HPC Benchmark Toolkit

Alberto Finardi   Giovanni La Gioia   Laura Paxton   Giulia Lionetti

January 2026

MeluXina HPC Cluster

Introduction

Project Overview

LLM Benchmarking Framework

- Reproducible: YAML-based configuration
- Scalable: Single to multi-node
- Observable: Real-time monitoring
- HPC-Native: Slurm + Apptainer
- Target: MeluXina HPC Cluster

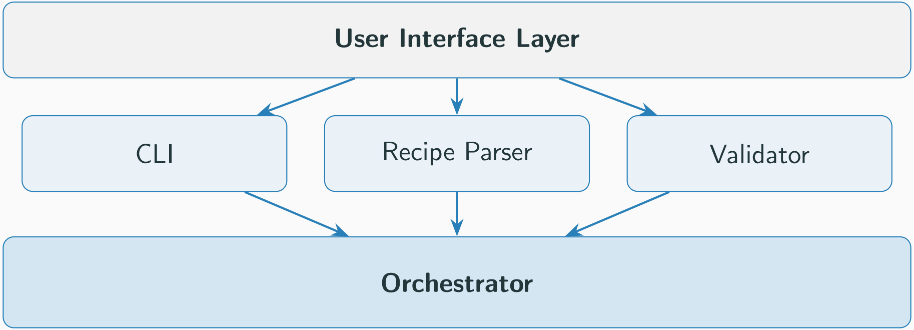
Supported Services

- **Ollama**: Local LLM inference via REST API
- **vLLM**: High-throughput LLM serving with OpenAI API compatibility
- **vLLM Distributed**: Multi-node tensor parallelism using Ray

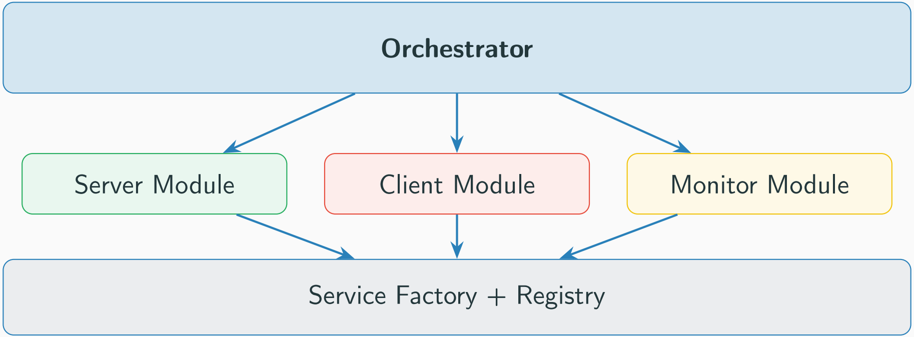
Architecture Highlights: Modular design, extensible service factory

How It Works

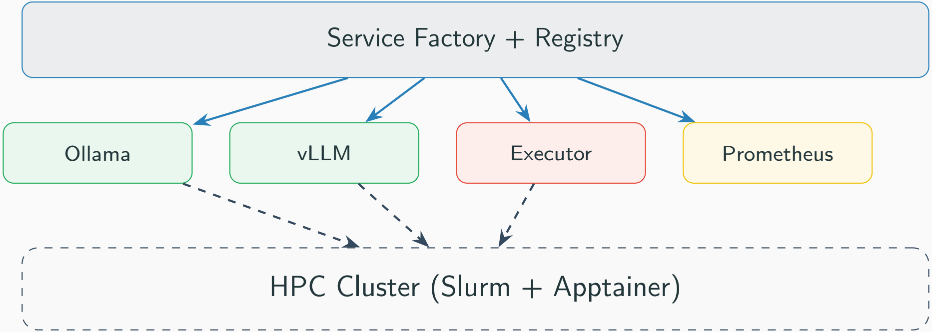
High-Level Architecture



High-Level Architecture (Contd.)



High-Level Architecture (Contd.)



