HPC AI Benchmarking Orchestrator

Project Report

Team 10 – EUMaster4HPC Challenge

January 12, 2026

Abstract

This report details the design, implementation, and results of the HPC AI Benchmarking Orchestrator, a tool designed for automated deployment and benchmarking of AI services on HPC clusters.

Contents

1 Introduction

The HPC AI Benchmarking Orchestrator is a modular Python-based system designed to execute containerized AI benchmarking workloads on High-Performance Computing (HPC) clusters via SLURM. As Artificial Intelligence workflows increasingly demand scalable infrastructure, the need for reproducible and automated benchmarking tools has become critical.

This system enables the automated deployment and benchmarking of various AI-centric services, including Large Language Model (LLM) inference servers, standard databases, and vector stores.

1.1 Architecture Overview

The orchestrator follows a modular design comprising five main components:

1. **CLI Interface:** User-facing management.
2. **Orchestrator:** The central engine handling logic.
3. **SSH Client:** Handles remote HPC operations.
4. **SLURM Integration:** Manages job queues and scheduling.
5. **Containers:** Encapsulates services and benchmarks.

The workflow operates by processing YAML recipes to generate SLURM batch scripts, which are then deployed to the cluster to spin up servers (such as Ollama or Redis) and clients (workload generators) to test performance metrics like latency, throughput, and resource utilization.

2 Services and Clients

This section details the specific architectures and configurations for the services integrated into the orchestrator.

2.1 Chroma

Chroma is integrated into the orchestrator as a CPU-based vector database service providing embedding storage and similarity search over HTTP. The integration follows the same service/client pattern as the other components: a long-running Chroma service is deployed on a compute node via SLURM, and one or more benchmark clients are submitted as separate jobs that connect to the service endpoint.

2.1.1 Chroma Service Setup

The service configuration is defined in `recipes/services/chroma.yaml`. The orchestrator builds (if needed) an Apptainer image from `docker://chromadb/chroma:latest` and stores it at `$HOME/containers/chroma_latest.sif`. The service is started with:

```
chroma run --host 0.0.0.0 --port 8000
```

and exposes port `8000`. A health check is configured against `/api/v1/heartbeat` to ensure the server is ready before clients start. Persistence is enabled (`IS_PERSISTENT=true`) with `PERSIST_DIRECTORY=/chroma/data`.

On the code level, the service is implemented in `src/services/chroma.py` as `ChromaService`. It reuses the default service script generation provided by the orchestrator base classes and mainly wires recipe fields (resources, environment, ports, container settings) into a concrete job instance.

2.1.2 Chroma Benchmark Client

Two client recipes are provided:

- `recipes/clients/chroma_benchmark.yaml`: single-run benchmark
- `recipes/clients/chroma_parametric.yaml`: parameter sweep over multiple configurations

Both clients use a lightweight Python container (`docker://python:3.11-slim`) and execute Python benchmark scripts uploaded to `$HOME/benchmark_scripts/`:

- `benchmark_scripts/chroma_benchmark.py`
- `benchmark_scripts/chroma_parametric_benchmark.py`

The client resolves the target endpoint via the orchestrator (`-target-service <SERVICE_ID>` or explicit `-target-endpoint`) and connects using Chroma's HTTP client (`chromadb.HttpClient`). The benchmark generates random, normalized embeddings, inserts documents in configurable batches, and then performs similarity search queries (`top_k`) while collecting throughput and latency statistics (average, P95, P99).

Parametric sweep. The parametric benchmark evaluates a full grid of configurations (in this report: 4 document counts $\times$ 4 embedding dimensions $\times$ 4 batch sizes = 64 runs). For each configuration it creates a dedicated collection (with a prefix and timestamp), runs insertion and query phases, and writes a single JSON result file named `chroma_parametric_{SLURM_JOB_ID}.json` to `$HOME/results/`.

2.1.3 Results Collection and Analysis

Downloaded results are stored under `results/` in the repository (e.g., `results/chroma_parametric_X.json`). Plots are generated with `analysis/plot_chroma_results.` which parses the JSON schema and produces seven figures (scaling plots and heatmaps). For the final report we include the pre-generated figures from `EUMasterChallenge_Group10/plotsChroma/`.

2.2 Ollama

Ollama is integrated into the orchestrator as a GPU-accelerated Large Language Model (LLM) inference service, providing text generation capabilities via HTTP API. The integration follows the standard service/client pattern: a persistent Ollama service runs on a GPU node via SLURM, and benchmark clients submit inference requests to evaluate performance under varying workload conditions.

2.2.1 Ollama Service Setup

The service configuration is defined in `recipes/services/ollama_with_cadvisor.yaml`. The orchestrator builds (if needed) an Apptainer image from `docker://ollama/ollama:latest` and stores it at `$HOME/containers/ollama_latest.sif`. Unlike CPU-bound services, Ollama requires GPU resources and is allocated via SLURM's `gres` mechanism (e.g., `gres: "gpu:1"`).

The service starts with:

`ollama serve`

and exposes port `11434`. A health check is configured against `/api/tags` to verify the server is ready before clients connect. Key environment variables include:

- `OLLAMA_HOST: "0.0.0.0:11434"`: bind to all interfaces
- `OLLAMA_KEEP_ALIVE: "5m"`: keep models loaded in GPU memory for 5 minutes after last request
- `CUDA_VISIBLE_DEVICES: "0"`: specify GPU device

On the code level, the service is implemented in `src/services/ollama.py` as `OllamaService`, which extends the base `Service` class and primarily wires recipe fields into a concrete SLURM job instance.

2.2.2 Ollama Benchmark Client

Two client recipes are provided:

- `recipes/clients/ollama_benchmark.yaml`: single-run benchmark
- `recipes/clients/ollama_parametric.yaml`: parameter sweep over multiple configurations

Both clients use a lightweight Python container (`docker://python:3.11-slim`) and execute Python benchmark scripts uploaded to `$HOME/benchmark_scripts/`:

- `benchmark_scripts/ollama_benchmark.py`
- `benchmark_scripts/ollama_parametric_benchmark.py`

The client resolves the target endpoint via the orchestrator (`-target-service <SERVICE_ID>` or explicit `-target-endpoint`) and connects to the Ollama HTTP API (`/api/generate`). The benchmark generates text-completion requests with configurable prompt lengths and output token limits, measuring throughput (requests/sec and tokens/sec) and latency statistics (mean, median, P95, P99).

Parametric sweep. The parametric benchmark evaluates a comprehensive grid of configurations:

- **Concurrent requests**: 1, 2, 5, 10, 20
- **Prompt lengths**: 50, 100, 200, 500 tokens
- **Max output tokens**: 50, 100, 200, 500

This yields $5 \times 4 \times 4 = 80$ configurations. For each configuration, the benchmark runs 20 inference operations using the `llama2` model, records performance metrics (throughput, latency, tokens per request), and writes a single JSON result file named `ollama_parametric_{SLURM_JOB_ID}.json` to `$HOME/results/`.

2.2.3 Results Collection and Analysis

Downloaded results are stored under `results/` in the repository (e.g., `results/ollama_parametric_*.jso`
Plots are generated with `analysis/plot_ollama_results.py`, which produces seven figures

analyzing throughput scaling, latency behavior, and efficiency metrics. For the final report, pre-generated figures from `report/plotsOllama/` are included.

2.3 Redis

Redis is integrated into the orchestrator as an in-memory key–value database service, deployed via Apptainer and benchmarked using the native `redis-benchmark` tool wrapped by lightweight Python scripts for structured results collection.

2.3.1 Redis Service Setup

The Redis service is defined in `recipes/services/redis.yaml`. It uses the Redis 7 Alpine container (`docker://redis:7-alpine`) converted to an Apptainer image (default: `$HOME/containers/redis_`
At job startup, the orchestrator generates a Redis configuration file on the node (under `$HOME/redis/confi`
and bind-mounts both configuration and data directories into the container:

- `$HOME/redis/config` → `/redis/config`
- `$HOME/redis/data` → `/redis/data`

The generated `redis.conf` configures Redis to listen on all interfaces (`bind 0.0.0.0`, port `6379`) with `protected-mode no` to allow client connections within the HPC node. Persistence is controlled via the environment variable `REDIS_PERSISTENCE` with supported modes:

- `none`: no persistence (AOF disabled, RDB saves disabled)
- `rdb`: periodic RDB snapshots enabled
- `aof`: append-only file enabled
- `both`: enables both RDB and AOF (default in the recipe)

Optionally, authentication can be enabled by setting `REDIS_PASSWORD`, which adds `requirepass` in the generated config.

The default SLURM allocation for the Redis service is CPU-only (single node/task) with a modest memory footprint (e.g., `mem: 4GB`) and a typical time limit of one hour. A basic health check loop is included to ensure `redis-cli ping` succeeds before reporting the endpoint.

