AI Factory Benchmarking Framework

Can Beydogan, Giuseppe Galardi, Mario Capodanno, Thies Weel
`Team1_EUMASTER4HPC2526`

January 2026

Abstract

We present a benchmarking framework AI service performance on High-Performance Computing (HPC) infrastructure. The framework automates the complete lifecycle of service deployment, load generation, metrics collection, and performance analysis. Our system introduces: (1) a recipe-driven architecture enabling reproducible experiments across heterogeneous HPC environments; (2) automated saturation point detection using curvature-based knee-point analysis for optimal concurrency identification; (3) bottleneck attribution through correlation of system metrics with performance degradation patterns; and (4) regression detection with configurable thresholds. The framework supports seven service types spanning inference engines (vLLM, Ollama), databases (PostgreSQL, Redis, MinIO), and vector stores (ChromaDB, Qdrant). Experimental validation on the MeluXina supercomputer demonstrates characterization of diverse scaling behaviors, from near-linear throughput scaling in Redis to early GPU saturation in LLM inference workloads.

1 Introduction

The convergence of large-scale AI deployment and high-performance computing has given rise to "AI factories": infrastructure platforms designed to serve machine learning models at scale, manage vector embeddings for retrieval augmented generation (RAG), and process high throughput database operations.

Traditional monitoring solutions provide real time visibility into the system but site reliability engineers and architects require answers to questions such as: *What is the maximum sustainable concurrency under latency service-level objectives (SLOs)? Which resource between CPU, GPU, memory, or network becomes the limiting factor? How do configuration changes affect throughput and tail latency?*

Our system integrates with HPC job schedulers (Slurm) and container runtimes (Apptainer), enabling automated experiments from declarative YAML specifications.

1.1 Problem Statement

Benchmarking AI services on HPC infrastructure presents unique challenges:

1. **Heterogeneous Resources**: Services may require GPUs, high-memory nodes, or specific interconnects.
2. **Job Scheduling**: Load must coordinate with batch schedulers rather than direct process management.
3. **Distributed Execution**: Clients and services could run on separate compute nodes with no shared memory.
4. **Reproducibility**: Environmental variations between runs confound result interpretation.

1.2 Contributions

1. A **recipe-driven architecture** that encapsulates complete benchmark specifications, including service configuration, client parameters, and resource requirements, enabling reproducible experiments.
2. **Automated analysis algorithms** comprising:
   - Saturation detection via maximum curvature analysis
   - Bottleneck attribution through resource correlation
   - Regression detection with configurable thresholds
3. A **dual-interface design** providing complete feature parity between command-line and web interfaces.
4. An **integrated monitoring stack** using Prometheus and Grafana with a novel filesystem-based heartbeat strategy for distributed client visibility.

2 Related Work

2.1 Benchmarking Frameworks

General-purpose load testing tools such as Apache JMeter [2], Locust [3], and k6 [4] provide flexible HTTP-based load generation but lack native integration with HPC job schedulers. Database-specific benchmarks like YCSB [5] and TPC-C [6] offer standardized workloads but require manual adaptation for containerized deployments.

For LLM inference, recent work includes vLLM's built-in benchmarking [7] and llmperf [8], which focus on token throughput and latency metrics. However, these tools assume direct process execution rather than HPC job submission.

3 System Architecture

The framework follows a layered architecture separating user interaction, orchestration logic, and cluster communication. Figure 1 illustrates the component hierarchy.

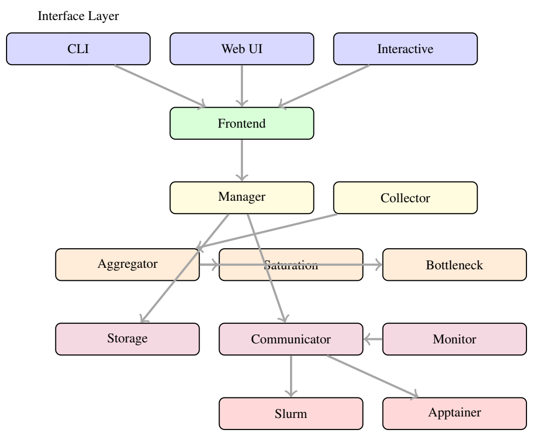

Figure 1: Layered system architecture. Components are organized into interface, orchestration, analysis, infrastructure, and execution layers.