Six-Phase Execution Model

1. **Initialization** – Parse recipe, validate configuration

2. **Service Deployment** – Start containers, health checks

3. **Monitoring Setup** – Initialize metrics collection

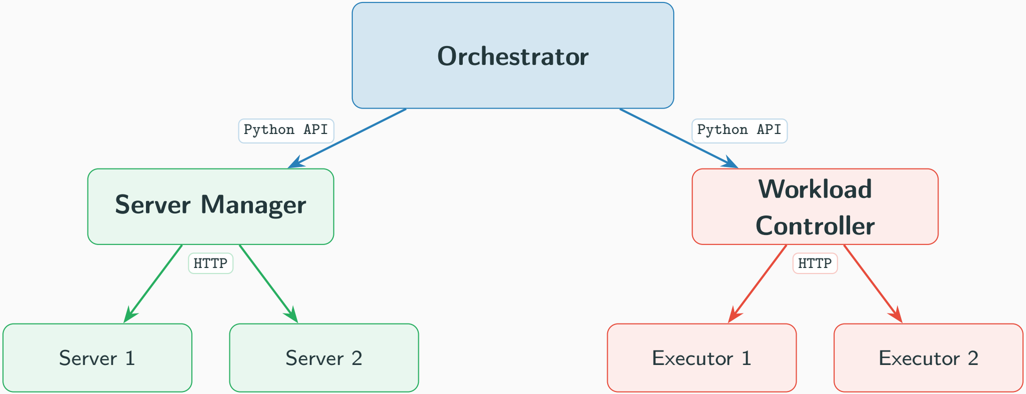
Six-Phase Execution Model (Contd.)

4. Client Launch – Deploy executors on client nodes

5. Execution – Run benchmark, collect metrics

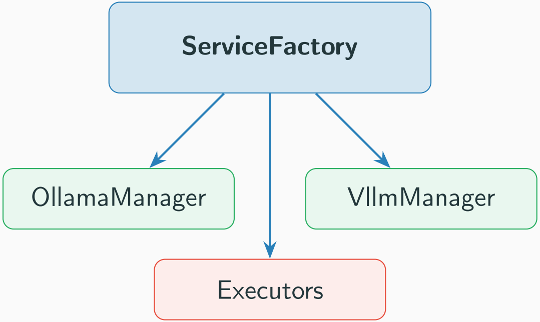
6. Teardown – Stop clients, services, monitoring

Control Plane Communication

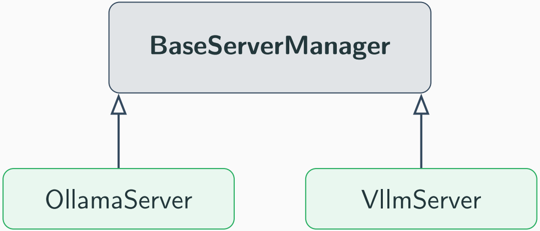


Design Patterns

Factory Pattern



Template Method Pattern



Infrastructure

Recipe Configuration System

YAML-Based Configuration for Reproducible Experiments

Orchestration

Slurm mode
Node allocation
Resource requests

Workload

Service selection
Duration & warmup
Model & parameters

Parameter Sweeps

Batch sizes
Concurrency levels
Automated trials

Features: JSON Schema validation • Service-specific configs • Automated experiment generation

Recipe Validation

- **Schema-driven validation**: YAML recipes are validated against a JSON Schema before job submission
- **Strict vs soft checks**: blocking errors stop execution, while warnings highlight suboptimal configurations
- **Service-aware rules**: validation adapts to the selected service (e.g. inference requires model and prompt length)
- **Clear diagnostics**: precise, user-friendly error and warning messages with field-level context
- **Reproducibility support**: schema versioning and validated defaults ensure consistent experiments

Slurm Integration

Recipe-Driven SLURM Job Setup

- The recipe is parsed to determine the required services, replicas, and node allocation.
- A SLURM job is created according to the configuration, reserving the correct number of nodes and resources.
- On each node, the appropriate Apptainer container is spawned:
  - Server nodes: start service Apptainer (e.g., Ollama)
  - Client nodes: start Python Apptainer for workload execution
  - Orchestrator/monitoring/logging node: start orchestrator and monitoring services

Slurm Integration (Contd.)

- The orchestrator coordinates all components, waits for services to be online, and provides clients with service type, URLs, and ports.
- Clients use multithreaded service worker implementations to send requests to the services.
- Only the orchestrator is stateful; all other components are stateless and report to the orchestrator.

SLURM Job Lifecycle: Example

Step 1: Recipe Configuration

User provides a YAML recipe:

- 2 server nodes (Ollama)
- 2 client nodes (100 clients per node)
- 1 node for orchestrator/monitoring/logging

Goal: Benchmark Ollama service with 200 concurrent clients.

Step 2: SLURM Job Creation

- The system parses the recipe and generates a SLURM job script.
- **Resources reserved:** 5 nodes (2 server, 2 client, 1 orchestrator/monitoring/logging)
- Each node is assigned a specific role based on the recipe.

