

OGX: An Open-Source, Vendor-Neutral Generative AI Application Server¹

Francisco Javier Arceo¹, Sébastien Han¹, Matthew Farrellee³, Charlie Doern¹, Yuan Tang¹,
Derek Higgins¹, Varsha Prasad Narsing¹, Gordon Sim¹, Sumanth Kamenani¹, Ben
Browning¹, Raghotham Murthy²

¹Red Hat AI ²Meta ³Independent

June 2026

Abstract

OGX (Open GenAI Stack) is an open-source AI application server and Python library that implements the APIs of major frontier labs (OpenAI, Anthropic, Google) with pluggable backend providers. Developers building agentic AI applications—such as retrieval-augmented generation pipelines, multi-turn agents, and tool-calling workflows—can develop against a single API surface and deploy with any combination of inference engine, vector database, and safety backend, without changing application code. OGX’s primary focus is the Responses API for server-side agentic orchestration, conforming to the Open Responses specification. The server also supports the Anthropic Messages API and Google GenAI Interactions API, decoupling SDK choice from model and deployment decisions. With over 20 inference providers, 13 vector store backends, and a companion Kubernetes Operator for production deployment, OGX serves as the self-hosted, model-agnostic backend for AI-powered developer tools including Claude Code, Codex CLI, OpenCode, and OpenHands. The project has over 8,400 GitHub stars, 242 contributors, and 4,000 commits across nearly two years of public development.

1 Introduction

AI application development today is tightly coupled to proprietary API providers. While inference-only workloads can increasingly be swapped across providers—vLLM, for example, supports the Responses API for basic inference—applications that rely on the full stack (retrieval, tool calling, conversation state, safety guardrails) remain difficult to migrate without rewriting significant application logic. This coupling limits deployment flexibility and prevents organizations from running AI workloads on their own infrastructure—a requirement in regulated industries and air-gapped environments.

The problem is compounded by the emergence of AI-powered developer tools. Tools like Claude Code, Codex CLI, OpenCode, and OpenHands need a backend that can serve models, execute tools, manage conversation state, and handle retrieval—all through standard APIs. Organizations that want to self-host these capabilities currently must assemble and integrate multiple systems: an inference engine, a vector database, a tool execution runtime, and custom glue code to connect them.

OGX addresses this by providing a complete, self-hosted AI application server that implements the OpenAI API surface—with the Responses API as its primary focus—alongside Anthropic Messages and Google GenAI Interactions compatibility layers. It decouples three decisions that are currently entangled: which SDK to use, which model to run, and where to deploy. Developers write

¹OGX (Open GenAI Stack): <https://github.com/ogx-ai/ogx> — Docs: <https://ogx-ai.github.io>

code against standard endpoints and swap the underlying infrastructure through configuration, not code changes.

This paper describes OGX’s design, architecture, and positioning in the AI application ecosystem. Section 2 surveys the state of the field. Section 3 details the software design. Section 5 describes the Kubernetes Operator. Section 6 discusses research impact and adoption.

2 State of the Field

The AI application ecosystem has stratified into layers that each solve a subset of the deployment problem. Table 1 summarizes the landscape.

Inference engines (vLLM [9], SGLang [20], Ollama) focus on efficient model serving. They optimize throughput and latency but do not provide retrieval, conversation state, tool execution, or safety guardrails. An application using vLLM for inference must separately integrate a vector database, implement its own agentic loop, and manage multi-turn state.

API gateways (LiteLLM, OpenRouter) provide a unified interface across multiple inference providers but act as pass-through proxies. They do not manage vector stores, execute tool calls, or maintain conversation history—they translate request formats between SDKs and providers.

Client-side frameworks (LangChain [10], LangGraph [11], LlamaIndex [12], CrewAI [4], Haystack [6]) provide rich developer abstractions for building agents and RAG pipelines. However, they execute orchestration client-side, distributing security-critical logic across application code. These frameworks are complementary to OGX: they compose agent logic while OGX provides the server-side execution target they call into.

