AI Factory Benchmarking Framework

Technical Report

Team 6 – EUMaster4HPC 2025-2026

Supervisor: Dr. Farouk Mansouri, LuxProvide
Institution: LuxProvide – MeluXina Supercomputer

January 12th, 2026

Abstract

The emergence of AI Factories in the European Union demands robust evaluation methodologies for AI infrastructure components. This report presents a comprehensive benchmarking framework designed to evaluate end-to-end performance for critical AI Factory components on the MeluXina supercomputer, one of Europe's most powerful supercomputers.

The AI Factory Benchmarking Framework provides unified benchmarking capabilities across diverse AI services. Supported services include Large Language Model (LLM) inference servers (Ollama, vLLM, Triton), vector databases (ChromaDB, Faiss, Weaviate), relational databases (PostgreSQL), object storage (S3-compatible MinIO), and file storage systems. The framework integrates production-grade monitoring using Prometheus and Grafana. It also supports SLURM-based HPC cluster orchestration and utilizes Apptainer containerization for reproducible deployments.

Key contributions include: (1) a modular Python-based architecture enabling extensible service benchmarking, (2) real-time monitoring with comprehensive metrics collection and visualization, (3) YAML-driven configuration for reproducible benchmark scenarios, (4) parallel execution support achieving up to 10× faster benchmarking, and (5) detailed reporting with statistical analysis and percentile calculations.

This framework equips researchers and practitioners with the tools necessary to design and operate future AI Factory workflows at scale, providing comparative insights, performance guidelines, and scalability assessments for AI infrastructure planning.

Keywords: High-Performance Computing, AI Benchmarking, LLM Inference, Vector Databases, Prometheus, Grafana, SLURM, Apptainer, GPU Acceleration, MeluXina

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and Motivation

The European Union is actively investing in AI Factories—high-performance computing facilities designed to accelerate artificial intelligence research, development, and deployment across member states. These AI Factories harness the power of next-generation HPC and AI systems to revolutionize data processing, analytics, and model deployment. As these facilities come online, there is a critical need for comprehensive benchmarking methodologies to evaluate their performance characteristics and optimize their configurations.

The MeluXina supercomputer, operated by LuxProvide in Luxembourg, serves as an ideal platform for developing and testing such benchmarking frameworks. With its heterogeneous architecture combining CPU and GPU resources, MeluXina provides the computational capabilities necessary to evaluate the diverse workloads typical of AI Factory operations.

1.2 Project Objectives

The primary objectives of this project are:

1. **Design a unified benchmarking framework** to evaluate end-to-end performance for critical AI Factory components
2. **Implement modular benchmark components** covering:
    - LLM inference servers (Ollama, vLLM, Triton)
    - Vector databases (ChromaDB, Faiss, Weaviate)
    - Relational databases (PostgreSQL)

- Object storage (S3-compatible systems)
- File storage (POSIX filesystem)

3. **Integrate production-grade monitoring** using Prometheus and Grafana
4. **Enable reproducible benchmarking** through YAML-based configuration and SLURM orchestration
5. **Provide comparative insights** with detailed performance metrics and statistical analysis

1.3 Challenge Overview

This project was conducted as part of the Student Challenge 2025-2026, spanning four months with the following phases:

Phase 1 – Onboarding: Introduction to MeluXina, exploration of tools and methodologies, design specification

Phase 2 – Prototyping: Development of applications, monitoring dashboards, and benchmarking scripts with iterative testing

Phase 3 – Evaluation: Deployment at realistic scales, performance measurements, resource profiling, and documentation

Phase 4 – Defense: Presentation of results and findings

1.4 Report Structure

This report is organized as follows:

- **Chapter 2:** Presents the system architecture and design decisions
- **Chapter 3:** Describes the core framework components in detail
- **Chapter 4:** Documents the supported services and their configurations
- **Chapter 5:** Explains the monitoring and visualization infrastructure
- **Chapter 6:** Details the HPC integration with SLURM and Apptainer
- **Chapter 7:** Covers implementation details and usage
- **Chapter 8:** Presents benchmark methodology and performance metrics

- **Chapter 9:** Summarizes conclusions and future work

Chapter 2

System Architecture

2.1 Architectural Overview

The AI Factory Benchmarking Framework follows a modular, layered architecture designed for extensibility, maintainability, and scalability. The system is composed of six core components orchestrated by a main controller, as illustrated in Figure 2.1.