3.1 Frontend Layer

The **Frontend** module serves as the unified entry point for all user interactions. It performs three primary functions:

1. **Recipe Parsing**: Validates YAML configuration and instantiates typed Python dataclasses representing service, client, and benchmark parameters.
2. **Identifier Generation**: Creates unique benchmark IDs using the format `BM-YYYYMMDD-NNN` with atomic counter increments.
3. **Interface Dispatch**: Routes requests to CLI handlers, Flask endpoints, or terminal UI depending on invocation mode.

3.2 Orchestration Layer

The **Manager** class implements the deployment logic shown in **Algorithm 1**.

3.3 Analysis Layer

The analysis pipeline transforms raw request data into more useful insights:

- **Aggregator**: Computes latency percentiles (P50, P90, P95, P99), throughput, and success rates from `requests.jsonl` files.
- **Saturation**: Implements knee-point detection for concurrency sweeps (Section 5.3).
- **Bottleneck**: Classifies resource limitations based on utilization patterns (Section 5.4).

Algorithm 1 Service Deployment Workflow

```
Require: Recipe $R$, Benchmark ID $B$
 1: $script \leftarrow$ GENERATESBATCH($R.service$)
 2: $job\_id \leftarrow$ SUBMITJOB($script$)
 3: while $\neg$ISRUNNING($job\_id$) do
 4:     WAIT(poll_interval)
 5: end while
 6: $host \leftarrow$ GETHOSTNAME($job\_id$)
 7: while $\neg$HEALTHCHECK($R.service.type, host$) do
 8:     WAIT(retry_interval)
 9: end while
10: for $i \leftarrow 1$ to $R.num\_clients$ do
11:     $client\_script \leftarrow$ GENERATECLIENTSBATCH($R, host$)
12:     SUBMITJOB(client_script)
13: end for
```

3.4 Infrastructure Layer

The **Communicator** abstract class defines the cluster interaction interface, with `SSHCommunicator` providing the concrete implementation via the Fabric library. This abstraction enables future adaptation to alternative transport mechanisms (e.g., REST APIs, message queues).

The **Storage** module persists benchmark state using pluggable backends. The current implementation uses local JSON files organized by benchmark ID.

4 Recipe Specification

Recipes encode complete experiment configurations in declarative YAML format. This design choice provides several advantages:

- **Reproducibility**: Recipes are self-documenting and version-controllable.
- **Validation**: Schema enforcement catches errors before job submission.

4.1 Recipe Structure

A recipe consists of four sections:

Listing 1: Complete vLLM benchmark recipe

```
configuration:
  target: meluxina

service:
  type: vllm
  partition: gpu
  num_gpus: 1
  time_limit: "02:00:00"
  settings:
    model: facebook/opt-125m
    tensor_parallel_size: 1

client:
  type: vllm_inference
  partition: cpu
  settings:
    prompt: "Explain quantum computing"
    max_tokens: 100
    warmup_delay: 30

benchmarks:
```

```
num_clients: 4
```

4.2 Service Type Abstraction

The framework supports seven service types across three categories (Table 1). Each service type encapsulates:

- Default container image and command
- Port configuration and exposure
- Health check implementation
- Compatible client types

Table 1: Supported service types with configuration defaults.

Category	Service	Port	Health Check
Inference	vLLM	8000	HTTP GET /health
	Ollama	11434	HTTP GET /api/tags
Database	PostgreSQL	5432	TCP connect + auth
	Redis	6379	PING command
	MinIO	9000	HTTP GET /minio/health/live
Vector DB	ChromaDB	8000	HTTP GET /api/v1
	Qdrant	6333	HTTP GET /collections

5 Metrics and Analysis

5.1 Data Collection

Each benchmark produces three primary artifacts:

- run.json: Complete metadata including embedded recipe, Slurm job IDs, node allocations, container digests, and timestamps.
- requests.jsonl: Per-request timing in newline-delimited JSON with microsecond-precision timestamps.
- summary.json: Aggregated statistics computed from raw data.