2.3.2 Redis Benchmark Client

The Redis benchmark client is defined in `recipes/clients/redis_benchmark.yaml` (single run) and `recipes/clients/redis_parametric.yaml` (parameter sweep). The client executes a host-level Python wrapper (uploaded to `$HOME/benchmark_scripts/`) which in turn invokes the native `redis-benchmark` binary inside the same Redis container image via Apptainer. This hybrid approach keeps the load generator containerized while enabling robust parsing and JSON result output in Python.

Single-run mode. The single-run wrapper (`benchmark_scripts/redis_benchmark.py`) maps recipe parameters to `redis-benchmark` flags, e.g.:

- `num_operations` → `-n`
- `clients` → `-c`
- `value_size` → `-d`

- `pipeline` → `-P`
- `native_tests` → `-t` (comma-separated operations)

The wrapper runs `redis-benchmark` with `-csv` output enabled and parses each row into a structured schema:

- `requests_per_second`: the `rps` column
- `metrics`: a fixed 6-value latency array [`avg`, `min`, `p50`, `p95`, `p99`, `max`] in milliseconds

The CSV header row is skipped during parsing to avoid schema inconsistencies.

Parametric mode. The parametric suite (`benchmark_scripts/redis_parametric_benchmark.py`) performs a sweep across a grid of parameters, typically varying:

- client counts (e.g., 1–500),
- payload sizes (e.g., 64B–64KB),
- pipeline depths (e.g., 1–256),

and running multiple Redis operations (e.g., `set`, `get`, list/set/hash/zset operations). For each configuration, the script records throughput and expands latency metrics into named fields (`latency_avg_ms`, `latency_p95_ms`, `latency_p99_ms`, etc.), enabling downstream analysis.

2.3.3 Startup and Target Resolution

A typical workflow is:

1. Start Redis service: `python main.py -recipe recipes/services/redis.yaml`
2. Run client against the service using `-target-service <SERVICE_ID>`, allowing the orchestrator to resolve the SLURM node hostname and construct the endpoint `<host>:6379`.

2.3.4 Results Collection and Analysis

Both single-run and parametric benchmarks write JSON results and copy them to `$HOME/results/` on the cluster. Result files follow the naming convention:

- `redis_benchmark_{SLURM_JOB_ID}.json` (single-run)
- `redis_parametric_{SLURM_JOB_ID}.json` (parametric)

They can be downloaded to the local repository via `python main.py -download-results`. For parametric runs, plots can be generated with `python analysis/plot_redis_results.py`, producing throughput/latency scaling plots and heatmaps under `analysis/plots/`.

2.4 MySQL

2.4.1 MySQL Service Setup

The MySQL service is deployed using a containerized approach, leveraging Docker images and HPC job scheduling through SLURM. The service configuration is defined in the YAML recipe file `recipes/services/mysql.yaml`, which specifies the complete setup for running a MySQL 8.0 database instance.

Container Configuration The service pulls the official MySQL 8.0 Docker image (`docker://mysql:8.0`) and converts it to a Singularity/Apptainer container image (`mysql_latest.sif`) stored at `/mnt/tier2/users/u_id/mysql_latest.sif`. The container is configured with persistent storage through bind mounts:

```
bind_mounts:
  - "/mnt/tier2/users/u103300/mysql:/mysql"
```

This ensures that database data persists across container restarts and is stored on the HPC cluster's shared filesystem.

Database Initialization and User Setup The MySQL service performs automatic database initialization and user configuration through an initialization script. The service starts MySQL with the following key arguments:

```
args: [
  "--datadir=/mysql/data",
  "--socket=/mysql/run/mysqld.sock",
  "--pid-file=/mysql/run/mysqld.pid",
  "--tmpdir=/mysql/tmp",
  "--skip-networking=0",
  "--bind-address=0.0.0.0",
  "--user=mysql"
]
```

The initialization script creates the benchmark database and sets up user permissions:

```
CREATE USER IF NOT EXISTS 'benchmark_user'@'%' IDENTIFIED WITH
  mysql_native_password BY 'benchmark_pass';
CREATE USER IF NOT EXISTS 'benchmark_user'@'localhost' IDENTIFIED WITH
  mysql_native_password BY 'benchmark_pass';
CREATE DATABASE IF NOT EXISTS benchmark_db;
GRANT ALL PRIVILEGES ON benchmark_db.* TO 'benchmark_user'@'%';
GRANT ALL PRIVILEGES ON benchmark_db.* TO 'benchmark_user'@'localhost';
ALTER USER 'root'@'localhost' IDENTIFIED BY 'mysecretpassword';
FLUSH PRIVILEGES;
```

This setup creates a dedicated `benchmark_db` database with a `benchmark_user` account that has full privileges on the database, allowing remote connections from any host.

Resource Allocation The service is allocated HPC resources through SLURM with the following specifications:

```
resources:
  time: "01:00:00"
  qos: default
  partition: cpu
  nodes: 1
  ntasks: 1
  ntasks_per_node: 1
  mem: "4GB"
```

Service Startup Command The MySQL service is started using the orchestrator's command-line interface:

```
python main.py --recipe recipes/services/mysql.yaml
```

This command loads the recipe, generates the appropriate SLURM batch script, and submits the job to the HPC cluster. The service runs MySQL daemon (`mysqld`) within the container, listening on port 3306.

2.4.2 MySQL Benchmark Client

The benchmark client is designed to stress-test the MySQL service using the industry-standard `sysbench` benchmarking tool. The client configuration is defined in `recipes/clients/mysql_benchmark.yaml`.

Client Container Configuration Similar to the service, the client uses a containerized approach, pulling the `severalnines/sysbench` Docker image (`docker://severalnines/sysbench`) which contains a pre-installed and configured instance of sysbench. This image is converted to `mysql_client.sif` and stored at `/mnt/tier2/users/u_id/mysql_client.sif`.

Benchmark Parameters The client executes a comprehensive OLTP (Online Transaction Processing) benchmark with the following parameters:

```
parameters:
  num_connections: 16
  transactions_per_client: 300
  tables: 10
  table_size: 100000
```

This configuration creates 10 test tables, each containing 100,000 rows (1 million total rows), and runs the benchmark for 300 seconds using 16 concurrent connections.

Connection Configuration The client connects to the MySQL service using environment variables that match the service's user setup:

```
environment:
  MYSQL_USER: "benchmark_user"
  MYSQL_PASSWORD: "benchmark_pass"
  MYSQL_DATABASE: "benchmark_db"
```

Benchmark Execution Phases The sysbench benchmark executes in three distinct phases within the client container:

1. **Prepare Phase**: Creates the test database schema and populates tables with initial data
2. **Run Phase**: Executes concurrent read/write transactions against the database for the specified duration
3. **Cleanup Phase**: Removes the test data and tables

The benchmark specifically uses the `oltp_read_write` test, which simulates typical OLTP workloads with a mix of SELECT, INSERT, UPDATE, and DELETE operations. During execution, sysbench reports performance metrics every 10 seconds, including transactions per second, query latency statistics, and 95th percentile response times.

Client Startup Command Once the MySQL service is running, the benchmark client is started by targeting the specific service instance:

```
python main.py --recipe recipes/clients/mysql_benchmark.yaml
--target-service <SERVICE_ID>
```

Where `<SERVICE_ID>` is the identifier returned when starting the service (e.g., `mysql_abc123`).

Result Collection Benchmark results are automatically saved to `/tmp/sysbench_results.txt` within the client container and copied to the submission directory's `results/` folder for analysis. The results include detailed performance metrics such as transaction throughput, query response times, and system resource utilization statistics.

2.4.3 Integration and Execution Flow

The service-client architecture enables seamless benchmark execution on HPC infrastructure. The MySQL service provides a standardized database environment, while the sysbench client generates realistic workload patterns that stress-test database performance under concurrent access. This setup allows for reproducible benchmarking across different HPC cluster configurations and facilitates performance analysis of MySQL deployments in high-performance computing environments.

3 Benchmark Results

This section presents the performance metrics and analysis gathered from the execution of benchmarks on the MeluXina supercomputer.

3.1 Chroma Benchmark Results

Chroma was benchmarked in a parametric configuration sweep against a single service endpoint (`http://mel0039:8000`). The evaluated parameter ranges were:

- **Documents per run**: 500, 1000, 2000, 5000
- **Embedding dimension**: 192, 384, 768, 1536
- **Insertion batch size**: 50, 100, 200, 500
- **Queries**: 1000 per configuration, **top-k**: 10

The sweep completed **64/64** successful runs (results file: `results/chroma_parametric_3772030.json`). Table 1 summarizes representative extrema across the sweep.

