XINITY

WHITEPAPER

On-Premises Enterprise Inference, Engineered for Sovereignty.

The open-source platform for serving large language models inside your own infrastructure. OpenAI-compatible, multi-node, production-ready. Zero mandatory data egress.

Sovereign by architecture, not by contract.

RUN AI WHERE YOUR DATA LIVES.

Contents

SECTION 01

Executive Summary

Xinity AI is an open-source, on-premises platform for managing and serving large language models (LLMs) in enterprise environments where data cannot leave the organization's infrastructure. Built for sectors such as media, manufacturing, healthcare, finance, and public institutions, Xinity provides the full operations layer around model inference: an OpenAI-compatible API, a management dashboard, multi-node GPU orchestration, load balancing, observability, and enterprise-grade access control. All of it runs with a guarantee of zero data egress.

This whitepaper describes the technical architecture, performance characteristics, and operational model of Xinity AI, with a focus on how it compares to running inference engines like Ollama directly, and how it supports high-availability, 24/7 production deployments.

Key Differentiators

/01 Best-of-both-worlds inference

Xinity combines the ease-of-use model management of Ollama with the throughput, speed, and robustness of vLLM, selectable per deployment.

/02 Transparent, auditable, and on-premises

All code is open-source and self-hosted. No data leaves the customer's infrastructure.

/03 Production-ready operations

Built-in load balancing, canary deployments, observability, and multi-node scaling. These capabilities go well beyond what an inference engine provides on its own.

SECTION 02

The On-Premises Challenge

Cloud AI platforms operate on the assumption that data can be sent externally. For many enterprises, this assumption does not hold, not by preference but by law. GDPR, banking secrecy, journalistic source protection, attorney-client privilege, and trade secret regulations make external data processing legally impossible for certain workloads.

Organizations in these sectors are not cloud-skeptical. They are sovereignty-blocked.

They require the capabilities of modern AI platforms, from model serving and API integration to observability, access management, and data labeling, without any data leaving their hardware. Xinity addresses this by providing a complete, self-hostable AI platform:

- Model orchestration and deployment management
- OpenAI-compatible API that works with existing client code
- Multi-node GPU scheduling and load balancing
- Web-based management dashboard with RBAC
- Usage tracking, data collection, and labeling pipelines
- SSO, 2FA, and enterprise authentication
- Prometheus metrics and Grafana dashboards

> THE UTILIZATION INVERSION
>
> Cloud pricing models assume roughly 15% utilization with bursty workloads. For always-on internal AI agents running at 80 to 90% utilization, dedicated on-premises infrastructure delivers approximately **80% cost savings** compared to equivalent cloud capacity. Sovereignty is not just a compliance requirement. It is an economic advantage.

SECTION 03

Architecture Overview

Xinity AI is a modular platform composed of independently deployable services that coordinate through a shared PostgreSQL database.

3.1 Component Summary

COMPONENT	RUNTIME	ROLE
Gateway	Bun native HTTP	Public API entry point; load balancing; request routing; usage logging
Dashboard	SvelteKit 2 + Bun	Management UI; oRPC API; deployment orchestration; data labeling
Daemon	Bun	Runs on each inference node; model lifecycle management; GPU auto-detection
Info Server	Bun	Stateless model catalog service; YAML-based model definitions
PostgreSQL	Postgres 17	Shared database for control-plane coordination
Redis	Redis 7	Gateway ephemeral state (auth cache, load balancer, response store)

3.2 Ollama + vLLM: Best of Both Worlds

Xinity AI supports both **Ollama** and **vLLM** as inference backends, allowing operators to choose the right engine for each use case:

- **Ollama** offers the convenience and agility of pulling and running models from a public registry with minimal configuration. It is ideal for rapid prototyping, smaller workloads, and environments where operational simplicity is the priority.
- **vLLM** provides significantly higher throughput and better performance under load, with features such as PagedAttention, continuous batching, and tensor parallelism. It is the engine of choice for high-performance production workloads and large models.

Operators can deploy either engine, or both side by side. Xinity's daemon manages the full lifecycle of both backends, and the gateway routes requests transparently to whichever engine is serving the requested model. This flexibility means teams can start with Ollama for agility and migrate to vLLM for scale, without changing their application code or API interface.