5.2 Metrics Computation

Latency Percentiles. For a sorted array of n latency values $\{v_1, \ldots, v_n\}$, the p-th percentile uses linear interpolation:

$$P_p = v_{\lfloor k \rfloor}(1 - f) + v_{\lceil k \rceil} \cdot f \tag{1}$$

where $k = (n-1) \cdot p/100$ and $f = k - \lfloor k \rfloor$ is the fractional component.

Throughput. Computed as requests per unit time over the measurement window:

$$\text{RPS} = \frac{N_{\text{success}}}{t_{\text{last}} - t_{\text{first}}} \tag{2}$$

LLM-Specific Metrics. For inference workloads, we additionally compute:

- Tokens per second (TPS): $\sum$ output_tokens/duration
- Time to first token (TTFT): Latency until streaming begins
- Inter-token latency (ITL): Mean time between consecutive tokens

5.3 Saturation Point Detection

For concurrency sweep experiments, identifying the optimal operating point is critical. We define the *saturation point* as the concurrency level where latency begins growing superlinearly relative to load.

Algorithm 2 Saturation Point Detection

```
Require: Concurrency levels $C = \{c_1, \ldots, c_n\}$, P99 latencies $L$
1: Normalize: $\tilde{c}_i \leftarrow (c_i - c_{\min})/(c_{\max} - c_{\min})$
2: Normalize: $\tilde{l}_i \leftarrow (l_i - l_{\min})/(l_{\max} - l_{\min})$
3: for $i \leftarrow 2$ to $n - 1$ do
4:     $y' \leftarrow (\tilde{l}_{i+1} - \tilde{l}_{i-1})/(\tilde{c}_{i+1} - \tilde{c}_{i-1})$
5:     $y'' \leftarrow (\tilde{l}_{i+1} - 2\tilde{l}_i + \tilde{l}_{i-1})/h^2$
6:     $\kappa_i \leftarrow |y''|/(1 + (y')^2)^{3/2}$
7: end for
8: return $c_{\arg\max_i \kappa_i}$
```

The algorithm computes discrete curvature at each point and returns the concurrency level with maximum curvature, indicating the transition from linear to saturated scaling.

5.4 Bottleneck Attribution

The framework classifies bottlenecks into four categories based on resource utilization patterns (Table 2).

Table 2: Bottleneck classification criteria.

Type	Indicators
GPU-bound	GPU utilization $> 80\%$, stable CPU, rising TTFT
CPU-bound	High CPU time, low GPU utilization, stable memory
Memory-bound	High RSS, OOM errors, latency spikes
Queueing	Throughput plateau with exploding P99 latency

Each classification includes supporting evidence extracted from Slurm accounting data (`sacct`) and generates actionable recommendations (e.g., "Consider adding GPUs" for GPU-bound workloads).

6 Monitoring Integration

The framework includes a Prometheus/Grafana monitoring stack for real-time observability.

6.1 Distributed Client Visibility

A key challenge in HPC benchmarking is visibility into clients running on remote compute nodes. Standard process monitoring (e.g., `psutil`) is ineffective when clients and scrapers run on different nodes.

We address this with a *filesystem-based heartbeat protocol*:

1. Each client periodically updates a timestamp file on shared storage.
2. The scraper monitors file modification times via the parallel filesystem.
3. Active clients are counted as those with recent (< 15s) heartbeats.

This approach leverages the high-performance Lustre filesystem available on MeluXina, avoiding the need for additional network services.

6.2 Metrics Exposition

Benchmark metrics are exposed in Prometheus format:

```
# HELP benchmark_requests_total Total requests
benchmark_requests_total{id="BM-001",status="success"} 9876
benchmark_requests_total{id="BM-001",status="failed"} 124
# HELP benchmark_latency_seconds Latency percentiles
benchmark_latency_seconds{id="BM-001",quantile="0.99"} 0.245
```

Pre-configured Grafana dashboards provide real-time visualization of throughput, latency distributions, and active client counts.

7 User Interfaces

The framework provides three interface modalities with complete feature parity.

