

Benchmarking AI Factories on MeluXina

A Modular Framework for Reproducible AI Workload Evaluation on EuroHPC Systems

EUMaster4HPC Student Challenge 2025-2026

Luca Lamperti
l.lamperti2002@gmail.com

January 2026

Abstract

This technical report presents a Python-based benchmarking framework designed for evaluating AI Factory components on the MeluXina supercomputer. The framework addresses the need for reproducible, modular, and scalable benchmarking of AI workloads—including LLM inference servers and object storage systems—in HPC environments. The system leverages SLURM for job orchestration, Apptainer for containerization, and Prometheus for metrics collection. This document details the system architecture, core components, execution flow, and performance analysis.

Contents

1	Executive Summary and Introduction	3
1.1	Context: AI Factories in Europe	3
1.2	Project Objective	3
2	System Architecture	3
2.1	High-Level Overview	3
2.1.1	Module Responsibilities	4
2.2	HPC Integration	4
2.2.1	SLURM Workload Manager Interaction	5
3	Core Components and Design	5
3.1	Abstract Base Classes and Contracts	5
3.1.1	The Executor Contract	5
3.1.2	Monitor and Logger Contracts	5
3.2	The Service Composition Layer	6
3.3	Server and Client Modules	6
3.4	Monitor Module: Prometheus Metrics Collection	6
3.5	Logger Module: Thread-Safe Logging	6
3.6	Design Patterns Summary	6
4	Execution Flow and Logic	7
4.1	The Recipe Concept	7
4.2	Sequence of Events	7
5	Implementation Details	9
5.1	Technology Stack	9
5.2	Error Handling Strategy	9
5.3	Extensibility: Adding New Workloads	9

6	Experimental Setup and Methodology	9
6.1	MeluXina Environment	9
7	Results and Performance Analysis	10
7.1	Metrics Visualization Infrastructure	10
7.2	vLLM Inference Benchmark Results	10
7.3	MinIO S3 Storage Benchmark Results	11
7.4	Output Data Structure	12
8	Conclusion and Future Work	12
8.1	Achievements	12
8.2	Future Work	12

1 Executive Summary and Introduction

1.1 Context: AI Factories in Europe

The European High-Performance Computing Joint Undertaking (EuroHPC JU) has established a network of petascale and pre-exascale supercomputers across Europe. Among these systems, MeluXina—hosted at LuxProvide in Luxembourg—represents critical infrastructure for AI, machine learning, and deep learning applications.

The concept of “AI Factories” has emerged as a paradigm for organizing computational resources dedicated to AI training, inference, and data processing at scale. These factories integrate:

- **Inference Servers:** High-throughput model serving (vLLM, TensorRT-LLM, Triton)
- **Object Storage:** Scalable data lakes for artifacts (MinIO, Ceph)
- **Vector Databases:** Semantic search support (ChromaDB, Milvus)
- **Orchestration:** Workflow management and scheduling

However, the lack of standardized benchmarking methodologies for these components on HPC systems presents a significant challenge.

1.2 Project Objective

This project delivers a **modular, reproducible, and scalable** benchmarking framework:

Modular Architecture

Four-module design (Servers, Clients, Monitors, Loggers) with abstract base classes enabling extension to new services.

Reproducible Execution

YAML-based “Recipe” configuration files capture complete experiment specifications.

Scalable Deployment

Native SLURM integration supports multi-node deployments with configurable parallelism.

HPC-Native Design

First-class Apptainer container support and module-based environment management.

2 System Architecture

2.1 High-Level Overview

The benchmarking framework implements a layered architecture comprising four principal modules orchestrated by a central **BenchmarkManager** component. Figure 1 illustrates the system design.

