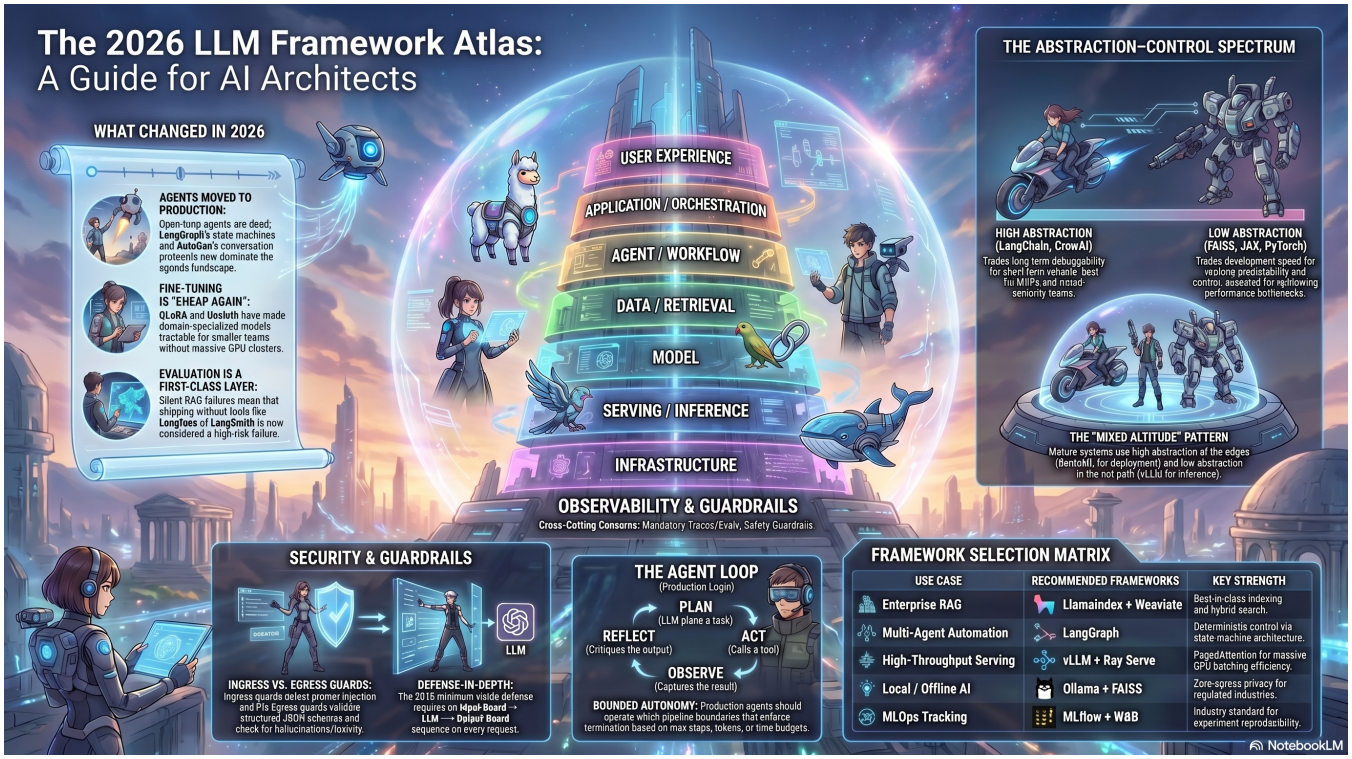
Building an LLM-powered system in 2026 is no longer a model choice — it is an architecture choice made across a dozen competing frameworks, each claiming a different layer of abstraction, deployment model, and philosophy about how intelligence should be composed into software. This atlas maps that landscape as a practitioner sees it. It is organised as five parts and seven plus three layered categories: core ML and deep-learning libraries, LLM orchestration, agent and workflow frameworks, vector databases and retrieval, model serving, end-to-end MLOps platforms, and the three layers that became non-negotiable between 2024 and 2026 — fine-tuning and training, observability and evaluation, and guardrails and safety. Each framework is catalogued against the same eleven attributes — language, abstraction level, architecture type, use cases, strengths, weaknesses, deployment model, maturity, learning curve, ecosystem fit, and (new in v2) community health. The atlas then distils the catalogue into decision heuristics, a selection flowchart, comparative heatmaps, per-category maturity radars, cost and latency envelopes for serving and retrieval, and four reference architectures — enterprise RAG, multi-agent automation, local-only stacks, and fine-tuned domain models — each delivered with working code. Where we cite concrete claims or benchmarks, sources are listed at the end of each section so individual statements can be traced. The intended use is as a working architect's reference: the kind of document you keep open on a second monitor while making framework decisions you have to live with in production for the next two to three years.

This document is technical in scope; mathematical detail is kept to what architects need to choose between frameworks, not to implement them from first principles.

Table of Contents

The Framework Atlas at a Glance

Before diving into the detailed landscape, the one-page map below captures the whole story at a glance: the six canonical layers of the AI stack, the cross-cutting concerns that wrap them, the abstraction–control spectrum every architect navigates, the agent loop that governs autonomy, and the selection matrix that recurs throughout Parts III and IV. Every element shown here is unpacked in detail in the chapters that follow.



A visual overview of the 2026 LLM framework landscape — layers, spectrum, agent loop, guardrails, and selection matrix — all of which are unpacked in detail throughout the report.

Executive Summary

Four years after LLMs entered the mainstream, the framework landscape has matured from an experimental collection of notebooks into a stratified stack with clearly defined roles. What looks like chaos from a distance — dozens of framework names jostling for attention — resolves at close range into ten categories of frameworks, each solving a well-posed problem: training a model, composing prompts, orchestrating agents, retrieving context, serving inference, and so on.

This atlas answers three questions an architect faces when designing an AI system in 2026:

- **What does the stack actually look like?** — Which frameworks sit at which layers, and what do they owe to one another?
- **How do I choose among them?** — Given my constraints (scale, latency, privacy, cost, team skill), which framework at each layer minimizes future regret?
- **What does a working system look like end-to-end?** — Not a toy demo, but a production architecture that ties the layers together into something debuggable and operable.

The short answer

There is no single AI stack. There is a *canonical shape* — application, agent, data, model, serving, infrastructure — with framework families competing inside each layer, wrapped top-to-bottom by two cross-cutting concerns: observability and guardrails. The choice within a layer is almost never about raw capability; it is about the deployment model you can live with, the ecosystem you can hire into, and the abstraction level that matches your team's seniority. A fast prototype on LangChain + Chroma and a production system on LlamaIndex + Weaviate are not really different stacks; they are different points on the same maturity curve.

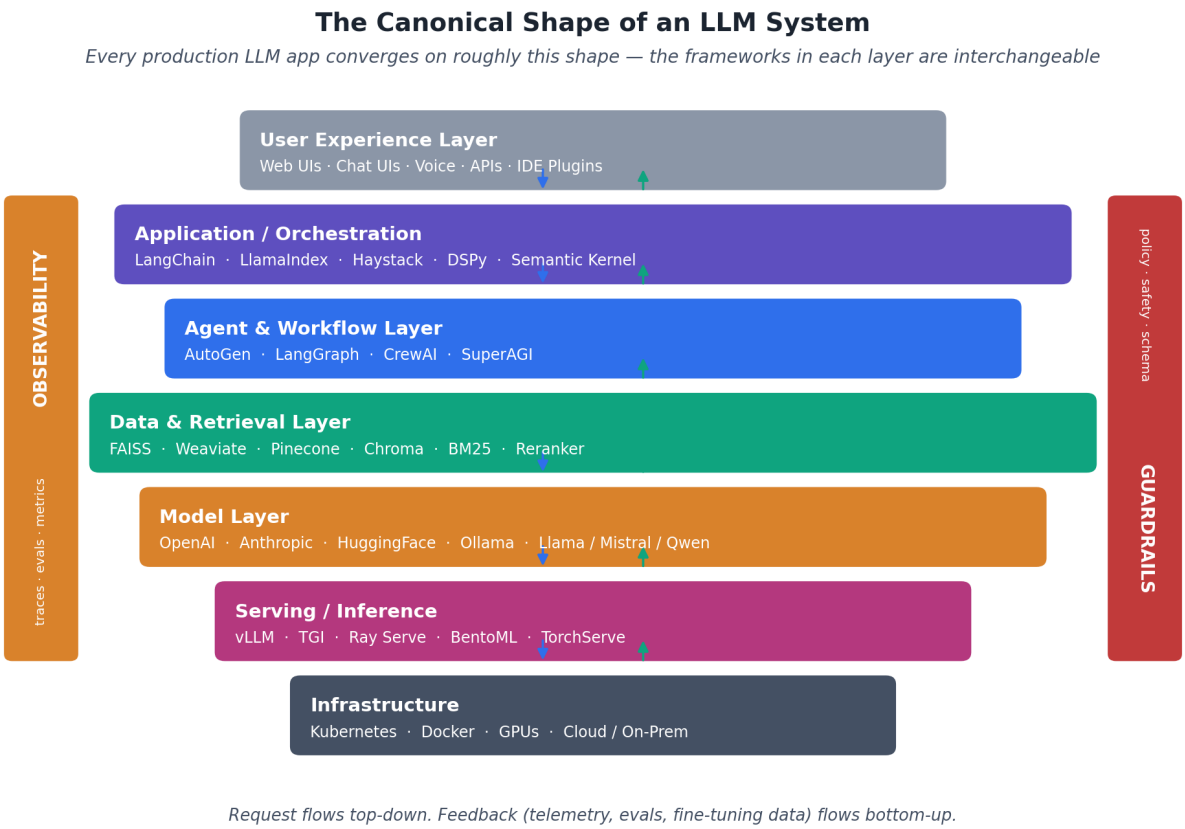


Figure 1 · The canonical shape of a production LLM system. Every layer is populated by interchangeable frameworks; the shape is what you commit to, not the boxes you pick.

What has changed in 2026

- **Agents moved from research to production.** LangGraph's state-machine model and AutoGen's conversation protocols have eclipsed the open-loop agents of 2023.
- **Evaluation is now a first-class layer.** You cannot ship an LLM system without Langfuse, LangSmith, or equivalent. The *silent failure* mode of naive RAG has made observability the single highest-ROI investment.
- **Fine-tuning is cheap again.** QLoRA, Unsloth, and per-request adapters have made domain-specialized models tractable without a training team.
- **Guardrails are no longer optional.** Prompt injection and data exfiltration are the new SQL injection; every production system needs an input/output guard.
- **Local inference is a live option.** Ollama + vLLM + Llama-3-class models make on-prem AI a realistic alternative for regulated industries.

How to read this atlas

Part I sets the big picture. Part II walks every layer of the stack, two-plus pages per category, with each framework catalogued against a standardized column set. Part III pulls the layers back together through

heuristics, heatmaps, and trade-offs. Part IV gives you four reference architectures you can lift wholesale. Part V is the outlook, glossary, and further reading. Skip around — the atlas is designed to be consulted, not read linearly.

> **Practitioner's note**
> Frameworks age faster than architectures. The stack shape you design today will still be valid in three years; the individual framework boxes you fill it with probably will not. Optimize for swappability at each layer, and you will age gracefully.

PART I

The Big Picture

A layered view of how modern AI systems are composed — and how to read the rest of this atlas.

1. The Canonical AI Stack

An AI system is not a monolith. Every non-trivial production deployment — whether a customer support chatbot, a document search engine, or a multi-agent workflow automation — is a composition of six or seven layers, each with a well-defined responsibility. The frameworks we catalog in this atlas each specialize in one or two of those layers; very few are credible at more than three.

The diagram below shows the canonical shape. Requests flow down from the application into the infrastructure; responses flow back up. Every framework in this document sits at one of these layers. Your job as an architect is to pick one framework per layer — and to make those choices independently enough that you can replace any single one without rewriting the rest.

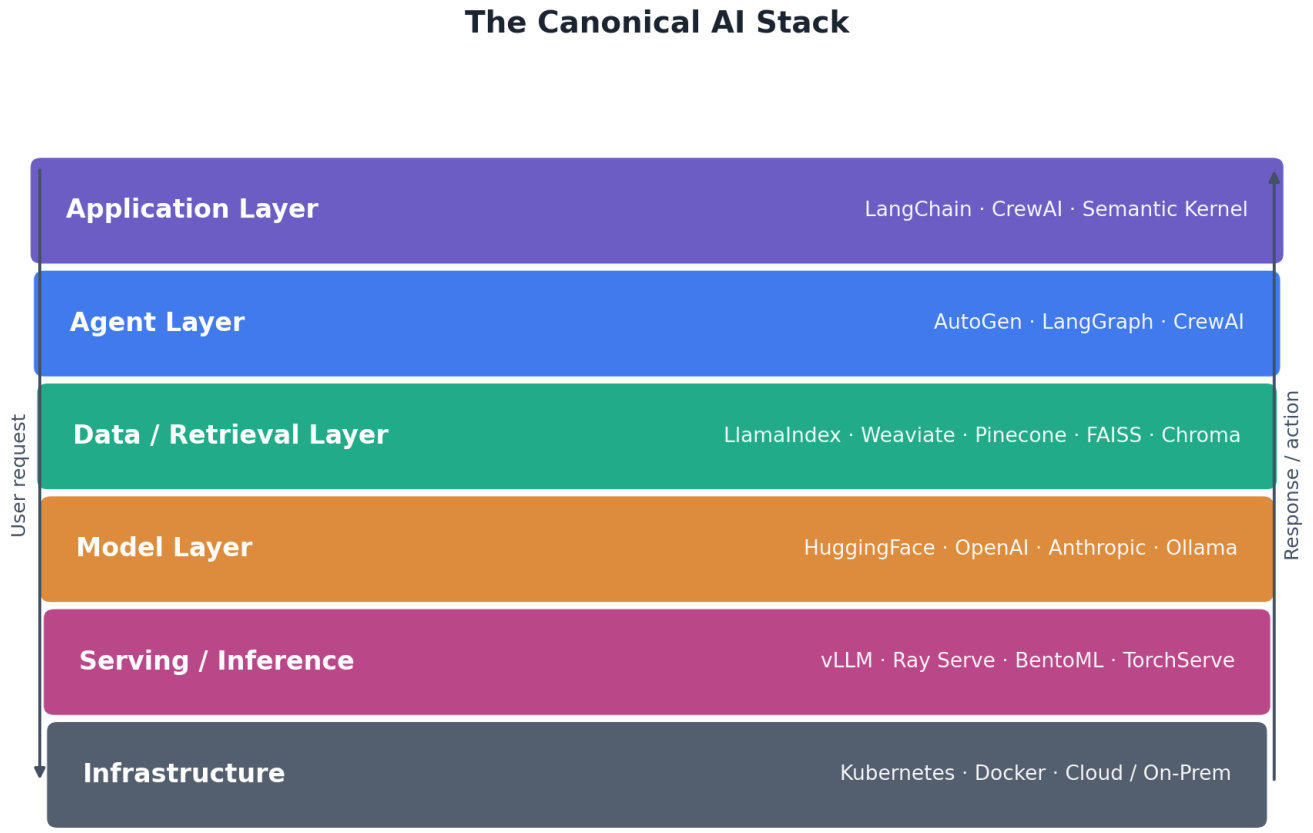


Figure 2 · The canonical AI stack. Each layer is a well-posed problem with its own framework family.

The six layers, briefly

Application layer. The surface your user interacts with — chat, API, internal tool. The framework here composes prompts, tools, and memory into coherent behavior. LangChain is the default; Semantic Kernel is

the Microsoft-native choice; CrewAI dominates when the app itself is agentic.

Agent layer. When a single LLM call is not enough — when the system needs to plan, call tools, critique itself, or coordinate among multiple specialized agents — this layer provides the loop. LangGraph is the most production-grounded; AutoGen is the most expressive; CrewAI is the easiest to reason about.

Data / retrieval layer. The memory of your system. Where long-term knowledge lives, how it is chunked, embedded, and fetched at query time. LlamaIndex leads on the orchestration side; Weaviate, Pinecone, FAISS, and Chroma compete at the storage layer, each tuned for a different operational profile.

Model layer. The foundation models themselves — OpenAI and Anthropic for frontier quality, Hugging Face for open weights, Ollama for local. This layer is increasingly commoditized, but swappability here is the single most important design invariant.

Serving / inference layer. How you turn a model into an endpoint. vLLM dominates throughput-bound workloads; Ray Serve scales horizontally; BentoML packages models into clean APIs; TorchServe and TF Serving are the framework-native defaults.

Infrastructure layer. Kubernetes, Docker, cloud, on-prem. This layer is out of scope for most of this atlas — but no framework choice is independent of the deployment target.

The three layers we added

The source document from which this atlas was built cataloged seven categories. Three more have earned their place since: **fine-tuning** (now cheap enough to be part of every serious system's playbook), **observability / evaluation** (without which LLM systems fail silently), and **guardrails** (prompt injection and data exfiltration are table-stakes concerns now).

> **Invariant**
> A mature production AI system touches every layer. If your system does not have an answer for each — even if that answer is 'we use the defaults' — it is under-designed. Every layer you ignore eventually becomes the layer you cannot debug.

2. How to Read This Atlas

Every framework in Part II is profiled against an identical set of eleven columns. The point of this uniform treatment is to make cross-layer comparisons legal: maturity in vector databases and maturity in agent frameworks mean the same thing, so you can compare risk across the stack.

The standard column set

For every framework you will see these attributes. Section 3 defines each in depth; here is the at-a-glance shape:

- **Framework** — canonical name as used in the community.
- **Language** — primary implementation languages and/or the language of the API surface the user writes in.
- **Abstraction Level** — High / Medium / Low; how opinionated the defaults are, and how much code you write to get a result.
- **Architecture Type** — the framework's structural model (pipeline, chain, graph, plugin, DB+API, etc.).
- **Typical Use Cases** — what the framework is most commonly reached for in production.
- **Strengths** — where the framework genuinely excels relative to its peers.
- **Weaknesses** — where it will slow you down or force workarounds.
- **Deployment Model** — cloud, local, on-prem, edge, or SaaS — and whether each is first-class.
- **Maturity** — Low / Medium / High / Very High — a composite of age, stability, and production footprint.
- **Learning Curve** — Low / Medium / High — expected time-to-first-productive-use for a competent engineer.
- **Ecosystem Fit** — the slot this framework naturally occupies in a larger stack.

How the atlas is organized

Each category section in Part II follows the same rhythm: a one-paragraph positioning of the category, a fan diagram showing the frameworks that compete in it, the standardized comparison table, a per-framework deep-dive paragraph, and a brief "when to reach for what" heuristic. Skim the diagram and table; read the deep-dives only for the frameworks on your shortlist.

> **How to use this document**
> Treat it as a reference, not a tutorial. The sections are self-contained; jumping directly to (say) §12 on guardrails will not cost you context from §5 on orchestration. The reference architectures in Part IV are the payoff — start there if you already know what layer you are stuck on.

3. Column Glossary

What Every Attribute Really Means

The comparison tables in Part II are only useful if everyone reading them shares a definition of the columns. This section is the dictionary. Each attribute below is defined, operationalized (how we actually decide the value for a given framework), and annotated with the architectural questions it tends to surface.

Language

What it captures: the primary implementation language and the language(s) of the API surface the engineer writes against. These often differ — FAISS is a C++ library with a Python API, vLLM is Python with a Rust-accelerated core.

Why it matters: language is a proxy for hireability, tooling, and performance ceilings. A Python-only framework closes off the JVM, the .NET world, and low-latency C++ inference paths. A polyglot framework like Semantic Kernel (C# + Python) is one of the few ways to bring LLM features into an enterprise Java or .NET shop without a protocol bridge.

Operational signal: if your organization has zero Python engineers, frameworks scoring Python-only are effectively excluded regardless of their other merits.

Abstraction Level

What it captures: how much code you write — and how much you delegate — to get a working result. A *high* abstraction framework hands you a five-line chatbot; a *low* abstraction framework hands you tensors and expects you to assemble the rest.

Why it matters: abstraction trades speed for control. LangChain's high abstraction makes the first demo fast and the tenth production fix slow, because you are debugging through someone else's default decisions. FAISS's low abstraction costs more lines but yields fewer surprises at 3am.