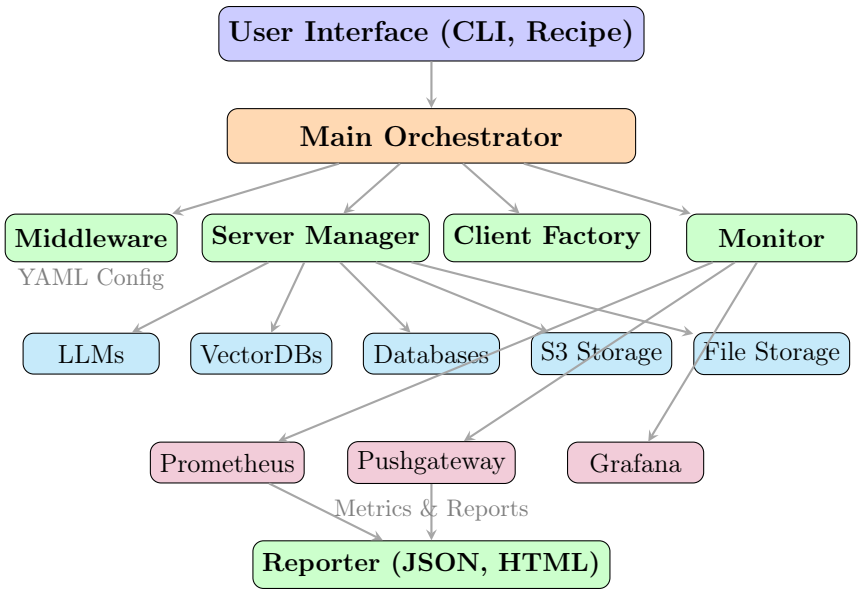

Figure 2.1: High-level system architecture of the AI Factory Benchmarking Framework showing the main orchestrator coordinating middleware, server management, client factory, and monitoring components.

2.2 Design Principles

The framework is built upon the following design principles:

Modularity: Each component is self-contained with well-defined interfaces, enabling independent development and testing

Extensibility: New service types can be added by implementing the appropriate client interface

Reproducibility: YAML-based configuration ensures benchmark scenarios can be exactly reproduced

Scalability: Parallel execution and SLURM integration support benchmarking at scale

Observability: Comprehensive metrics collection and real-time monitoring provide insights into system behavior

2.3 Data Flow

The benchmark execution follows a structured workflow:

1. **Configuration Loading:** The middleware interface loads and validates the recipe YAML configuration
2. **Service Initialization:** The server manager starts required services via SLURM and Apptainer
3. **Endpoint Discovery:** Service endpoints are retrieved from SLURM job outputs
4. **Benchmark Execution:** The client factory creates appropriate clients and executes benchmark operations
5. **Metrics Collection:** The monitor component records performance metrics in SQLite
6. **Report Generation:** The reporter generates comprehensive reports with statistical analysis
7. **Cleanup:** Services are stopped and resources are released

Chapter 3

Core Components

3.1 Main Orchestrator

The Main Orchestrator (`src/main.py`) serves as the central coordinator for the entire benchmarking workflow. It is responsible for:

- Initializing all framework components
- Managing the benchmark lifecycle
- Coordinating parallel execution of benchmarks
- Handling errors and cleanup procedures
- Generating final reports

3.1.1 Execution Modes

The orchestrator supports two execution modes:

Parallel Mode (Default): Uses Python's `ThreadPoolExecutor` to run benchmarks concurrently, achieving up to 10× faster execution

Sequential Mode: Executes benchmarks one at a time, useful for debugging and resource-constrained environments

```
# Parallel execution (default)
python src/main.py --recipe recipe.yml --max-workers 10

# Sequential execution
python src/main.py --recipe recipe.yml --sequential
```

Listing 3.1: Command-line execution options

3.2 Middleware Interface

The Middleware Interface (`src/middleware/interface.py`) provides the configuration management layer. It handles:

- Loading and parsing YAML recipe files
- Validating configuration parameters
- Providing service metadata to other components
- Managing benchmark identifiers

3.3 Server Manager

The Server Manager (`src/server/server.py`) manages the lifecycle of benchmark services running in Apptainer containers via SLURM. Key capabilities include:

- Automatic SLURM batch script generation
- Container image management (pulling from Docker registries)
- Service health checking
- Endpoint discovery and URL construction
- Graceful shutdown via SLURM job cancellation

3.3.1 SLURM Script Generation

The server manager dynamically generates SLURM submission scripts based on service configuration:

```
#!/bin/bash -l
#SBATCH --job-name=ollama-llama2
#SBATCH --time=01:00:00
#SBATCH --partition=gpu
#SBATCH --account=p200981
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --output=logs/ollama-llama2_%j.out

module add Apptainer
export COMPUTE_NODE=$(hostname -s)
export SERVICE_PORT=11434

# Save endpoint for discovery
```

```
echo "$COMPUTE_NODE:$SERVICE_PORT" > \
    $HOME/.cache/service_endpoints/ollama-llama2_endpoint.txt

# Run service in Apptainer container
apptainer exec --nv containers/ollama.sif ollama serve &
wait $!
```

Listing 3.2: Example generated SLURM script structure

3.4 Unified Client Factory

The Unified Client Factory (`src/client/unified_client.py`) creates specialized benchmark clients based on service type. It provides a unified interface while delegating to appropriate backend implementations:

- **BenchmarkClient:** HTTP-based clients for LLM inference (Ollama, vLLM, Triton)
- **PostgreSQLBenchmarkClient:** Database operations client
- **S3BenchmarkClient:** Object storage client
- **FileStorageBenchmarkClient:** POSIX filesystem client
- **VectorDBBenchmarkClient:** Vector database client (ChromaDB, Faiss, Weaviate)

3.5 Monitor

The Monitor component (`src/monitor/monitor.py`) collects and stores performance metrics in a SQLite database. Features include:

- Batch metric storage for efficiency
- Indexed queries by benchmark ID
- Raw data export to JSON
- Thread-safe operations for parallel benchmarking

The database schema captures:

- Timestamp
- Service name
- Client ID

- Request duration
- Success/failure status
- Status code
- Error messages
- Response size

3.6 Metrics Interceptor

The Metrics Interceptor (`src/interceptor/interceptor.py`) provides real-time metrics recording during benchmark execution. It captures:

- Request timing with microsecond precision
- Success/failure tracking
- Error classification
- Response size measurements

3.7 Reporter

The Reporter component (`src/reporter/reporter.py`) generates comprehensive reports from collected metrics. Report features include:

- Summary statistics (total, successful, failed requests)
- Timing analysis (average, median, min, max, standard deviation)
- Percentile calculations (P50, P90, P95, P99)
- Per-service breakdown
- Per-client analysis
- Throughput calculations (requests per second)

3.7.1 Report Format

Reports are generated in JSON format with the following structure:

```
{
  "benchmark_id": "benchmark_1234567890",
  "generated_at": "2025-01-04T12:00:00",
```

```
  "summary": {
    "total_requests": 3000,
    "successful_requests": 2950,
    "failed_requests": 50,
    "success_rate": 98.33,
    "avg_duration": 0.523,
    "median_duration": 0.501
  },
  "services": {
    "service-name": {
      "timing": {...},
      "percentiles": {"p50": 0.501, "p95": 0.812},
      "throughput": {"requests_per_second": 9.98}
    }
  }
}
```

Listing 3.3: Example report structure

Chapter 4

Supported Services

The AI Factory Benchmarking Framework supports comprehensive benchmarking across five major service categories, as summarized in Table 4.1.

Table 4.1: Supported services and their capabilities

Service Category	Backends	GPU Support	Protocol
LLM Inference	Ollama, vLLM, Triton	✓	HTTP/gRPC
Vector Databases	ChromaDB, Faiss, Weaviate	✓	HTTP/Native
Relational Databases	PostgreSQL	✓	psycopg2
Object Storage	MinIO (S3-compatible)	✓	S3 API
File Storage	POSIX Filesystem	✓	Direct I/O

4.1 LLM Inference Services

4.1.1 Ollama

Ollama provides a simple HTTP API for running large language models locally. The framework supports:

- Model selection (Llama2, Mistral, etc.)
- Configurable prompt generation
- Response streaming support

4.1.2 vLLM

vLLM is a high-performance inference engine optimized for large language models. Features include:

- OpenAI-compatible API
- PagedAttention for efficient memory management
- Continuous batching for high throughput