Proprietary platforms (OpenAI’s Responses API [16], Databricks Mosaic AI [5]) offer integrated experiences but couple applications to a specific vendor’s infrastructure and pricing.

OGX occupies a distinct position: a self-hosted, vendor-neutral server that implements the full API surface—inference, retrieval, tool execution, conversation management, and safety—with pluggable providers at every layer. Its conformance to the Open Responses specification [15] ensures interoperability with any client that speaks the same protocol.

System	Inference	RAG	Tools	State	Safety	Deployment
vLLM / SGLang	✓	–	–	–	–	Self-hosted
Ollama	✓	–	–	–	–	Local
LiteLLM	✓*	–	–	–	–	Gateway
LangChain / LangGraph	✓*	✓	✓	✓	–	Client-side
OpenAI Platform	✓	✓	✓	✓	✓	SaaS
OGX	✓	✓	✓	✓	✓	Local & Self-hosted

Table 1: Feature comparison across AI application system categories. *Proxy/wrapper—delegates to external provider.

3 Software Design

3.1 Provider Architecture

OGX’s core abstraction is the pluggable provider. Each API capability—inference, vector storage, safety, tool runtime, file processing—is defined by a Protocol interface. Concrete providers

implement these interfaces for specific backends (Figure 1):

- **Inference:** `remote::openai`, `remote::anthropic`, `remote::vllm`, `remote::ollama`, `remote::bedrock`, `remote::gemini`, and 15+ others.
- **Vector stores:** `inline::faiss`, `inline::sqlite-vec`, `remote::pgvector`, `remote::qdrant`, `remote::milvus`, `remote::weaviate`, `remote::chromadb`, and others.
- **Safety:** Content moderation providers for input/output guardrails.
- **Tools:** Built-in tools (file search, web search, code interpreter) and MCP [1] integration for external tool servers.

A routing layer dispatches requests to provider instances based on logical resource identifiers, enabling multiple providers to serve the same API simultaneously. For example, Ollama can handle local models while OpenAI handles hosted models, both behind `/v1/chat/completions`. The routing table tracks which provider owns each resource (model, vector store, file), enabling auto-dispatch without client-side logic.

