SoftwareX 30 (2025) 102183

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

Hive: A secure, scalable framework for distributed Ollama inference

Domen Vake [a,b,*], Jernej Vičič [a,c], Aleksandar Tošić [a,b]

[a] University of Primorska, Faculty of Mathematics, Natural Sciences and Information Technologies, Koper, Slovenia
[b] InnoRenew CoE, Izola, Slovenia
[c] Research Centre of the Slovenian Academy of Sciences and Arts, The Fran Ramovš Institute, Ljubljana, Slovenia

ARTICLE INFO

Keywords:
Distributed inference
Large language models
Ollama
Proxy

ABSTRACT

Large Language Models (LLMs) require substantial computational resources, often necessitating distributed inference across multiple machines. However, organizations frequently struggle to unify scattered resources, whether due to firewalls, private networks, or heterogeneous hardware. Traditional approaches demand complex VPN configurations, direct network exposure, or manual workload balancing, making large-scale deployment impractical for many. We present Hive, an open-source framework designed to seamlessly integrate fragmented compute resources into a single, unified inference system. Hive consists of HiveCore, a central proxy that handles client requests, authentication, and task queuing, and HiveNode, a lightweight worker agent that connects securely to HiveCore and executes inference locally. By relying solely on outbound connections, Hive allows remote and isolated machines running Ollama to contribute computational capacity without requiring public network exposure. This architecture enables organizations to leverage both high-performance clusters and legacy hardware, dynamically scaling LLM inference without operational overhead.

Code metadata

Current code version	v0.1.0
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-25-00149
Permanent link to Reproducible Capsule	Not available
Legal Code License	MIT
Code versioning system used	git
Software code languages, tools, and services used	Java, Rust, Shell scripts
Compilation requirements, operating environments & dependencies	Requires Java (17+) with Maven, Rust (1.70+), Ollama locally, SQLite (for HiveCore)
If available Link to developer documentation/manual	https://github.com/VakeDomen/HiveCore and https://github.com/VakeDomen/HiveNode
Support email for questions	domen.vake@famnit.upr.si

1. Motivation and significance

Large Language Models (LLMs) are increasingly employed in various domains, including industry, research, and education. Due to the computational demands of LLM inference, it is common to employ multiple GPU-equipped machines and aggregate their capacity for higher throughput [1]. However, setting up a distributed LLM environment can be challenging when machines are spread across different networks, often hidden behind firewalls or NAT layers. Furthermore, the increasing demand for computational power has driven research institutes and companies to invest heavily in GPU infrastructure long before the advent of LLMs. However, the emergence of LLMs has exacerbated this demand, creating a significant divide between organizations with access to state-of-the-art hardware and those constrained by budgetary limitations, leaving the latter reliant on older, fragmented GPU resources.

These complexities are not just technical but also organizational. DevOps teams face considerable overhead if each node must be publicly reachable or integrated via custom VPN configurations. Security concerns further amplify these challenges, as exposing multiple nodes or juggling complex firewall rules can become a liability. Researchers and

* Corresponding author at: University of Primorska, Faculty of Mathematics, Natural Sciences and Information Technologies, Koper, Slovenia.
E-mail address: domen.vake@famnit.upr.si (Domen Vake).

https://doi.org/10.1016/j.softx.2025.102183
Received 7 March 2025; Received in revised form 14 April 2025; Accepted 28 April 2025
Available online 14 May 2025

smaller institutions, in particular, risk being left behind if they cannot seamlessly unify their limited GPU resources into a cohesive cluster. Although some rely on tools like *nginx* [2] or *ngrok* to relay traffic and bypass NAT, these generic solutions do not address multi-node scheduling, secure key distribution, or dynamic model assignments for varied inference needs.

Hive was conceived to streamline this process by abstracting away low-level networking details and presenting a unified interface for deploying inference tasks across diverse hardware. Rather than placing the burden on every node to handle incoming requests or manage secure connections, *Hive* centralizes these responsibilities within a single proxy, while worker nodes remain privately networked. By consolidating multiple machines — be they cutting-edge GPUs or legacy hardware — under a cohesive system, *Hive* aims to democratize access to high-performance LLM inference, particularly for smaller labs or organizations with limited budget.

