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

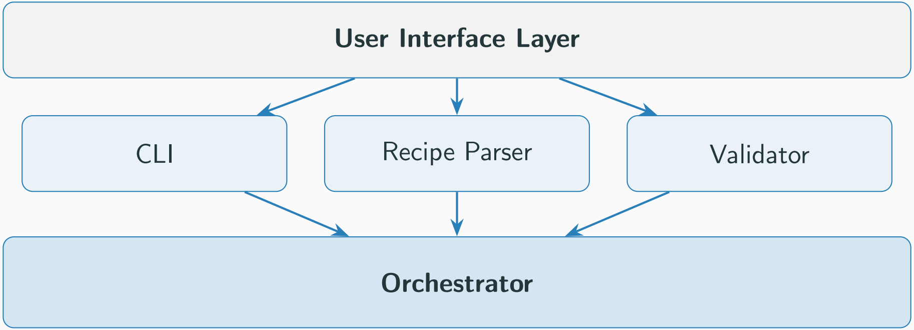
Supported Services

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

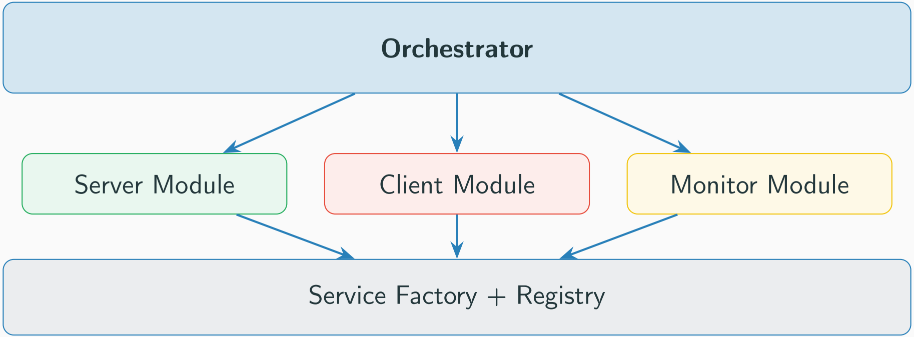
Architecture Highlights: Modular design, extensible service factory

How It Works

High-Level Architecture



High-Level Architecture (Contd.)



Orchestrator

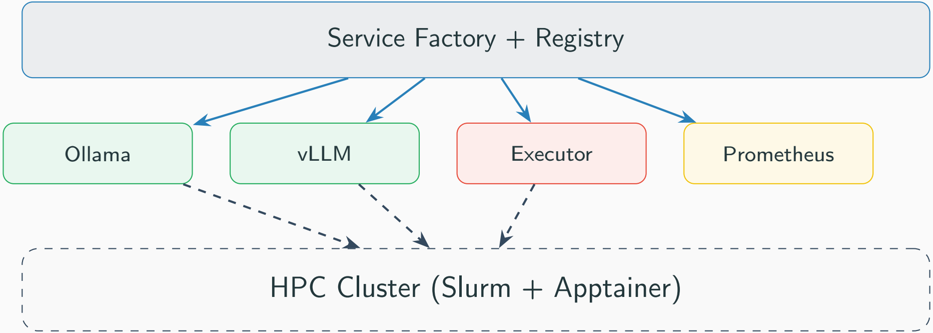
Server Module

Client Module

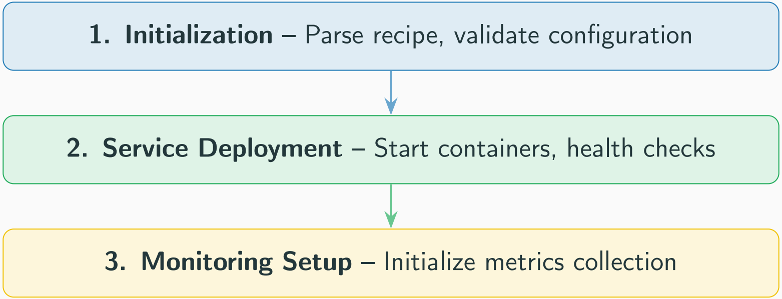
Monitor Module

Service Factory + Registry

High-Level Architecture (Contd.)