Operational signal: match abstraction to team seniority. Junior teams over-value high abstraction; senior teams over-value low. A mixed team benefits from a *medium* default.

Architecture Type

What it captures: the structural metaphor the framework organizes work around. Chain, graph, pipeline, DB-plus-API, role-based, declarative, functional — each implies a different mental model and a different failure mode.

Why it matters: architecture type constrains what is easy to express. LangGraph's state-machine architecture makes deterministic retries trivial and free-form branching expensive. CrewAI's role-based architecture makes delegation natural and fine-grained flow control hard.

Operational signal: pick the architecture type that matches your domain's natural structure. If your problem is inherently a pipeline (ETL, RAG ingest), use a pipeline-native framework; if it is a negotiation among specialists, use a role-based one.

Typical Use Cases

What it captures: what the framework is most commonly used for in production today, not what it *could* do in principle. Every framework can technically do RAG; only a few are actually good at it.

Why it matters: frameworks mature along their use-case frontier. The weight of production deployments at a given use case is the single best predictor of how many of your bugs are already fixed. Using a framework outside its typical use cases means you are the beta tester.

Strengths / Weaknesses

What it captures: opinionated, relative judgments — not absolute. PyTorch's "weak native prod tooling" weakness is relative to TensorFlow's TF Serving, not an absolute claim. SuperAGI's "immature" weakness is relative to LangGraph, not to a greenfield project.

Why it matters: these cells are the closest the atlas gets to vendor advocacy. They reflect the author's working experience and the consensus of production practitioners. Treat them as priors, not as axioms.

Deployment Model

What it captures: where the framework can actually run. Options: *Cloud*, *On-prem*, *Local*, *Edge*, *Mobile*, *SaaS-only*. Many frameworks are technically deployable in all of these but operationally painful in most.

Why it matters: this column is a compliance gate. A healthcare or finance deployment that cannot leave the enterprise VPC immediately eliminates SaaS-only frameworks like Pinecone, regardless of other merits. A mobile product eliminates everything that cannot run inside 50MB of RAM.

Operational signal: lead with this column when shortlisting. It is the one attribute that cannot be worked around with more engineering effort.

Maturity

What it captures: a composite of project age, API stability, number of known production deployments at scale, and responsiveness of the maintainer community. A *Very High* maturity framework has been running in production at multiple Fortune 500 companies for three or more years without a breaking API change in the core.

Why it matters: maturity is a proxy for risk. Immature frameworks change their APIs, lose maintainers, and ship bugs that you will discover in production. Every notch down in maturity is an order of magnitude more on-call time.

Learning Curve

What it captures: expected time for a competent engineer (2+ years of Python, familiarity with web services) to reach productive output on the framework. *Low* means a day or two; *Medium* means a couple of weeks;

High means a month or more of serious investment before shipping.

Why it matters: this is the column architects under-weight most often. The difference between a *Low* and *High* learning curve, multiplied by team size, is often larger than any performance benefit the higher-curve framework delivers.

Ecosystem Fit

What it captures: the slot this framework naturally occupies in a larger stack — and by implication, what other frameworks it plays well with. "Default LLM stack" (LangChain), "Research + LLM" (PyTorch), "Enterprise RAG" (LlamaIndex + Weaviate), etc.

Why it matters: frameworks are not chosen in isolation. Choosing LlamaIndex pulls in a set of reasonable defaults (OpenAI embeddings, Weaviate or Pinecone downstream) that save you from litigating every decision. Fighting the ecosystem is one of the most expensive mistakes in framework selection — it looks cheap in the decision meeting and bleeds for months afterward.

> **Reading the columns together**
> No single column is dispositive. **Maturity × Learning Curve × Deployment Model** — the three hardest columns to work around — usually determine the shortlist. The rest determine the winner.

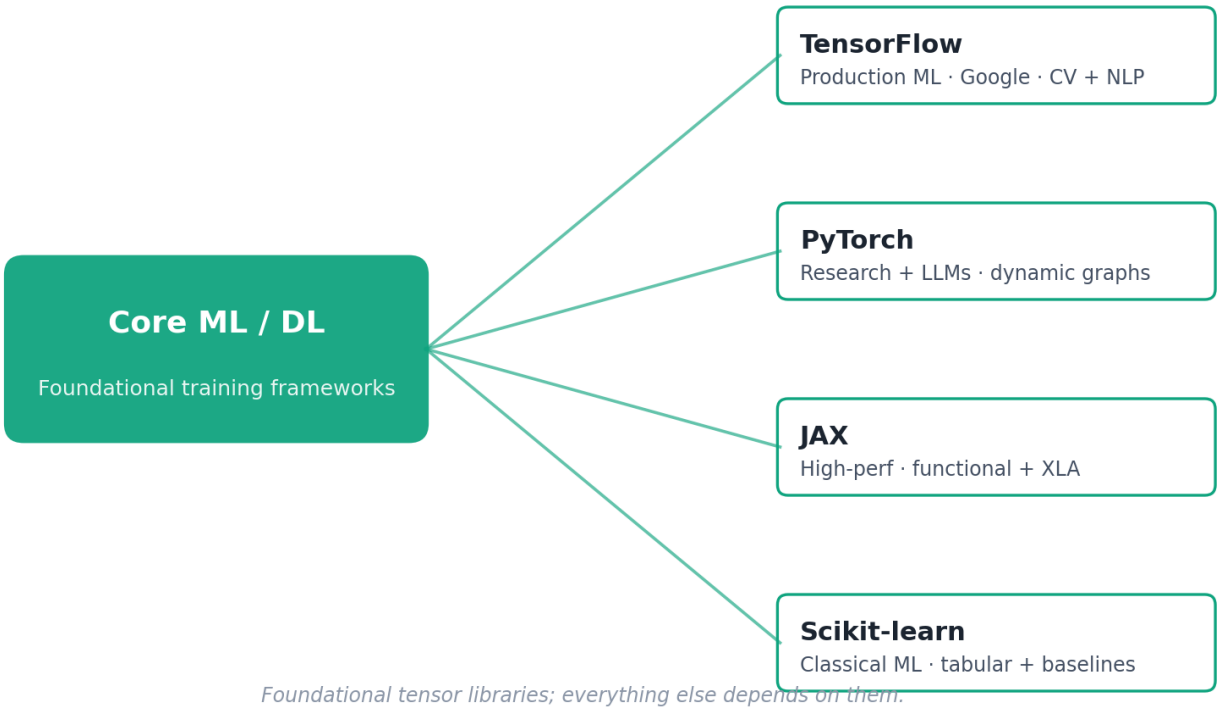
PART II

Framework Deep-Dives

Every layer of the stack, one section at a time. Each framework catalogued against the same eleven columns, with deep-dives on production trade-offs.

4. Core ML / Deep Learning

Every other framework in this atlas ultimately reduces to tensor operations on one of four libraries. TensorFlow and PyTorch are the dominant general-purpose frameworks; JAX is the high-performance outlier; Scikit-learn remains the unkilled champion of classical ML on tabular data. You do not reach for these frameworks directly unless you are training, fine-tuning, or implementing a novel architecture — but the framework your LLM was trained in dictates much of what you can do downstream.



Foundational tensor libraries; everything else depends on them.

Figure 3 · Core ML / DL training frameworks.

Comparison

Framework	Language	Abstraction	Architecture	Use cases	Strengths	Weaknesses	Maturity
TensorFlow	Python, C++	High + Low	Static + eager	Production ML, CV, NLP	Scalable; TFLite, TF Serving	Verbose; complex debugging	Very High
PyTorch	Python	Medium	Dynamic graph	Research, LLMs	Flexible, intuitive	Weak native prod tooling	Very High

Framework	Language	Abstraction	Architecture	Use cases	Strengths	Weaknesses	Maturity
JAX	Python	Low	Functional + XLA	High-perf ML, transformers	Speed, composability	Niche; debugging harder	Medium
Scikit-learn	Python	High	Classical ML	Regression, clustering	Simple, stable	No deep learning	Very High

Framework deep-dives

TensorFlow

Google's production workhorse. If your organization already deploys TF models behind TF Serving or on TFLite mobile runtimes, the momentum argument is decisive. The flip side: TF's eager mode narrowed the ergonomic gap with PyTorch, but debugging a Keras model that crosses into tf.function territory remains the worst developer experience in this atlas. Reach for TF when production maturity and deployment breadth (edge, mobile, browser) matter more than research velocity.

PyTorch

The default tensor library of 2026. Every major LLM (Llama, Mistral, DeepSeek) is trained in PyTorch, and the Hugging Face ecosystem is PyTorch-native. Its weakness — historically weak production tooling — is largely solved by torch.compile, TorchServe, and integration layers like BentoML and Ray Serve. Use PyTorch unless you have a specific reason not to.

JAX

The performance-first library. Google-internal for much of its use; externally dominant in transformer research (Pallas, Flax) and scientific ML. The programming model (pure functions, explicit PRNG, vmap/pmap) is a culture shock for engineers trained on PyTorch; debugging compiled XLA code is its own discipline. Reach for JAX when you need TPU-class throughput or are doing custom transformer research.

Scikit-learn

Still the first library you install for any tabular ML problem. No deep learning, but for a regression, clustering, or classification baseline on structured data, nothing else is this fast from zero to working model. Under-used as a sanity check against over-engineered LLM solutions: if your problem fits in a gradient-boosted tree, it does not need an LLM.

> **Practitioner heuristic**
> Start every production LLM project in PyTorch, every research exploration in JAX, every mature enterprise deployment in TensorFlow, and every tabular data problem in Scikit-learn. In 2026, the crossover points between these decisions are clearer than they have been in years.

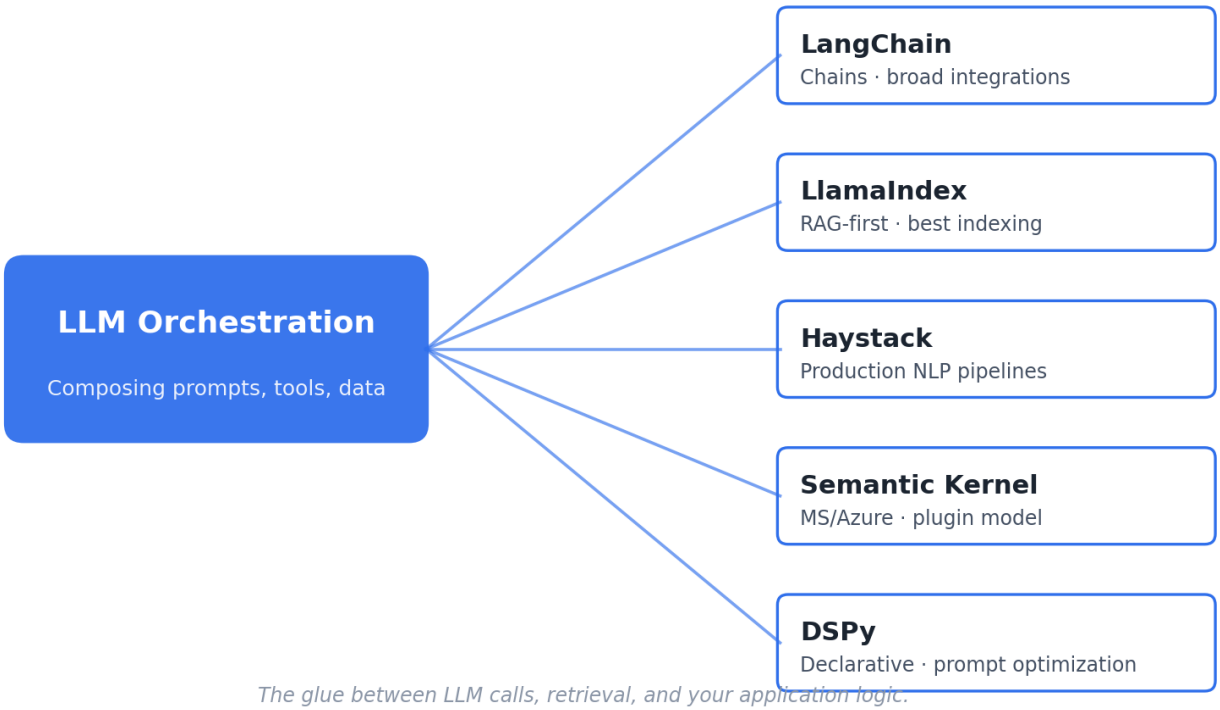
References & further reading

1. TensorFlow official docs — tensorflow.org
2. PyTorch docs and tutorials — pytorch.org/docs
3. JAX — Bradbury et al., 2018. github.com/google/jax

4. Scikit-learn: Machine Learning in Python — Pedregosa et al., JMLR 2011. scikit-learn.org
5. PyTorch 2.x torch.compile — pytorch.org/docs/stable/torch.compiler

5. LLM Orchestration

The glue layer. When a system needs more than a single LLM call — when it needs to inject context, call tools, branch on outputs, or chain multiple models — this layer is where that logic lives. The stakes here are high: the orchestration framework you pick dictates your debuggability, your cost profile, and the half-life of your code.



The glue between LLM calls, retrieval, and your application logic.

Figure 4 · LLM orchestration frameworks.

Comparison

Framework	Language	Abstraction	Architecture	Use cases	Strengths	Weaknesses	Maturity
LangChain	Python, JS	High	Chain-based	Chatbots, RAG, tools	Integrations, modular	Over-abstraction	High
LlamaIndex	Python	Medium	Data-centric	RAG, doc search	Best indexing layer	Limited agent logic	High
Haystack	Python	Medium	Pipeline-based	Search, QA	Production-ready	Heavy infra	High
Semantic Kernel	C#, Python	Medium	Plugin-based	Copilots	Enterprise integration	MS ecosystem bias	Medium

Framework	Language	Abstraction	Architecture	Use cases	Strengths	Weaknesses	Maturity
DSPy	Python	Medium	Declarative	Prompt optimization	Structured prompting	New paradigm	Medium

Framework deep-dives

LangChain

The default LLM framework and the most criticized one. Its strength is breadth — hundreds of integrations to vector DBs, tool providers, and model APIs; its weakness is that the abstraction cost compounds. A LangChain app is fast to prototype and, by the third month, slow to debug because you are reading chains of wrapper classes to understand why a prompt changed. Use LangChain for MVPs and any system where LangGraph (its state-machine sibling) will eventually take over the agent logic.

LlamaIndex

Where LangChain tries to be everything, LlamaIndex is unapologetically a RAG-first framework. Its indexing abstractions — hierarchical, router, auto-retrieval — are the most thoughtful in the ecosystem, and its integrations with Weaviate, Pinecone, and ingestion tools are tighter than LangChain's equivalents. Reach for LlamaIndex when retrieval quality is the axis you are optimizing.

Haystack

The production NLP pipeline framework. Deepset built it for industrial search and QA at enterprises that predated the LangChain wave. The component model is clean, the production story is real (Docker + REST + monitoring out of the box), but the infra footprint is heavy. Choose Haystack when your team has strong backend engineering discipline and you are embedding LLMs into an existing search product.

Semantic Kernel

Microsoft's answer to LangChain, built for the .NET and Python ecosystems with native Azure integration. Its plugin model is enterprise-friendly: declarative function schemas, built-in auth, telemetry through the Microsoft observability stack. The cost is ecosystem lock-in — SK outside Azure is technically possible but culturally awkward. The right call for Microsoft-shop copilots; the wrong call for a polyglot cloud.

DSPy

A declarative approach to prompting. You describe the transformation (Signature) and the orchestration style (Module), and DSPy's optimizer searches for the prompts that achieve it. The paradigm shift is real — DSPy programs read like type systems rather than string templates — and the structured approach scales better to complex pipelines. The cost is novelty: your team will spend weeks unlearning prompt-as-string habits. Use DSPy when prompt robustness across models is your top risk.

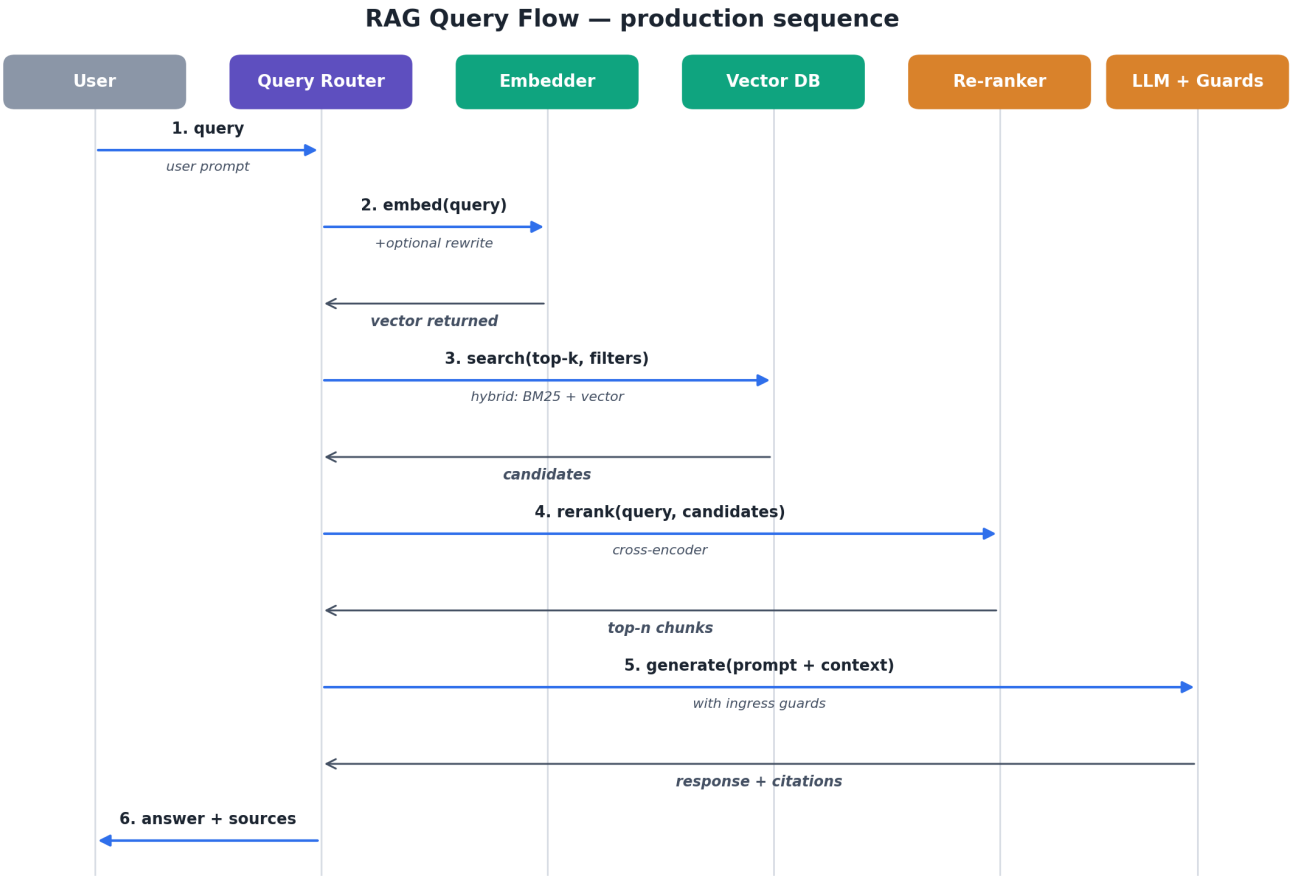


Figure 4a · The RAG query flow — how an orchestrator sequences embedding, retrieval, reranking, and generation in a production system.