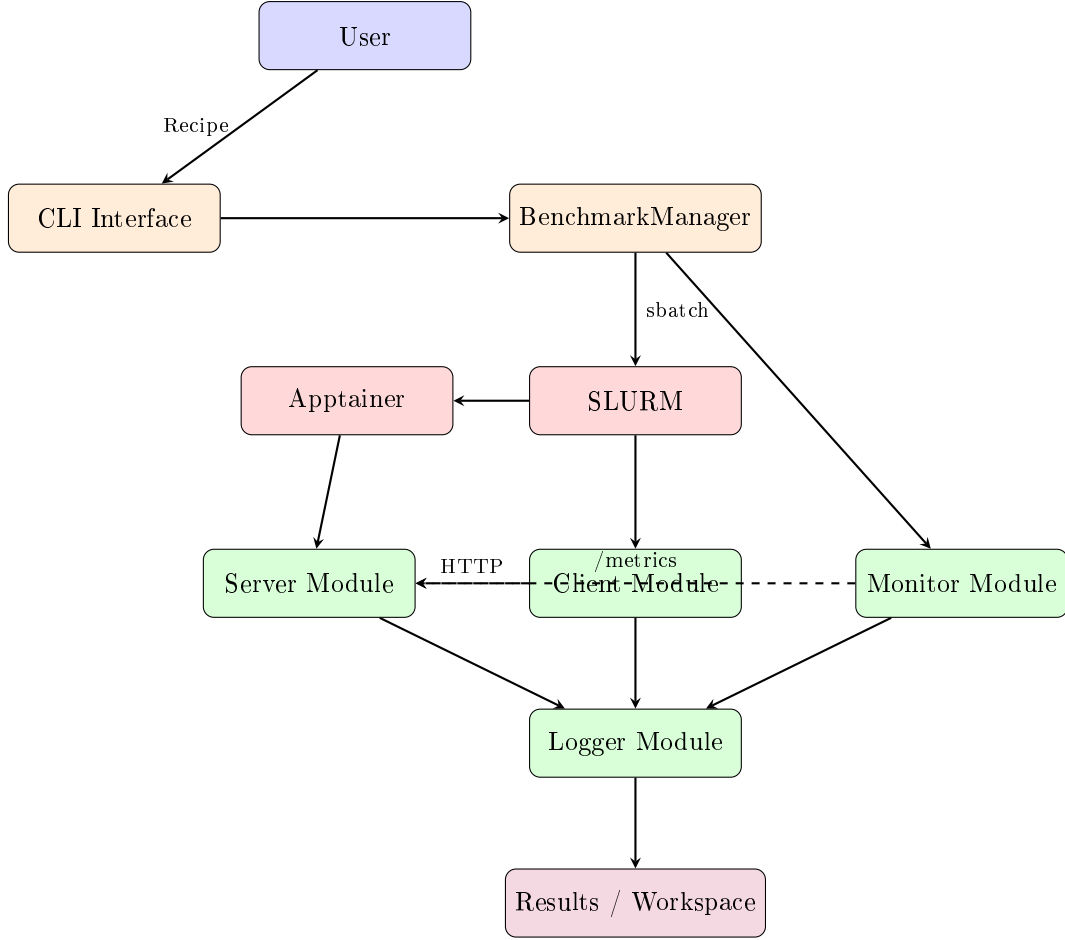


Figure 1: High-level system architecture showing User, Interface, HPC, Compute, and Storage layers.

2.1.1 Module Responsibilities

- **Server Module:** Encapsulates long-running services such as vLLM inference servers or MinIO object storage. Servers are launched via executors, monitored for health via HTTP healthchecks, and expose Prometheus-compatible metrics endpoints.
- **Client Module:** Implements load generation workloads that stress-test servers. Clients execute configurable request patterns with concurrent workers, collecting latency distributions and throughput metrics.
- **Monitor Module:** Performs pull-based metrics scraping from server endpoints at configurable intervals. The `PrometheusMonitor` parses Prometheus exposition format and persists time-series data to JSON files.
- **Logger Module:** Provides thread-safe, centralized logging for distributed components. Supports JSON and plaintext formats.