Unlike existing inference frameworks such as NVIDIA Triton [3], vLLM [4], and Petals [5], Hive adopts a fundamentally different approach tailored to the specific needs of organizations with fragmented, geographically dispersed resources. NVIDIA Triton is a widely used inference server optimized for centralized, high-performance GPU clusters within data centers, delivering low-latency, high-throughput performance for homogeneous hardware setups. Similarly, vLLM explicitly focuses on maximizing inference performance through sophisticated batching and GPU optimization strategies but expects uniform and centrally managed hardware environments. Petals, on the other hand, takes a decentralized, community-driven approach by allowing volunteers to contribute compute resources; however, this introduces challenges related to reliability, consistency, and administrative control. Hive strikes a balance by merging the administrative and security advantages of central management with the resource flexibility found in decentralized systems.

Hive addresses these infrastructure hurdles by clearly separating the proxy component (*HiveCore*) from the worker component (*HiveNode*). Each node merely requires an outbound connection to *HiveCore*, which handles incoming client requests, key management, and queuing. By focusing on this proxy-worker architecture, *Hive* drastically reduces deployment complexity. Behind the scenes, each worker uses *Ollama*,[1] a lightweight open-source LLM runtime, to execute inference locally. This Ollama integration provides a consistent interface for text generation, embeddings, and other advanced language tasks.

Although Ollama's documentation [6] provides general advice on running single-machine inference, its scope does not fully tackle the networking and organizational complexities of multi-node deployments. Issues such as coordinating multiple geographically dispersed machines, dealing with firewalls or NAT configurations, and managing secure credentials across different teams exceed the coverage of basic setup instructions. As a result, labs and institutions with fragmented GPU resources — often working under strict security policies — struggle to adapt these single-node guidelines for large-scale environments.

Additionally, *Hive* centralizes key and node management in *HiveCore*. Administrators can generate, update, or revoke worker credentials, inspect node load, and even forcibly disconnect or reboot nodes for maintenance. The entire platform focuses on ease of replication and configuration, allowing new nodes to come online with minimal adjustments.

2. Software description

Hive is composed of two primary components: *HiveCore* (the central proxy) and *HiveNode* (the worker node). The repositories for both components are open-source and made available under the MIT License. The ecosystem unifies distributed computing resources behind a single proxy endpoint while minimizing the network configuration overhead typically associated with multi-machine clusters (see Fig. 1).

HiveCore is a Java-based server application that is responsible for orchestrating client requests, managing worker node connections, and enabling administrative control through a dedicated management interface. Internally, it leverages a small embedded SQLite [7] database to store and validate security tokens. This approach makes it straightforward for administrators to create, revoke, or modify tokens used to authenticate nodes and clients. In addition, HiveCore enforces role-based access control as it distinguishes between client tokens (for standard inference requests), worker tokens (for nodes joining the network), and administrator tokens (for full management privileges).

HiveNode is implemented in Rust [8] and is running on the same system from which the inference server is directly reachable. Each worker node connects to HiveCore via an outbound TCP connection and continuously polls for new tasks. Once authenticated, HiveNode advertises which local LLM models it supports. HiveNode then forwards any assigned inference requests to its local *Ollama* server, a lightweight LLM runtime that handles text generation and other language model operations. This modularity allows each worker machine to run as many or as few models as resources permit, supporting both single-GPU and multi-GPU scenarios. When a client (e.g., a user or another system) sends an inference request to HiveCore, the request is examined to determine where it should be queued. There are two types of queues:

1. **Model-based queues.** If the request specifies a model name (for example, "llama3.2"), HiveCore places the request in the corresponding model queue. Once a node that supports that model is available, the request is dispatched to that node.
2. **Worker-based queues.** If the request header explicitly targets a specific worker node by its name (for instance, via an HTTP header `Node: <nodename>`), HiveCore places the request into that worker's personal queue. This is typically used for administrative or specialized workloads where a user wants precise control over which node processes the request (for example pulling end deleting models on worker machine).

If an incoming request does not specify a valid model or a valid target worker node, *HiveCore* cannot determine the right queue and rejects the request. In this way, all traffic on the client server must fit into either a model queue or a worker queue.