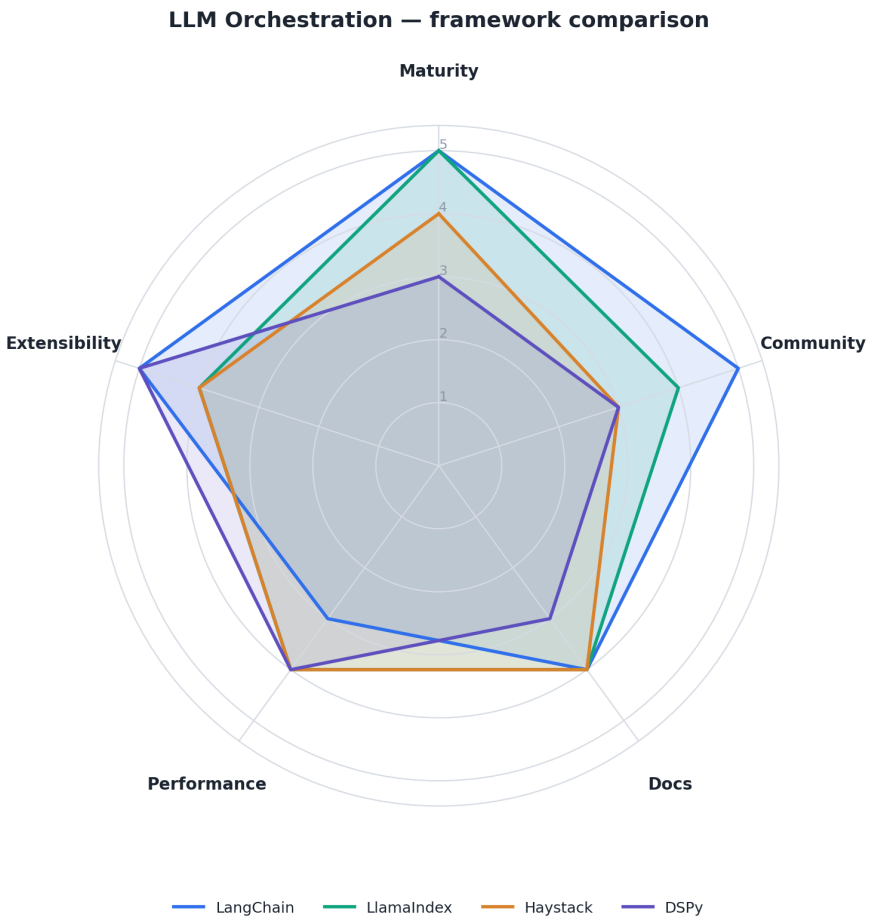


Figure 4b · Orchestration frameworks compared on maturity, community, docs, performance, and extensibility (1–5 scale).

Practitioner heuristic

For a fast prototype: LangChain. For serious RAG: LlamaIndex. For existing Python search systems: Haystack. For Microsoft shops: Semantic Kernel. For prompt stability across model upgrades: DSPy. The right answer is often to pick LangChain for the application layer and LlamaIndex for the retrieval layer; they compose cleanly and each plays to its strength.

References & further reading

1. LangChain documentation — python.langchain.com
2. LlamaIndex documentation — docs.llamaindex.ai
3. Haystack documentation (deepset) — haystack.deepset.ai
4. Semantic Kernel — learn.microsoft.com/semantic-kernel
5. DSPy: Compiling Declarative Language Model Calls — Khattab et al., 2023. arXiv:2310.03714
6. Retrieval-Augmented Generation — Lewis et al., NeurIPS 2020. arXiv:2005.11401

6. Agent Frameworks

An agent is a loop: the LLM decides, a tool is called, the result feeds the next decision. Agent frameworks codify that loop — how state is carried, how tools are invoked, how multi-agent conversations are coordinated, how failures are retried. In 2026, autonomous agents have moved from research demos into targeted production use (triage, routing, research synthesis); they have not displaced deterministic pipelines for most work. Choose the right framework for the scope of autonomy you are willing to grant.

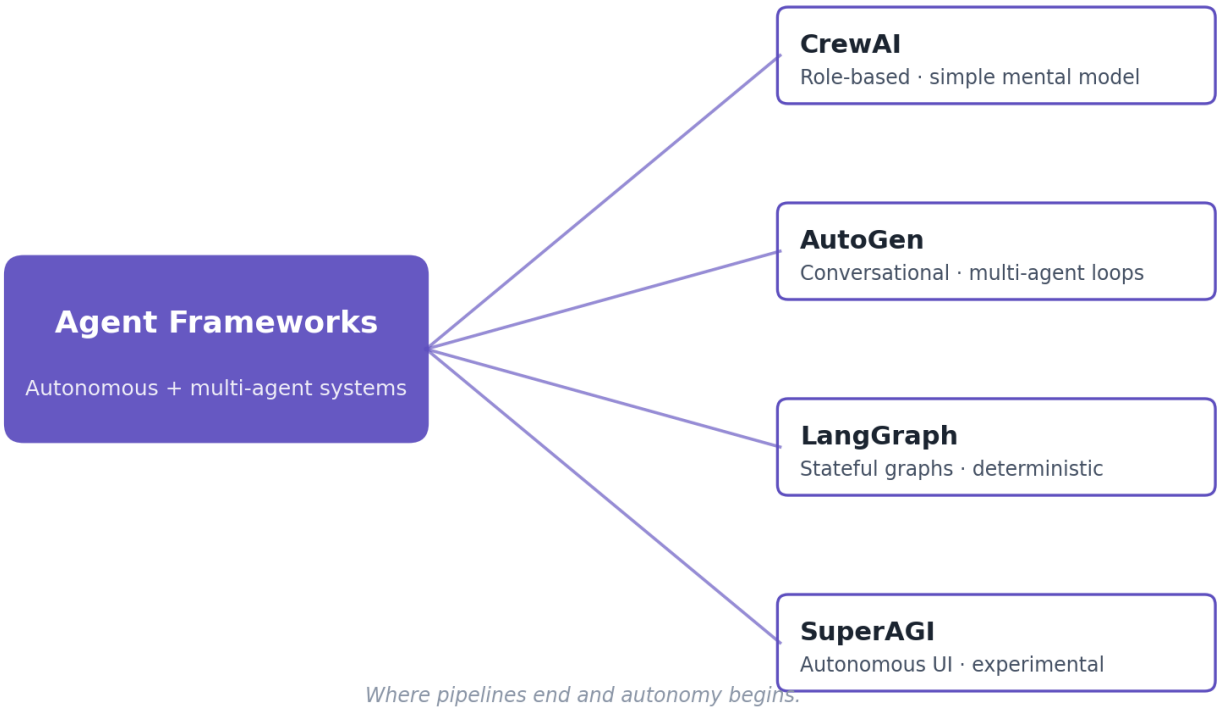


Figure 5 · Agent frameworks across the autonomy spectrum.

Comparison

Framework	Language	Abstraction	Architecture	Use cases	Strengths	Weaknesses	Maturity
CrewAI	Python	High	Role-based agents	Task automation	Simple mental model	Limited control	Medium
AutoGen	Python	Medium	Conversational agents	Multi-agent systems	Powerful coordination	Complex tuning	Medium
LangGraph	Python	Medium	Graph / state machine	Stateful agents	Deterministic control	LangChain dependency	High

Framework	Language	Abstraction	Architecture	Use cases	Strengths	Weaknesses	Maturity
SuperAGI	Python	High	Autonomous agents	Autonomous workflows	UI + automation	Immature	Low

Framework deep-dives

CrewAI

The role-based framework: you declare agents with personas ("Researcher", "Editor", "Critic") and CrewAI orchestrates the collaboration. The mental model is accessible to non-ML engineers and surprisingly productive for document workflows, research synthesis, and content pipelines. The cost is control: fine-grained flow branching, retry policies, and state machines are not its native idiom. Best for mid-autonomy, document-shaped problems.

AutoGen

Microsoft Research's contribution; models multi-agent systems as conversations. Each agent is a participant; the coordination layer routes messages and enforces termination. The expressiveness is unmatched — you can model critique-revise loops, planner-executor separations, or committee-of-experts voting in a few dozen lines. The cost is tuning: conversations can spiral, token budgets can explode, and the debugging story is still maturing. Use AutoGen when you need genuine agent-to-agent negotiation.

LangGraph

The production-grounded choice. LangGraph models an agentic system as a stateful graph: nodes are functions (often LLM calls), edges are conditional transitions, state is an explicit typed object. The explicit state model is what production debugging demands: every step is inspectable, every transition is replayable, every branch is testable. LangGraph has displaced LangChain's original agent primitives and is the right default for any agentic system headed to production.

SuperAGI

An autonomous agent with a UI, marketplaces for plugins, and ambitions toward an "operating system" for agents. Fascinating to demo; still immature for production. The right framing: SuperAGI and its peers (Agent OS, OpenInterpreter) are the research frontier, not the current production surface.

Figure 5a · The canonical agent loop — Plan, Act, Observe, Reflect. Every production agent framework is an opinionated implementation of this cycle.

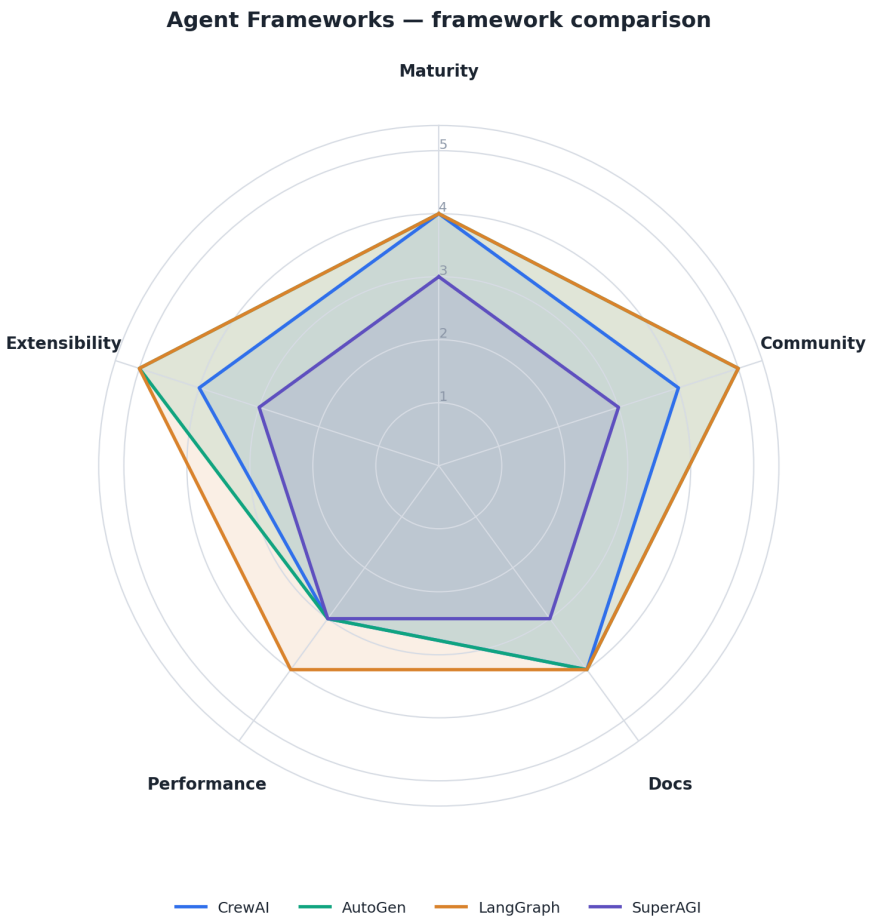


Figure 5b · Agent frameworks compared on maturity, community, docs, performance, and extensibility.

Practitioner heuristic

For production agents, LangGraph. For multi-agent conversations, AutoGen. For lightweight document workflows, CrewAI. Avoid autonomous agent frameworks in production unless the failure cost is bounded (e.g., the agent drafts but a human approves).

References & further reading

1. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation — Wu et al., Microsoft 2023. arXiv:2308.08155
2. LangGraph documentation — langchain-ai.github.io/langgraph
3. CrewAI documentation — docs.crewai.com
4. ReAct: Synergizing Reasoning and Acting — Yao et al., 2022. arXiv:2210.03629
5. Reflexion: Language Agents with Verbal Reinforcement Learning — Shinn et al., 2023. arXiv:2303.11366
6. SuperAGI — superagi.com

7. Vector DB / Retrieval

The memory layer. When an LLM needs to know something that was not in its training data — your company's policies, your customers' past tickets, yesterday's product updates — a vector database is how that knowledge is retrieved. The choice here is dominated by operational profile: how much data, how fast, from where. There is no universal winner; the right call depends on your scale and your willingness to run infrastructure.

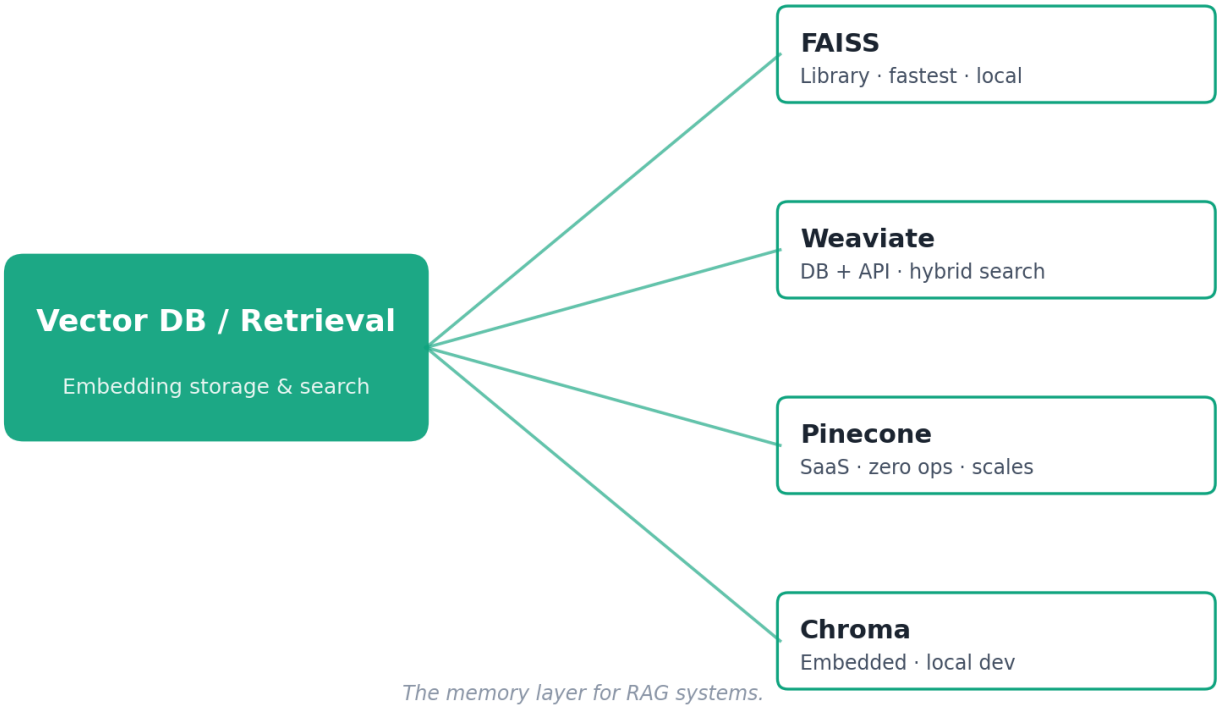


Figure 6 · Vector databases and retrieval libraries.

Comparison

Framework	Language	Abstraction	Architecture	Use cases	Strengths	Weaknesses	Maturity
FAISS	C++, Python	Low	Library	Similarity search	Fastest	No server layer	Very High
Weaviate	Go	High	DB + API	RAG, semantic search	Built-in ML	Ops overhead	High
Pinecone	Managed	High	SaaS DB	Scalable RAG	Zero ops	Cost	High

Framework	Language	Abstraction	Architecture	Use cases	Strengths	Weaknesses	Maturity
Chroma	Python	High	Embedded DB	Local apps	Lightweight	Not scalable	Medium

Framework deep-dives

FAISS

Facebook's similarity search library, and the underlying engine inside most higher-level vector databases. Not a database — no server, no multi-tenancy, no replication — just the fastest approximate nearest-neighbor index you can get. Reach for FAISS directly when you are building a vector DB yourself, embedding search into a local tool, or operating at sub-10M-vector scale where a single machine suffices.

Weaviate

The open-source vector database that treats ML as a first-class citizen: native modules for text vectorizers (OpenAI, Cohere, Hugging Face), built-in hybrid search (BM25 + vector), and a production-grounded Go implementation. Ops overhead is real (you run the cluster) but the self-hosted model is a compliance unlock for regulated industries. The default choice for enterprise RAG where SaaS is not viable.

Pinecone

The zero-ops SaaS vector database. Instant provisioning, pod-based scaling, and a focus on operational simplicity that makes it the fastest path from prototype to production. The cost model is the catch: at high volume Pinecone gets expensive relative to self-hosted Weaviate, and the SaaS posture is a non-starter for data-sovereignty-sensitive deployments. The right call for startups and teams where engineer-hours cost more than SaaS bills.

Chroma

The embedded vector database for local development and small-scale production. Runs in-process, persists to disk, has a Pythonic API. Its strength is frictionlessness; its weakness is that it does not scale horizontally and was never meant to. Use Chroma for development environments, demos, and on-device AI; graduate to Weaviate or Pinecone when you cross single-machine limits.

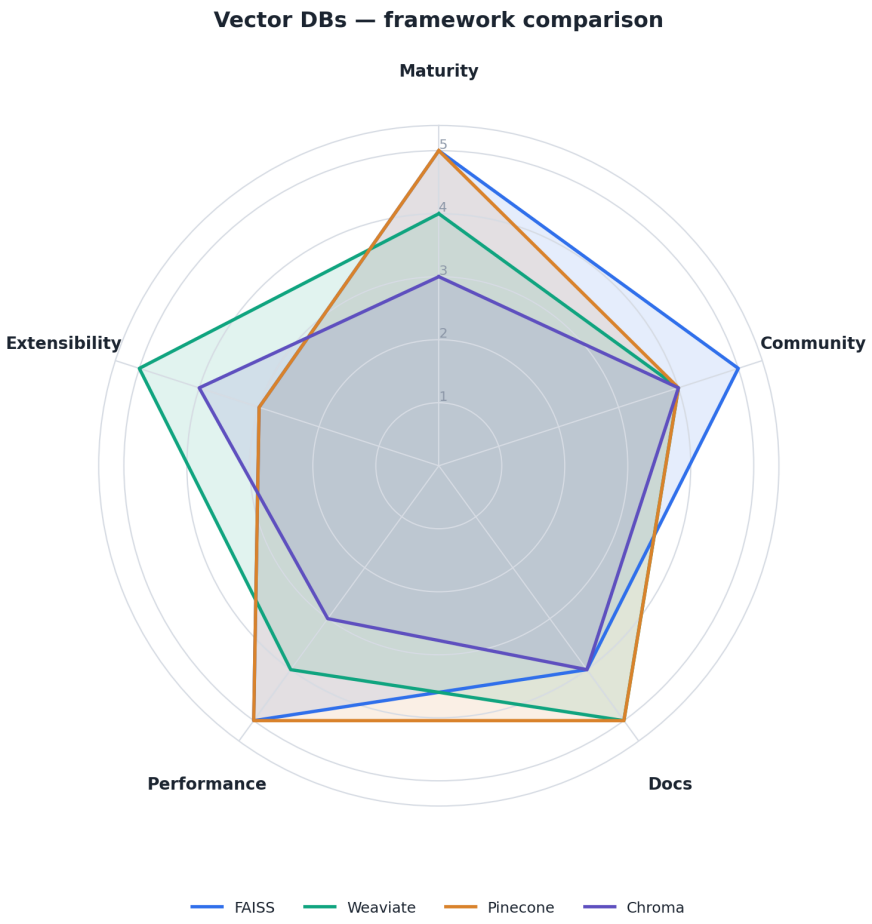


Figure 6a · Vector DBs compared on maturity, community, docs, performance, and extensibility.

Cost & latency envelope (order-of-magnitude)

Option	P50 search latency	Scale ceiling	$ / 1M vectors / month
FAISS (in-process)	1 – 5 ms	Single-machine (~100M)	~$0 (compute only)
Chroma (embedded)	5 – 20 ms	Single-machine (~10M)	~$0 (compute only)
Weaviate (self-host)	10 – 40 ms	Cluster, 1B+	~$200 – $800
Pinecone (SaaS, serverless)	15 – 60 ms	Cluster, 10B+	~$500 – $3,000
pgvector (Postgres)	20 – 80 ms	~100M	~$100 – $400

Practitioner heuristic
Prototype on Chroma (or in-process FAISS). Ship to Pinecone when engineer-hours are the bottleneck. Ship to Weaviate when data residency or cost dominates. FAISS directly only when you are building infrastructure.