Step 3: Node Setup

- **Server nodes:** Start Ollama Apptainer service
- **Client nodes:** Start Python Apptainer, each running 100 client threads
- **Orchestrator/monitoring/logging node:** Start orchestrator, monitoring, and logging services

Step 4: Orchestrator Actions

- Waits for all services to be online
- Discovers service types, URLs, and ports
- Provides clients with connection details
- Coordinates the experiment, tracks state, and aggregates results

Note: Orchestrator is the only stateful component; all others are stateless.

Step 5: Client Behavior

- Each client thread uses the provided service worker implementation
- Sends requests to the correct service endpoint (Ollama)
- Collects metrics and reports results to the orchestrator

Result: Scalable, reproducible benchmarking with full orchestration and monitoring.

Distributed vLLM with Ray

- **Ray-based vLLM (Distributed):** Uses a master/worker architecture (Ray Head + Ray Workers).
- **Single Endpoint:** All distributed workers are managed by the Ray Head node, exposing a single API endpoint for clients.
- **Tensor Parallelism:** Multiple GPUs across several nodes are coordinated for high-throughput inference.
- **Contrast with Ollama/Non-Distributed vLLM:**
  - **Ollama:** Each server node runs a fully separated service instance, each with its own endpoint.
  - **vLLM (non-distributed):** Single-node, single service, single endpoint.
  - **Ray Distributed:** Multiple worker nodes are managed under one master, with all services accessible via a unique, unified endpoint (more realistic for production).
- **Implication:** Clients connect to one endpoint, and Ray transparently distributes requests across all available GPUs/nodes.

Monitoring

Monitoring Architecture

Prometheus + Grafana Stack

- **Executors:** Expose metrics via HTTP endpoints (`/metrics/prometheus`)
- **Prometheus:** Time-series database, scrapes metrics periodically
- **Grafana:** Visualization dashboards with custom panels
- **Deployment:** Docker Compose for easy setup (local + HPC)

Key Metrics Collected:

- Request count, errors, success rate
- Latency (average, P50, P90, P99)
- Throughput (requests per second)
- System resources (CPU, memory, active threads)

MeluXina Integration

Challenge: Compute nodes not directly accessible from outside

Solution: Automated SSH tunneling

- `setup_monitoring.sh` script identifies Slurm node allocation
- Establishes SSH tunnels from login node to each executor endpoint
- Makes metrics accessible to Prometheus
- Handles cleanup when job completes

Dashboards:

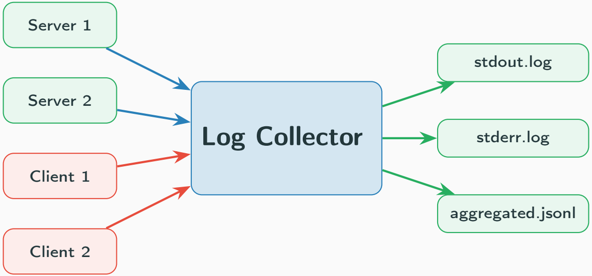
- **Ollama Dashboard:** API performance, response times
- **vLLM Dashboard:** API endpoints, cluster performance

Metrics collected after benchmark completion for analysis

Logging System

Logging System Overview

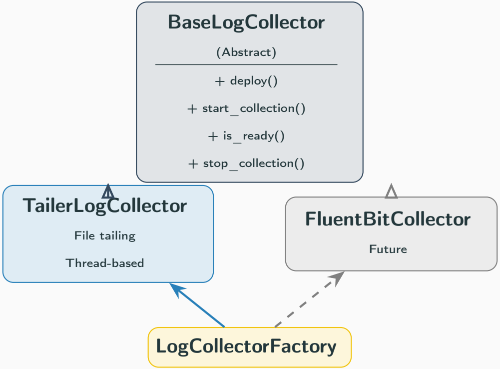
Purpose: Aggregate distributed logs from multi-node HPC benchmarks



Key Features: Multi-node aggregation, Structured JSON, Thread-based tailing

Architecture: Strategy Pattern

Design: Abstract interface with pluggable implementations



LogSource Dataclass

Purpose: Structured representation of log sources

```
@dataclass
class LogSource:
  node: str
  component: str
  container_name: str
```

Example:

```
LogSource(node="mel2120",
  component="server",
  container_name="ollama_0")
```

BaseLogCollector Interface

Abstract Methods: Contract for all implementations

deploy(sources)

Prepare infrastructure

Create output files

is_ready()

Check readiness

Verify flag files

start_collection()

Begin log capture

Launch threads

stop_collection()