Once a request is placed into the appropriate queue, *HiveCore* finds an available *HiveNode* that can service that queue. For model-based queues, any node that advertises the requested model is eligible. For worker-based queues, the matching node must be the one named in the header. Once a node is selected, *HiveCore* forwards the request to that node over the existing authenticated connection. *HiveNode* then communicates with *Ollama* to carry out the inference or embedding, ultimately streaming back chunks of results. These chunks flow through *HiveCore* to the client, preserving the interactive nature of LLM outputs.

By separating the duties of a central proxy from local inference engines, Hive creates a secure and scalable environment. Nodes remain isolated behind their own networks, requiring no inbound ports, and only need to maintain a secure outbound socket channel to *HiveCore*. Meanwhile, administrators operate at the proxy layer, provisioning tokens, observing queue lengths, and monitoring node status from a single location. This design enables consistent load balancing, straightforward horizontal scaling, and a seamless client experience, regardless of where the compute resources reside.

2.1. Software architecture

Within HiveCore, three logical servers run concurrently in a single process. Although they share some internal services (like the database), each one exposes its own listening port and handles a distinct role:

[1] https://ollama.com/

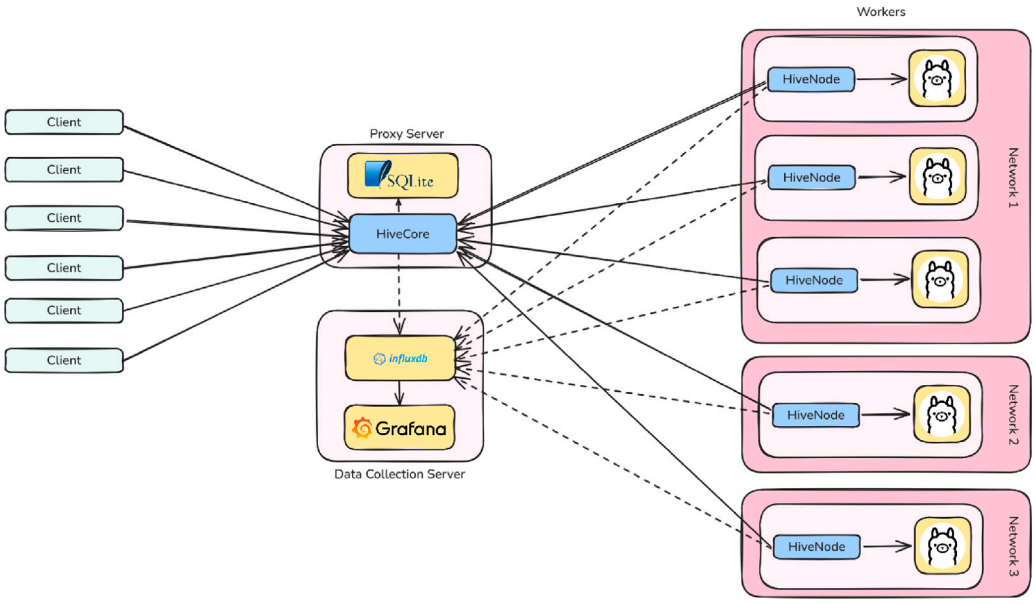

Fig. 1. Diagram of connected components of a possible Hive system setup.

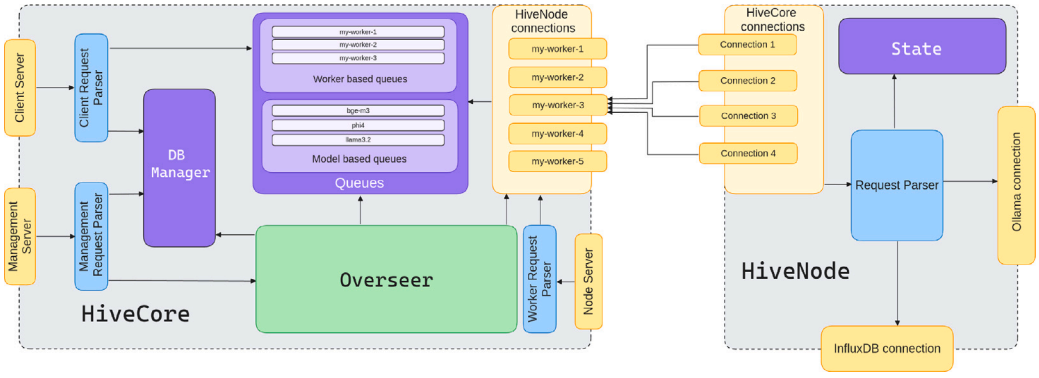

Fig. 2. Diagram of the architecture of HiveCore and HiveNode.