References & further reading

1. FAISS: A Library for Efficient Similarity Search — Johnson, Douze, Jégou, 2017. arXiv:1702.08734
2. Weaviate documentation — weaviate.io/developers/weaviate
3. Pinecone documentation — docs.pinecone.io
4. Chroma documentation — docs.trychroma.com
5. pgvector extension — github.com/pgvector/pgvector
6. HNSW: Efficient and robust approximate nearest neighbor search — Malkov & Yashunin, 2016. arXiv:1603.09320
7. ANN-Benchmarks — ann-benchmarks.com (reference for the latency ranges above).

8. Model Serving / Inference

The last mile. A trained model becomes a product only when it is exposed as a low-latency, concurrent, observable endpoint. The serving layer is where your cost and latency bounds are actually determined; it is also where most teams discover that their model works but does not fit in the latency budget. Frameworks here differ in their optimization target: throughput, latency, flexibility, or operational simplicity.

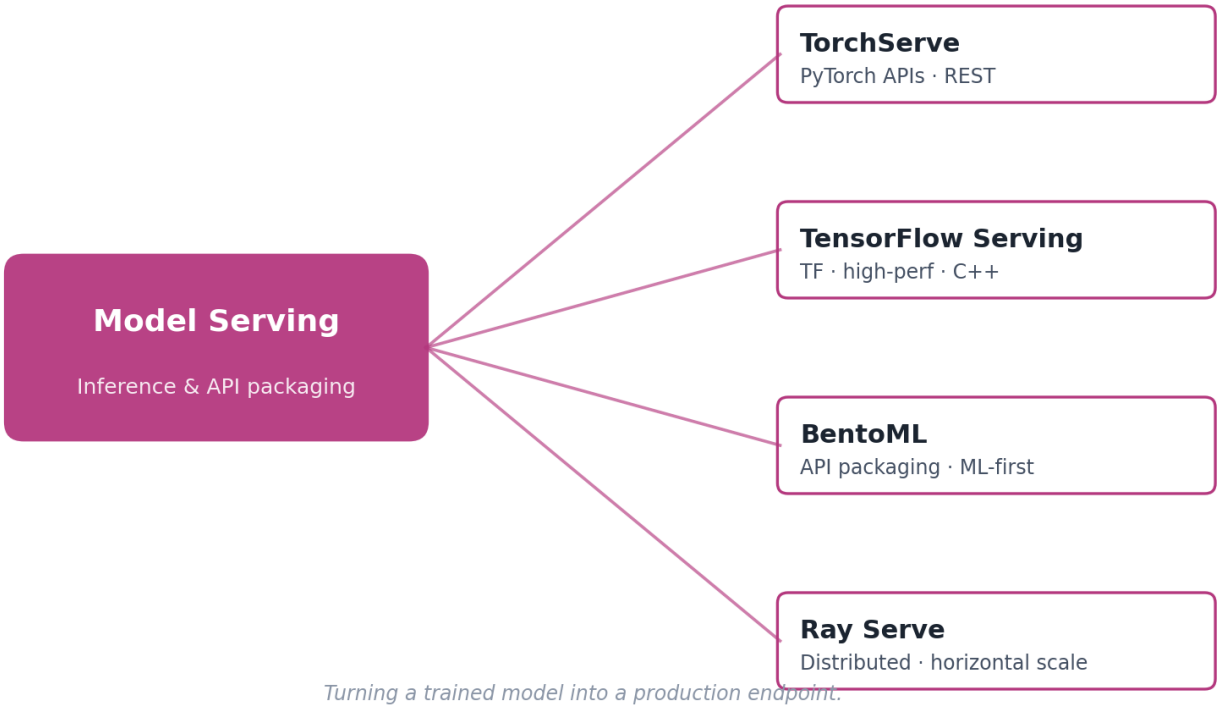


Figure 7 · Model serving and inference frameworks.

Comparison

Framework	Language	Abstraction	Architecture	Use cases	Strengths	Weaknesses	Maturity
TorchServe	Python	Medium	REST serving	PyTorch APIs	Easy integration	Limited features	Medium
TF Serving	C++	Low	High-perf serving	Production ML	Fast, robust	TF only	Very High
BentoML	Python	High	API packaging	Model APIs	Simple deploy	Smaller ecosystem	High
Ray Serve	Python	Medium	Distributed serving	Scalable inference	Horizontal scaling	Complexity	High

Framework deep-dives

TorchServe

PyTorch's native serving layer. REST endpoints, batching, versioning, management API. Everything you need to put a PyTorch model behind HTTP and not much more. The default when your organization has standardized on PyTorch and does not need horizontal scaling or LLM-specific optimizations.

TF Serving

Google's production serving stack for TensorFlow models. C++ at the core, battle-tested at internet scale, extremely robust — and strictly TensorFlow. If your model is TF-SavedModel, this is the serving system; if it isn't, look elsewhere.

BentoML

The modern API-packaging framework. Treats a served model as a Python class with decorators: dependencies are declared in a bento, the runtime is a Docker image, the deployment is a single CLI call. Plays well across PyTorch, TF, Scikit-learn, and custom runtimes. The right call for teams that want to think about models, not serving infrastructure.

Ray Serve

The distributed serving layer, built on Ray. Supports model composition (pipelines of models), horizontal scaling across GPUs, and dynamic batching. The operational cost is running a Ray cluster; the payoff is that you can serve a multi-model pipeline — embedder, retriever, reranker, generator — as a single coherent deployment. The right call at scale, especially when paired with vLLM for LLM workloads.

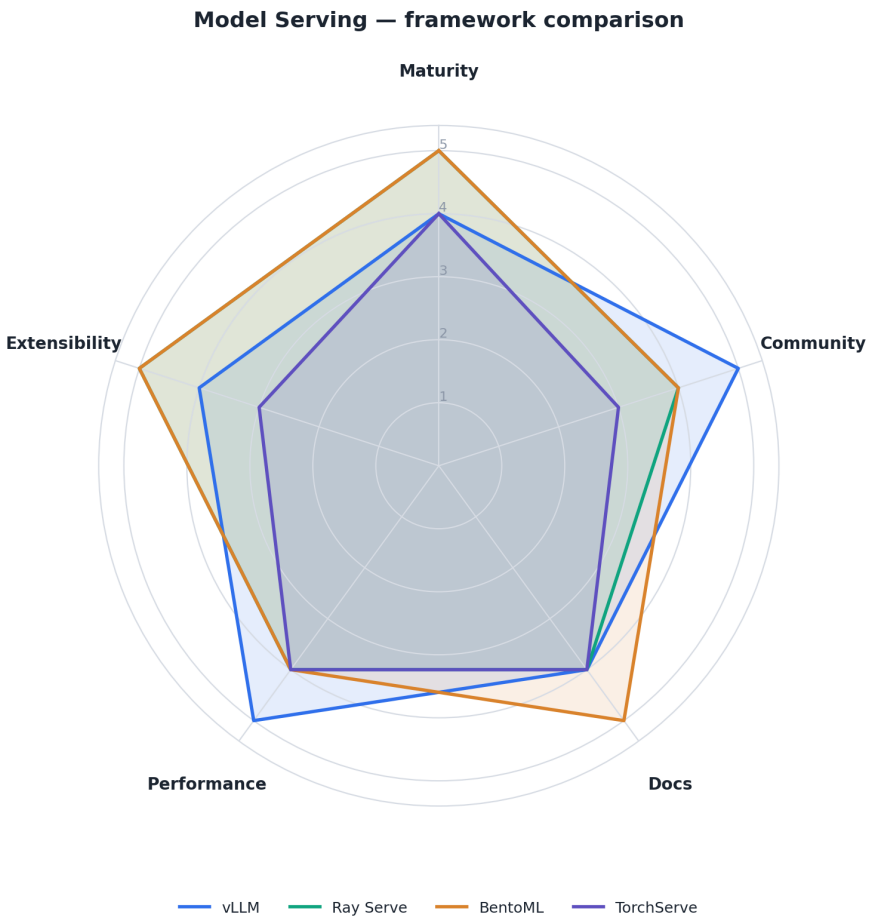


Figure 7a · Serving frameworks compared on maturity, community, docs, performance, and extensibility.

LLM serving — typical cost & latency

Runtime	7B on 1 × A10G	70B on 2 × H100	Best-fit use case
vLLM (PagedAttention)	~40 – 80 tok/s, P50 ~25 ms	~70 – 120 tok/s	High-throughput shared inference
TGI (HF)	~35 – 70 tok/s	~60 – 110 tok/s	Multi-model, OpenAI-compatible API
Ollama (llama.cpp)	~15 – 35 tok/s (CPU + GPU)	n/a (CPU-bound)	Local dev, on-device
TorchServe	~20 – 45 tok/s	~40 – 80 tok/s	Non-LLM workloads + PyTorch shops
Ray Serve + vLLM	scales linearly	scales linearly	Multi-model pipelines at scale

Practitioner heuristic

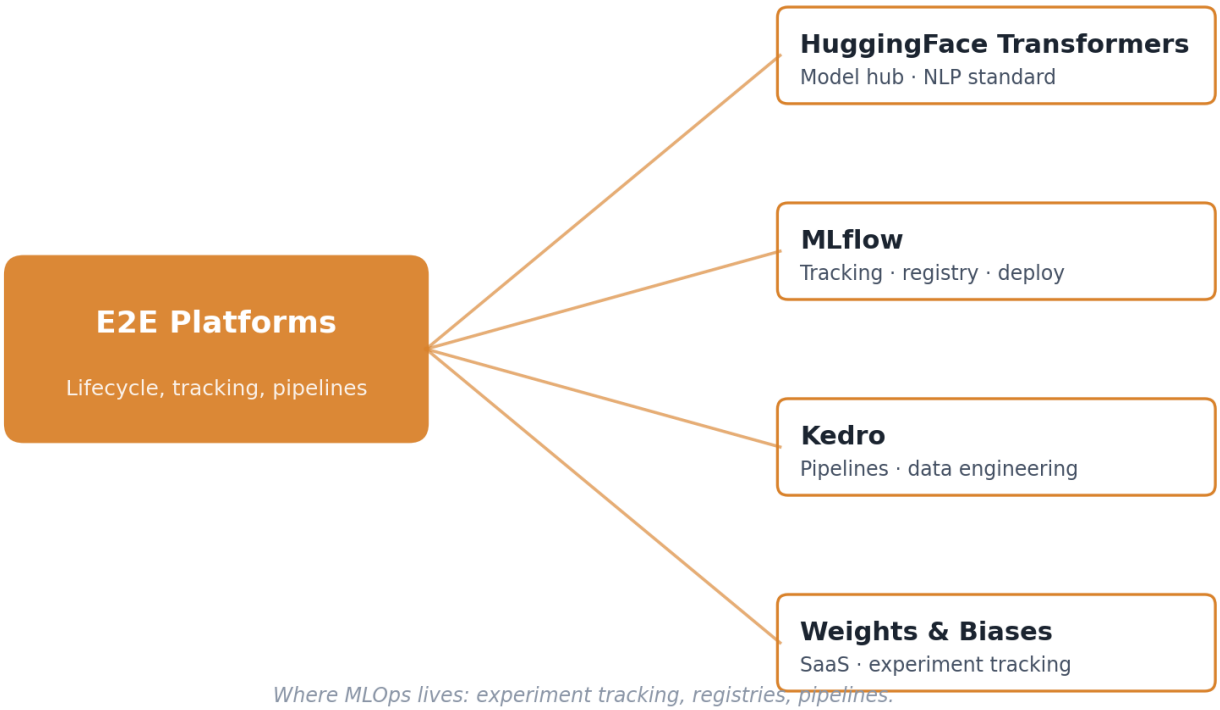
BentoML for simplicity; Ray Serve for scale; vLLM (see §13) specifically for LLM throughput. Use TorchServe or TF Serving only when ecosystem alignment outweighs flexibility.

References & further reading

1. vLLM: Easy, Fast, and Cheap LLM Serving with PagedAttention — Kwon et al., 2023. arXiv:2309.06180
2. Hugging Face TGI — github.com/huggingface/text-generation-inference
3. Ray Serve documentation — docs.ray.io/serve
4. BentoML documentation — docs.bentoml.com
5. TorchServe — pytorch.org/serve
6. Ollama — ollama.com
7. Latency figures are indicative and depend on sequence length, batch size, and quantization.

9. End-to-End Platforms

The MLOps layer. Once you have more than one model and more than one engineer, you need experiment tracking, model registry, pipeline orchestration, and artifact versioning. This is the layer that makes machine learning reproducible — which, left unsolved, is the layer that makes your ML unfalsifiable.



Where MLOps lives: experiment tracking, registries, pipelines.

Figure 8 · End-to-end MLOps platforms.

Comparison

Framework	Language	Abstraction	Architecture	Use cases	Strengths	Weaknesses	Maturity
HF Transformers	Python	Medium	Model hub	NLP, LLMs	Huge ecosystem	Heavy models	Very High
MLflow	Python	High	Lifecycle mgmt	Tracking, registry	End-to-end ML ops	UI limitations	Very High
Kedro	Python	Medium	Pipeline	Data workflows	Clean architecture	Setup overhead	High
Weights & Biases	Python	High	SaaS tracking	Experiment logging	Best UI	Paid scaling	Very High

Framework deep-dives

Hugging Face Transformers

Not a platform in the MLOps sense — it is the model hub and model-loading library that everything else integrates with. The de facto standard for open-weight LLMs. If your system loads a model by name, that name is a HF model ID. The weakness — models and their weights are large, downloads are slow — is mostly a function of the problem domain, not the framework.

MLflow

The open-source MLOps tooling that most enterprises end up with. Experiment tracking, a model registry, pluggable deployment targets, and a stable REST API that other tools integrate against. The UI is functional, not beautiful; the abstractions are generic enough to outlive specific frameworks. Reach for MLflow when you want vendor neutrality and self-hosted posture.

Kedro

QuantumBlack's data pipeline framework. Kedro's opinionated structure — nodes, pipelines, data catalogs, configuration — is an antidote to the notebook-to-production nightmare. Setup cost is real (you learn the framework before you ship your first pipeline), but the payoff is a codebase that scales to many engineers without becoming an archaeology project. The right call when data engineering hygiene matters more than velocity.

Weights & Biases

The SaaS-first experiment tracking tool with the best UI in the category. Integrations everywhere, powerful reports, collaborative features. The cost model pushes you to the paid tier quickly at production scale, and the SaaS posture is a blocker for some data-sensitive deployments. The best developer experience in this layer; whether it is worth the price is an organizational question.

> **Practitioner heuristic**
> Every team needs experiment tracking on day one: W&B if you can afford it and the SaaS posture works; MLflow if you cannot or it does not. Kedro when pipeline hygiene is a bottleneck. Hugging Face Transformers is not optional for any open-weight LLM work.

References & further reading

1. MLflow documentation — mlflow.org/docs
2. Kedro documentation — kedro.readthedocs.io
3. Weights & Biases documentation — docs.wandb.ai
4. Hugging Face Transformers — Wolf et al., 2020. arXiv:1910.03771; huggingface.co/docs/transformers
5. Hugging Face Hub — huggingface.co/docs/hub

10. Fine-Tuning & Training

The layer that returned from exile. In 2023, fine-tuning was expensive and fragile; by 2026, QLoRA and Unsloth have made it cheap enough to be part of every serious system's playbook. You fine-tune not to teach the model new facts (RAG does that better) but to shape its style, enforce structured outputs, and reduce your per-call token cost. The frameworks here differ mainly in how much training infrastructure they hide.

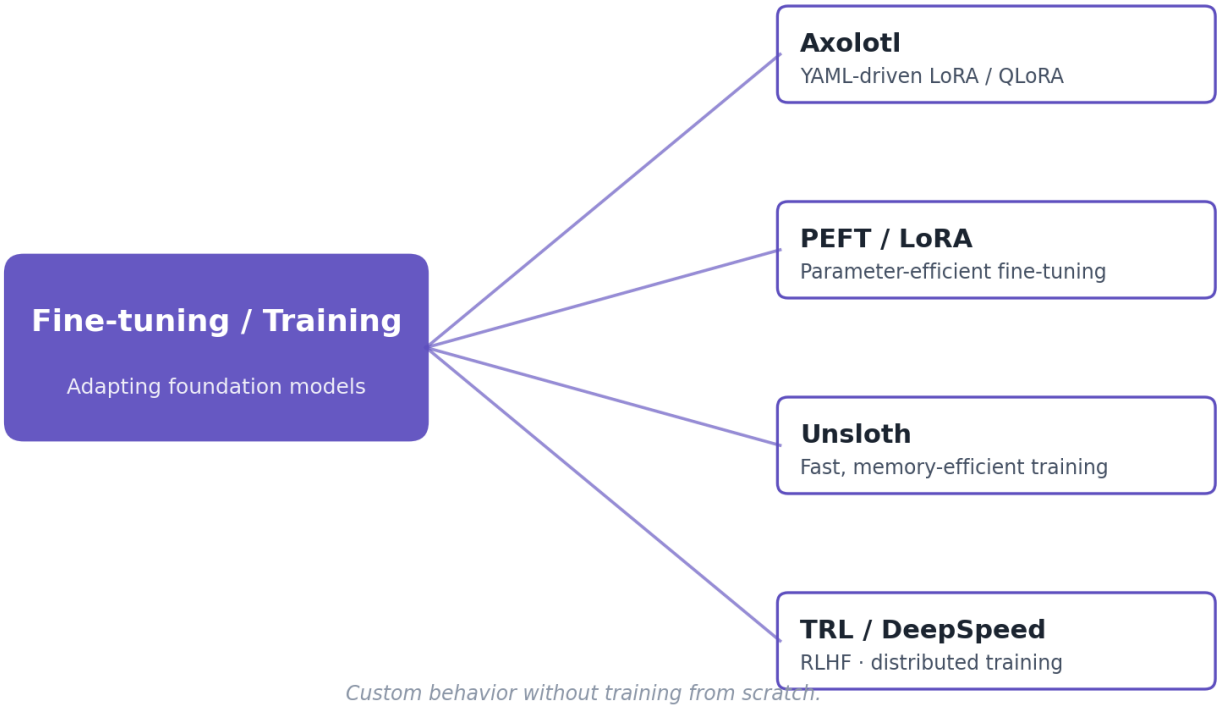


Figure 9 · Fine-tuning and training frameworks.

Comparison

Framework	Language	Abstraction	Architecture	Use cases	Strengths	Weaknesses	Maturity
Axolotl	Python	High	YAML-driven trainer	LoRA / QLoRA fine-tune	Opinionated, fast iteration	Less flexibility	High
PEFT / LoRA	Python	Medium	Hugging Face PEFT library	Parameter-efficient tuning	Small adapters, swappable	Full-fine-tune uncommon	High
Unsloth	Python	Medium	Kernel-optimized trainer	Fast LoRA on small GPUs	2-5× training speedup	Model support limited	Medium

Framework	Language	Abstraction	Architecture	Use cases	Strengths	Weaknesses	Maturity
TRL / DeepSpeed	Python	Low	RLHF / distributed	Alignment, large training	Full control	Steep learning curve	High

Framework deep-dives

Axolotl

YAML-driven LoRA fine-tuning. Point it at a dataset, specify a base model, declare your adapter rank, and train. Abstracts away the trainer, the data collator, the distributed setup. The fastest path from "I have a dataset" to "I have a fine-tuned model" in the ecosystem. Reach for Axolotl for 90% of LoRA use cases.

PEFT / LoRA

Hugging Face's parameter-efficient fine-tuning library. The underlying implementation most higher-level frameworks (Axolotl, Unsloth) depend on. Use PEFT directly when you need fine-grained control over the adapter structure — targeted layers, custom rank schedules, mixed precision — that the opinionated wrappers do not expose.

Unsloth

Custom Triton kernels that deliver 2-5× training speedups on consumer GPUs. Transformative for teams without a dedicated GPU cluster. Supported model list lags the frontier by a few weeks. Reach for Unsloth when training throughput is your bottleneck and your base model is in its supported list.