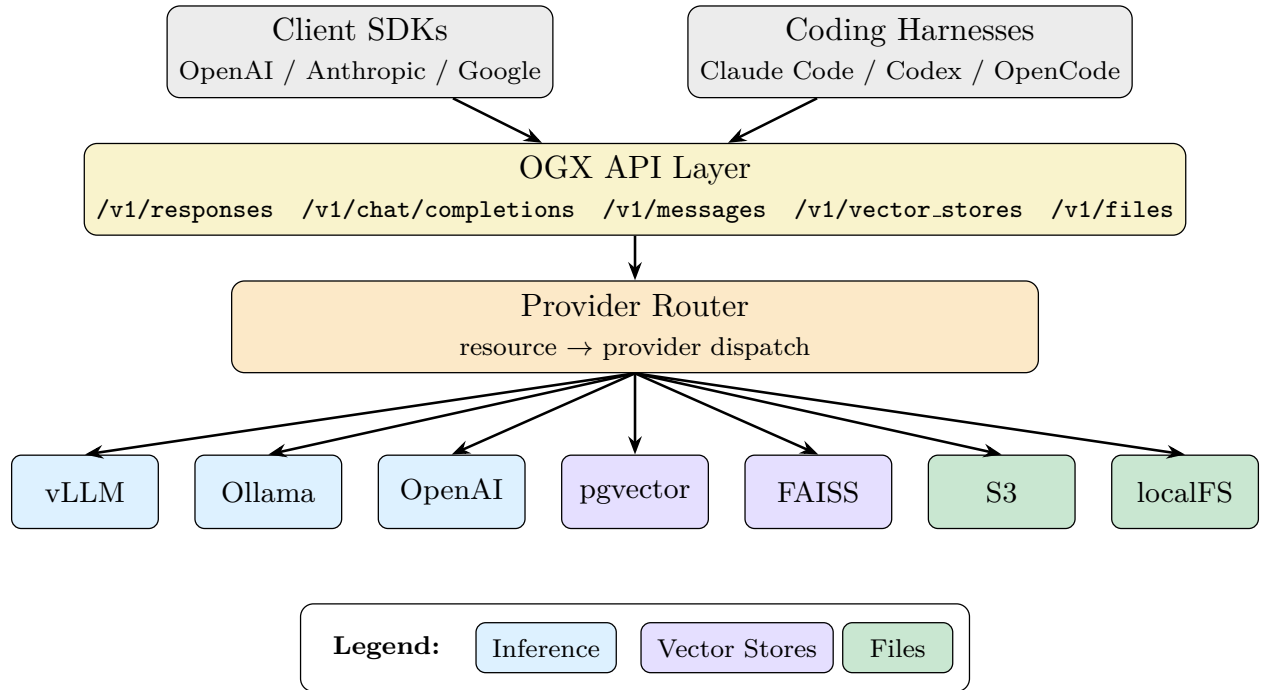


Figure 1: Provider architecture: client requests are dispatched through the API layer and router to pluggable backend providers. Representative APIs and providers shown; OGX additionally exposes Containers, Skills, Safety, Prompts, Conversations, and Telemetry APIs.

3.2 Distributions

A *distribution* packages a specific set of providers and configuration into a deployable unit, decoupling application logic from infrastructure selection. Developers prototype with lightweight inline providers (Ollama for inference, sqlite-vec [7] for vectors) and deploy to production backends (vLLM [9], pgvector) by changing the distribution configuration, not the application code. OGX ships a **starter** distribution for quick setup and supports custom distributions for production environments.

3.3 Dual Deployment Model

OGX runs in two modes:

- **Server mode:** HTTP endpoints accessible from any language or tool. This is the production deployment model.
- **Library mode:** Direct Python import with zero network overhead. Suitable for notebooks, scripts, and rapid prototyping.

Both modes use identical provider routing and API semantics. The recommended progression is: start with the library for prototyping, graduate to the server when you need multi-language access, scaling, or production isolation.

3.4 Multi-SDK Compatibility

OGX serves three client SDK protocols from a single server instance:

- **OpenAI-compatible endpoints** (`/v1/chat/completions`, `/v1/responses`, `/v1/vector_stores`): the primary interface. The Responses API implementation conforms to the Open Responses specification [15].
- **Anthropic Messages endpoint** (`/v1/messages`): native compatibility for teams using the Anthropic SDK.
- **Google GenAI Interactions endpoint** (`/v1alpha/interactions`): native compatibility for teams using the Google GenAI SDK.

This means SDK choice, model selection, and deployment target are fully independent decisions. A team can use the Anthropic SDK to call an open-weight model served by vLLM through OGX, or use the OpenAI SDK to call Anthropic’s Claude—the server handles the translation.

3.5 Server-Side Agentic Orchestration

The Responses API [16] is OGX’s primary API focus and implements server-side agentic orchestration: the inference→tool→inference loop executes within the server process, not the client. This design centralizes several critical concerns:

- **Tool authorization:** Tool calls are executed server-side with centralized authorization, not delegated to potentially untrusted clients.
- **Conversation state:** Multi-turn state is managed server-side with tenant-scoped isolation.
- **Safety guardrails:** Input and output safety checks are applied at each step of the agentic loop.
- **Context management:** A Compaction API summarizes long conversation histories to manage context window limits.

Built-in tools include file search (RAG over vector stores with hybrid dense/sparse retrieval), web search, code interpretation, computer use, and Model Context Protocol (MCP) [1] integration for external tool servers.

For multitenant deployments, OGX provides attribute-based access control (ABAC) that enforces tenant isolation at the retrieval, tool execution, state management, and API routing layers. The security properties of this architecture—specifically, how ABAC-gated retrieval eliminates cross-tenant data leakage while adding negligible overhead—have been formally analyzed and empirically validated [2].