Table 1: Summary of Chroma Parametric Benchmark (64 configurations)

Metric	Best/Worst Configuration	Value
Peak insertion throughput	500 docs, dim=192, batch=500	578.4 docs/s
Lowest insertion throughput	500 docs, dim=192, batch=50	34.3 docs/s
Best query P99 latency	500 docs, dim=192, batch=200	6.06 ms
Worst query P99 latency	500 docs, dim=1536, batch=200	17.83 ms
Query throughput range	across all configurations	117.2–207.8 qps

3.1.1 Insertion Throughput

Figure 1 shows that insertion throughput scales strongly with batch size: moving from batch 50 to batch 500 yields an order-of-magnitude improvement for small collections. Across the tested ranges, the batch size dominated insertion performance more than embedding dimension or document count. The heatmap in Figure 4 (left) highlights the same effect for a fixed embedding dimension (384).

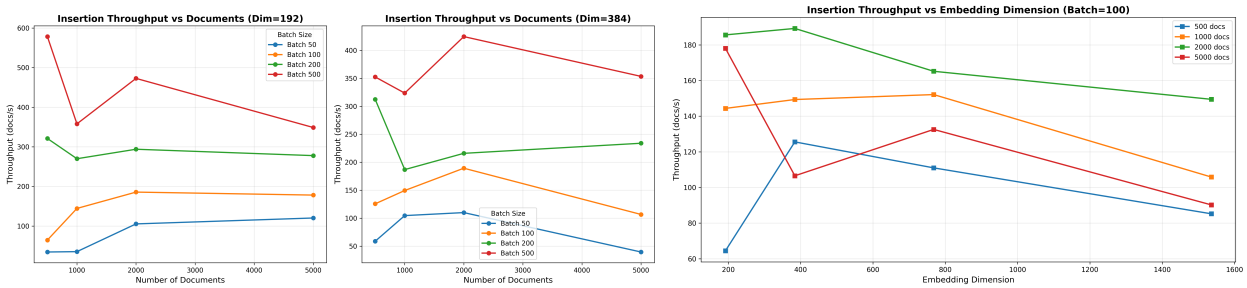

Figure 1: Insertion throughput scaling: (left) throughput vs. documents for different batch sizes; (right) throughput vs. embedding dimension (batch=100).

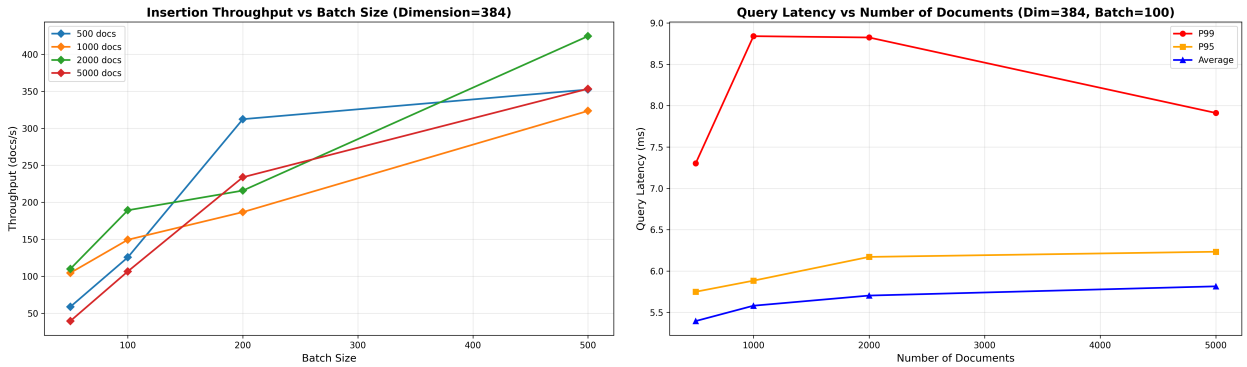

Figure 2: Chroma benchmark: (left) insertion throughput vs. batch size (dimension=384); (right) query latency vs. number of documents (dimension=384, batch=100).

3.1.2 Query Latency and Throughput

Query performance remained stable across the tested dataset sizes (up to 5000 documents) with P99 latencies typically in the single-digit milliseconds for common configurations (e.g., dimension=384, batch=100). Increasing embedding dimension raises the compute cost per distance evaluation and is reflected in higher latency, as shown in Figure 3. The overall query throughput across the sweep ranged from **117** to **208** queries per second.

Overall, the benchmarks indicate that Chroma provides low-latency similarity search for the evaluated scales, while ingestion performance depends heavily on batching. For larger experiments, increasing batch size and provisioning additional memory for persistent storage are the most impactful tuning knobs.

3.2 Ollama Benchmark Results

Ollama was benchmarked using a parametric configuration sweep to evaluate the performance of the `llama2` model under varying concurrency levels, prompt lengths, and output token limits. The service was deployed on a GPU node (`mel2042`) and subjected to a grid of 80

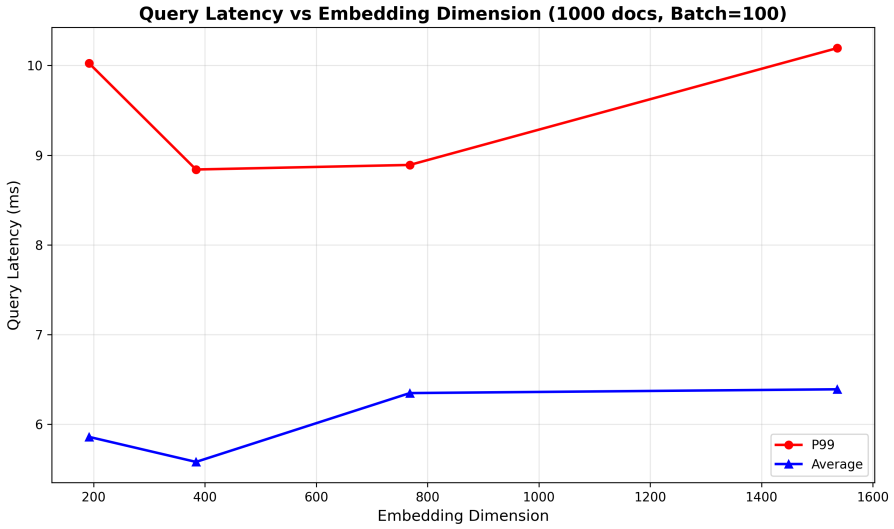

Figure 3: Query latency as a function of embedding dimension (1000 documents, batch=100).

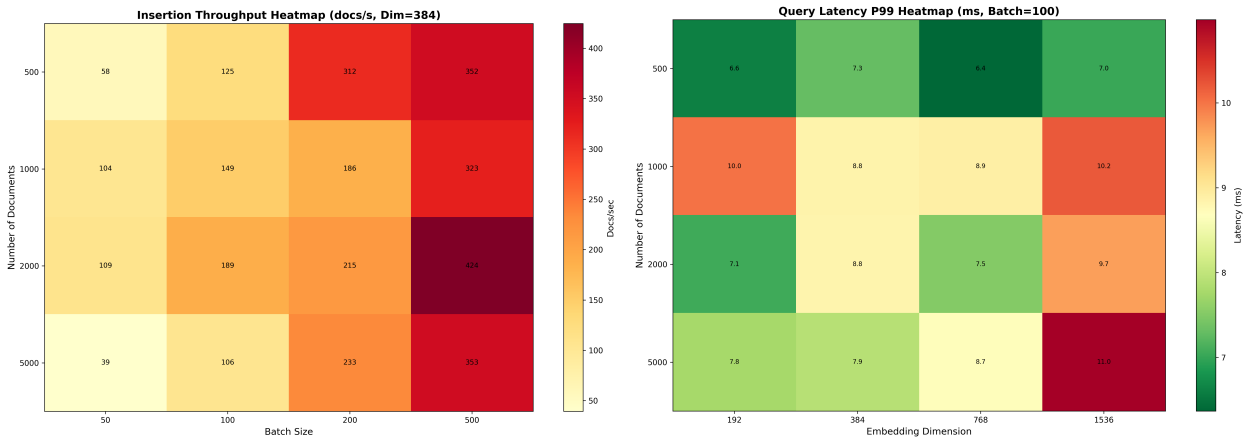

Figure 4: Heatmaps: (left) insertion throughput vs. documents and batch size (dimension=384); (right) query P99 latency vs. documents and embedding dimension (batch=100).

configurations. All 80 runs completed successfully, consuming approximately 42 minutes of total execution time.

3.2.1 Parameter Space and Overview

The evaluated parameter ranges were:

- **Concurrent requests**: 1, 2, 5, 10, 20
- **Prompt length**: 50, 100, 200, 500 tokens
- **Max output tokens**: 50, 100, 200, 500
- **Operations per configuration**: 20 inference requests
- **Model**: llama2

Table 2 summarizes key performance metrics across the parameter space.