TRL / DeepSpeed

The low-level training stack. TRL adds RLHF, DPO, and preference optimization on top of Hugging Face Transformers; DeepSpeed adds ZeRO and tensor parallelism for distributed training at scale. The combination is what you reach for when fine-tuning a 70B-plus model or implementing a custom alignment algorithm. Not for most teams; essential for the teams that need it.

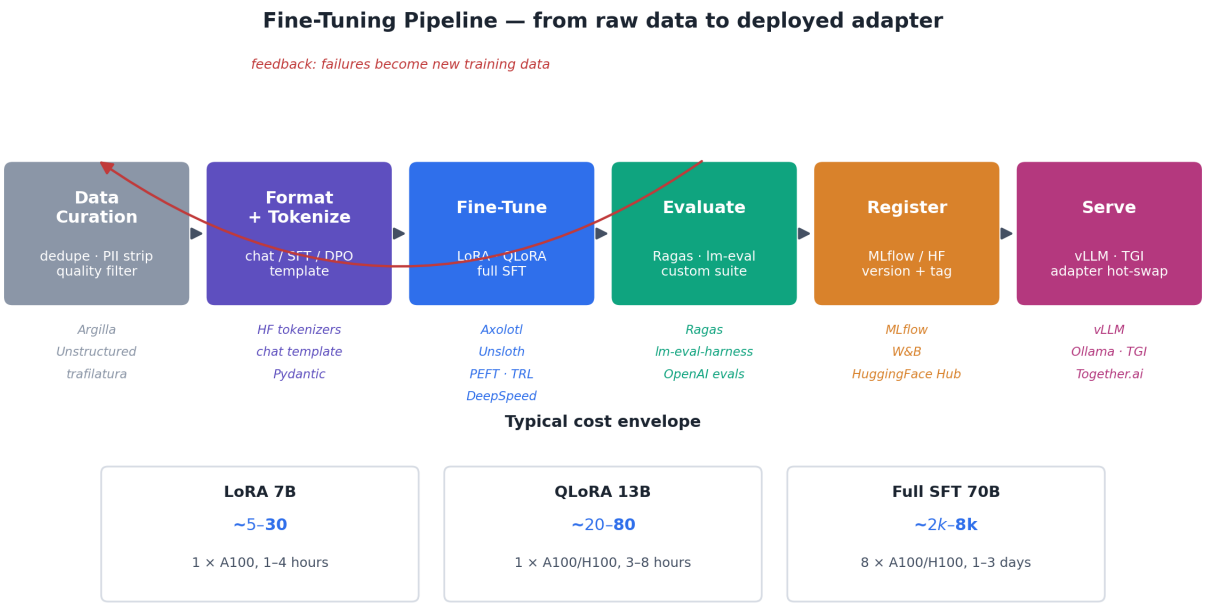


Figure 9a · The fine-tuning pipeline — from raw data curation through evaluation to adapter deployment — with representative tooling and cost envelopes.

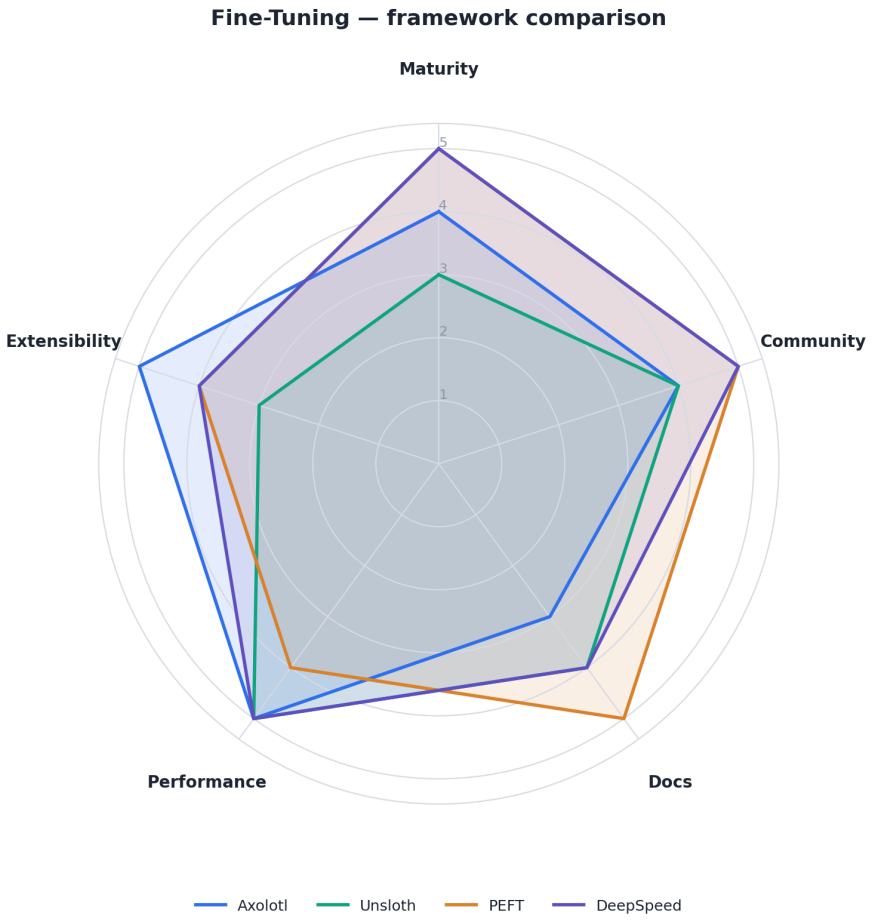


Figure 9b · Fine-tuning frameworks compared on maturity, community, docs, performance, and extensibility.

Practitioner heuristic

For most LoRA fine-tunes, Axolotl. For consumer-GPU speed, Unsloth. Drop to PEFT directly only when you need adapter control. Use TRL + DeepSpeed only when you are doing alignment research or large-model training.

References & further reading

1. LoRA: Low-Rank Adaptation of Large Language Models — Hu et al., 2021. arXiv:2106.09685
2. QLoRA: Efficient Finetuning of Quantized LLMs — Dettmers et al., NeurIPS 2023. arXiv:2305.14314
3. PEFT library — Hugging Face. github.com/huggingface/peft
4. TRL: Transformer Reinforcement Learning — Hugging Face. huggingface.co/docs/trl
5. DeepSpeed — Microsoft Research. deepspeed.ai
6. Axolotl — github.com/axolotl-ai-cloud/axolotl
7. Unsloth — github.com/unslothai/unsloth
8. DPO: Direct Preference Optimization — Rafailov et al., 2023. arXiv:2305.18290

11. Observability & Evaluation

The layer you will regret skipping. LLM systems fail silently — the answer looks right but is wrong, the agent loops without progress, the hallucination is confident and factually incorrect. Traditional monitoring is not enough; you need traces, evaluations, and human-in-the-loop review. In 2026, shipping an LLM system without observability is like shipping a web service without logs.

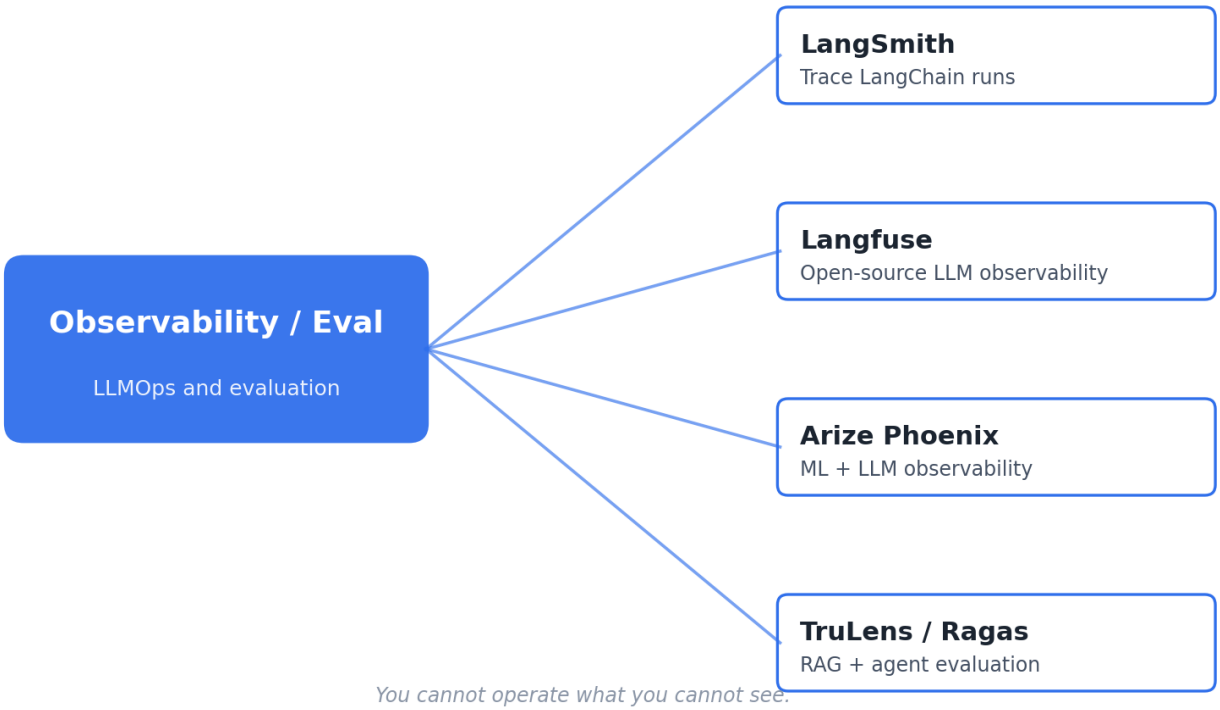


Figure 10 · Observability and evaluation frameworks.

Comparison

Framework	Language	Abstraction	Architecture	Use cases	Strengths	Weaknesses	Maturity
LangSmith	SaaS / Py	High	Trace + eval platform	LangChain debugging	Deep LangChain integration	LangChain-centric	High
Langfuse	Py / Self-host	High	Trace + eval platform	General LLM observability	Open-source, self-hostable	Smaller ecosystem	Medium
Arize Phoenix	Python	Medium	ML + LLM observability	Drift, embeddings	Classical + LLM unified	UX heavier	Medium
TruLens / Ragas	Python	Medium	Evaluation library	RAG + agent eval	Structured metrics	Less tracing	Medium

Framework deep-dives

LangSmith

LangChain's commercial observability platform. Trace every chain, every retrieval, every LLM call; annotate runs as datasets; run evaluations against those datasets as you iterate. The default if you are committed to the LangChain ecosystem. The cost is LangChain dependency — LangSmith is technically framework-neutral, culturally not.

Langfuse

The open-source, self-hostable alternative. Trace ingestion is framework-agnostic, the UI is quickly closing the gap with LangSmith, and the self-host story makes it deployable in data-sovereign environments. The right call when you cannot use SaaS observability and still want the trace-and-eval workflow.

Arize Phoenix

Arize's open-source tool, born from classical ML observability (drift, explainability) and extended to LLMs. The embedding-drift visualizations are genuinely differentiated; the UX is heavier than Langfuse's. Reach for Phoenix when you are running both classical ML and LLMs and want one tool for both.

TruLens / Ragas

Evaluation libraries, not full observability platforms. Ragas provides structured RAG metrics (faithfulness, context recall, answer relevancy); TruLens extends to agents and provides a feedback-function abstraction. Use these inside CI or batch evaluation runs, not as primary observability.

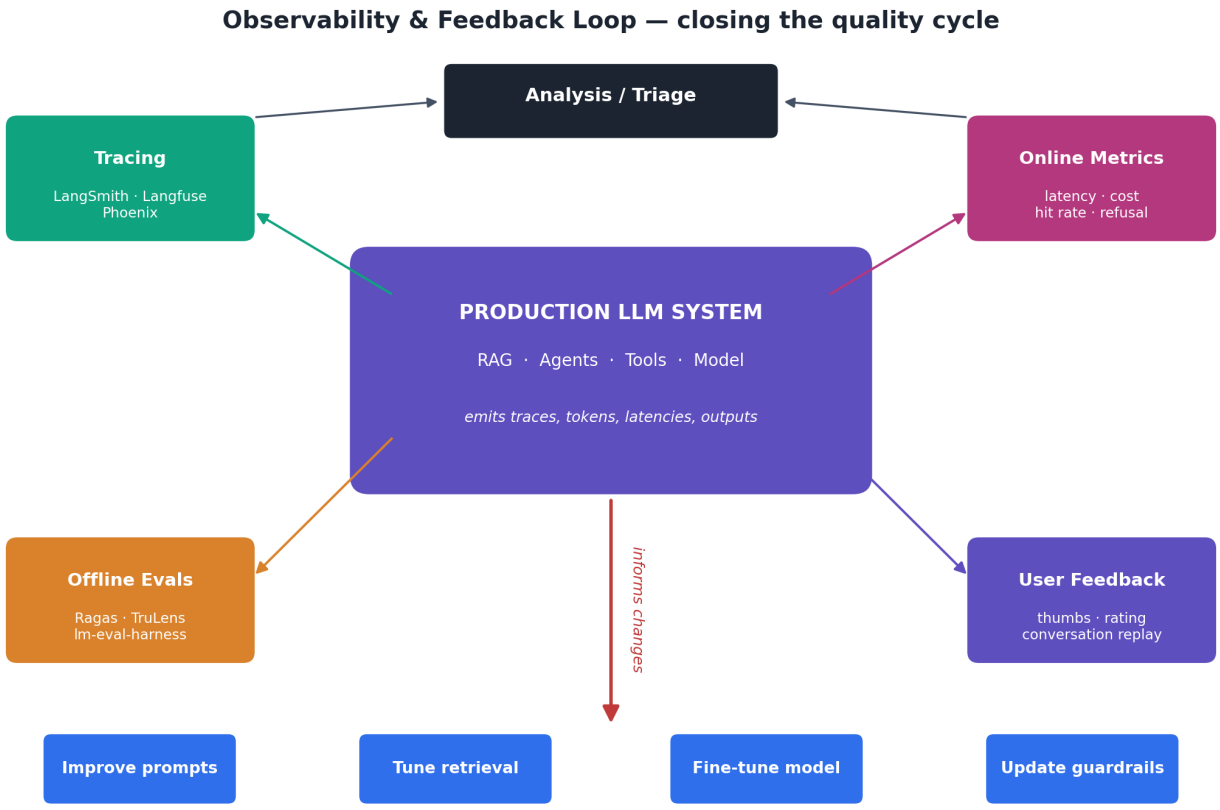


Figure 10a · The observability feedback loop — traces and evals feed triage; triage informs prompt, retrieval, model, and guardrail changes.

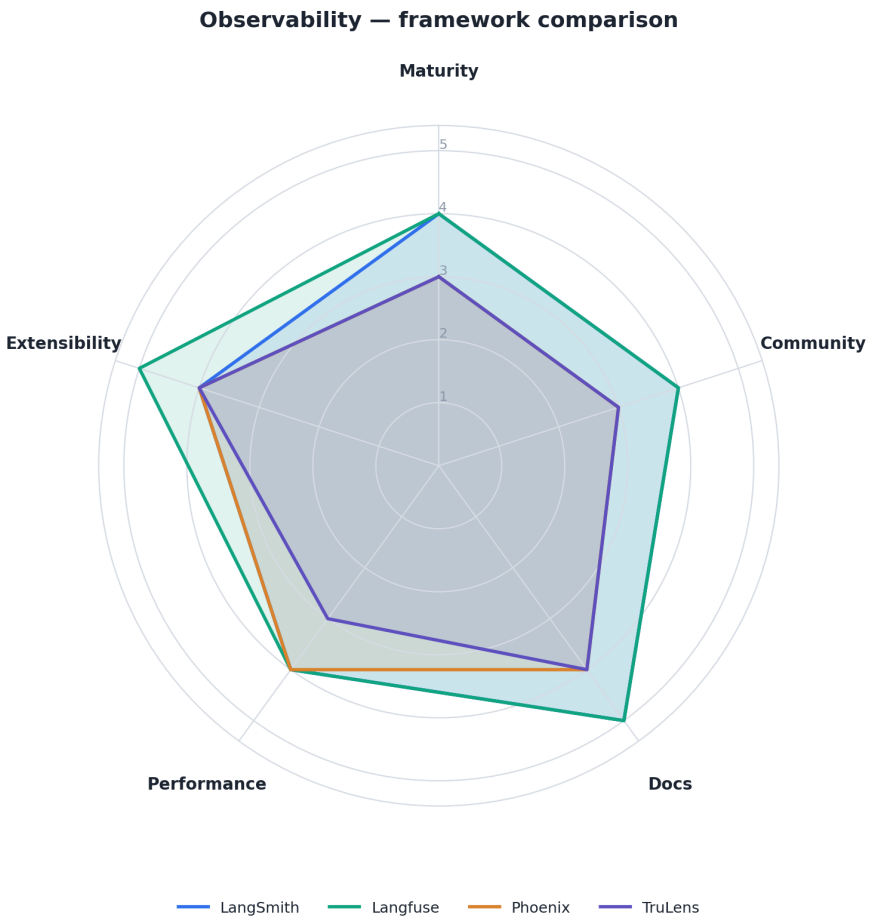


Figure 10b · Observability frameworks compared on maturity, community, docs, performance, and extensibility.

Practitioner heuristic

Pick one trace/eval platform (LangSmith or Langfuse) and adopt it before writing a line of production code. Add Ragas in CI for RAG-specific quality metrics. Non-negotiable: there is no production LLM system without traces.

References & further reading

1. LangSmith — docs.smith.langchain.com
2. Langfuse — langfuse.com/docs
3. Arize Phoenix — docs.arize.com/phoenix
4. TruLens — trulens.org
5. Ragas: Automated Evaluation of Retrieval Augmented Generation — Es et al., 2023. arXiv:2309.15217
6. OpenTelemetry GenAI conventions — opentelemetry.io/docs/specs/semconv/gen-ai

12. Guardrails & Safety

The defensive perimeter. An LLM that takes user input and produces structured output is an attack surface. Prompt injection lets users rewrite your system prompt; jailbreaks bypass your policies; hallucinated structured outputs break downstream systems. Guardrails frameworks sit between the model and the user on both ingress and egress, validating inputs against policies and outputs against schemas. This category did not exist in the original FW document; in 2026 it is non-negotiable.

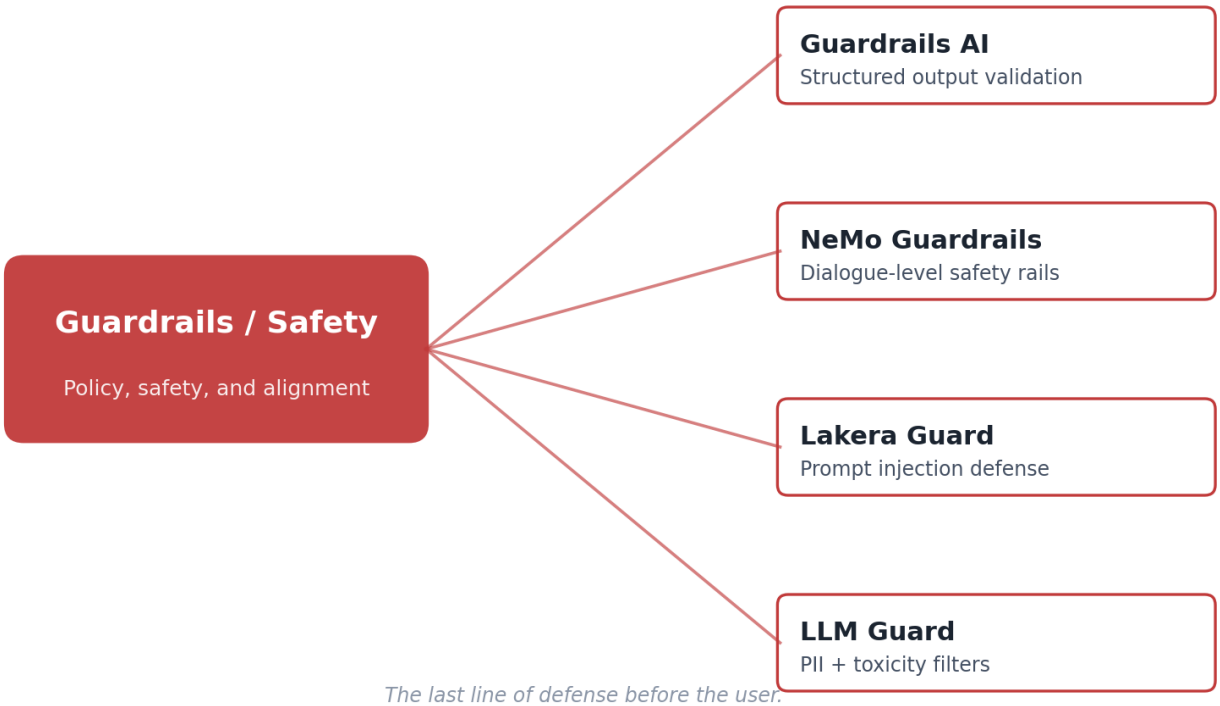


Figure 11 · Guardrails and safety frameworks.