(1) **Client Server**: Receives client-facing requests on a configurable port (default `6666`). Most requests ask for text generation or embedding operations using a particular model (e.g., ‘‘`mistral-nemo`’’). If no specific model is provided, HiveCore rejects the request unless it explicitly targets a worker node by name.
(2) **Node Server**: Accepts inbound connections from *HiveNode* instances on a separate port (default `7777`). Each node must provide a valid worker token during setup and then declares its capabilities, such as the Ollama version and *HiveNode* version. The Node Server keeps track of each node’s operational state (e.g., *SettingUp*, *Polling*, *Working*, and *Closed*), verifying and removing inactive nodes if they fail to respond in a timely manner (see Fig. 2).
(3) **Management Server**: Offers an administrative interface (default `6668`) for creating, revoking, and listing authentication keys, as well as viewing queue lengths and node statuses. Only users or scripts presenting an *Admin* token can perform these operations.

To enable a seamless flow of tasks, the HiveCore maintains a set of queues in memory, keyed either by model name (for model-based requests) or by target node name (for worker-specific requests). When a new request arrives from the Client Server, HiveCore places it into the appropriate queue. On startup, each *HiveNode* opens N simultaneous socket connections out to HiveCore, where N equals the number of concurrent requests the node can handle. Each socket connection persists as a streaming channel for passing data between Ollama and the proxy in real time.

When a *HiveNode* first connects, it announces both its own software version and the version of Ollama running on the system. On the first poll for work, the node reports the set of models it can serve. Internally, *HiveCore* consults any worker-specific queue first, dispatching requests explicitly targeted at that node; if that queue is empty, the node begins consuming requests from the model-based queues. The sequence in which the node pulls models follows the order initially declared by the worker (e.g., `mistral7b`, `llama3.1`, `mistral-nemo`). However, once the node processes a request, *HiveCore* reorders the sequence by placing the last pulled model at the front of the list (e.g., if `llama3.1` was the first used model, the new pull order becomes `llama3.1`, `mistral-nemo`, `mistral7b`). This rotation-based approach helps reduce the likelihood of excessive model switching on the worker side when there is many diverse requests to be handled (see Fig. 3).

On the worker side, each connection uses asynchronous Rust (via `tokio` [9]) to poll for tasks, and once assigned a request, the node issues an HTTP call to Ollama. It streams back generated tokens in small chunks, which are relayed through *HiveCore* to the client. This streaming design preserves interactivity, allowing the client to read partial outputs before the entire response completes.

If a node encounters a fatal error or becomes unreachable, *HiveCore* transitions it to a *Closed* state and removes it from the active pool. The node may subsequently reconnect by re-authenticating with the Node

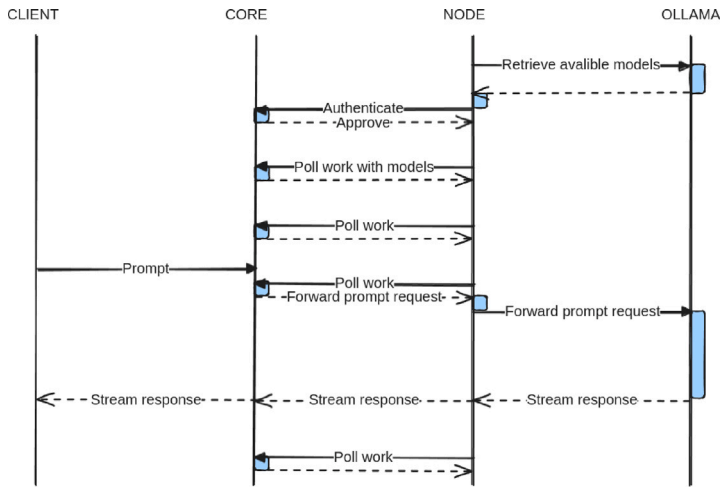

Fig. 3. Request flow across client, HiveCore, HiveNode, and Ollama.