4.1.3 Triton Inference Server

NVIDIA Triton provides a platform for deploying AI models at scale:

- Multi-framework support (TensorRT, PyTorch, TensorFlow)
- Dynamic batching
- Model versioning

4.2 Vector Databases

4.2.1 ChromaDB

ChromaDB is an AI-native open-source embedding database:

- HTTP REST API
- Collection-based organization
- Configurable embedding dimensions

4.2.2 Faiss

Facebook AI Similarity Search (Faiss) provides efficient similarity search:

- In-memory operation (no server required)
- GPU-accelerated indexing
- Multiple index types (Flat, IVF, HNSW)

4.2.3 Weaviate

Weaviate is a cloud-native vector search engine:

- GraphQL and REST APIs
- Semantic search capabilities
- Hybrid search support

4.3 Relational Databases

4.3.1 PostgreSQL

PostgreSQL benchmarking supports standard database operations:

- SELECT, INSERT, UPDATE, DELETE operations
- Configurable operation mix
- Connection pooling support
- Batch operations

4.4 Storage Systems

4.4.1 Object Storage (MinIO)

MinIO provides S3-compatible object storage:

- PUT, GET, LIST, DELETE operations
- Configurable object sizes
- Bucket management
- Multi-part upload support

4.4.2 File Storage

POSIX filesystem benchmarking supports:

- Read, Write, Stat, Delete operations
- Configurable file sizes
- Direct I/O option
- Sync mode for durability testing

Chapter 5

Monitoring and Visualization

5.1 Monitoring Architecture

The framework integrates a production-grade monitoring stack based on Prometheus and Grafana, specifically designed for HPC environments. The architecture, shown in Figure 5.1, supports deployment on MeluXina compute nodes using Apptainer containers.

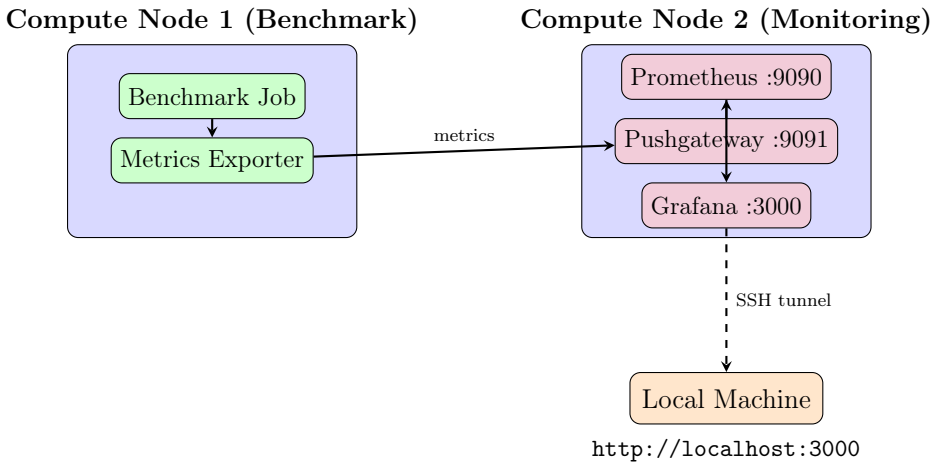

Figure 5.1: HPC monitoring architecture with SSH tunneling for visualization access

5.2 Prometheus Integration

Prometheus serves as the time-series database for metrics storage:

- Scrapes metrics from Pushgateway
- Stores time-series data for historical analysis
- Supports PromQL queries for data exploration

- Configurable retention and alerting

5.2.1 Key Metrics

The framework exports the following Prometheus metrics:

```
# Request rate (throughput)
rate(benchmark_requests_total[1m])

# Average latency
benchmark_request_duration_seconds_sum /
benchmark_request_duration_seconds_count

# Success rate
(benchmark_requests_successful /
benchmark_requests_total) * 100

# P95 latency
histogram_quantile(0.95,
  rate(benchmark_request_duration_seconds_bucket[5m]))
```

Listing 5.1: Prometheus metric queries

5.3 Pushgateway

The Pushgateway serves as an intermediary for batch and ephemeral jobs to push metrics to Prometheus. This is essential for HPC environments where benchmark jobs have limited lifetimes:

```
python src/monitoring/prometheus_exporter.py benchmark_id \
  --push \
  --gateway-url http://mel2110:9091 \
  --job ai_factory_benchmark
```