Comparison

Framework	Language	Abstraction	Architecture	Use cases	Strengths	Weaknesses	Maturity
Guardrails AI	Python	High	Schema + validator DSL	Structured output	Rich validator library	Runtime overhead	Medium
NeMo Guardrails	Python	Medium	Dialogue-level rails (Colang)	Conversation safety	Declarative policies	Learning curve	Medium
Lakera Guard	SaaS	High	Injection-detection API	Prompt injection defense	Low-latency classifier	SaaS only	Medium

Framework	Language	Abstraction	Architecture	Use cases	Strengths	Weaknesses	Maturity
LLM Guard	Python	Medium	Pluggable scanner library	PII / toxicity / injection	Open-source, flexible	Needs tuning	Medium

Framework deep-dives

Guardrails AI

The structured-output framework. Define a schema — types, constraints, validators — and Guardrails wraps the LLM call, re-prompting or correcting on validation failure. The primary protection against structured-output breakage and one category of hallucination ("the model said Q4 revenue was $17 million but that field was missing"). The cost is latency: validators run on every call.

NeMo Guardrails

NVIDIA's dialogue-level safety framework, with its own policy DSL (Colang). Rails are defined declaratively: "if the user asks about competitor products, do not answer." Excellent for chat-shaped use cases where the policy surface is conversational. The learning curve is real; Colang is a new language to learn.

Lakera Guard

The injection-detection SaaS. A fast classifier (20-50 ms) that runs on every prompt; returns a probability that the input is a jailbreak or injection attempt. The right call when you need SOC-2-grade injection defense and can tolerate SaaS posture.

LLM Guard

ProtectAI's open-source, composable scanner library. Input scanners (PII detection, injection detection, toxicity) and output scanners (bias, hallucination heuristics). Reach for LLM Guard when you need the flexibility of composable scanners and cannot use SaaS.

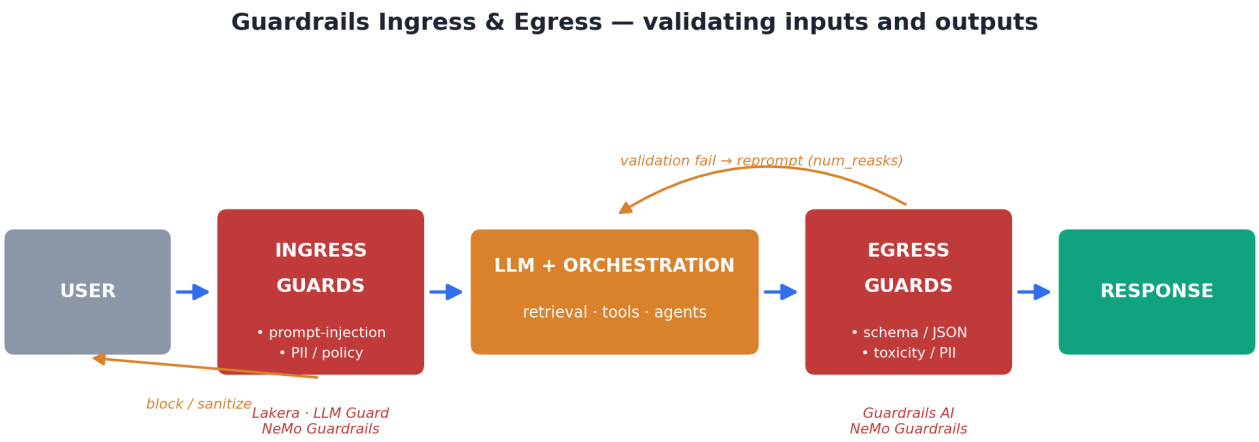


Figure 11a · Ingress and egress guardrails — every guard is a classifier, validator, or policy check that caps catastrophic failures at the cost of some latency.

Practitioner heuristic

For structured outputs, Guardrails AI. For dialogue-shape policies, NeMo Guardrails. For prompt injection, Lakera (SaaS-ok) or LLM Guard (on-prem). Layer them: input guard → LLM → output guard is the minimum viable defense in 2026.

References & further reading

1. OWASP Top 10 for LLM Applications — 2025. genai.owasp.org/llm-top-10
2. Guardrails AI — guardrailsai.com/docs
3. NVIDIA NeMo Guardrails — docs.nvidia.com/nemo/guardrails
4. Lakera Guard — lakera.ai/lakera-guard
5. LLM Guard (ProtectAI) — llm-guard.com
6. Ignore Previous Prompt: Attack Techniques for Language Models — Perez & Ribeiro, 2022. arXiv:2211.09527
7. NIST AI Risk Management Framework — nist.gov/itl/ai-risk-management-framework

13. Supporting & Emerging

A short list of things that don't fit neatly in a layer above but are too important to omit. Small surface, outsized impact.

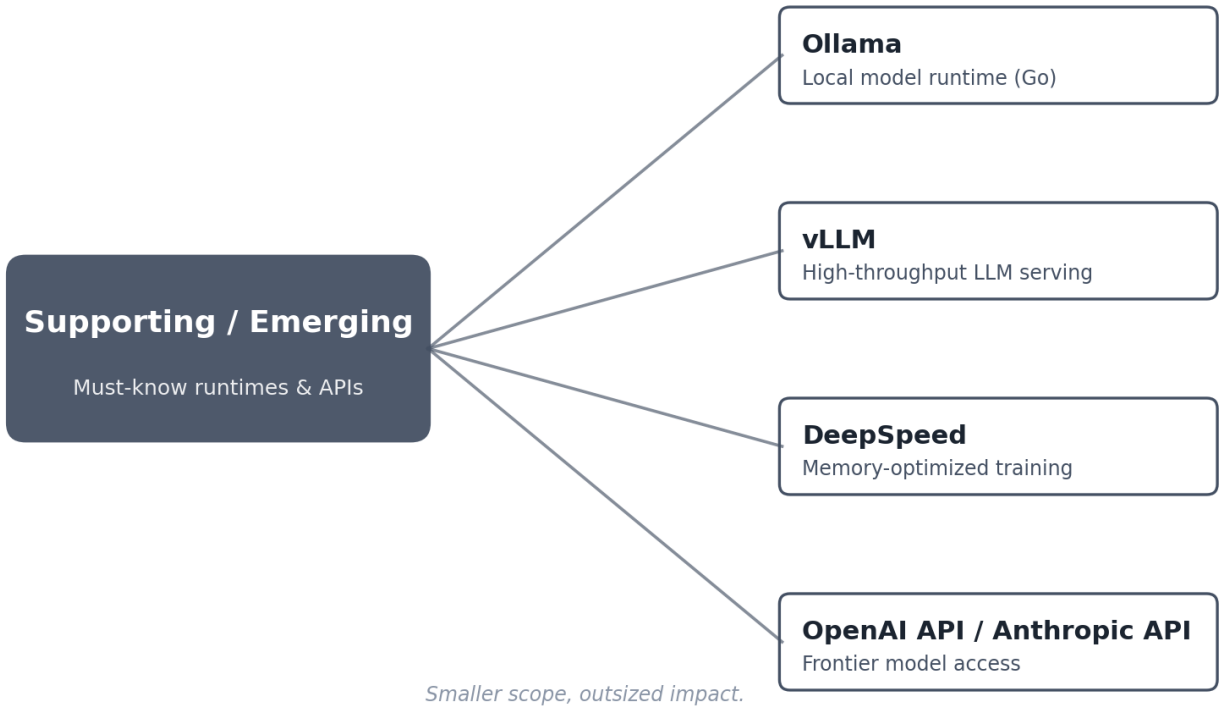


Figure 12 · Supporting / emerging frameworks.

Comparison

Framework	Language	Category	Use case	Key insight	Maturity
Ollama	Go	Runtime	Local LLM apps	Simplifies local inference	High
vLLM	Python	Inference	High-throughput LLM serving	Paged attention for GPU batching	High
DeepSpeed	Python	Training	Large model training	ZeRO memory optimization	High
OpenAI API / Anthropic API	API	SaaS	Frontier LLM apps	Frontier model quality	Very High

Framework deep-dives

Ollama

The runtime that made local LLM apps easy. One command to download, one API to serve. Ships GGUF quantized models that run on commodity GPUs and recent CPUs. The default runtime for local development, on-device demos, and on-prem proofs of concept.

vLLM

The throughput-optimized LLM serving engine. Paged attention (borrowing from OS virtual memory) lets vLLM batch requests with wildly different lengths without memory fragmentation. The result is 3-10× the throughput of naive Transformers serving. Pair with Ray Serve for horizontal scale.

DeepSpeed

Microsoft's distributed training library. ZeRO (Zero Redundancy Optimizer) partitions optimizer states, gradients, and parameters across GPUs to train models that don't fit on a single device. Essential for >13B parameter training; overkill below that.

OpenAI / Anthropic APIs

The frontier model APIs. Still the quality leaders for most tasks, still the fastest path from prototype to product, still expensive at scale. The architectural question in 2026 is not "API or local" but "which workloads justify the API's cost premium over a fine-tuned local model?"

> **Practitioner heuristic**
> Ollama for local dev. vLLM for serving scale. DeepSpeed only when your training problem demands it. OpenAI/Anthropic for quality-first workloads and as a baseline to compare local fine-tunes against.

References & further reading

1. Ollama — ollama.com
2. vLLM — docs.vllm.ai
3. DeepSpeed — deepspeed.ai
4. OpenAI API reference — platform.openai.com/docs
5. Anthropic API documentation — docs.anthropic.com
6. Model Context Protocol — modelcontextprotocol.io
7. llama.cpp — github.com/ggerganov/llama.cpp

PART III

Cross-Framework Analysis

Pulling the layers back together. Decision heuristics, comparative heatmaps, and the trade-offs every architect negotiates.

14. The Abstraction–Control Spectrum

Every framework in this atlas sits at a point on a single continuum: how much code do you write to get a result, and how much control do you surrender in exchange? High-abstraction frameworks (LangChain, CrewAI) get you to a running demo in an afternoon by deciding everything for you. Low-abstraction frameworks (FAISS, JAX) get you to exactly the implementation you designed, one tensor at a time.

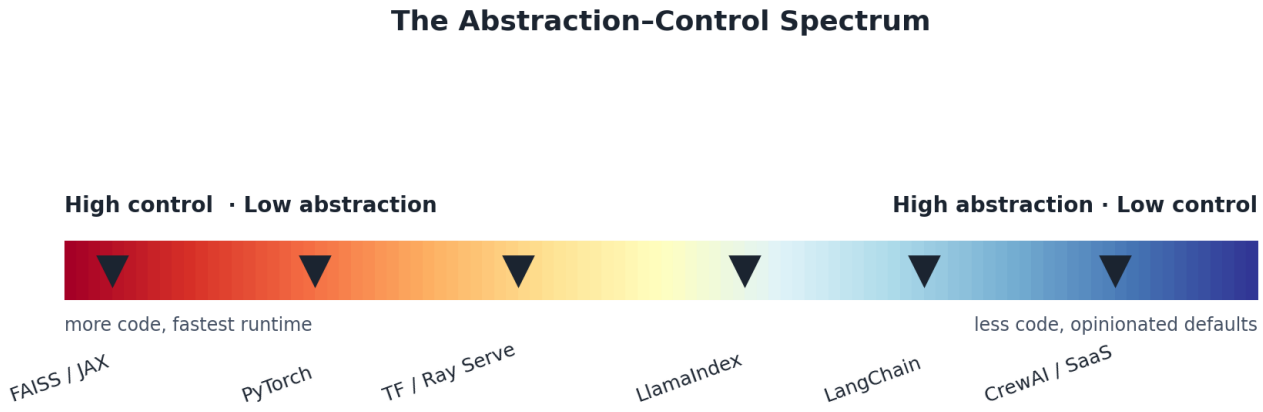


Figure 13 · The abstraction–control spectrum. Left: fastest runtime, hardest to learn. Right: fastest to ship, hardest to debug.

When to slide left, when to slide right

Slide left (toward lower abstraction) when you are deep into production, optimizing a bottleneck you understand well, or hitting a wall the high-abstraction framework's defaults cannot scale through. You give up development speed for runtime predictability — a trade you want to make only at the layer that matters.

Slide right (toward higher abstraction) when you are still discovering what your system should even do, when the team is mixed-seniority, or when the cost of a wrong framework choice is low (e.g., throwaway prototypes). You trade long-term debuggability for short-term velocity — valuable when you are still learning what to build.

Mix altitudes deliberately. A mature production system is rarely uniform in abstraction. The common pattern is high-abstraction at the edges (LangChain for the app layer, BentoML for deployment) and low-abstraction in the hot path (FAISS for retrieval, vLLM for inference). Resisting the temptation to be uniform is one of the marks of a mature architect.

> **Anti-pattern**
> Low abstraction everywhere is not a purist virtue; it is an unshipped product. High abstraction everywhere is not pragmatism; it is a system you cannot debug. The craft is in the mix.

15. Decision Heuristics & Flowchart

The most common framework decisions reduce to a handful of questions. The flowchart below is the shortcut; the table that follows expands it with requirement-to-framework mappings distilled from a decade of production deployments.

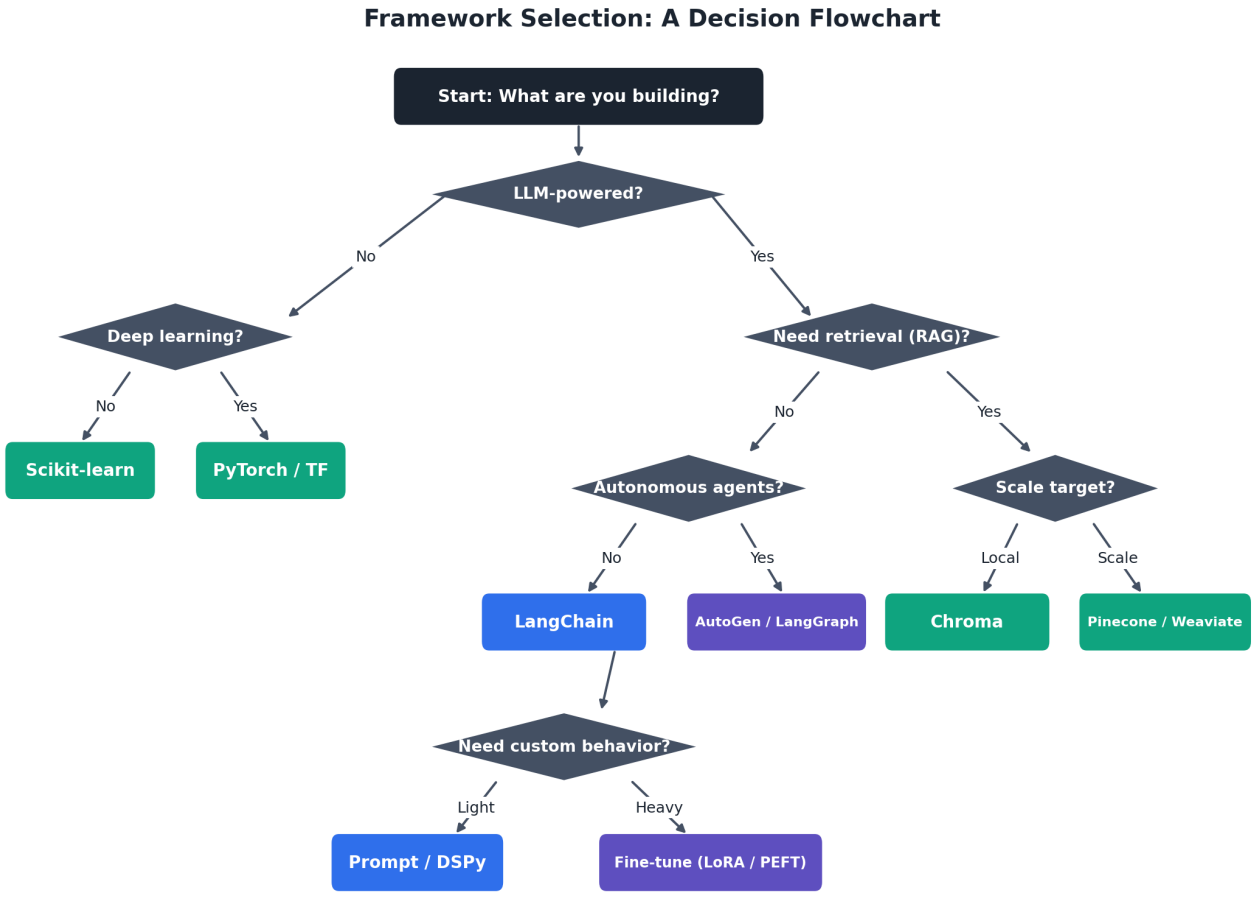


Figure 14 · Framework selection flowchart — the minimal path from requirement to starting framework.

Requirement-to-framework mappings

Requirement	Preferred choice
Fast LLM prototype	LangChain + Chroma + OpenAI API
Enterprise-grade RAG	LlamaIndex + Weaviate + LangSmith
Multi-agent workflow	LangGraph (+ AutoGen for conversations)
High-throughput inference	vLLM + Ray Serve
Local / offline AI	Ollama + FAISS + LangChain-local

Requirement	Preferred choice
Full MLOps pipeline	MLflow + Kedro + W&B
Domain-specialized model	Axolotl + PEFT + MLflow
Regulated industry deployment	Weaviate + Llama-3 on vLLM + LLM Guard
Microsoft-shop copilot	Semantic Kernel + Azure AI Search
Production agent w/ tools	LangGraph + Guardrails AI + Langfuse

The three questions that actually matter

1. Where must this run? The deployment model is the first constraint. A SaaS-only framework (Pinecone, LangSmith) is a non-starter for a data-sovereign deployment no matter how good it is.

2. Who will operate it? A framework with a *High* learning curve is a liability on a team that cannot dedicate weeks to learning it. Abstraction level must match team seniority.

3. How will it change? Frameworks with low maturity will break your code with upgrades; frameworks in the wrong ecosystem will trap you when you want to swap a model. Optimize for swappability.

16. Comparative Heatmaps

The comparison tables in Part II catalog each framework in its own row. The heatmaps here do the opposite: they put all frameworks on a shared scale so you can see, at a glance, the clusters and outliers.

Heatmap 1 · Frameworks × attributes

The first heatmap scores every framework on five dimensions: maturity, ease of use (inverse of learning curve), abstraction level, ecosystem size, and production readiness. Scores are 0-5; they are opinionated and reflect production consensus, not benchmarks.