3.6 API Surface

OGX exposes over 20 APIs covering the full lifecycle of AI applications (Table 2):

- **Inference APIs:** Chat Completions and Embeddings provide standard inference endpoints. The Responses API serves as the central agentic orchestration endpoint, coordinating multi-turn conversations, tool calling, and server-side execution [16].
- **Data APIs:** Vector Stores and Search APIs support dense, sparse, and hybrid retrieval with structured metadata filtering for tenant isolation. The Files and File Processors APIs provide tenant-scoped file storage (S3, GCS, PVCs, local filesystem) with parsing and chunking for vector store ingestion. The Batches API supports offline processing.
- **Agent and Tool APIs:** Built-in tools include file search (RAG over vector stores), web search, code interpreter, shell (via Containers), image generation, and computer use. External tools integrate via MCP [1]. Function Calling and Connectors enable custom tool integration.
- **Execution APIs:** The Containers API (`/v1/containers`) manages sandboxed execution environments—matching the OpenAI Containers API—enabling models to execute shell commands in isolated Docker, Podman, or Kubernetes containers via a `shell` tool in the Responses API. The Skills API (`/v1alpha/skills`) manages versioned skill bundles that package tools, prompts, and configuration into reusable, composable units for domain-specific capabilities.
- **State APIs:** The Conversations API persists multi-turn history with tenant-scoped isolation, eliminating client-side session stores. The Prompts API provides versioned prompt template management. A resource registry tracks models, vector stores, files, and tool groups as first-class server objects with ownership metadata. The Compaction API automatically summarizes long conversation histories to manage context window limits. This server-side state layer is a distinguishing feature—most inference engines and API gateways treat requests as stateless, requiring applications to build their own persistence. OGX manages this natively, enabling it to function as a complete application server rather than a stateless proxy.
- **Safety APIs:** Content moderation providers apply input and output guardrails at each step of the agentic loop.
- **Admin APIs:** Model registry for managing available models across providers. Telemetry via OpenTelemetry (OTEL) for observability and tracing of agent execution, with built-in MLflow [19] tracing integration (which supports OTEL) for logging spans, tool calls, and retrieval steps to existing ML experiment tracking infrastructure.

Category	APIs
Inference	Chat Completions, Responses (agentic orchestration), Embeddings
Data	Vector Stores, Search, Files, File Processors, Batches
Agent/Tools	File search, web search, code interpreter, shell, image generation, computer use, MCP, Function Calling, Connectors
Execution	Containers (sandboxed environments), Skills (versioned bundles)
State	Conversations, Prompts, Compaction, Resource Registry
Safety	Content moderation (input/output guardrails)
Admin	Models, Telemetry (OTEL, MLflow tracing)

Table 2: OGX API surface organized by category.

4 Example Usage

The following examples demonstrate OGX’s portability: the same client code works regardless of which inference provider or vector store backend is configured on the server.

4.1 RAG Agent with the OpenAI SDK

Building a RAG agent requires only standard OpenAI SDK calls. The server handles document chunking, embedding, vector storage, retrieval, and context injection transparently:

```
from openai import OpenAI

client = OpenAI(base_url="http://localhost:8321/v1",
                api_key="unused")

# Create a vector store and upload documents
vector_store = client.vector_stores.create(name="docs")
client.vector_stores.files.upload(
    vector_store_id=vector_store.id,
    file=open("manual.pdf", "rb"),
)

# Query with server-side RAG via the Responses API
response = client.responses.create(
    model="meta-llama/Llama-3.2-3B-Instruct",
    input="What are the installation requirements?",
    tools=[{"type": "file_search",
            "vector_store_ids": [vector_store.id]}],
)
print(response.output_text)
```

Switching from a local Ollama backend to a production vLLM cluster requires changing only the server’s distribution configuration—the client code above remains identical.

4.2 Published Use Cases

OGX’s provider portability has been demonstrated across diverse enterprise backends:

- **IBM watsonx.ai + Milvus:** Enterprise RAG pipeline using watsonx.ai for inference and watsonx.data Milvus for vector storage, with Llama Stack as the unifying orchestration layer [8].
- **Oracle Cloud Infrastructure:** Generative AI application development using OCI AI Blueprints with Llama Stack for standardized API access [17].
- **Red Hat OpenShift:** An intelligent operations agent combining agentic RAG, web search, and MCP tool integration (OpenShift cluster management, Slack notifications) for automated incident response [18].