Table 2: Summary of Ollama Parametric Benchmark (80 configurations)

Metric	Configuration	Value
Peak request throughput	10 conc., 50 prompt, 50 tokens	3.15 req/s
Peak token throughput	5 conc., 500 prompt, 50 tokens	1534 tok/s
Best latency (mean)	1 conc., 50 prompt, 50 tokens	0.37 s
Worst latency (mean)	20 conc., 500 prompt, 500 tokens	35.5 s
Request throughput range	across all configurations	0.29–3.15 req/s
Token throughput range	across all configurations	108–1534 tok/s
Success rate	all configurations	100%

3.2.2 Concurrency Scaling

Figure 5 shows how throughput scales with increasing concurrent requests. For the baseline configuration (prompt=100, max_tokens=100), **both request and token throughput remain relatively stable** across concurrency levels. Request throughput hovers around 1.4–1.6 requests per second, while token throughput stays in the 230–260 tokens/sec range, regardless of concurrency level from 1 to 20.

This stability indicates that for this moderate workload configuration, the system achieves consistent performance across different concurrency levels. The GPU appears to maintain similar processing rates whether handling requests sequentially or concurrently, suggesting that the inference pipeline is well-balanced for this prompt and output length combination.

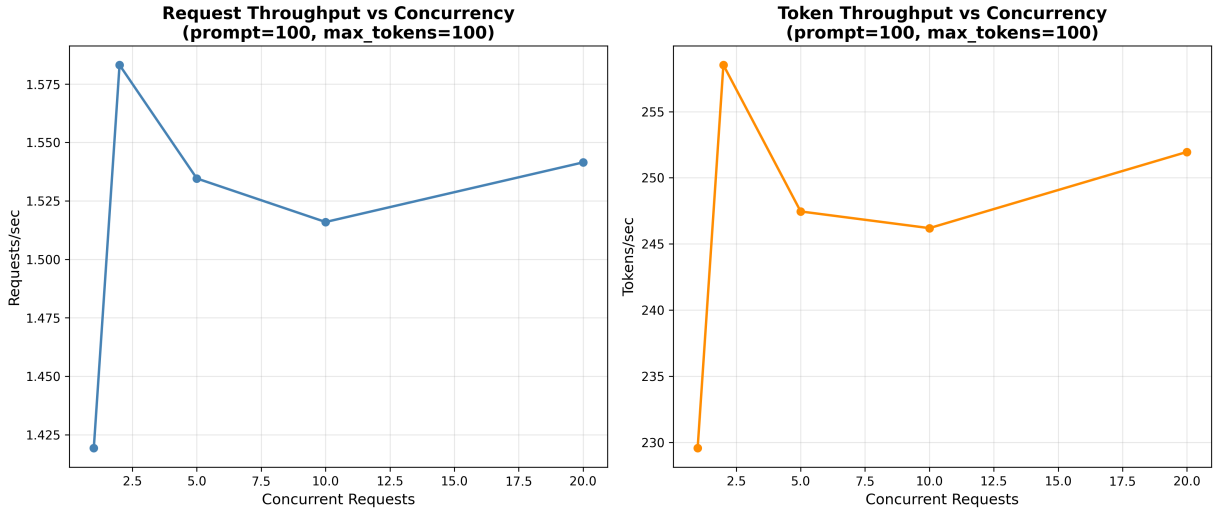

Figure 5: Throughput vs. concurrent requests: (left) request-level throughput; (right) token-level throughput (prompt=100, max_tokens=100).

Latency behavior (Figure 6) shows a near-linear increase with concurrency: mean latency rises from 0.70s (1 concurrent) to 6.67s (20 concurrent) for the baseline configuration. This linear scaling is expected as higher concurrency means more requests compete for GPU resources, increasing queue wait times even though throughput remains stable.

3.2.3 Prompt Length and Output Token Impact

Figure 7 demonstrates the impact of prompt length on throughput. **Request throughput** (left) degrades slightly as prompt length increases from 50 to 500 tokens, with the steepest drop

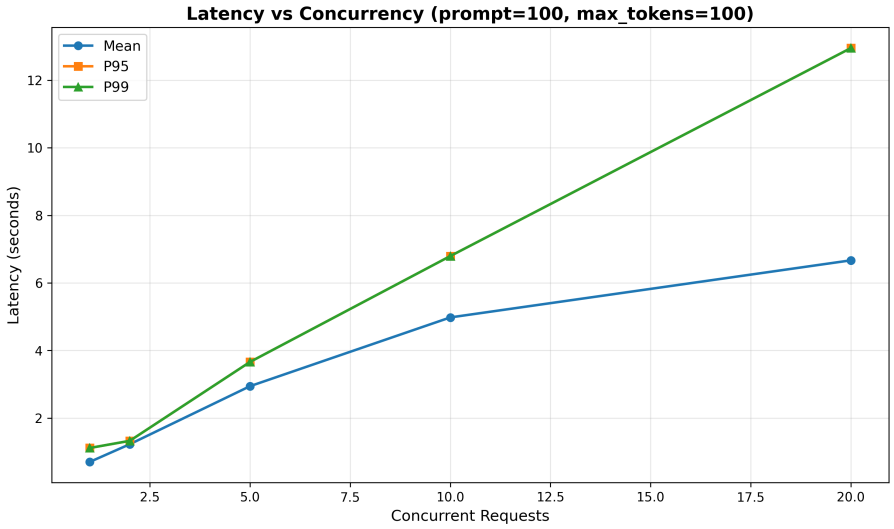

Figure 6: Latency vs. concurrent requests (prompt=100, max_tokens=100). Mean and P95 latencies increase linearly with load.

observed at higher concurrency levels (e.g., 20 concurrent requests drop from 0.8 to 0.4 req/s). This behavior is attributed to increased prompt encoding time for longer inputs.

Token throughput (right) shows a more nuanced pattern: for low concurrency (1–2 requests), token throughput increases with prompt length, peaking at prompt=200 before declining. At higher concurrency (10–20 requests), token throughput remains more stable across prompt lengths, hovering around 400–600 tok/s. This suggests that batch processing benefits saturate at moderate prompt lengths.

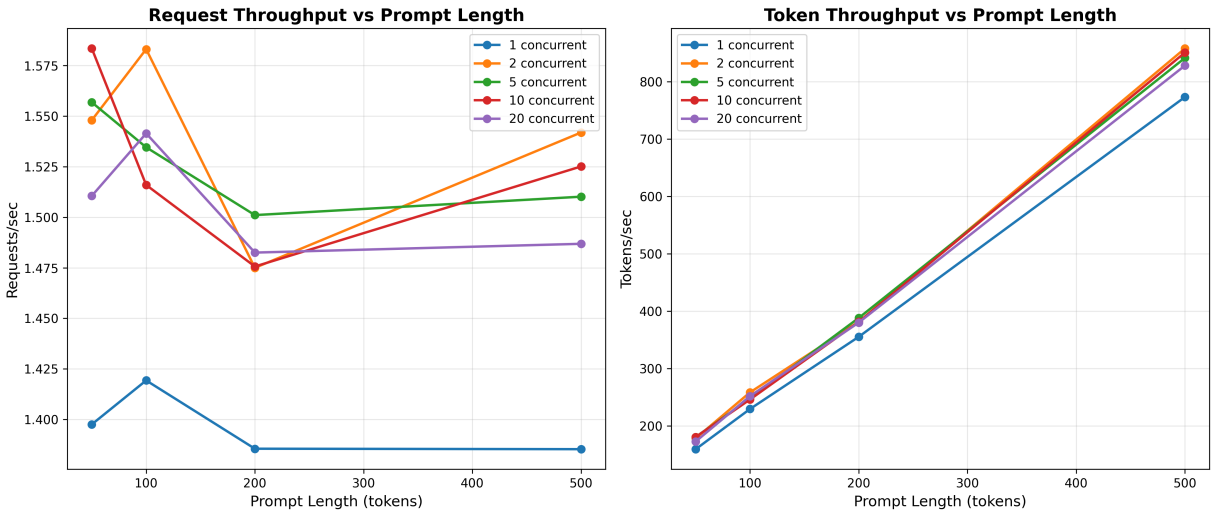

Figure 7: Throughput vs. prompt length (max_tokens=100): (left) request throughput; (right) token throughput. Different lines represent varying concurrency levels.

The impact of **max output tokens** is illustrated in Figure 8. As expected, increasing the output length from 50 to 500 tokens drastically reduces request throughput (left panel): throughput drops from over 2.5 req/s (50 tokens) to approximately 0.3 req/s (500 tokens)—nearly a 10× reduction. This is intuitive: generating 10× more tokens requires proportionally more time. Token throughput (right panel) varies significantly with both output length and concurrency, showing more complex behavior that depends on the specific prompt length and workload configuration.