Server. Since concurrency limits, available models, and rotation states are all re-declared upon reconnection, *HiveCore* can smoothly adapt when nodes join or leave the cluster.

2.2. Software functionalities

Hive's design enables distributed inference while minimizing repetitive configurations. A *queue-based system* in *HiveCore* categorizes requests by either model name or a specific worker node. This approach balances load across multiple nodes for common requests, yet also supports targeted routing when an `Admin` specifies a particular worker (e.g., specialized hardware or debugging scenarios).

Security and authentication revolve around three roles: `Client`, `Worker`, and `Admin`. Administrators manage these tokens through the Management Server, and each node must present a valid `Worker` token before receiving tasks. Additionally, administrators can optionally require `Client` tokens for incoming inference requests, depending on the needs.

Hive includes several built-in mechanisms for handling failure scenarios, maintaining system stability, and ensuring graceful degradation during faults. On the worker side, if an inference request results in a recoverable error from the local Ollama instance (e.g., malformed input, unsupported model, or internal runtime errors), the response is transparently proxied back to the client, preserving the original error message and status code. If, however, the worker encounters a critical issue — such as network timeouts, unreachable Ollama services, or malformed responses — the request is aborted and the client receives a 500 Internal Server Error.

In the event of network interruptions or internal inconsistencies (e.g., duplicate worker keys, invalid authentication), HiveNode and HiveCore disconnect and initiate a reconnection sequence with exponential backoff, beginning at 10 s increasing with each failed attempt. This ensures network flaps or transient misconfigurations do not cause cascading errors. HiveCore continuously monitors node connectivity and marks unresponsive or invalid nodes as closed. These nodes are automatically removed from the active pool and must re-authenticate to resume participation.

On the client side, HiveCore handles incorrect or malformed requests gracefully. Invalid model names or unavailable workers result in a 405 response, invalid or missing authentication tokens yield a 401, and requests made with insufficient privileges return a 403. These structured responses help clients handle errors programmatically and reduce ambiguity in failure modes.

Monitoring and logging capabilities are built into both components. HiveCore tracks request queues and node statuses in real time, while each *HiveNode* can optionally emit system metrics (e.g., GPU usage) to external logging services like InfluxDB [10]. This enables end-to-end visibility over resource utilization. Additionally, the *HiveNode* repository includes an exportable *Grafana* [11] dashboard that administrators can import to visualize live GPU usage, concurrency levels, and request latencies in a unified interface (see Fig. 4).

Lastly, *scalability and replication* are inherent to Hive's design. Adding new workers is as simple as starting a *HiveNode* with a valid token.

Table 1
Table shows the setup of workers during testing. Worker count is the number of workers with the same configuration connected to the proxy. The GPU and the count denote what hardware the worker was equipped with, and concurrent shows how many requests Ollama and the worker node were set to process in parallel.

WORKERS	GPU	Concurrent
3	1x NVIDIA H100	10
2	1x NVIDIA RTX A6000	4
2	2x NVIDIA GeForce GTX 1080 Ti	4
5	1x NVIDIA GeForce RTX 3080	4

To assess the runtime characteristics of *HiveCore*, two sets of tests were conducted.

The first experiment focused on resource consumption under load. A fixed set of *HiveNode* workers was connected to *HiveCore* as denoted in Table 1, and system resource usage was recorded while putting the system under load. In each configuration, a number of clients (ranging from 10 to 1000) repeatedly sent embedding requests containing five short texts over a one-minute interval. Each worker supported the requested model. Meanwhile, *HiveCore*'s CPU usage, memory footprint, open file descriptors (FDs), and thread count were sampled at one-second intervals. A two-minute idle period was used as baseline. The results (summarized in Table 2) show that even under high concurrency, the resource usage of *HiveCore* remained modest. For instance, CPU usage increased only marginally from an idle average of 0.108 to maximum 0.139 under load, while memory remained under 300MB throughout. Open FDs increased predictably with thread count, confirming that each request opened and closed connections efficiently.