Finalize outputs

Return summary

TailerLogCollector: 5-Phase Workflow

1. Deploy: Create stdout.log, stderr.log, aggregated.jsonl

2. Start: Launch thread per source, create loggers_ready flag

3. Tail: Monitor log files, read new lines continuously

4. Process: Add timestamp, node, component metadata

5. Stop: Close files, return summary (line counts, paths)

Log Output Formats

stdout.log – Plain text with metadata

```
[2026-01-12T14:32:15Z] [mel2120] [server] Model loaded
```

stderr.log – Error messages

```
[2026-01-12T14:35:20Z] [mel2148] [client] Connection timeout
```

aggregated.jsonl – Structured JSON Lines

```
{"timestamp": "2026-01-12T14:32:15Z",
 "node": "mel2120", "component": "server"}
```

Configuration in Recipe

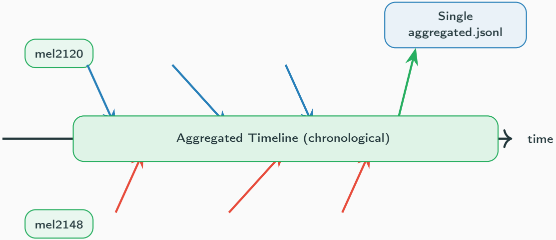
```yaml
logging:
  type: "tailer"
  create_jsonl: true
  flush_interval: 5

  outputs:
    stdout: "stdout.log"
    stderr: "stderr.log"
    aggregated: "aggregated.jsonl"
```

type: Collector type
create_jsonl: JSON output
flush_interval: Buffer time

outputs: File paths
Relative to experiment dir

Multi-Node Aggregation

Challenge: Collect logs from distributed nodes into single files



Solution: Timestamps enable natural chronological merge

Integration with Orchestrator

Phase 1: Initialization

Phase 2: Service Deployment

Phase 3: Deploy & Start Logging — ready flag

Phase 4: Client Launch

Phase 5: Continuous Log Capture — active

Phase 6: Stop & Finalize Logs — summary

Phase 7: Report Generation

Validation and Testing

Comprehensive validation ensures logging correctness

Automated Tests:

- File existence
- Content size
- Line count
- Format validation
- Multi-node aggregation
- Component coverage

Test Script: `validate_logging.sh`

Tests:

- 16 automated tests
- JSON validity
- Log distribution
- Timestamp format
- Multi-node check
- Line consistency

Benchmarking

Performance Analysis: Ollama Setup

- Service: **Ollama (single-service deployment)**
- Model: `llama2`
- Execution time: **2 minutes**
- Threads: **10**
- Total requests: **57**

Performance Analysis: Ollama Results

Metric	Value
Average latency	24.04 s
P50 latency	23.72 s
P90 latency	36.37 s
P99 latency	38.34 s
Throughput	0.38 req/s
Errors	0

Performance Analysis: Ollama Bottlenecks

- Latency dominated by **model inference time**
- Limited batching and concurrency handling
- Suitable for low-throughput, interactive use cases

Performance Analysis: vLLM Distributed Setup

- Service: **vLLM Distributed (Ray-based)**
- Deployment: **Multi-node GPU execution**
- Job submission: **Successful**
- SLURM job ID: **3944353**
- Architecture: Single API endpoint, distributed workers

Performance Analysis: Ollama vs vLLM

Aspect	Ollama	vLLM Distributed
Deployment complexity	Low	High
Latency focus	High latency	Optimized for batching
Throughput	Low	High (expected)
Scaling	Single-node	Multi-node
Endpoint model	One per node	Single unified endpoint

Performance Analysis: Key Takeaways

- Ollama provides a stable but latency-heavy baseline
- Tail latency is a critical limitation for high load
- vLLM distributed targets scalable, production-like inference
- The framework supports both extremes with the same recipe model

Division of Work

Alberto Finardi

System Architect & Core Implementation

- **Infrastructure:** Core architecture, Factory pattern, Service registry
- **Server Managers:** Ollama, vLLM, Ray orchestration
- **Workload System:** Controllers, Executors, Benchmarking
- **Logging:** Base information logging system
- **Integration:** CLI interface, Slurm orchestration, Documentation

Monitoring & Observability

- **Prometheus:** Configuration, Metric collection, Scraping setup
- **Grafana:** Dashboard design, Visualization

Laura Paxton

Recipe Validation & Benchmark Execution

- **Validation:** JSON Schema, Error formatting
- **Benchmarking:** Tested Ollama execution, vLLM execution, Parameter sweeps, Result analysis

Logging System Architecture

- **Architecture:** BaseLogCollector, Abstract interfaces, LogSource design, Strategy pattern
- **Implementation:** TailerLogCollector, Remote collection, Structured JSON, Aggregation

Thank You!

Questions?

`hpc-benchmark-toolkit` | MeluXina HPC Cluster