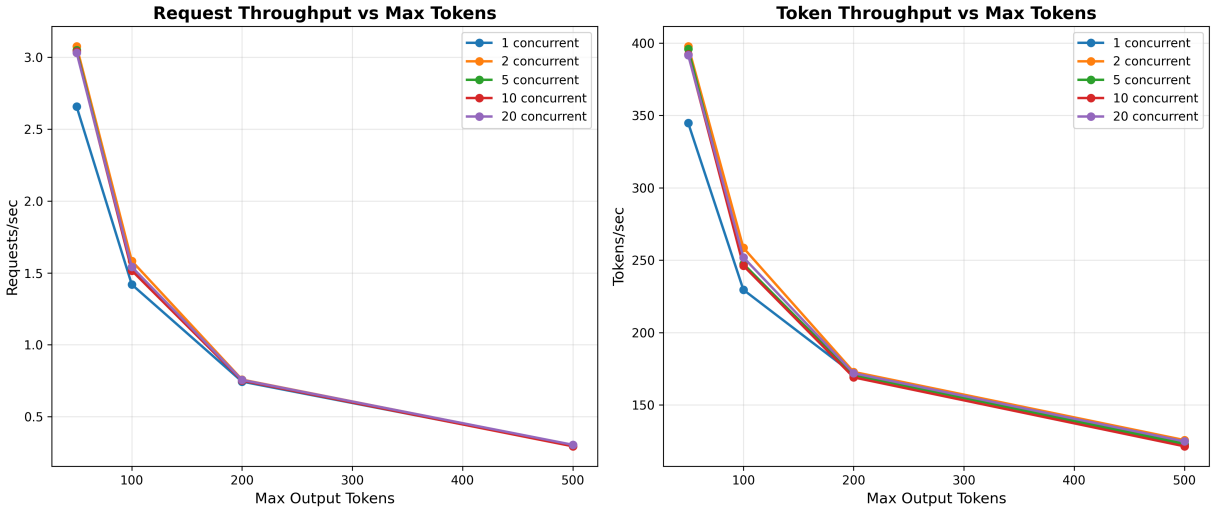

Figure 8: Throughput vs. max output tokens (prompt=100): (left) request throughput declines sharply with longer outputs; (right) token throughput scales better with concurrency.

3.2.4 Performance Heatmaps

Figures 9 and 10 provide heatmap visualizations of the performance landscape. Figure 9 (request throughput) reveals that optimal request throughput is achieved at low concurrency with short prompts and short outputs (top-left corner), reaching 2–2.5 req/s. Performance degrades uniformly across both axes as workload intensity increases.

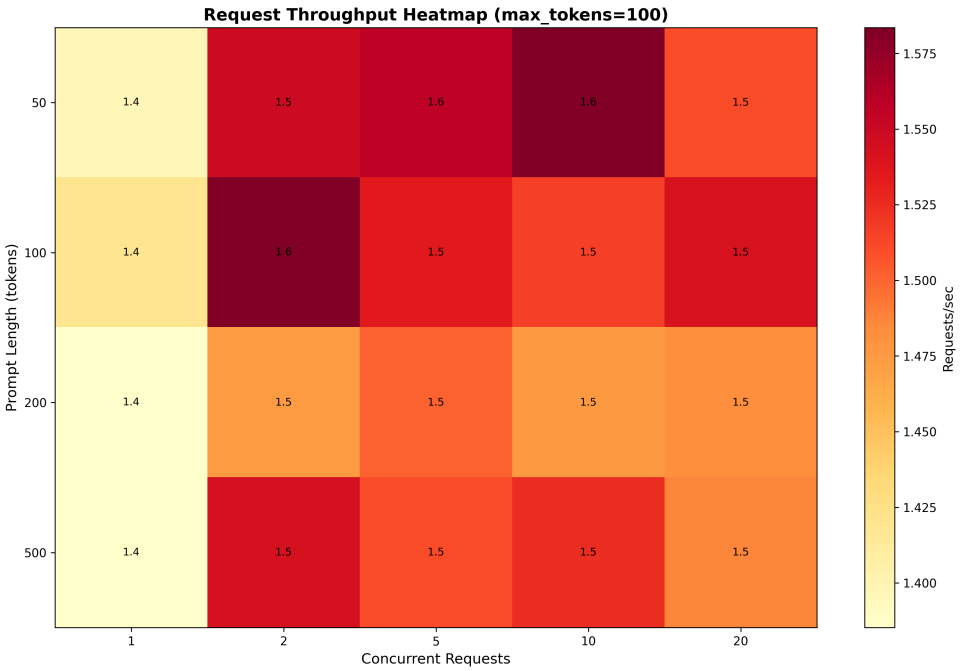

Figure 9: Heatmap of request throughput (req/s) vs. concurrent requests and prompt length (max_tokens=100). Lighter colors indicate higher throughput.

Conversely, Figure 10 (token throughput) shows a **sweet spot** in the high-concurrency, short-to-medium prompt region (bottom-right quadrant), where token throughput exceeds 600 tok/s. This confirms that the token-generation phase benefits significantly from concurrent request batching, whereas prompt encoding overhead dominates at longer prompt lengths.

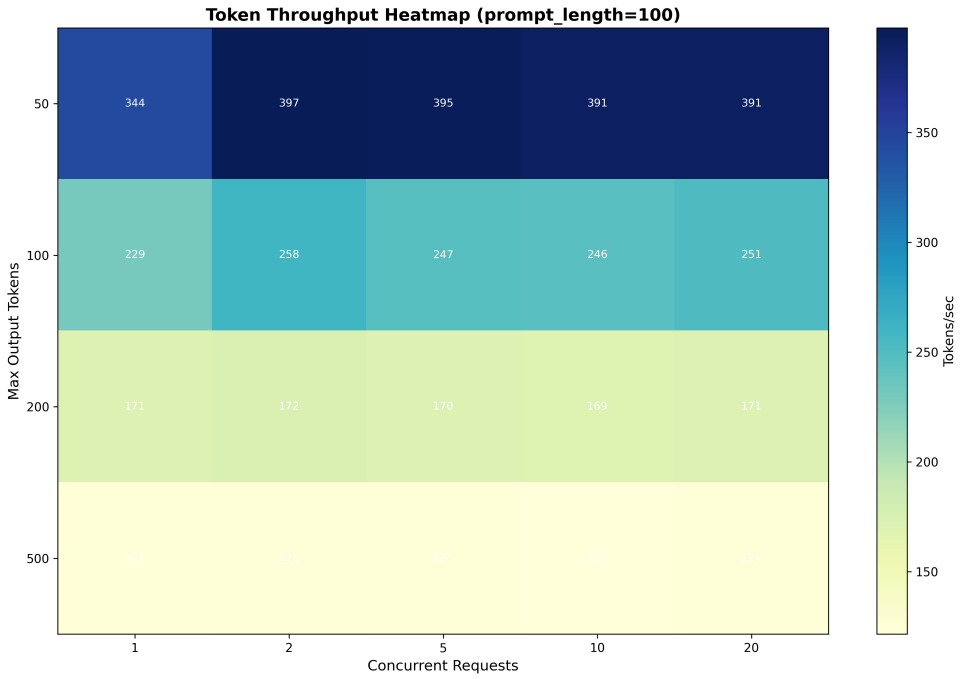

Figure 10: Heatmap of token throughput (tok/s) vs. concurrent requests and prompt length (max_tokens=100). High concurrency with moderate prompts yields peak token throughput.

3.2.5 Efficiency Analysis

Figure 11 presents a combined efficiency analysis, plotting both request throughput and token throughput against concurrent requests for different prompt lengths. The divergence between request-level and token-level throughput curves highlights the architectural trade-off: while the system cannot significantly increase the rate of *completed requests* under high concurrency, it can substantially increase the rate of *generated tokens* by batching token decoding across multiple active sequences.

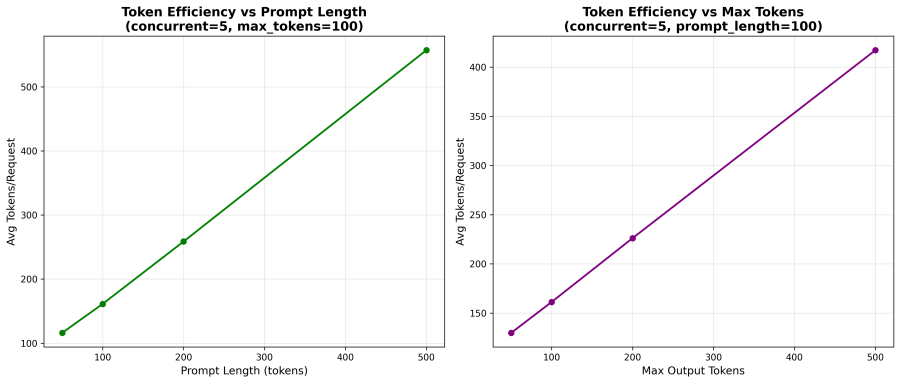

Figure 11: Efficiency analysis: request throughput vs. token throughput across concurrency levels and prompt lengths (max_tokens=100). Token throughput scales favorably with concurrency.

3.2.6 Summary

The Ollama benchmarks reveal several key characteristics for GPU-accelerated LLM inference in HPC environments:

1. **Throughput stability at moderate workloads**: For balanced configurations (prompt=100,

max_tokens=100), both request and token throughput remain stable (1.4–1.6 req/s, 230–260 tok/s) across concurrency levels 1–20.