The second test measured performance overhead introduced by the proxy architecture. In this setup, a single worker with access to an H100 GPU was used. First, requests were issued directly to Ollama via 100 concurrent clients locally on the machine. Next, the same machine was connected to *HiveCore* as its only worker, and requests were routed through the proxy. For both configurations the concurrency was set to handle 10 concurrent requests while other requests were waiting in a queue. In both cases, the number of completed requests and timing statistics were collected over multiple runs. Table 3 shows that the

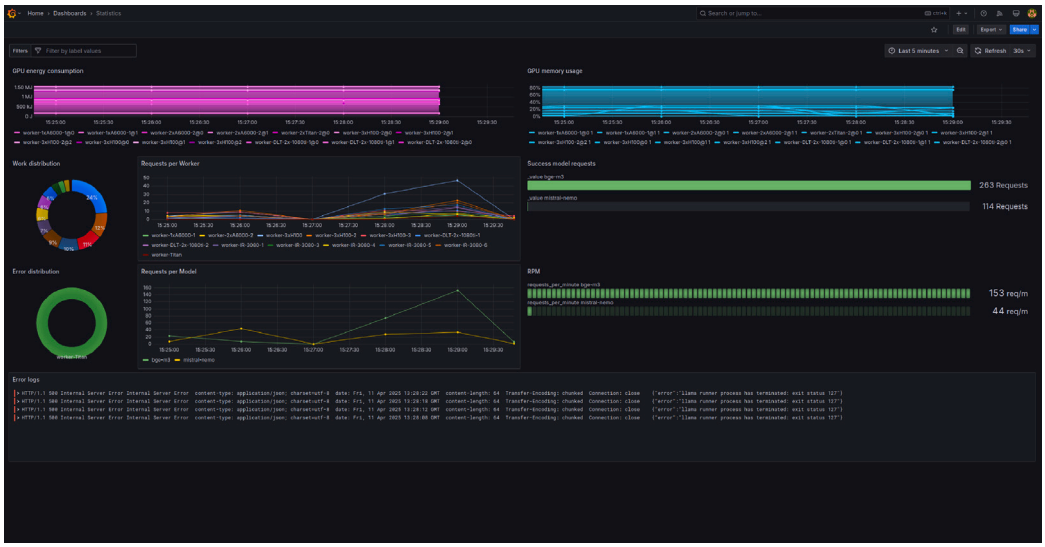

Fig. 4. Example Grafana dashboard provided in HiveNode repository.

Table 2
Maximum resource usage of HiveCore when handling increasing numbers of concurrent client threads. Each client repeatedly submitted embedding requests over a 1 min interval. Columns report: average CPU usage in cores, peak RAM usage in megabytes, maximum thread count observed during the interval (Max Threads), and the maximum number of open file descriptors (FDs).

Label	CPU	Max RAM (MB)	Max Threads	Max FDs
Idle	0.108	231.89	97	83
10 Clients	0.122	244.31	102	485
50 Clients	0.139	269.13	102	708
100 Clients	0.126	252.70	102	826
500 Clients	0.127	266.85	115	1209
1000 Clients	0.125	284.41	184	2511

Table 3
Comparison between direct Ollama performance(Direct) and performance of requests being proxied via *HiveCore*(Proxied). The Requests column denotes how many requests were issued on average in the span of 60 s and the Time column denotes how long it took to complete each of the requests including the time in queue. Both show average results over multiple runs with Std denoting standard deviation of the measured results.

Type	Requests		Time	
	Average	Std	Average	Std
Direct	545.17	12.48	11.79	0.28
Proxy	543.17	13.67	11.85	0.32

proxy induced only minimal overhead—less than 0.5% difference in mean request count and average response time.

These results confirm that *HiveCore* maintains low system overhead even at high concurrency and introduces negligible latency assuming sufficient bandwidth, validating its suitability as a lightweight proxy for large-scale, distributed LLM inference.