3.3 Service Communication Model

The architecture follows a **shared-database pattern** for control-plane coordination:

- The **Dashboard** writes deployment plans (`modelInstallation` rows) to PostgreSQL and triggers immediate updates via PostgreSQL `LISTEN/NOTIFY`.

- The **Daemon** on each GPU node reads these plans, manages model installations (pulling via Ollama or starting via vLLM), and reports actual state (`modelInstallationState`) back to the database.
- The **Gateway** reads the database for API key verification, model resolution, and host selection. It is the only service that communicates directly with the daemon: inference requests are forwarded over HTTP(S) to the target node.

This design keeps each service thin, independently deployable, and resilient to individual service failures.

3.4 Request Flow

01 Client sends a request to the gateway (e.g., `POST /v1/chat/completions`).

02 The gateway authenticates the request via API key (cached in Redis).

03 The requested model is resolved through the deployment registry.

04 A load-balanced inference node is selected.

05 The request is forwarded to the target daemon, which proxies it to the local inference driver (Ollama or vLLM).

06 The response streams back through the same path to the client.

07 Usage events and (optionally) full request/response data are logged to PostgreSQL.

SECTION 04

Load Balancing & Throughput

4.1 Gateway Overhead

The gateway layer adds enterprise features (authentication, routing, usage logging, retries, admission control) on top of the raw inference pipeline. It is designed to add minimal overhead while providing substantial operational benefits. In benchmark tests, per-stream token generation throughput through the gateway was identical to the underlying inference engine, meaning the gateway does not become a bottleneck.

Under heavy load, the gateway actually improves throughput compared to calling an inference engine directly. The gateway's bounded request admission keeps the inference engine in its efficient operating range, and its retry layer catches and transparently re-dispatches transient errors before the client ever sees a failure. The result is significantly higher completion rates under sustained high concurrency. That is the difference between a usable production system and one that falls apart under real-world traffic.

4.2 Load Balancing Strategies

The gateway supports three load-balancing strategies, configurable via `LOAD_BALANCE_STRATEGY`:

STRATEGY	BEHAVIOR
least-connections (`default`)	Picks the node with the fewest in-flight requests, tracked in Redis. Falls back to random on Redis errors.
round-robin	Atomic Redis counter per model, rotating through nodes.
random	Uniform random selection.

4.3 Prefix-Cache Affinity

For `least-connections` and `random` strategies, the gateway implements **prefix-cache affinity**: conversation message prefixes are hashed and mapped to nodes in Redis (TTL: 5 minutes). This ensures repeat conversations are routed to the same node, improving KV-cache hit rates on the inference engine. With `least-connections`, affinity is honored only when the hinted node is within 2 connections of the least-loaded node.

4.4 Canary Deployments

Xinity supports gradual model rollouts through canary deployments:

- **Manual mode:** Traffic splits at a fixed percentage until manually advanced.
- **Time-based mode:** Progress interpolates linearly over a configured time window, automatically moving all traffic to the primary model when the window expires.

This allows safe A/B testing of new model versions in production without downtime.

SECTION 05

Multi-Node Scaling

5.1 Deployment Strategies

The Dashboard's deployment sync service supports four node-selection strategies:

STRATEGY	BEHAVIOR
first-fit (`deterministic`)	First available node that fits the model's requirements.
balanced (`default`)	Selects the node with the most absolute free VRAM, naturally spreading load for HA.
bin-pack	Tightest fit to consolidate workloads, keeping some nodes idle and drainable.
proportional	Lowest percent utilization for fair spread across heterogeneous hardware.

5.2 Supported Hardware

The Daemon auto-detects GPUs across vendors:

VENDOR	DETECTION METHOD
NVIDIA	`nvidia-smi` (GPU name, VRAM per card)
AMD	Sysfs (`/sys/class/drm`), fallback to `rocm-smi`
Intel	`xpu-smi` discovery

Unified memory systems (e.g., DGX Spark) are detected when GPUs report zero VRAM, with 90% of system RAM allocated as usable capacity. CPU-only inference is supported when no GPUs are present.

5.3 Inference Engine Lifecycle

ENGINE	MODE	DETAILS
Ollama	Pull-based	Models pulled from Ollama registry with streaming progress. Up to 2 concurrent pulls.
vLLM (`systemd`)	Service-managed	Manages `vllm-driver@{id}.service` template units. Auto-computes GPU memory utilization with overhead, capped at 90%.
vLLM (`Docker`)	Containerized	Runs in containers on an egress-blocked network (`HF_HUB_OFFLINE=1`). Models pre-downloaded by the daemon. Blocks all outbound internet from the inference process.