2.2 HPC Integration

The framework integrates with MeluXina through multiple touchpoints:

1. **Environment Modules:** The `run_benchmark.sh` script initializes Lmod, loading Python and Apptainer modules.

2. **Scratch Filesystem:** All workspace directories use scratch storage for high-performance I/O.
3. **GPU Partition:** SLURM job scripts target the GPU partition with A100 allocation.

2.2.1 SLURM Workload Manager Interaction

The `SlurmExecutor` class provides programmatic SLURM job management. Upon receiving a command string, the executor first checks whether an Apptainer container image is configured; if so, it wraps the command inside an `apptainer exec` invocation. The executor then writes the command to a temporary shell script, constructs a complete `sbatch` command line with configured resource parameters (nodes, GPUs, time limit, account), and submits the job via subprocess. The returned job ID is stored for subsequent status queries or cancellation.

The executor maintains internal state tracking the current job ID, enabling the `status()` method to query SLURM via `squeue` and return the job state. The `stop()` method issues `scancel` to terminate jobs gracefully.

3 Core Components and Design

This section analyzes the framework’s core components, examining object-oriented design patterns and implementation strategies.

3.1 Abstract Base Classes and Contracts

The framework establishes a contract-based architecture through three abstract base classes in `src/Core/abstracts.py`.

3.1.1 The Executor Contract

The `Executor` abstract class defines the universal interface for all execution backends. It mandates three methods: `run()` accepts a command payload and returns an identifier, `stop()` terminates execution, and `status()` returns current state. This abstraction decouples orchestration logic from specific execution mechanisms.

Four concrete executors implement this contract:

Table 1: Executor implementations and their behaviors

Executor	Behavior
<code>SlurmExecutor</code>	Submits jobs to SLURM via <code>sbatch</code> , tracks job IDs, queries status via <code>squeue</code> , cancels via <code>scancel</code> . Supports GPU allocation and Apptainer wrapping.
<code>ApptainerExecutor</code>	Launches commands inside Apptainer containers.
<code>ProcessExecutor</code>	Executes shell commands as subprocesses with process group isolation. Spawns daemon threads to stream output to the logger.
<code>WorkloadExecutor</code>	Runs in-process Python workloads on background threads. Retrieves runner functions from the workload registry.

3.1.2 Monitor and Logger Contracts

The `Monitor` abstract class requires `start()`, `collect()`, and `stop()` methods, establishing a lifecycle protocol for metrics collection. The `Logger` abstract class mandates `log()` for message recording and `export()` for persistence.

3.2 The Service Composition Layer

The **Service** class implements the **Composition Pattern**, assembling an executor, optional monitor, and optional logger into a cohesive unit. When `start()` is invoked, the service first logs the startup event, activates the monitor’s background scraping, and delegates command execution to the executor. The `stop()` method reverses this sequence with exception handling to prevent cascading failures.

3.3 Server and Client Modules

- The **Server** class extends **Service** with server-specific behavior, storing command configuration and implementing `_resolve_command()` to normalize input (string or dictionary). The `start_service()` method resolves the command and delegates to the parent.
- The **Client** class extends **Service** for workload generation, storing workload specifications and using `_resolve_payload()` to determine executor input. When executor type is “workload”, the entire configuration dictionary passes to the **WorkloadExecutor**.
- The **BenchmarkManager** supports client scaling through the `instances` field. The `_expand_client_specs()` method creates deep copies for each instance, appending indices to IDs (e.g., “vllm-loadgen-1”, “vllm-loadgen-2”).

3.4 Monitor Module: Prometheus Metrics Collection

The **PrometheusMonitor** implements a pull-based scraping model. Upon initialization, it receives scrape targets, intervals, and output paths.

When `start()` is called, the monitor spawns a daemon thread running a scrape loop. This loop invokes `collect()`, then waits on a threading Event with the scrape interval as timeout. The Event-based wait allows immediate termination when `stop()` signals.