2. **Configuration-dependent scaling**: Throughput behavior varies significantly with prompt length and output token limits. Peak token throughput (1534 tok/s) is achieved with long prompts and short outputs at moderate concurrency.
3. **Prompt length impacts performance**: Very long prompts (500 tokens) can achieve high token throughput when paired with short outputs, but degrade request throughput.
4. **Output length dominates latency**: Increasing max output tokens from 50 to 500 reduces request throughput by $\sim$10$\times$, as generation time scales nearly linearly with output length.
5. **Optimal configuration for balanced workloads**: Moderate concurrency (5–10 requests), prompt length 100–200 tokens, max output tokens 50–100 provides consistent performance with latencies under 5s mean.

These findings suggest that for HPC-based LLM serving, the Llama2 model exhibits stable performance at moderate workload configurations. Workload managers should focus on **balancing prompt and output lengths** based on use-case requirements, as these parameters have more impact than concurrency level for typical configurations. Peak token throughput can be achieved by exploiting long-prompt, short-output scenarios, while balanced workloads maintain consistent latency across varying concurrency.

3.3 Redis Benchmark Results

The Redis in-memory database was benchmarked using the native `redis-benchmark` tool across multiple configurations. Both single-run tests (50 clients, 256B payload) and comprehensive parametric sweeps were conducted to characterize performance across different client counts, payload sizes, and pipeline depths.

3.3.1 Baseline Performance

Figure 12 presents the baseline throughput for core Redis operations at the default configuration (50 clients, 256-byte payload, no pipelining). Read operations (GET) achieve the highest throughput at 62.7K requests per second, followed by PING operations, with write operations (SET) achieving 49.3K req/s—approximately 21% lower than reads.

3.3.2 Latency Analysis

Table 3 summarizes the latency characteristics across operations. All operations exhibit sub-millisecond average latencies, with GET achieving the lowest at 0.74ms. Notably, the SET operation displays a significant tail latency anomaly, with a maximum latency of 35.5ms—approximately 13$\times$ higher than other operations. This outlier is attributed to Redis's persistence mechanism (AOF/RDB snapshots) momentarily blocking the event loop during background saves.

3.3.3 Pipeline Depth Impact

The most significant performance lever identified is command pipelining. Figure 13 demonstrates that increasing the pipeline depth from 1 to 64 yields a **20$\times$ throughput improvement**, reaching approximately 1.4 million requests per second. Beyond pipeline depth 64,

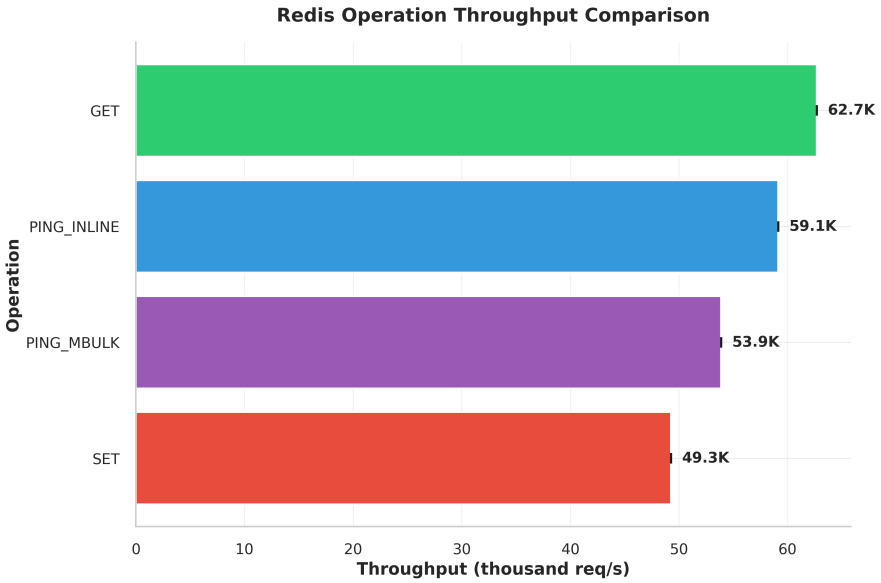

Figure 12: Baseline Redis operation throughput (50 clients, 256B payload, pipeline=1).

Table 3: Redis Latency Distribution by Operation (ms)

Operation	Min	Avg	P95	P99	Max
GET	0.02	0.74	1.32	1.69	2.62
SET	0.05	0.96	1.42	1.71	**35.52**
PING_INLINE	0.02	0.78	1.38	1.74	2.62
PING_MBULK	0.02	0.86	1.48	1.85	2.80

performance saturates and begins to decline slightly for higher client counts, likely due to increased memory pressure and scheduling overhead.

3.3.4 Parameter Space Analysis

Figure 14 visualizes GET throughput across the parameter space of client count and payload size. Key observations include:

- **Optimal region**: 100–200 clients with 64–256B payloads achieve peak throughput (120–136K req/s).
- **Payload sensitivity**: Throughput degrades significantly with larger payloads; 64KB payloads achieve only 10–18K req/s—approximately 7× lower than 64B payloads.
- **Client scaling**: Performance peaks at intermediate client counts (10–200) before declining due to connection management overhead.

3.3.5 P99 Latency Scaling

Figure 15 shows P99 latency behavior as client count increases. Both SET and GET operations exhibit similar scaling patterns, with P99 latency remaining below 2ms up to 50 clients, then increasing to 3–3.5ms at 200 clients. The non-monotonic behavior observed (dips at certain client counts) reflects Redis's single-threaded architecture efficiently batching concurrent requests.

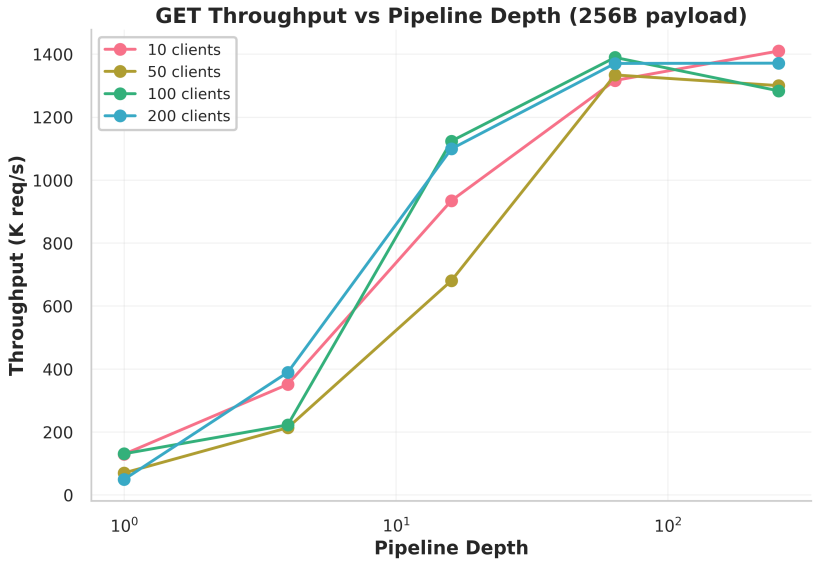

Figure 13: GET throughput scaling with pipeline depth (256B payload). Pipeline depth of 64 achieves near-optimal performance.

3.3.6 Summary

The Redis benchmarks reveal several key characteristics relevant to HPC deployments:

1. **High baseline throughput**: 50–60K req/s achievable with minimal configuration.
2. **Pipeline optimization critical**: Enabling pipelining (depth 16–64) can improve throughput by 10–20×.
3. **Latency predictability**: Sub-2ms P99 latency at moderate load, though write operations exhibit occasional tail latency spikes during persistence operations.
4. **Optimal configuration**: For throughput-sensitive workloads, 100–200 clients with small payloads (64–256B) and pipeline depth 64 provides the best performance profile.

3.4 MySQL Benchmark Results

A comprehensive performance analysis was conducted on the MySQL service using the `sysbench` OLTP read-write workload. The benchmark utilized 16 concurrent threads over a duration of 300 seconds to evaluate throughput stability, latency distribution, and resource contention on the HPC node.

3.4.1 Overview of Metrics

The system processed a total of 34,418 transactions (approx. 688,360 queries) during the test run. The aggregate performance metrics are summarized in Table 4. While the average throughput was respectable at 114.67 transactions per second (TPS), the latency metrics indicate significant variance, with a "long tail" phenomenon where maximum latency reached 811.30 ms—nearly six times the average.

3.4.2 Stability and Resource Contention

The temporal analysis of the benchmark reveals a distinct instability in performance. As illustrated in Figure 16, the system exhibits a "saw-tooth" behavior rather than a steady state.