3. Illustrative examples

3.1. Single HiveCore with multiple distributed nodes

First, an administrator logs into *HiveCore*'s management interface (by default on port 6668) and creates a new `Worker` key. For instance, issuing a request such as:

```
curl -X POST http://hivecore.example.com:6668/
    key \
   -H "Authorization: Bearer <admin-token>" \
   -H "Content-Type: application/json" \
   -d '{"name":"myWorker1","role":"Worker"}'
```

The response returns a generated key (e.g., `c228f8df-8df3-4649-a964-...`), which is then added to the `.env` file on the remote machine that will run *HiveNode*:

```
HIVE_CORE_URL=hivecore.example.com:7777
HIVE_KEY=c228f8df-8df3-4649-a964-...
OLLAMA_URL=http://localhost:11434
CONCURRENT_RQEUESTS=2
```

Next, the remote machine runs:

```
cargo run --release
```

Upon startup, *HiveNode* connects to `cloud.example.com:7777`, presents its worker key, and announces the local models it can serve. HiveCore authenticates the node and adds it to its active pool.

Once worker(s) are online, a user can send an inference request, for instance:

```
curl http://hivecore.example.com:6666/api/
    generate \
  -d '{"model":"mistral-nemo","prompt":"Hello,
    world!"}'
```

HiveCore places that request in the `mistral-nemo` queue. One of the nodes that supports `mistral-nemo` then claims the job and streams partial responses back through *HiveCore* to the client, preserving interactive semantics even in a distributed setting.

3.2. Targeting a particular node for pulling models

Suppose a node named `myWorker1` is already running as a **HiveNode**. An `Admin` user wants to add a new model called `mistral-nemo` specifically to that worker. They can send the following request to HiveCore:

```
curl -X POST http://hivecore.example.com:6666/
    api/pull \
  -H "Authorization: Bearer <admin-token>" \
  -H "Node: myWorker1" \
  -H "Content-Type: application/json" \
  -d '{"model":"mistral-nemo"}'
```

Because the `Node` header is set to `myWorker1`, HiveCore queues this job specifically for that node. When *myWorker1* polls for tasks, it will receive the pull instruction and proceed to load (or download) the specified model locally. This granular level of control lets administrators upgrade or modify individual nodes via Hive without affecting other machines in the cluster.

3.3. Monitoring queue sizes via the management server

Finally, an `Admin` user may want to inspect the current load on each model or node queue. Since HiveCore exposes management endpoints on port 6668, the admin could run:

```
curl -X GET http://hivecore.example.com:6668/
    queue \
  -H "Authorization: Bearer <admin-token>"
```

If successful, HiveCore responds with a JSON structure indicating how many requests are queued per model, and how many jobs are specifically waiting for each node (if any). For example, the returned data might look like:

```
{
  "Model: mistral-nemo": 2,
  "Model: bge-m3": 1,
  "Node: myWorker1": 0
}
```

This overview helps administrators gauge which models are in high demand and whether certain worker-based queues are experiencing bottlenecks.

4. Impact

Hive offers a strong foundation for organizations and researchers aiming to unify distributed resources for LLM inference. By separating the proxy and node roles, the solution improves scalability, and maintainability. Nodes remain shielded behind outbound connections, lowering potential attack surfaces. Administrators can monitor everything via a consolidated management port, which reduces the operational overhead of manually managing connections or firewall rules for each node.

Moreover, *Hive*'s reliance on *Ollama* (an increasingly popular LLM server) makes it versatile and relatively lightweight. Users can run multiple *HiveNode* processes on a single machine, perhaps assigning each process to a different GPU, or replicate them across VMs, cloud providers, or on-prem HPC clusters.

MIT License ensures that Hive can be adapted for a variety of use cases. Use cases can integrate specialized logging, custom load-balancing heuristics, or new user authentication flows. In its current form, the open repositories (`HiveCore` and `HiveNode`) have served as a stable, real-world solution for bridging on-prem GPUs, cloud instances, and even ephemeral lab machines.

The software is deployed in production at the University of Primorska and is actively used in research as well as internally developed chatbots that help students as well as administrative staff based on Retrieval-Augmented Generation (RAG). *Hive* also connects the compute cluster of InnoRenew CoE, an independent research institute established in 2017. Collaborations between researchers from multiple institutes can use *Hive* to join computing resources, unlocking use cases otherwise computationally unreachable by individual entities.