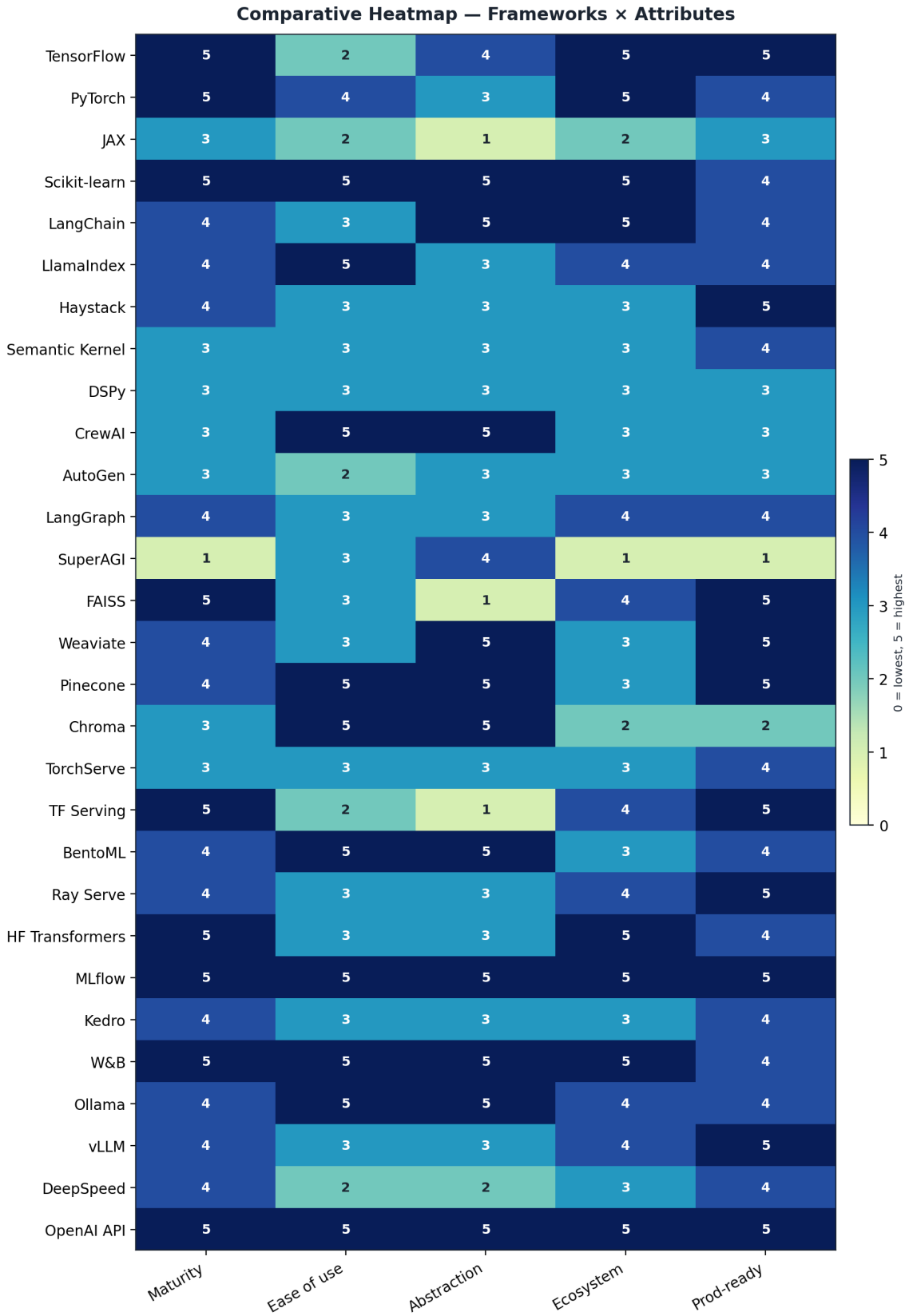


Figure 15 · Every framework, five axes. The blue clusters identify frameworks you can deploy in production with minimal justification.

What the heatmap reveals

- **The top row is blue.** Frameworks that score 4-5 across all five dimensions (Scikit-learn, MLflow, W&B, OpenAI API) are the "boring" picks — the ones you never get fired for choosing.
- **The research frameworks (JAX, DeepSpeed) have low ease-of-use scores.** That is feature, not defect; it reflects that they expose control the higher-abstraction frameworks hide.
- **The emerging frameworks (SuperAGI, DSPy) have lower maturity scores.** They may prove foundational in two years, but today their adoption is a bet, not a default.

Heatmap 2 · Use case × framework fit

The second heatmap inverts the view: for each common use case, which frameworks actually fit? The cell scores are **0 = poor fit** to **5 = excellent fit**. A row of reds means that use case needs a different toolbox; a row with two or three strong columns is your shortlist.

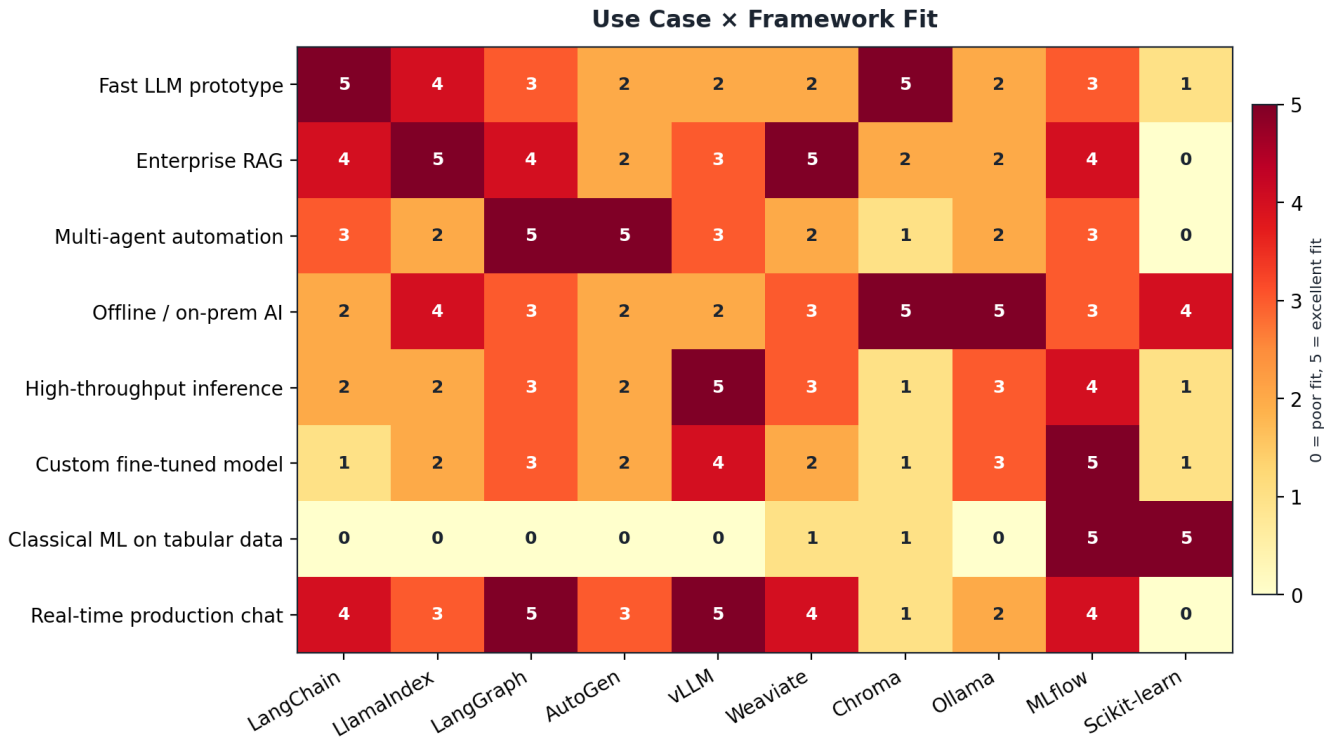


Figure 16 · Use-case fit. Read rows, not columns — the strongest cells in each row are your shortlist for that use case.

Reading the matrix

The matrix confirms a few things that the prose in Part II only implied:

- **Multi-agent automation** is dominated by LangGraph and AutoGen; most other frameworks are poor fits.
- **Classical ML on tabular data** is a row of near-zeros for everything except MLflow and Scikit-learn. Do not reach for an LLM when a gradient-boosted tree will do.
- **High-throughput inference** is vLLM's row. Everything else is a consolation prize at serious scale.
- **Offline / on-prem AI** has more strong columns (Ollama, Chroma, Weaviate, LangChain-local) than most practitioners expect. Privacy is no longer a viable excuse for not shipping.

17. Key Trade-offs

Every framework choice is a negotiation among three axes: abstraction versus control, local versus cloud, agents versus pipelines. These are the irreducible trade-offs — the ones no framework can eliminate, only relocate.

Abstraction versus control

Abstraction wins on day one. A LangChain demo is running in an hour. A FAISS implementation of the same demo takes a week. On day one, the velocity gap is dispositive.

Control wins on day 100. A LangChain production bug leads you through wrapper classes to a decision made by the framework author in 2023. A FAISS bug lives in code you wrote. At day 100, the debuggability gap is dispositive.

The compromise is layering. High-abstraction at the cold edges (app, CLI, deployment), low-abstraction in the hot path (retrieval, inference). This mixed-altitude posture is the single most consistent pattern in mature production systems.

Local versus cloud

Cloud wins on velocity, breadth, and scale. Frontier model quality, zero-ops vector databases, elastic inference. The SaaS ecosystem is an economic argument against running anything yourself.

Local wins on cost, privacy, and predictability. At high volume, cloud API costs swamp the capex of GPUs. In regulated industries (healthcare, finance, defense), SaaS is a non-starter. For deterministic latency, in-process retrieval beats a cross-network call.

The compromise is hybrid. Frontier models in the cloud for quality-critical tasks; local fine-tuned models for volume-critical ones; local vector DBs or self-hosted Weaviate for data-sovereign retrieval. The hybrid pattern is not a transition state; it is the stable architecture for serious 2026 deployments.

Agents versus pipelines

Pipelines win on predictability. A fixed DAG of LLM calls is debuggable, testable, and has bounded latency. Most problems that *sound* like they need agents actually fit as pipelines.

Agents win on open-endedness. When the path depends on intermediate results in ways that cannot be fixed in advance — triage, research, exploratory analysis — agents are the only credible approach. The cost is unbounded latency and token spend.

The compromise is bounded autonomy. Use agents inside pipeline boundaries: the pipeline enforces termination (max steps, max tokens, time budget), the agent decides within that envelope. LangGraph's state-machine model makes this compromise explicit.

The meta-trade-off
Every framework decision above is really a decision about **who absorbs the complexity**: the framework author (high abstraction), your team (low abstraction), the user (unbounded agents), or the architecture (bounded pipelines). Pick deliberately.

PART IV

Reference Architectures

Four production-grounded architectures, each with a diagram, a component breakdown, and working code you can lift.

18. Enterprise RAG

Problem. Your company has a decade of documentation, policies, contracts, and knowledge-base articles. Employees cannot find what they need; customers ask questions your support team answers the same way a hundred times a month. You want a system that answers those questions from your actual documents, cites its sources, and fails safely when it does not know.

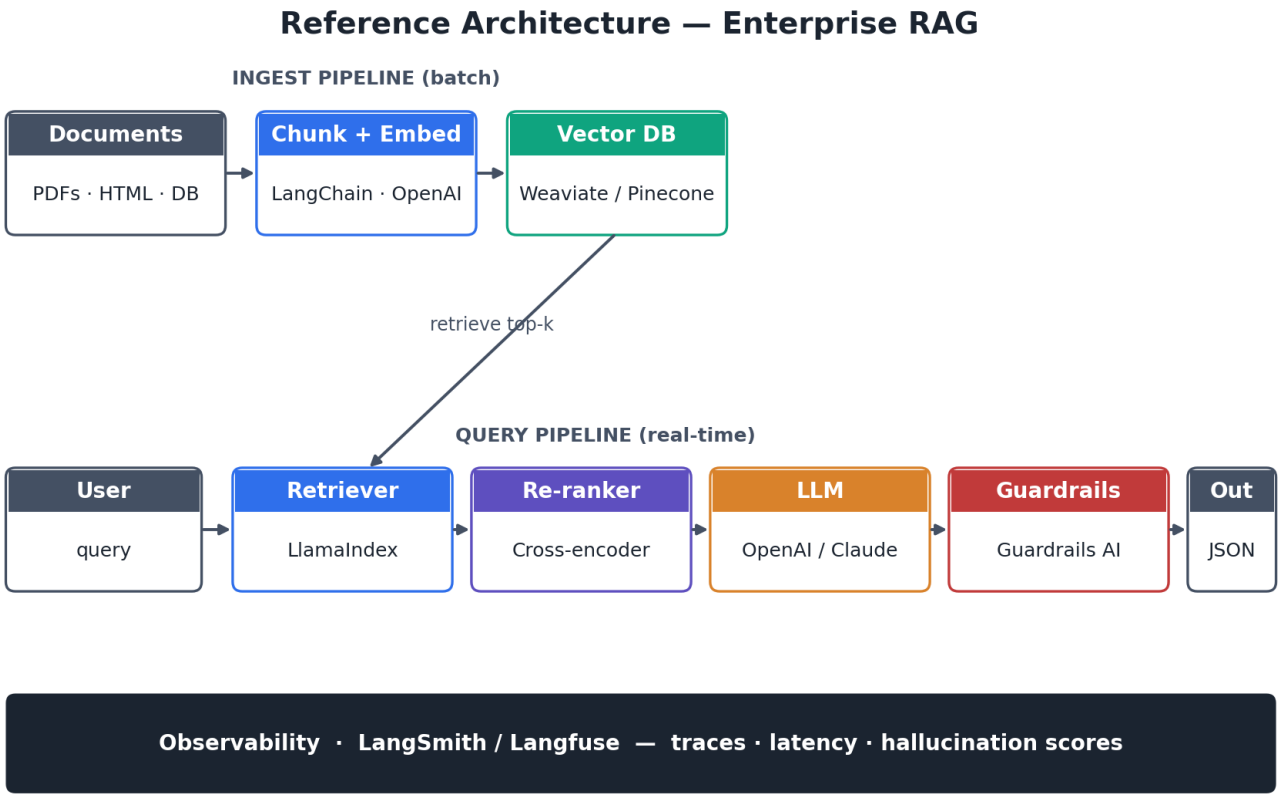


Figure 17 · Enterprise RAG. Two pipelines (ingest, query), one observability plane.

Component breakdown

The architecture separates into two pipelines. The **ingest pipeline** is batch: documents are chunked, embedded, and written to a vector database. The **query pipeline** is real-time: user queries flow through a retriever, a re-ranker, the LLM, and a guardrail before returning structured output. Both pipelines emit traces to a shared observability plane.

- **Documents.** Source content in multiple formats — PDFs, HTML, database rows. Ingest parsers normalize everything to plain text plus structured metadata (source, author, date).
- **Chunk + embed.** LangChain's document loaders + a semantic chunker + an embedding model (OpenAI text-embedding-3 or a local bge-m3). Chunk sizes tuned to 500-1000 tokens with 100-token overlap for most document types.

- **Vector DB.** Weaviate for self-hosted; Pinecone for SaaS. Metadata filters preserved so queries can be scoped (e.g., "only 2025 policy documents").
- **Retriever.** LlamaIndex orchestrates hybrid search (vector + BM25) and routes across multiple indices if you segment by document type.
- **Re-ranker.** A cross-encoder (bge-reranker, Cohere Rerank) scores the top-k candidates against the query. Critical for precision — vector search alone is recall-oriented.
- **LLM.** OpenAI / Anthropic for quality-critical deployments; a local fine-tuned Llama for volume-critical ones. The choice is orthogonal to the rest of the architecture.
- **Guardrails.** Structured-output validation (Guardrails AI) enforces the response schema — citations, confidence, refusal flag — and re-prompts on violation.
- **Observability.** Langfuse or LangSmith ingest traces from every step. Latency, token cost, retrieval hit rate, and hallucination scores are monitored continuously.

Reference code: the query pipeline

A minimal, production-shaped LlamaIndex query pipeline with reranking and guardrails. In real deployments you would add retries, timeouts, and metric emission at each stage.

```
from llama_index.core import VectorStoreIndex, StorageContext
from llama_index.vector_stores.weaviate import WeaviateVectorStore
from llama_index.core.postprocessor import SentenceTransformerRerank
from llama_index.llms.openai import OpenAI
from guardrails import Guard
from pydantic import BaseModel, Field

class Answer(BaseModel):
    answer: str = Field(description="Direct answer to the user.")
    citations: list[str] = Field(description="Source doc IDs used.")
    confidence: float = Field(ge=0, le=1)

# 1. Load index from Weaviate (ingested offline)
store = WeaviateVectorStore(weaviate_client=client, index_name="docs")
ctx   = StorageContext.from_defaults(vector_store=store)
index = VectorStoreIndex.from_vector_store(store, storage_context=ctx)

# 2. Build a query engine with re-ranking
reranker = SentenceTransformerRerank(model="BAAI/bge-reranker-v2-m3", top_n=4)
engine = index.as_query_engine(
    llm=OpenAI(model="gpt-4o"),
    similarity_top_k=20,
    node_postprocessors=[reranker],
)

# 3. Guardrails enforces the output schema
guard = Guard.from_pydantic(output_class=Answer)

def answer_query(q: str) -> Answer:
    raw = engine.query(q).response
    validated, _ = guard.parse(raw, num_reasks=2)
    return validated
```

Production note
The code above is a template, not a recipe. In production you will want: query rewriting (expand short queries), citation verification (the LLM cited doc 7; does doc 7 actually support the claim?), and a refusal path when retrieval hit rate is below a threshold. Those additions are what separate a demo from a system.

19. Multi-Agent Automation

Problem. A task that a single LLM call cannot reliably do in one shot: research a topic, write code against it, test the code, and iterate on failures. The path is open-ended — the number of critique-revise rounds depends on what the reviewer finds — but the envelope is bounded (a few minutes, a few dozen tool calls, a few dollars in tokens).

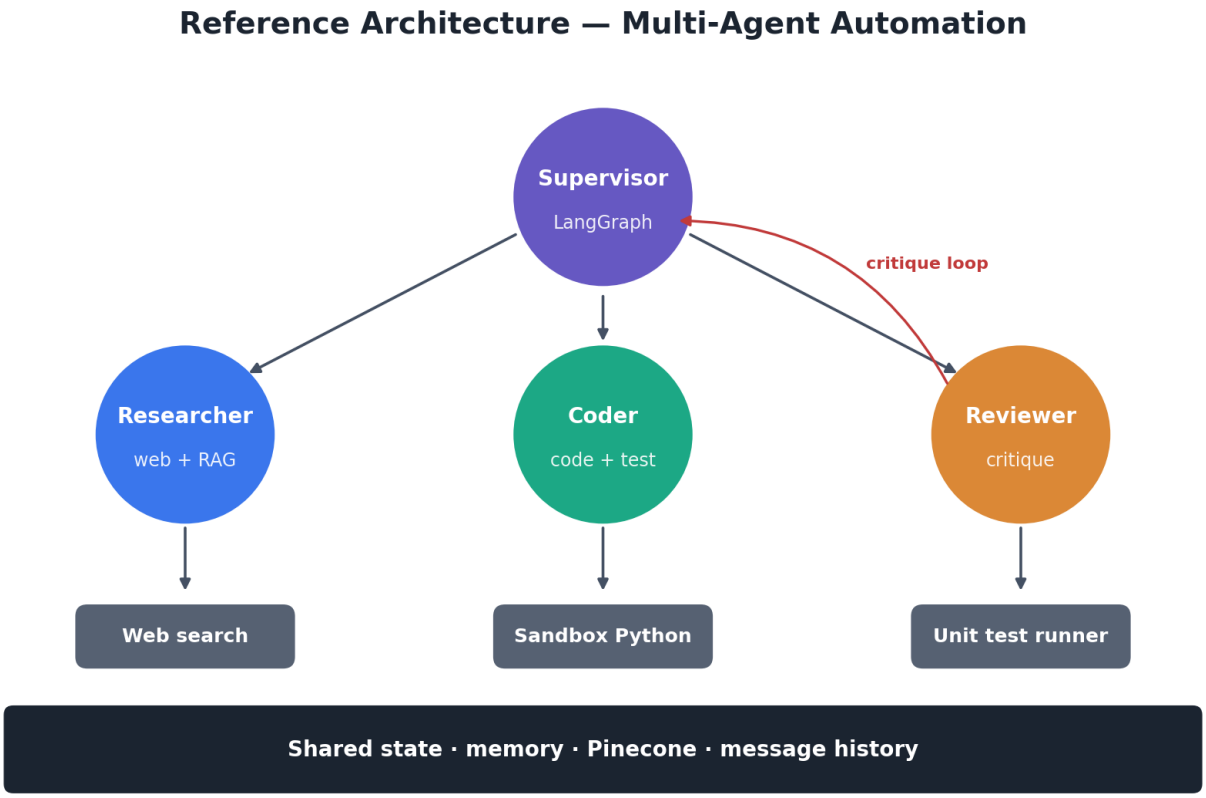


Figure 18 · Multi-agent automation with a supervisor, three specialists, and a critique loop.