The `collect()` method iterates over targets, constructs metrics URLs, and issues HTTP GET requests with 3-second timeout. Successful responses are stored as raw text; failures store error messages. Each collection appends a timestamped entry to an in-memory buffer.

Periodically, the monitor writes the buffer to disk in two formats: raw JSON and parsed JSON with structured metric samples including names, labels, values, and HELP/TYPE metadata.

3.5 Logger Module: Thread-Safe Logging

The **FileLogger** provides robust logging for multi-threaded execution. A `threading.Lock` protects all write operations, ensuring atomic writes from concurrent workload workers. Each entry captures ISO timestamp, level, and message.

3.6 Design Patterns Summary

Factory Pattern

The **BenchmarkManager** uses factory methods (`_create_executor`, `_create_monitors_map`, `_create_loggers_map`) for component instantiation.

The workload registry provides `get_workload_runner()` as a factory function.

Template Method Pattern

`Service.start()` defines the algorithm template; subclasses implement hooks like `_resolve_command()` and `_resolve_payload()`.

Strategy Pattern

The executor abstraction allows runtime selection of execution strategies without modifying orchestration logic.

4 Execution Flow and Logic

4.1 The Recipe Concept

Recipes are YAML configuration files that declaratively specify benchmark experiments. Located in **Recipes/**, each recipe defines metadata, global settings, services, clients, monitors, loggers, and execution parameters.

Table 2 documents the primary configuration fields:

Table 2: Recipe configuration field reference

Field	Description
<code>meta.name</code>	Unique identifier for the benchmark experiment
<code>global.workspace</code>	Base directory for outputs (supports <code>\${VAR}</code> expansion)
<code>services[].executor.type</code>	Executor strategy: process , slurm , apptainer
<code>services[].command</code>	Shell command or dict for service startup
<code>services[].healthcheck</code>	HTTP endpoint and timeout for readiness verification
<code>clients[].instances</code>	Number of client replicas to spawn
<code>clients[].workload.type</code>	Workload implementation: vllm-inference , s3-upload
<code>monitors[].targets</code>	List of host:port endpoints for Prometheus scraping
<code>execution.duration</code>	Maximum benchmark runtime in seconds
<code>execution.post_actions</code>	Ordered list of cleanup actions

4.2 Sequence of Events

Figure 2 illustrates the detailed execution flow. The benchmark proceeds through five phases:

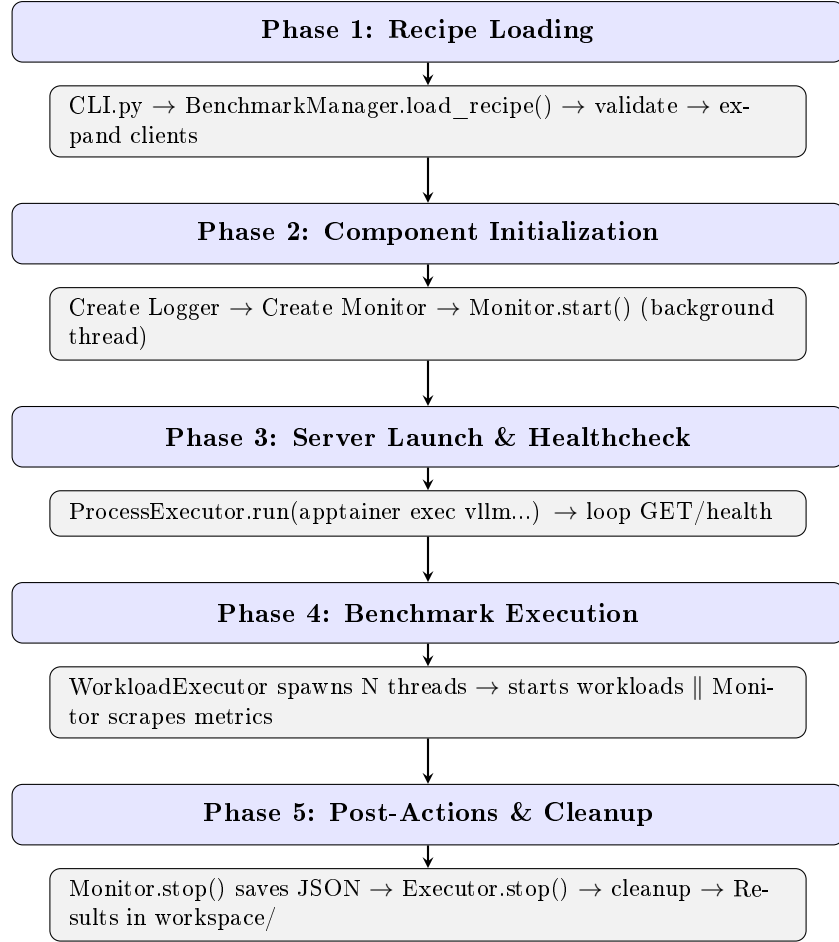


Figure 2: Five-phase benchmark execution flow: Recipe Loading, Component Initialization, Server Launch, Benchmark Execution, and Post-Actions.

Phase 1: Recipe Loading. The `BenchmarkManager.load_recipe()` method parses the YAML file, expands environment variables, validates required sections, and expands client instances.

Phase 2: Component Initialization. Factory methods instantiate monitors, loggers, executors, servers, and clients. Monitors begin background scraping threads.

Phase 3: Server Launch. Each server’s `start_service()` executes via the assigned executor. The manager polls HTTP health endpoints until servers report ready or timeout expires.

Phase 4: Benchmark Execution. Client instances start, spawning concurrent worker threads. The main loop polls client status at configured intervals. Prometheus monitors scrape metrics concurrently.

Phase 5: Post-Actions. Upon duration expiry, the manager executes `collect_metrics` and `stop_services`, persists metrics, and performs cleanup.

5 Implementation Details

5.1 Technology Stack

Table 3: Technology stack and dependencies

Category	Technology	Version
Language	Python	3.9+
Configuration	PyYAML	≥ 6.0
Metrics	prometheus_client	$\geq 0.17.0$
S3 Client	boto3	$\geq 1.28.0$
Container Runtime	Apptainer	1.2+
Scheduler	SLURM	21.08+

5.2 Error Handling Strategy

The framework implements defense-in-depth error handling:

Healthcheck Timeout: Polling loop with configurable retry interval. Timeout raises `TimeoutError`, triggering graceful shutdown.

Executor Stop Isolation: `Service.stop()` wraps executor termination in try-except, preventing cascading failures.

Workload Error Backoff: Worker threads increment consecutive error counters; exceeding threshold aborts the worker. Configurable backoff delay between retries.

Monitor Resilience: `collect()` catches all HTTP exceptions, storing error messages rather than crashing.

5.3 Extensibility: Adding New Workloads

To add a new workload type (e.g., ChromaDB):

Step 1: Create `src/Core/workloads/chromadb_query.py` with a `run(config, logger, stop_event)` function.

Step 2: Register in `WORKLOAD_REGISTRY` dictionary in `workloads/__init__.py`.

Step 3: Create a recipe YAML with the new workload type.

6 Experimental Setup and Methodology

6.1 MeluXina Environment

Table 4: MeluXina hardware specifications (GPU partition)

Component	Specification
GPU Nodes	200 nodes
GPUs per Node	4× NVIDIA A100-40
CPUs per Node	2× AMD Rome (32 cores)
Memory per Node	512 GB
Interconnect	HDR InfiniBand

7 Results and Performance Analysis

This section presents the experimental results from two benchmark scenarios executed on MeluXina: vLLM inference serving and MinIO S3 object storage. Metrics were collected via Prometheus scraping and visualized through Grafana dashboards.

7.1 Metrics Visualization Infrastructure

The framework includes a FastAPI-based visualization server (`src/Interface/fastapi_server.py`) that implements the Grafana SimpleJSON datasource protocol. This enables real-time metrics visualization without external dependencies.