Beyond LLM inference, Hive's architecture may support research in fields that rely on distributed or constrained computing environments. For example, Hive could be used to deploy models across university networks, research labs, or partner institutions without requiring public IPs or VPNs—particularly useful in education, digital humanities, and collaborative scientific projects. In areas like environmental monitoring or edge computing, Hive's outbound-only connectivity makes it suitable for running local inference on field devices or isolated systems. The planned support for hybrid routing to commercial APIs like ChatGPT or Gemini could also enable dynamic workflows that combine local and cloud-based AI services, which is especially relevant for small research groups or institutions with limited infrastructure.

5. Conclusions

Hive consolidates multiple network-isolated or geographically separated *Ollama* servers into a single distributed inference system, offering a seamless user experience. This proxy-worker architecture simplifies node connectivity and streamlines tasks such as key-based authentication, node monitoring, and robust management APIs. The approach fosters an environment where hardware can be quickly added or removed, while the client experience remains consistent.

However, several limitations still define its current scope. Inference is performed entirely on individual worker nodes, with currently no support for distributed execution across multiple machines. This choice prioritizes simplicity, resilience, and ease of deployment over fine-grained model parallelism. Although nodes automatically reconnect after network interruptions, Hive does not yet provide persistent request queuing or retry mechanisms for in-flight tasks interrupted by system failures.

Future work will also focus on further refining load-balancing strategies, integrating advanced concurrency controls, and enhancing observational analytics in the management layer. Features like auto-scaling or dynamic GPU assignment can further optimize resource usage. Additional expansions may also introduce alternative model backends that emulate *Ollama*'s API and official Docker containers for LLMs, extending *Hive*'s applicability to a wider ecosystem of LLM serving solutions. Beyond backend flexibility, we plan to support fine-grained API key controls, allowing administrators to restrict access to specific models or operations per key, enabling safer multi-tenant or role-based deployments. Hive will also be extended to support hybrid inference setups by allowing requests to be routed through the proxy to external APIs such as OpenAI's ChatGPT, Anthropic's Claude, and Google Gemini, either as a fallback when local resources are unavailable or as a deliberate configuration choice for specific models.

CRediT authorship contribution statement

Domen Vake: Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization. **Jernej Vičič:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization. **Aleksandar Tošić:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The authors gratefully acknowledge the European Commission for funding the InnoRenew CoE project (H2020 Grant Agreement #739574) and the SRC-EDIH project (DIGITAL-2021-EDIH-01 call, project number: 101083351). We also thank our collaborators and open-source contributors for testing and supporting HiveCore and HiveNode in various deployments.

References

[1] Chowdhery A, Narang S, Devlin J, Bosma M, Mishra G, Roberts A, et al. Palm: Scaling language modeling with pathways. J Mach Learn Res 2023;24(240):1–113.

[2] Reese W. Nginx: the high-performance web server and reverse proxy. Linux J 2008;2008(173):2.

[3] NVIDIA Corporation. Triton inference server: An optimized cloud and edge inferencing solution. 2024, https://github.com/triton-inference-server/server. Accessed 11 April 2024.

[4] Kwon W, Li Z, Zhuang S, Sheng Y, Zheng L, Yu CH, et al. Efficient memory management for large language model serving with PagedAttention. In: Proceedings of the ACM SIGOPS 29th symposium on operating systems principles. 2023.

[5] Borzunov A, Baranchuk D, Dettmers T, Riabinin M, Belkada Y, Chumachenko A, et al. Petals: Collaborative inference and fine-tuning of large models. In: Proceedings of the 61st annual meeting of the association for computational linguistics (volume 3: system demonstrations). 2023, p. 558–68, URL https://arxiv.org/abs/2209.01188.

[6] Ollama FAQ. 2023, https://github.com/ollama/ollama/blob/main/docs/faq.md. Accessed 01 September 2023.

[7] Owens M. The Definitive Guide to SQLite. Springer; 2006.

[8] Matsakis ND, Klock FS. The rust language. In: Proceedings of the 2014 ACM sIGAda annual conference on high integrity language technology. 2014, p. 103–4.

[9] Tokio - An asynchronous Rust runtime. 2023, https://tokio.rs/. Accessed 01 September 2023.

[10] InfluxDB documentation. 2023, https://docs.influxdata.com/influxdb/. Accessed 15 September 2023.

[11] Grafana documentation. 2023, https://grafana.com/docs/. Accessed 15 September 2023.