7.1 Command-Line Interface

The CLI supports the full workflow:

```
# Execute benchmark
python src/frontend.py recipe.yaml

# Monitor and analyze
python src/frontend.py --watch BM-20260110-001
python src/frontend.py --report BM-20260110-001
python src/frontend.py --compare BM-001 BM-002

# Sweep analysis
python src/frontend.py --sweep-report BM-001,BM-002,BM-003
```

7.2 Web Interface

The Flask-based web UI provides:

- **Dashboard**: Benchmark overview with status badges
- **Deploy**: Recipe selection with parameter customization
- **Status**: Real-time job monitoring with auto-refresh
- **Logs**: Interactive log browser with search
- **Reports**: Rendered analysis with embedded charts

8 Experimental Validation

We validated the framework through comprehensive experiments on MeluXina.

8.1 Experimental Setup

Hardware. MeluXina comprises 573 CPU nodes (AMD EPYC 7H12, 256 GB RAM) and 127 GPU nodes (4$\times$ NVIDIA A100 40GB). Nodes are interconnected via InfiniBand HDR (200 Gbps).

Services Tested. We evaluated all seven supported service types with varying concurrency levels (1–128 clients) and workload parameters.

8.2 Redis Scaling Characterization

Redis demonstrated near-linear throughput scaling (Figure 2):

- **Linear Region**: 1–64 clients showed $0.95\times$ scaling efficiency
- **Saturation Onset**: Detected at 96 clients via curvature analysis
- **Payload Sensitivity**: Values $>$ 16KB caused earlier saturation due to memory bandwidth constraints

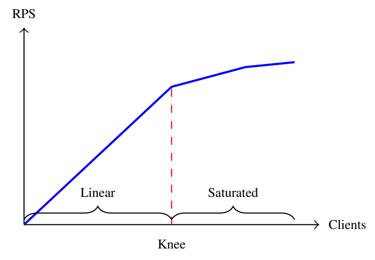

Figure 2: Conceptual Redis scaling curve showing knee-point detection.

8.3 LLM Inference Characterization

vLLM with OPT-125M on A100 GPUs exhibited GPU-bound behavior:

- **Optimal Batch Size**: 4 concurrent requests maximized tokens/second
- **TTFT Scaling**: Superlinear growth beyond 8 clients indicated KV-cache pressure
- **Bottleneck**: Correctly classified as GPU-bound with recommendation to increase tensor parallelism

9 Reproducibility

Every benchmark produces a self-contained artifact bundle:

```
results/<benchmark_id>/
  run.json           # Embedded recipe + metadata
  requests.jsonl     # Per-request timing
  summary.json       # Aggregated metrics
  logs/              # Service and client logs
```

The `run.json` file captures:

- Complete recipe YAML (embedded)
- Container image digests
- Slurm node allocations
- Lifecycle timestamps (submit, start, end)

Experiments can be exactly replicated via:

```
python src/frontend.py --rerun BM-20260110-001
```

10 Conclusion and Future Work

Key contributions include support for seven service types, automated saturation detection via curvature analysis, resource-aware bottleneck attribution, and real-time monitoring integration with novel heartbeat-based distributed visibility.

Future work includes:

- Multi-node client deployment for load generation beyond single-node capacity
- Machine learning-based anomaly detection for automatic degradation identification
- Integration with additional HPC environments (PBS, LSF)
- Power-aware efficiency metrics (tokens per watt)

Acknowledgments

We thank LuxProvide for access to the MeluXina supercomputer, the EUMASTER4HPC program for supporting this research, and Dr. Farouk Mansouri for guidance throughout the project.

References

[1] LuxProvide, "MeluXina Supercomputer," https://luxprovide.lu/meluxina/, 2024.

[2] Apache Software Foundation, "Apache JMeter," https://jmeter.apache.org/.

[3] Locust Contributors, "Locust: An open source load testing tool," https://locust.io/.

[4] Grafana Labs, "k6: Modern load testing for developers," https://k6.io/.

[5] B. F. Cooper et al., "Benchmarking cloud serving systems with YCSB," in *SoCC*, 2010.

[6] Transaction Processing Performance Council, "TPC-C Benchmark," https://www.tpc.org/tpcc/.

[7] W. Kwon et al., "Efficient memory management for large language model serving with PagedAttention," in *SOSP*, 2023.

[8] Anyscale, "LLMPerf: Evaluate LLM API endpoints," https://github.com/ray-project/llmperf.

[9] L. Adhianto et al., "HPCToolkit: Tools for performance analysis," *CCPE*, 2010.