Failed vLLM processes are automatically restarted up to 3 times (configurable). Fatal errors (GPU OOM, CUDA errors) trigger immediate failure without retries.

SECTION 06

High Availability & 24/7 Operation

6.1 Operational Philosophy

Because Xinity runs entirely on customer-owned infrastructure, uptime guarantees depend on the customer's own infrastructure choices and operational practices. Xinity cannot and does not make hard SLA commitments. Instead, the platform is architected to **make high availability straightforward** for operators who want it.

6.2 Built-in Resilience

FEATURE	DESCRIPTION
Horizontal scalability	Every service (gateway, daemon, dashboard) can be deployed as multiple replicas running alongside each other.
Automatic failover	If a daemon node goes offline, the gateway's load balancer immediately routes around it. Offline nodes are detected and excluded from host selection.
Redundant inference	Deploying the same model on multiple nodes (via multi-replica deployments) ensures inference continues if any single node fails.
Daemon process recovery	Daemon-managed inference processes auto-restart up to 3 times on failure.
Health monitoring	The Dashboard Compute page provides real-time views of all GPU nodes with auto-polling every 12 seconds.

6.3 Recommended HA Setup

For a 24/7 production deployment:

01 **Multiple gateway replicas** behind a load balancer (e.g., Caddy, nginx, or a hardware load balancer).

02 **Managed PostgreSQL cluster** (replication, failover, automated backups).

03 **High-availability Redis** (Redis Sentinel or cluster mode).

04 **Multi-node inference pool** with the same models deployed across several GPU machines using `DEPLOYMENT_STRATEGY=balanced`.

05 **Redundant infrastructure** for DNS, TLS termination, and network connectivity.

With this level of redundancy, outages become highly improbable. The platform makes it possible; the operational implementation is in the customer's hands.

6.4 Deployment Sync Reliability

The Dashboard's deployment sync service runs continuously (every 5 minutes, with immediate triggers via PostgreSQL `NOTIFY`). It maintains consistency between desired state (what the dashboard plans) and actual state (what daemons are running), automatically correcting drift.

SECTION 07

Monitoring & Observability

7.1 Prometheus Metrics

All three core services expose Prometheus metrics at `/metrics`:

Gateway metrics include:

- Request rates and latencies by endpoint
- Time-to-first-token histograms per deployment
- Token counts (input/output) per model, API key, and organization
- Generation throughput (tokens/second)
- Backend error rates and client disconnects

Daemon metrics include:

- GPU utilization, memory, temperature, power draw
- GPU throttle state and ECC errors
- Cumulative energy consumption (Wh)
- Node health (`daemon_up` gauge)

Dashboard metrics include:

- HTTP request rates by route
- Node.js runtime metrics (CPU, memory, event loop lag, heap, GC)

Operators can extend the metrics surface by exporting additional Prometheus counters, histograms, or gauges from their own application logic. Xinity's metrics endpoint is fully compatible with custom instrumentation.

7.2 Auto-Discovery

The Dashboard exposes an HTTP service discovery endpoint (`GET /metrics/sd/daemons`) that returns one target group per registered node. Prometheus polls this every 3 minutes and automatically adds/removes nodes as they come online or go offline.

7.3 Pre-built Grafana Dashboards

Four pre-built Grafana dashboards are included:

DASHBOARD	CONTENT
Xinity Overview	High-level stats: request rate, error rate, active requests, daemons up, GPU count, total GPU power
Xinity Gateway	Traffic panels (rate, latency, status, errors), model health (TTFT, backend errors, failure rate), org-level usage
Xinity GPU / Compute	GPU utilization, memory, temperature, power vs. limit, throttling timelines
Xinity Logs (`NixOS only`)	Systemd journal logs for all services via Loki/Promtail

All dashboards are tagged `xinity-ai` and cross-linked via a shared navigation dropdown.