Component breakdown

- **Supervisor (LangGraph).** The top-level state machine. Receives the task, dispatches to specialists, enforces the termination envelope (max steps, max tokens, max wall clock). Every transition is a pure function of the shared state.
- **Researcher.** Web search + RAG over internal docs. Produces grounded context for the coder.
- **Coder.** Writes code against the researcher's context; executes it in a sandbox.
- **Reviewer.** Examines the coder's output for correctness, style, and edge cases. Issues critiques back to the supervisor.

- **Shared memory.** A typed state object plus a long-term memory in a vector DB for recall across sessions.
- **Tools.** Web search, Python sandbox, unit test runner. Each tool is a pure function of input state.

Reference code: the supervisor state machine

LangGraph makes the multi-agent loop an explicit graph. Nodes are specialists; edges are conditional transitions driven by the reviewer's verdict.

```
from langgraph.graph import StateGraph, END
from typing import TypedDict, Annotated, List
import operator

class State(TypedDict):
    task: str
    research: str
    code: str
    critique: str
    iteration: int
    messages: Annotated[List, operator.add]

def researcher(state: State) -> State:
    # web + RAG search, return grounded context
    return {"research": do_research(state["task"]), "messages": [...]}

def coder(state: State) -> State:
    code = write_and_run(state["task"], state["research"])
    return {"code": code, "messages": [...]}

def reviewer(state: State) -> State:
    verdict = critique(state["code"], state["task"])
    return {"critique": verdict, "iteration": state["iteration"] + 1}

def route(state: State) -> str:
    if "APPROVED" in state["critique"] or state["iteration"] >= 3:
        return END
    return "coder"  # loop back

graph = StateGraph(State)
graph.add_node("researcher", researcher)
graph.add_node("coder", coder)
graph.add_node("reviewer", reviewer)
graph.set_entry_point("researcher")
graph.add_edge("researcher", "coder")
graph.add_edge("coder", "reviewer")
graph.add_conditional_edges("reviewer", route, {"coder": "coder", END: END})
app = graph.compile()

final = app.invoke({"task": "...", "iteration": 0, "messages": []})
```

Why LangGraph and not AutoGen here
LangGraph makes termination a first-class concern (iteration counter, conditional edges). AutoGen is more expressive for free-form conversations but harder to bound. For automation with strict latency and cost envelopes, LangGraph's explicit state wins; for research-style agent-to-agent discussions, AutoGen's conversations win.

20. Local / Offline AI Stack

Problem. A deployment that cannot send data outside the VPC — a hospital, a law firm, a defense contractor. SaaS APIs are off the table. Every component must run on hardware you control, with models you can audit.

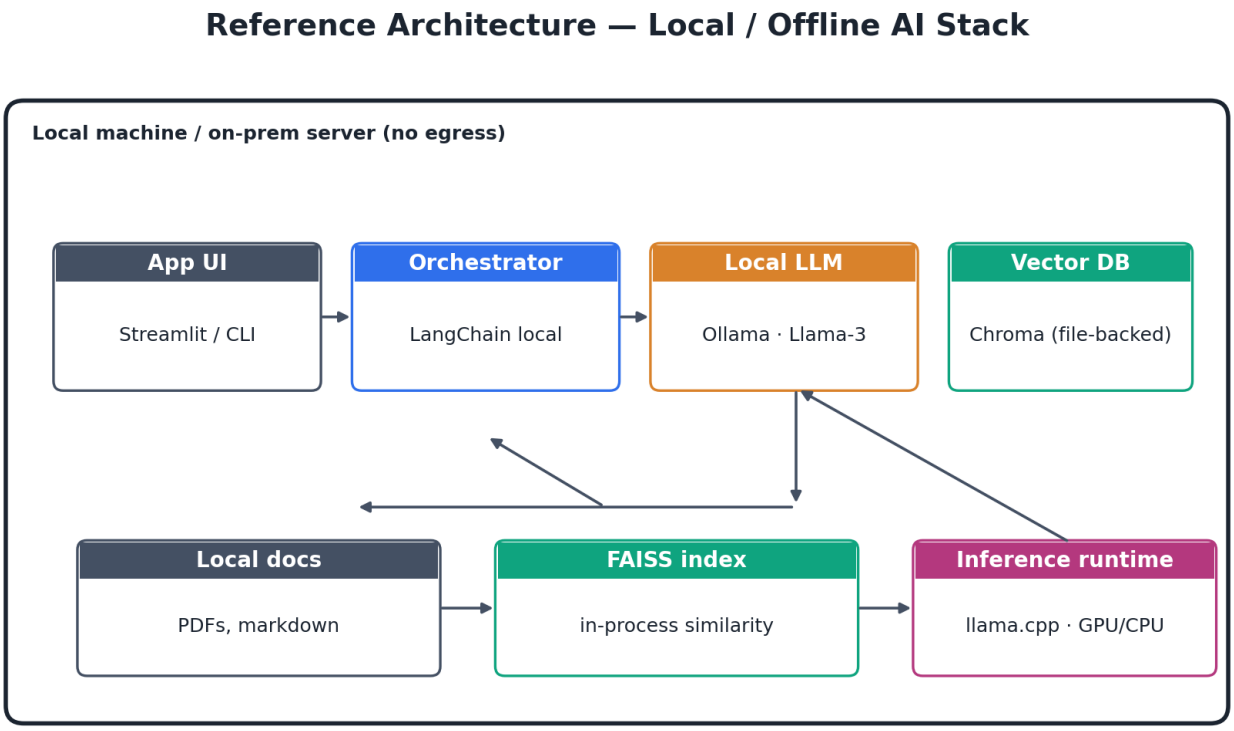


Figure 19 · Local / offline AI. Every component inside the security perimeter; zero egress.

Component breakdown

- **App UI.** Streamlit for internal tools, a REST API for integration with existing systems. Everything in the same VPC.
- **Orchestrator.** LangChain or LlamaIndex, configured to use local embeddings and local model endpoints — no OpenAI, no Anthropic.
- **Local LLM.** Ollama serving a Llama-3 (8B or 70B) or Mistral model. Quantized to Q5_K_M for the throughput-quality sweet spot.
- **Vector DB.** Chroma for file-backed persistence; Weaviate for multi-machine deployments.
- **Inference runtime.** llama.cpp under Ollama on CPU; vLLM on a dedicated GPU if throughput demands it.
- **Local docs + FAISS index.** The knowledge base lives on disk; FAISS provides in-process similarity search for the hot path.

Reference code: a fully local RAG chain

```
from langchain_community.llms import Ollama
from langchain_community.embeddings import OllamaEmbeddings
from langchain_community.vectorstores import Chroma
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.chains import RetrievalQA
from langchain_community.document_loaders import DirectoryLoader

# 1. Load and chunk local documents — nothing leaves the disk
docs = DirectoryLoader("/data/kb", glob="**/*.md").load()
splitter = RecursiveCharacterTextSplitter(chunk_size=800, chunk_overlap=120)
chunks = splitter.split_documents(docs)

# 2. Embed locally with an Ollama-served model
embeddings = OllamaEmbeddings(model="nomic-embed-text")
vectordb = Chroma.from_documents(
    chunks, embeddings, persist_directory="/data/chroma"
)

# 3. Run inference locally
llm = Ollama(model="llama3:8b-instruct-q5_K_M")
qa = RetrievalQA.from_chain_type(
    llm=llm,
    retriever=vectordb.as_retriever(search_kwargs={"k": 5}),
    return_source_documents=True,
)

result = qa.invoke({"query": "What is our data retention policy?"})
# No network egress. No SaaS. Audit-ready.
```

What you give up

Frontier quality (GPT-4/Claude-class) and zero-ops scale. What you get: full data sovereignty, predictable per-token cost (amortized GPU), and an auditable model. At 2026 frontier-minus-one quality, the trade is increasingly reasonable.

21. Fine-Tuned Domain Model

Problem. You have a large volume of similar interactions — support tickets, insurance claims, medical notes — and a base LLM that almost-but-not-quite handles them. You want a model specialized to your domain: cheaper per call, faster, more consistent in format.

Reference Architecture — Fine-Tuned Domain Model Pipeline

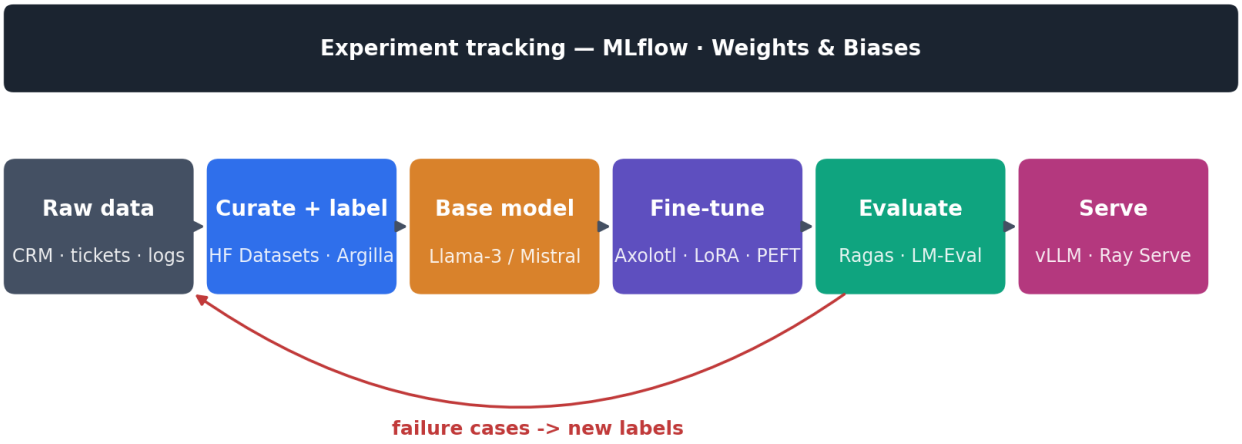


Figure 20 · Fine-tuning pipeline with a failure-case feedback loop.

Component breakdown

- **Raw data.** Historical interactions — tickets, logs, labeled outputs. The volume to aim for: 1,000-10,000 examples for LoRA, 100k+ for full fine-tuning (which you usually should not do).
- **Curate + label.** Argilla or Label Studio for human curation; Hugging Face Datasets for storage. Quality of labels dominates quantity.
- **Base model.** Llama-3-8B for small, latency-critical use cases; Llama-3-70B or Mistral-Large when quality matters and cost allows.
- **Fine-tune.** Axolotl with a LoRA adapter; rank 16-64 for most use cases. Training run is hours on a single A100.
- **Evaluate.** Ragas for RAG metrics if relevant, plus a domain-specific eval suite. Failures feed back to the labeling stage.
- **Serve.** vLLM with the LoRA adapter loaded at startup; swap adapters at request time for multi-tenant use.

- **Tracking.** MLflow + W&B for experiment tracking. Every run logged, every artifact versioned.

Reference code: an Axolotl LoRA config

```
# axolotl/lora_domain.yml
base_model: meta-llama/Meta-Llama-3-8B-Instruct
model_type: LlamaForCausalLM
load_in_4bit: true                  # QLoRA: 4-bit base, 16-bit adapter

datasets:
  - path: data/train.jsonl
    type: alpaca

adapter: lora
lora_r: 32
lora_alpha: 64
lora_target_modules: [q_proj, k_proj, v_proj, o_proj]

sequence_len: 4096
micro_batch_size: 2
gradient_accumulation_steps: 4
num_epochs: 3
learning_rate: 0.0002

output_dir: ./out/lora_domain
wandb_project: my-domain-tune

# Launch:
#   accelerate launch -m axolotl.cli.train axolotl/lora_domain.yml
# Merge (optional) and serve with vLLM:
#   python -m vllm.entrypoints.openai.api_server \
#     --model ./out/lora_domain/merged
```

> **The unglamorous truth**
> The training run is a weekend. The data curation is six months. The eval harness is ongoing forever. If you treat fine-tuning as a modeling problem instead of a data problem, you will end up with a model that is worse than the base. Do not make that mistake.

PART V

Outlook & Appendices

Where the stack is heading, a glossary of the terms used in this atlas, and the sources worth following.

22. The 2026 Outlook

The framework landscape is consolidating. Between 2023 and 2026, dozens of candidate frameworks launched at every layer; most have been selected out or absorbed. The remaining survivors tend to specialize within a layer rather than spanning layers. The following trends are the ones worth designing for.

Trend 1 · The agent layer is becoming the compile target

LangGraph, AutoGen, and CrewAI are diverging less than they appear: they all compile down to a loop over an LLM with tool use and state. Expect a convergence on a common IR (something like OpenAI's agent tool-use schema, standardized) and frameworks that merely compile *to* that IR from different developer experiences. The agent layer will look like the deep learning layer did in 2018: multiple frontends, one runtime.

Trend 2 · Retrieval is eating search

The distinction between a vector database and a search engine has collapsed. Elasticsearch, OpenSearch, and Postgres all ship vector indexes; Weaviate and Pinecone ship BM25. The primitives are converging; the differentiator is operational maturity, not feature breadth. Expect the vector DB category to re-merge with search in the 2027-2028 window.

Trend 3 · Evaluation is the new testing

The standard engineering practice is "write a unit test, make it pass, merge." The LLM-era practice is "write an eval, measure baseline quality, regress if quality drops." Langfuse and LangSmith are building that workflow into CI. Expect eval-driven development to be as baseline in 2028 as test-driven development is today.

Trend 4 · Small specialized models displace large generalists

For volume workloads (classification, structured extraction, routing), a fine-tuned Llama-3-8B outperforms GPT-4 on accuracy, cost, and latency. The frontier models remain the quality ceiling for open-ended reasoning, but the volume frontier will increasingly be served by specialists. The architectural implication: the model layer splits into a frontier path and a specialist path, routed by the orchestration layer.

Trend 5 · On-device and edge AI become default for privacy

Apple's on-device models, Qualcomm's NPUs, and the GGUF ecosystem have pushed competent inference onto consumer devices. For any product touching PII, the default posture is shifting from "send it to the cloud" to "keep it on the device." Frameworks that do not have a credible edge story will be ceded to those that do.

Trend 6 · Security becomes a framework concern

Guardrails started as an afterthought library and are becoming a runtime. Expect a future where model invocations go through a security middleware by default — prompt injection detection, PII scrubbing, policy enforcement — just as HTTP requests today go through a CORS and auth middleware. The frameworks that lead here will look like Cloudflare: ubiquitous, invisible, essential.

> **What to bet on**
> A stack you control at the orchestration layer (LangGraph), an interchangeable model layer (fine-tuned local + frontier API), a retrieval layer that is search-shaped (hybrid, not vector-only), and a security layer you treat as critical infrastructure. The specific framework names will evolve; the shape will not.

23. Glossary

Abstraction level	How much code you write to achieve a result, and how much the framework decides for you. High abstraction = fewer lines, more defaults.
Adapter (LoRA)	A small set of trainable parameters (typically <1% of the base model) added on top of a frozen base model. Enables domain specialization without re-training.
Agent	An LLM-driven loop: the model decides the next action, a tool is called, the result feeds back into the next decision. Different from a pipeline in that the path is not pre-declared.
Chain (LangChain sense)	A fixed sequence of LLM calls and transformations, distinct from an agent because the sequence is pre-declared, not chosen at runtime.
Chunk	A substring of a document, typically 500-1000 tokens, embedded as a unit for retrieval.
Cross-encoder	A ranking model that scores a query-document pair directly, unlike a bi-encoder that embeds each side separately. Slower but more precise — used for reranking the top-k retrieval results.
DSL	Domain-specific language. NeMo Guardrails uses one (Colang) for dialogue policies.
Embedding	A dense vector representation of a piece of text, produced by a model trained so that semantically similar texts have similar vectors.
Fine-tuning	Further training of a pre-trained model on domain-specific data. LoRA / QLoRA are the common parameter-efficient variants.
Guardrails	Policies and validators enforced between the model and the user on input and output — injection detection, schema validation, PII scrubbing.
Hybrid search	Retrieval that combines vector similarity with lexical search (BM25). Generally outperforms either alone for enterprise content.
Inference	Running a trained model to produce outputs (as opposed to training it). The serving layer exists to make inference fast and concurrent.
LoRA	Low-Rank Adaptation. A parameter-efficient fine-tuning method that trains a small rank-constrained adapter matrix instead of the full model weights.
MLOps	The operational discipline around machine learning — experiment tracking, model registry, deployment, monitoring. Covered by MLflow, W&B, Kedro.
Observability (LLM)	Trace-level visibility into LLM-driven systems: every call, every retrieval, every token, attributed and measurable. Langfuse and LangSmith are the primary tools.
Orchestration	The logic that composes multiple LLM calls, tool invocations, and retrievals into a coherent application behavior. LangChain, LlamaIndex, Semantic Kernel.

PEFT	Parameter-Efficient Fine-Tuning. Hugging Face's library implementing LoRA, QLoRA, prefix tuning, and other techniques that modify a small fraction of model parameters.
Prompt injection	An attack where user input is crafted to override the system prompt's instructions. The LLM-era equivalent of SQL injection.
QLoRA	LoRA applied to a 4-bit-quantized base model. Drastically reduces VRAM requirements, making fine-tuning possible on consumer GPUs.
RAG	Retrieval-Augmented Generation. The pattern of retrieving relevant documents at query time and providing them to the LLM as context.
Re-ranker	A model that scores retrieval candidates against the query and reorders them by relevance. A cross-encoder used on the top-k output of a fast retriever.
Semantic kernel (concept)	A plugin architecture for LLM applications: each plugin is a declarative function the LLM can call. Also the name of Microsoft's framework.
State machine	An execution model where state is explicit and transitions are conditional. LangGraph's core abstraction.
Token	A unit of text used by LLMs — roughly a word or subword. Pricing, context windows, and throughput are all measured in tokens.
Vector database	A database optimized for similarity search over high-dimensional vectors (embeddings). Weaviate, Pinecone, Chroma, FAISS.
vLLM	A high-throughput LLM serving engine using paged attention for efficient request batching.
ZeRO	Zero Redundancy Optimizer — DeepSpeed's technique for partitioning optimizer state, gradients, and parameters across GPUs. Enables training of models larger than any single GPU.

24. Further Reading

The framework landscape moves too fast for a printed atlas to be current indefinitely. The sources below are the ones that move faster than this document and are worth following.

Primary project documentation

- **LangChain / LangGraph / LangSmith** — python.langchain.com
- **LlamaIndex** — docs.llamaindex.ai
- **Hugging Face Transformers + Datasets + PEFT + TRL** — huggingface.co/docs
- **vLLM** — docs.vllm.ai
- **Ray / Ray Serve** — docs.ray.io
- **MLflow** — mlflow.org
- **Weaviate** — weaviate.io/developers/weaviate
- **Axolotl** — github.com/axolotl-ai-cloud/axolotl
- **Guardrails AI / NeMo Guardrails** — guardrailsai.com / github.com/NVIDIA/NeMo-Guardrails

Analytical sources

- **State of AI Report** — stateof.ai — annual, strategic view.
- **The Generative AI Landscape (a16z)** — quarterly market map.
- **Papers with Code** — paperswithcode.com — benchmarks and SOTA.
- **Hugging Face Open LLM Leaderboard** — current model quality.
- **EleutherAI / LMSYS Chatbot Arena** — adversarial model evals.

Practitioner blogs worth following

- Simon Willison — simonwillison.net — daily notes on the LLM ecosystem.
- Eugene Yan — eugeneyan.com — production ML, eval-driven development.
- Hamel Husain — hamel.dev — pragmatic LLM application engineering.
- Jason Liu — jxnl.co — structured outputs, agents, DSPy-style thinking.
- Chip Huyen — huyenchip.com — MLOps and design patterns for ML systems.

Source document

This atlas was built on top of a framework comparison document (FW.docx) summarizing seven framework categories with an eleven-column attribute matrix. The atlas expands that matrix with three additional layers (fine-tuning, observability, guardrails), detailed column definitions, comparative visualizations, and four reference architectures with working code.

End of atlas — v4.0 · April 2026