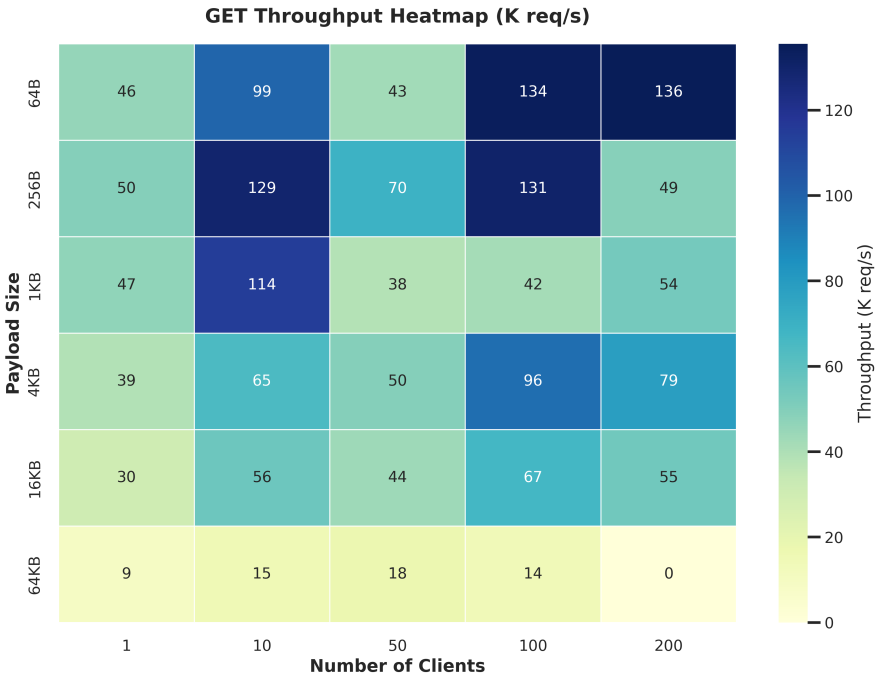

Figure 14: GET throughput heatmap showing the interaction between client count and payload size (pipeline=1).

Table 4: Summary of MySQL Sysbench OLTP Results (300s Run)

Metric	**Value**
Total Transactions	34,418
Average Throughput	114.67 TPS
Average Query Rate	2,293.50 QPS
Latency	
Minimum	6.03 ms
Average	139.52 ms
95th Percentile	267.41 ms
Maximum	811.30 ms

There is a visible inverse relationship between throughput and latency: periods of high latency (spikes) coincide perfectly with drops in transaction throughput.

This behavior is further characterized in Figure 17, which plots the 95th percentile latency against throughput. The distinct negative slope indicates that the system is likely suffering from **resource stalling**. Unlike a typical saturation curve where latency increases as throughput rises, here the throughput drops *because* the system is blocking. This pattern suggests intermittent bottlenecks, likely due to disk I/O contention or database locking mechanisms, which cause requests to pile up (increasing latency) and momentarily halt processing (decreasing TPS).

3.4.3 Workload Distribution

Despite the performance fluctuations, the workload generator maintained a consistent query mix throughout the execution. Figure 18 confirms a steady Read/Write ratio of approximately

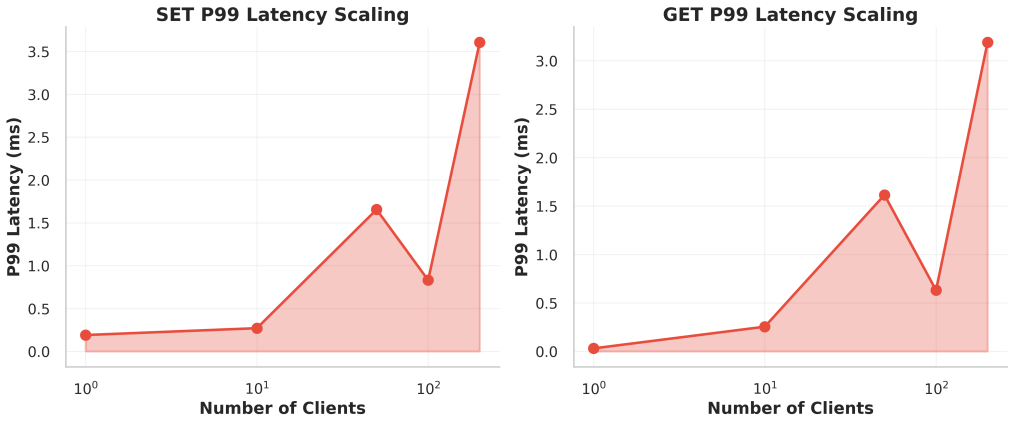

Figure 15: P99 latency scaling with increasing client count (256B payload, pipeline=1).

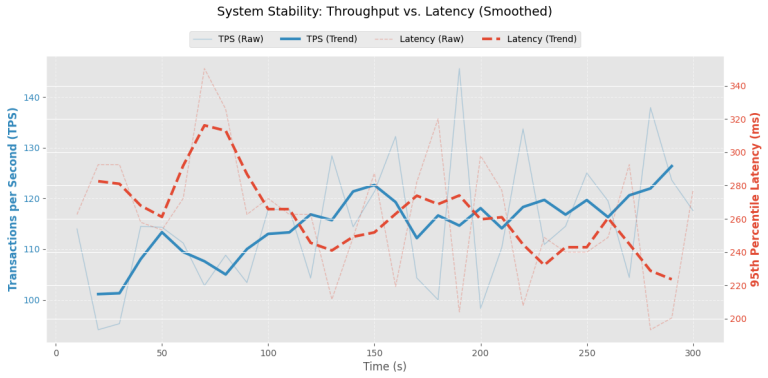

Figure 16: Time series of TPS vs. Latency (Smoothed) showing the inverse "saw-tooth" pattern.

3.5:1. This stability in the applied load confirms that the performance variance observed is intrinsic to the system's handling of the requests, rather than an artifact of the benchmark client's behavior.

3.4.4 Latency Distribution Analysis

Figure 19 highlights the severity of the latency variance. While the minimum latency confirms the hardware is capable of rapid response (6 ms), the 95th percentile (267 ms) and maximum (811 ms) show that a subset of requests experience severe delays. In an HPC context, this degree of jitter suggests that while the containerized database functions correctly, the underlying storage or scheduler configuration may require tuning to eliminate I/O wait states and smooth out performance.

4 Container-Level Monitoring

In addition to application-level benchmarking, the orchestrator provides infrastructure for **container-level resource monitoring** using Prometheus and cAdvisor. This monitoring capability enables observation of CPU, memory, network, and disk utilization at the container granularity, complementing the application-specific performance metrics collected by benchmark clients.

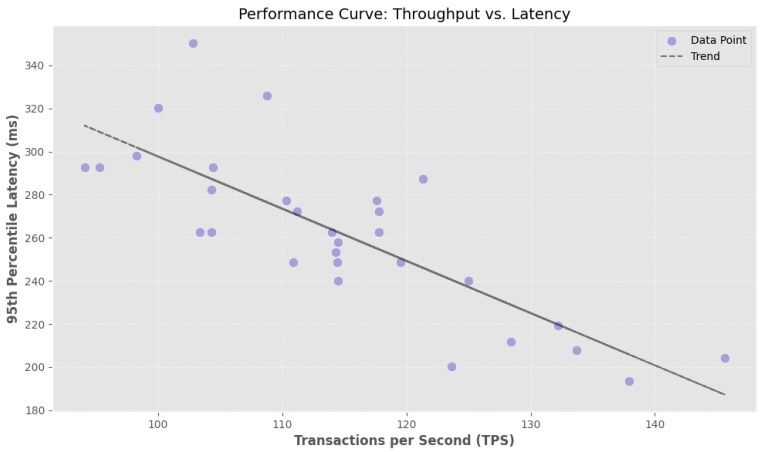

Figure 17: Correlation between Throughput and Latency, indicating stalling behavior.

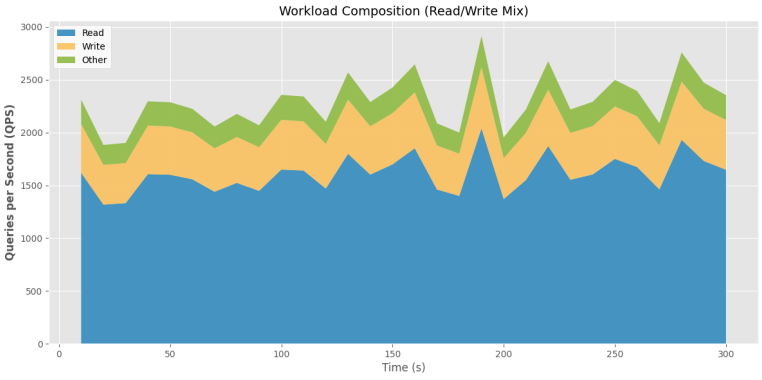

Figure 18: Stacked area chart of Query Per Second (QPS) breakdown, showing a stable Read/Write mix.