The framework has been presented at Meta Connect [13] and IBM TechXchange [3] as a standardization layer for enterprise AI applications.

5 Kubernetes Operator

The OGX Kubernetes Operator [14] provides declarative, production-grade deployment through the `OGXServer` custom resource definition (CRD). Written in Go using the `operator-sdk` framework, it automates the full lifecycle of OGX server deployments on Kubernetes and OpenShift.

5.1 Custom Resource Model

A single `OGXServer` CR specifies:

- **Distribution:** Which AI stack variant to deploy (e.g., `starter`, custom distributions).
- **Workload:** Replica count, persistent storage size and mount paths, environment variable overrides for inference model configuration.
- **Network:** External access configuration (hostname-based routing) and network policies (enabled by default per-CR).

The operator reconciles the desired state expressed in the CR with the actual cluster state, handling creation, scaling, updates, and teardown of OGX server pods.

5.2 Operational Features

- **ConfigMap-driven image overrides:** Administrators update the `ogx-operator-config` ConfigMap with an `image-overrides` key, and all matching `OGXServer` resources restart with the new image—enabling fleet-wide image updates without redeploying the operator.
- **ConfigMap-based configuration:** Users supply `config.yaml` content via ConfigMaps; the operator watches for changes and automatically restarts pods to load updated configuration.
- **Multi-architecture builds:** Supports `linux/amd64` and `linux/arm64` with FIPS-compliant images built using native architecture-matched CI runners.
- **Dual platform support:** First-class support for both vanilla Kubernetes (using cert-manager for webhook TLS) and OpenShift (using built-in service-serving-cert-signer).
- **Quickstart scripts:** `hack/deploy-quickstart.sh` enables rapid setup with provider and model flags.

5.3 Isolation Topologies

The operator supports three deployment topologies:

1. **Shared instances:** Multiple tenants share a single OGX server with ABAC-enforced logical isolation.
2. **Per-tenant instances:** Namespace-level isolation with Kubernetes RBAC, each tenant receiving a dedicated OGX server.
3. **Hybrid:** Shared inference with per-tenant data paths, balancing cost efficiency with isolation requirements.

6 Research Impact and Adoption

6.1 Production Deployments

OGX is deployed in production across enterprises in telecommunications, semiconductor manufacturing, financial services, insurance, and consulting. These deployments use the full stack: pluggable inference providers, tenant-isolated vector stores, server-side agentic orchestration, and Kubernetes-based deployment via the operator. Published use cases span IBM watsonx.ai with

Milvus vector storage [8], Oracle Cloud Infrastructure with OCI AI Blueprints [17], and Red Hat OpenShift with MCP-based operational agents [18].

6.2 Academic Validation

The security architecture of OGX’s multitenant isolation model was formally analyzed and empirically validated in a peer-reviewed publication at the ACM Conference on AI and Agentic Systems (CAIS ’26) [2]. The evaluation demonstrated that ABAC-gated retrieval eliminates cross-tenant data leakage (0% cross-tenant leakage rate) while adding approximately 19ms to the search path. The defense operates at the retrieval layer, making it resilient to prompt injection attacks regardless of model behavior.

6.3 Community and Ecosystem

As of June 2026, the project has:

- Over 8,400 GitHub stars and 1,300 forks.
- 242 unique contributors and over 4,000 commits.
- 68 releases across nearly two years of public development (since July 2024).
- Weekly community contributor calls and an active Discord server.
- Integrations contributed by external organizations including Red Hat, IBM, Oracle, and Infinispan.

OGX conforms to the Open Responses specification [15] and serves as a reference implementation for open, vendor-neutral agentic AI APIs.

6.4 AI-Powered Developer Tools

OGX is designed to serve as the backend for AI-powered developer tools that implement OpenAI-compatible APIs, including Claude Code, Codex CLI, OpenCode, and OpenHands. By providing a self-hosted, model-agnostic server that speaks these protocols, OGX enables organizations to use these tools with any model on their own infrastructure.