Six-Phase Execution Model

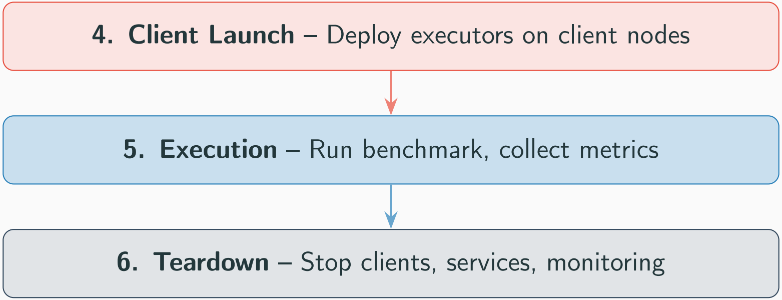


1. **Initialization** – Parse recipe, validate configuration

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

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

Six-Phase Execution Model (Contd.)

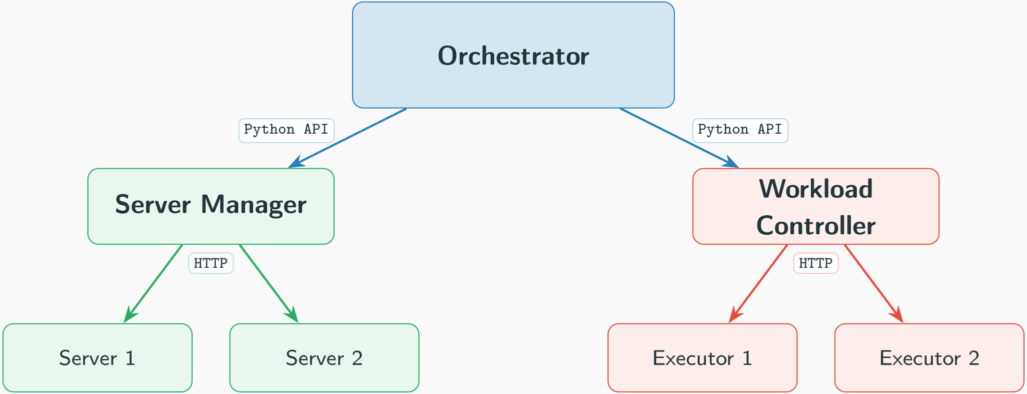


4. Client Launch – Deploy executors on client nodes

5. Execution – Run benchmark, collect metrics

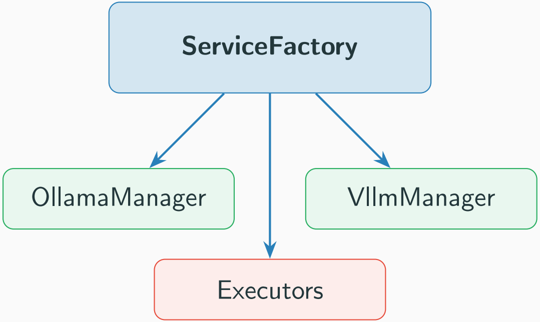
6. Teardown – Stop clients, services, monitoring

Control Plane Communication

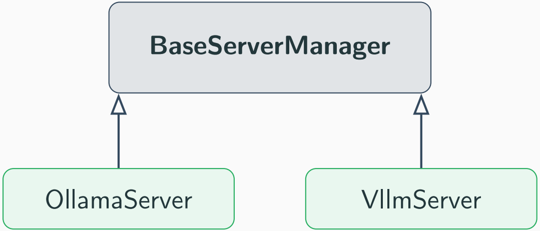


Design Patterns

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

PLACEHOLDER:

Laura’s Section

- JSON Schema validation implementation
- Validation rules and error messages
- Custom validators for service-specific configs
- Schema versioning support
- Interactive validation mode

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 Architecture

PLACEHOLDER:

Giulia's Section **Overall Design:**

- Log sources and destinations
- Aggregation strategy for distributed logs

BaseLogCollector:

- Abstract interface design
- LogSource dataclass

TailerLogCollector:

- File tailing implementation
- Remote node log collection

Log Categories and Storage

PLACEHOLDER:

Giulia's Section **Log Categories:**

- Application logs
- System logs (Slurm)
- Benchmark logs
- Infrastructure logs

Log Format:

- Structured logging (JSON)
- Timestamps and correlation IDs
- Storage organization
- Retention policies

Benchmarking

Performance Analysis

PLACEHOLDER:

Laura's Benchmarking Results **Bottleneck Analysis:**

- Latency breakdown for LLM inference
- GPU compute vs memory transfer
- Network overhead measurements

Scaling Analysis:

- Single-node vs multi-node performance
- Strong/weak scaling efficiency

Include benchmark charts and graphs here

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

Giovanni La Gioia

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

Giulia Lionetti

Logging System Architecture

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

Questions?

Thank You!

Questions?

`hpc-benchmark-toolkit` | MeluXina HPC Cluster