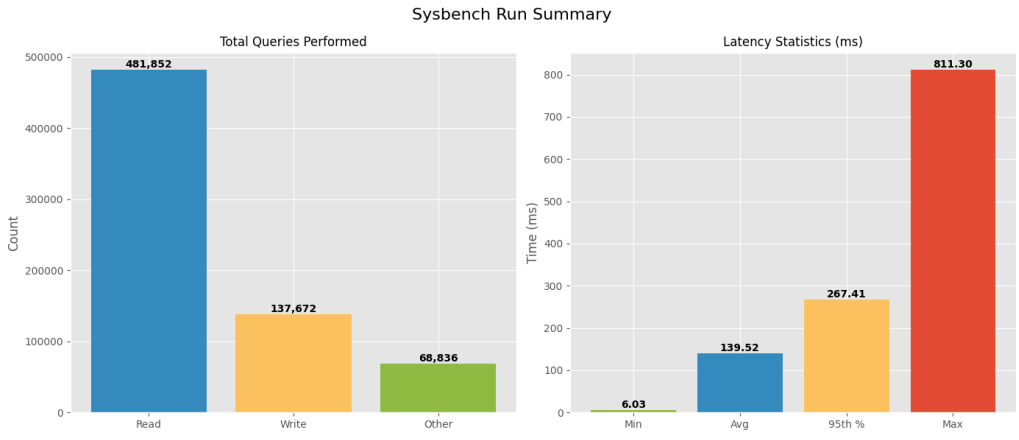

Figure 19: Summary statistics showing the disparity between Average and Maximum latency.

4.1 Monitoring Architecture

The monitoring system follows a three-tier architecture:

1. **cAdvisor**: A lightweight container monitoring agent deployed alongside each service container on HPC compute nodes. cAdvisor collects real-time resource usage statistics (CPU, memory, network I/O, filesystem I/O) by interfacing with the container runtime (Apptainer/Singularity) and exposes these metrics via an HTTP endpoint (`/metrics`) on port 8080.
2. **Prometheus**: A time-series database and monitoring service deployed on a dedicated CPU node. Prometheus periodically scrapes metrics from all registered cAdvisor endpoints (default: every 15 seconds), stores the time-series data, and provides a query interface (PromQL) for analysis and visualization.
3. **SSH Tunnel + Web UI**: Since HPC compute nodes are not directly accessible from external networks, the orchestrator facilitates SSH port forwarding from the user's local machine to the Prometheus instance. Users can then access the Prometheus web UI at `http://localhost:9090` to query metrics, visualize time-series data, and inspect container behavior during benchmarks.

Figure 20 illustrates the complete monitoring data flow across HPC nodes.

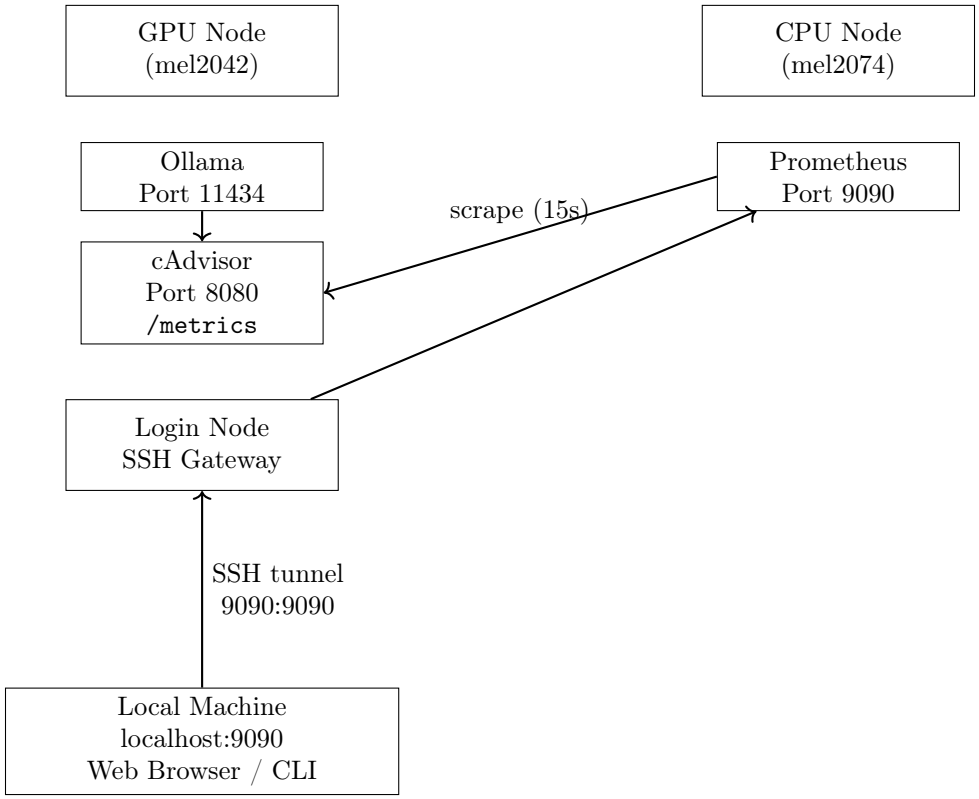

Figure 20: Monitoring architecture: cAdvisor collects container metrics on service nodes, Prometheus aggregates and stores time-series data, and users access the web UI via an SSH tunnel.

4.2 Usage Workflow

To enable monitoring for a benchmark session, users follow this workflow:

1. **Enable cAdvisor in service recipe**: Set `enable_cadvisor: true` in the service YAML recipe (e.g., `recipes/services/ollama_with_cadvisor.yaml`). The orchestrator automatically downloads and starts cAdvisor on the allocated compute node.
2. **Start the service**: Deploy the service using the standard CLI command:

   ```
   python main.py --recipe recipes/services/ollama_with_cadvisor.yaml
   ```

   Note the assigned service ID (e.g., `ollama_abc123`) and node hostname.
3. **Configure Prometheus**: Edit the Prometheus recipe (`recipes/services/prometheus_with_cadv` to include the service ID as a monitoring target. The orchestrator resolves service IDs to node hostnames and generates the Prometheus scrape configuration.
4. **Start Prometheus**: Deploy Prometheus on a CPU node:

   ```
   python main.py --recipe recipes/services/prometheus_with_cadvisor.yaml
   ```

   Note the Prometheus service ID.
5. **Create SSH tunnel**: Use the orchestrator to generate and establish an SSH tunnel:

   ```
   python main.py --create-tunnel <prometheus_service_id>
   ```

   This command outputs the SSH command, which the user executes to forward local port 9090 to the Prometheus instance.
6. **Access Prometheus UI**: Open a web browser and navigate to `http://localhost:9090`. The Prometheus web UI allows interactive querying of metrics via PromQL, inspection of active scrape targets, and time-series graph visualization.
7. **Start benchmark client**: With monitoring active, start the benchmark client to generate load on the service. cAdvisor will capture resource utilization throughout the benchmark run.

Alternatively, the orchestrator provides automated commands that execute steps 1–5 in a single invocation:

```
python main.py --start-session \
    recipes/services/ollama_with_cadvisor.yaml \
    recipes/clients/ollama_benchmark.yaml \
    recipes/services/prometheus_with_cadvisor.yaml
```

5 Conclusion

The design and implementation of the HPC AI Benchmarking Orchestrator successfully demonstrate a streamlined approach to deploying and evaluating containerized AI workloads on high-performance computing infrastructure. By decoupling the benchmark logic from the underlying SLURM scheduling and container runtime complexities, the system ensures reproducibility and ease of use—critical factors for analyzing scalable AI services.

The experimental results obtained on the MeluXina supercomputer highlight distinct performance characteristics across the evaluated domains:

- **Inference Workloads:** For the Ollama LLM service, performance was found to be highly sensitive to prompt and output token configurations. While the service maintained stability under concurrent loads, the analysis identified specific "sweet spots" for token generation throughput, emphasizing the need for workload-aware tuning rather than simple resource scaling.
- **Vector and Data Stores:** Both Chroma and Redis demonstrated that high throughput in an HPC environment is contingent on efficient client-side patterns. The dramatic performance gains observed with Redis pipelining (up to 20×) and Chroma batch insertion underscore that network latency and protocol overheads are significant bottlenecks that must be managed, even on high-speed interconnects.
- **System Stability:** The MySQL benchmarks revealed the challenges of running I/O-intensive transactional databases on shared HPC storage, as evidenced by the "saw-tooth" latency behavior and resource stalling. This reinforces the value of the integrated monitoring stack (cAdvisor and Prometheus), which allows operators to identify resource contention and storage I/O wait states that application logs alone might miss.

In summary, the project confirms that a modular, automated orchestration framework is essential for efficiently navigating the complex parameter spaces of modern AI applications. The tool not only facilitates performance profiling but also provides the infrastructure observability required to optimize AI services for the unique architectural constraints of supercomputing environments. Future work could extend the orchestrator to support multi-node distributed training benchmarks and more granular GPU profiling.