SECTION 08

Security & Compliance

8.1 Authentication

Xinity supports multiple authentication methods:

- **Email/password** with optional invite-only signup
- **SSO via OIDC:** configurable at the instance level or per-organization (with domain verification via DNS TXT records)
- **Passkeys (WebAuthn/FIDO2):** hardware security keys or platform authenticators (Touch ID, Windows Hello)
- **TOTP 2FA:** time-based one-time passwords with backup codes

8.2 Role-Based Access Control (RBAC)

Six granular roles control access across seven resource types (API keys, API calls, responses, deployments, models, applications, audit log):

ROLE	ACCESS
owner	Full access, including organization deletion
admin	Full access to resources, cannot delete organization
member	Full access except audit log
labeler	Read calls, full labeling access, read models/applications
viewer	Read-only across all resources
pending	No access (placeholder awaiting role assignment)

8.3 Data Sovereignty

> ZERO MANDATORY DATA EGRESS
>
> The fundamental design principle of Xinity AI is **zero mandatory data egress**. Inference requests, responses, labels, model weights, and GPU telemetry all stay within the customer's infrastructure by default. There is no analytics, no external telemetry, no external API calls required for core operation.

There are a few optional services that *can* reach out to the wider internet, but only when explicitly enabled by the operator:

- **Info Server:** When customers host their own Info Server (self-hosted model catalog), it reads from a local YAML catalog. When not deployed, the dashboard defaults to a model metadata endpoint hosted by Xinity (`sysinfo.xinity.ai`). It is used solely for model catalog data (names, specs, tags, driver strings), never for inference or customer data.

- **Web search & fetching:** Optional integrations (Google, Bing, Brave, Tavily, SearXNG, etc.) enable web search and page fetching capabilities in the gateway's Responses API. When enabled, these require internet access to fulfill their function. They are opt-in features, never part of the default inference path.

Operators in strict-zero-egress environments can deploy everything locally and disable all optional outbound integrations, achieving full data sovereignty.

8.4 TLS Support

The gateway supports TLS termination with customer-provided certificates. Per-node TLS is tracked in the database, and the gateway automatically negotiates the correct protocol. Inference daemons authenticate with per-node tokens.

SECTION 09

Deployment Options

Xinity recommends three production deployment routes. Other setups are possible as well:

ROUTE	BEST FOR
Xinity CLI	Any Linux server with systemd. One-command installation: `xinity up all`
Docker Compose	Servers with Docker, or cloud VMs. Includes optional profiles for Caddy (HTTPS), SeaweedFS (S3 storage), Infoserver (self-hosted model registry), and Prometheus/ Grafana (monitoring)
NixOS Flake	NixOS infrastructure with declarative, reproducible configuration

9.1 Adding Inference Nodes

The control plane (gateway, dashboard, database, Redis) is deployed once. Each additional GPU node is added independently:

```
xinity up daemon    # on each GPU machine
```

The daemon connects to the shared PostgreSQL instance, registers itself, and begins receiving model deployments. This workflow is identical regardless of how the control plane was deployed.

9.2 Secret Management

All deployment targets support the `_FILE` convention for secrets: any environment variable `KEY` can reference a file path via `KEY_FILE`. The service reads the file at startup and uses its trimmed contents.

SECTION 10

Conclusion

Xinity AI provides a complete, on-premises inference platform for enterprises that cannot route data externally. Its architecture, modular services coordinated through a shared database, enables horizontal scalability, fault tolerance, and operational independence.

By combining the convenience and agility of Ollama with the throughput, speed, and robustness of vLLM, Xinity gives operators the freedom to choose the right engine for each workload. The platform adds the operations layer that production AI demands: load balancing, observability, access control, data labeling, and multi-node scaling. All of it with zero data egress, fully auditable source code, and deployment flexibility spanning Docker, NixOS, and bare-metal systemd.

Further Reading

Architecture documentation

`github.com/xinity-ai/xinity-ai/blob/fix/small-optimizations/docs/architecture.md`

Gateway (load balancing, canary, metrics)

`github.com/xinity-ai/xinity-ai/blob/fix/small-optimizations/docs/features/gateway.md`

Daemon (GPU management, model lifecycle)

`github.com/xinity-ai/xinity-ai/blob/fix/small-optimizations/docs/features/daemon.md`

Deployment guide

`github.com/xinity-ai/xinity-ai/blob/fix/small-optimizations/deployment/README.md`

Monitoring (Prometheus, Grafana)

`github.com/xinity-ai/xinity-ai/blob/fix/small-optimizations/docs/features/monitoring.md`

Run AI where your data lives.
No exceptions.

SOVEREIGN BY ARCHITECTURE, NOT BY CONTRACT.