Key features of the visualization server:

- **Auto-detection:** Automatically detects service type (vLLM, MinIO) from metric prefixes
- **Rate calculation:** Applies `rate()` to counter metrics (`_total`, `_count`) automatically
- **Recommended metrics:** Provides service-specific default metrics for optimal dashboards
- **Histogram support:** Parses bucket metrics for heatmap visualizations

7.2 vLLM Inference Benchmark Results

The vLLM benchmark was executed on December 31, 2025, using the facebook/opt-125m model with 2 client instances (4 threads each).

Table 5: vLLM inference benchmark configuration and results

Parameter	Value
Model	facebook/opt-125m
Client Instances	2 (4 threads each)
Benchmark Duration	120 seconds
Total Requests	$\approx 7,400$
Success Rate	99.9%
Average Latency	122–130 ms
Throughput	≈ 62 req/s
Concurrent Requests	6–8 (average)

Figure 3 shows the Grafana dashboard for the vLLM benchmark. Panel 1 displays the HTTP request duration count (rate-transformed), demonstrating consistent throughput around 60–66 requests per second. Panel 2 shows the number of concurrent requests running in the vLLM engine, fluctuating between 4–8 during load. Panel 3 presents a heatmap of request latency distribution, with the majority of requests completing in the lower latency buckets (green/blue regions).

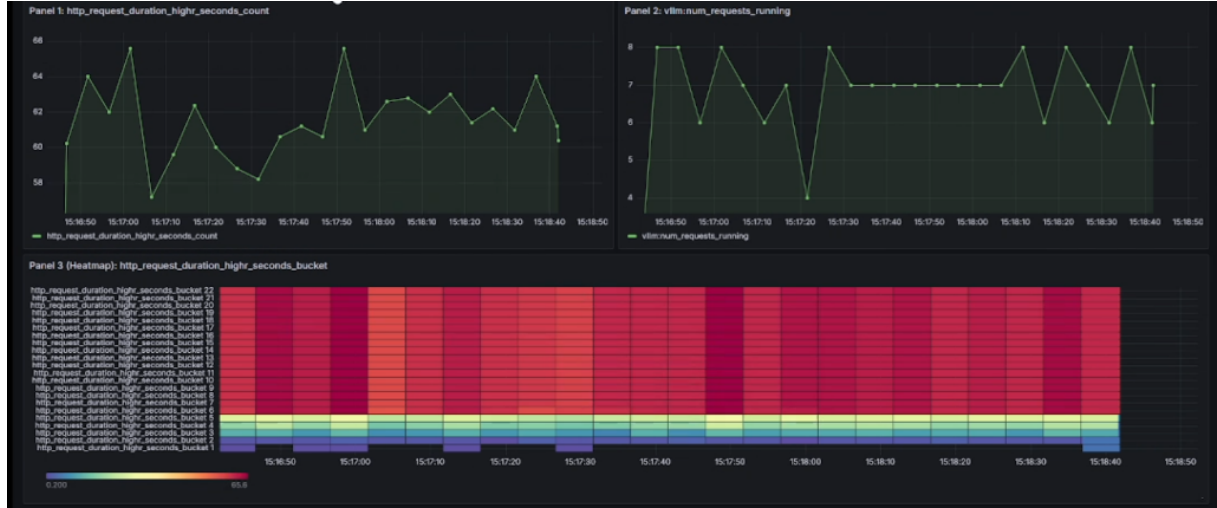


Figure 3: Grafana dashboard for vLLM inference benchmark showing: (1) HTTP request rate, (2) concurrent requests running, (3) latency distribution heatmap.

7.3 MinIO S3 Storage Benchmark Results

The S3 benchmark was executed using MinIO as the object storage backend, testing upload and download operations with varying object sizes.