7 Acknowledgements

We thank Meta for creating and open-sourcing Llama Stack, the foundation from which OGX evolved. We are grateful to Red Hat for supporting the development of OGX. We thank the OGX contributor community for their sustained contributions to the project.

References

- [1] Anthropic. *Model Context Protocol Specification*, 2026. <https://modelcontextprotocol.io/specification/>.
- [2] Francisco Javier Arceo and Varsha Prasad Narsing. Securing the agent: Vendor-neutral, multitenant enterprise retrieval and tool use. In *Proceedings of the ACM Conference on AI and Agentic Systems*, CAIS ’26, pages 862–872, New York, NY, USA, 2026. Association for Computing Machinery.

- [3] Cedric Clyburn. Llama stack: Kubernetes for RAG and AI agents in generative AI, 2025. https://mediacenter.ibm.com/media/Llama+Stack+Kubernetes+for+RAG+AI+Agents+in+Generative+AI/1_xl78upq2.
- [4] CrewAI, Inc. CrewAI: Framework for orchestrating role-playing autonomous AI agents, 2024. <https://github.com/crewAIInc/crewAI>.
- [5] Databricks. *Mosaic AI Agent Framework*, 2025. <https://www.databricks.com/product/machine-learning/retrieval-augmented-generation>.
- [6] deepset. Haystack: End-to-end LLM framework for building production-ready applications, 2023. <https://github.com/deepset-ai/haystack>.
- [7] Alex Garcia. sqlite-vec: A vector search SQLite extension, 2024. <https://github.com/asg017/sqlite-vec>.
- [8] IBM Community. Build RAG with Llama Stack and watsonx.data Milvus, 2025. <https://community.ibm.com/community/user/blogs/divya13/2025/05/08/build-rag-with-llama-stack-and-watsonxdata-milvus>.
- [9] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles*, pages 611–626, New York, NY, USA, 2023. Association for Computing Machinery.
- [10] LangChain, Inc. LangChain: Build context-aware reasoning applications, 2023. <https://github.com/langchain-ai/langchain>.
- [11] LangChain, Inc. LangGraph: Build resilient language agents as graphs, 2024. <https://github.com/langchain-ai/langgraph>.
- [12] LlamaIndex. LlamaIndex: Data framework for LLM applications, 2022. https://github.com/run-llama/llama_index.
- [13] Meta. Llama stack: Chapter one, 2024. <https://developers.facebook.com/m/meta-connect-developer-sessions/llama-stack-chapter-one/>.
- [14] OGX Contributors. OGX kubernetes operator, 2026. <https://github.com/ogx-ai/ogx-k8s-operator>.
- [15] Open Responses Community. *Open Responses Specification*, 2026. <https://www.openresponses.org/>.
- [16] OpenAI. *Responses API Reference*, 2025. <https://platform.openai.com/docs/api-reference/responses>.
- [17] Oracle. Accelerating enterprise gen AI applications development on OCI with Llama Stack and OCI AI blueprints, 2025. <https://blogs.oracle.com/ai-and-datascience/accelerating-enterprise-gen-ai-applications-development-on-oci-with-llama-stack-and-oci-ai-blueprints>.

- [18] Red Hat. Generative AI applications with Llama Stack: A notebook-guided journey to an intelligent operations agent, 2025. <https://www.redhat.com/en/blog/generative-ai-applications-llama-stack-notebook-guided-journey-intelligent-operations-agent>.
- [19] Matei A. Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, Fen Xie, and Corey Zumar. Accelerating the Machine Learning Lifecycle with MLflow. *IEEE Data Eng. Bull.*, 41:39–45, 2018.
- [20] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kober, Liang Shi, Chien-Sheng Wu, Hao Zhang, Ying Sheng, Joseph E. Gonzalez, Ion Stoica, and Wei-Lin Ma. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems*, volume 37. Curran Associates, Inc., 2024.