Table 6: MinIO S3 benchmark configuration and results

Parameter	Value
Storage Backend	MinIO (containerized)
Client Instances	4
Benchmark Duration	300 seconds
Peak Request Rate	$\approx 60\text{--}70$ req/s
Traffic Sent	400–700 MB/interval
TTFB Distribution	Consistent across buckets

Figure 4 presents the MinIO benchmark dashboard. Panel 1 shows the total S3 requests over time with multiple client streams (color-coded). Panel 2 displays traffic sent in bytes, showing periodic bursts corresponding to batch uploads. Panel 3 provides a heatmap of Time-To-First-Byte (TTFB) latency distribution across histogram buckets.

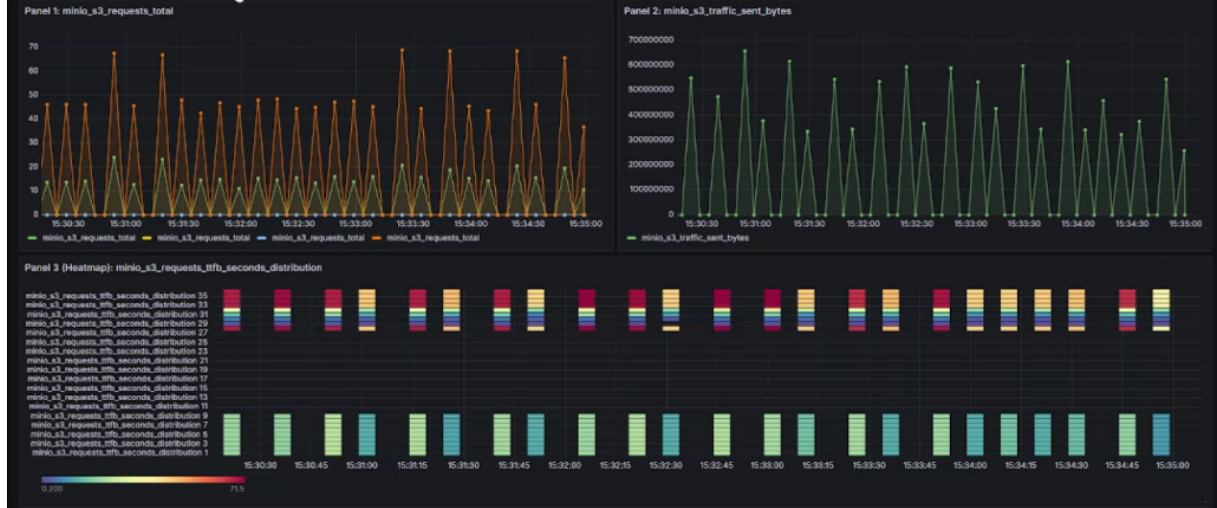


Figure 4: Grafana dashboard for MinIO S3 benchmark showing: (1) total requests per client, (2) traffic sent bytes, (3) TTFB seconds distribution heatmap.

7.4 Output Data Structure

Benchmark results are persisted in timestamped directories with the following structure:

- `logs/` – JSON-lines application logs
- `slurm_logs/` – SLURM stdout/stderr from job execution
- `*_metrics.json` – Raw Prometheus scrapes (text format)
- `*_metrics_parsed.json` – Structured JSON with metric names, labels, and values

8 Conclusion and Future Work

8.1 Achievements

This project delivered a production-ready benchmarking framework:

1. Modular four-module architecture with abstract base classes for extensibility.
2. First-class SLURM support with Apptainer containerization.
3. YAML-based Recipe specifications for reproducibility.
4. Prometheus-compatible monitoring with parsed JSON output.
5. Validated object storage and vLLM inference.

8.2 Future Work

- Extended workload suite (vector DBs).
- Multi-node scaling with distributed client load generation.
- Workflow DAG support for multi-stage benchmarks.
- Real-time web dashboard for live monitoring.

References

- [1] EuroHPC Joint Undertaking. <https://eurohpc-ju.europa.eu/>
- [2] LuxProvide MeluXina. <https://luxprovide.lu/meluxina/>