Listing 5.2: Pushing metrics to Pushgateway

5.4 Grafana Dashboards

Grafana provides visualization and analytics capabilities. The pre-configured dashboard includes:

- **Request Rate (Throughput):** Time-series graph of requests per second
- **Success Rate Gauge:** Visual indicator with configurable thresholds

- **Response Time:** Latency graph with percentile overlays
- **Total Requests Counter:** Stat panel for overall volume
- **Failed Requests Counter:** Error tracking
- **Requests by Service:** Pie chart for workload distribution
- **Success vs Failure:** Stacked bar chart

5.5 Accessing Monitoring from Local Machine

Since compute nodes are not directly accessible from external networks, SSH tunneling is used to access Grafana:

```
# Create SSH tunnel to compute node
ssh -L 3000:mel2345:3000 \
    -L 9090:mel2345:9090 \
    -L 9091:mel2345:9091 \
    <username>@login.lxp.lu -p 8822

# Access Grafana at http://localhost:3000
# Login: admin / admin
```

Listing 5.3: SSH tunnel setup for monitoring access

Chapter 6

HPC Integration

6.1 MeluXina Supercomputer

MeluXina is Luxembourg's petascale supercomputer, operated by LuxProvide. It provides:

- Heterogeneous architecture with CPU and GPU nodes
- SLURM workload manager for job scheduling
- High-speed interconnect for distributed computing
- Shared filesystems for data management

6.2 SLURM Integration

The framework provides deep integration with SLURM for job management:

6.2.1 Configuration Parameters

```
slurm:
  partition: gpu          # or 'cpu'
  account: p200981        # Project account
  time: "01:00:00"        # Maximum runtime
  nodes: 1                # Number of nodes
  ntasks: 1               # Tasks per job
  qos: default            # Quality of service
  gres: gpu:1             # GPU resources
```

Listing 6.1: SLURM configuration in recipe.yml

6.2.2 Job Lifecycle

1. **Script Generation:** Dynamic SLURM scripts based on service configuration
2. **Job Submission:** Automatic `sbatch` submission
3. **Status Monitoring:** `squeue` integration for job tracking
4. **Endpoint Discovery:** Reading node hostname from job output
5. **Job Cancellation:** `scancel` for cleanup

6.3 Apptainer Containerization

Apptainer (formerly Singularity) provides containerization support for HPC environments:

6.3.1 Container Management

```
# Pull containers from Docker registry
apptainer pull ollama.sif docker://ollama/ollama
apptainer pull vllm.sif docker://vllm/vllm-openai:latest
apptainer pull chromadb.sif docker://chromadb/chroma:latest

# Run with GPU support
apptainer exec --nv chromadb.sif chroma run --port 8000
```

Listing 6.2: Container operations

6.3.2 Bind Mounts for Persistent Data

Different services require specific bind mounts for data persistence:

- **ChromaDB:** `-bind $DATA_DIR:/chroma/data`
- **PostgreSQL:** `-bind $PGDATA:/var/lib/postgresql/data`
- **MinIO:** `-bind $MINIO_DATA:/data`
- **Weaviate:** `-bind $WEAVIATE_DATA:/var/lib/weaviate`

6.4 GPU Acceleration

All services support GPU acceleration through NVIDIA containers:

- Use `-nv` flag with Apptainer for GPU passthrough

- Request GPU resources via SLURM `gres` parameter
- LLM inference benefits significantly from GPU acceleration
- Vector databases can use GPU for accelerated similarity search

Chapter 7

Implementation Details

7.1 Technology Stack

The framework is built using the following technologies:

Table 7.1: Technology stack

Category	**Technologies**
Programming Language	Python 3.12
Orchestration	SLURM, Python ThreadPoolExecutor
Containerization	Apptainer (Singularity)
Monitoring	Prometheus, Pushgateway, Grafana
Data Storage	SQLite (metrics), JSON (reports)
GPU Support	NVIDIA CUDA via Apptainer `-nv`
Configuration	YAML (PyYAML)
HTTP Client	requests
CLI	rich, argparse

7.2 Dependencies

Core Python dependencies are managed via `requirements.txt`:

```
PyYAML>=6.0            # Configuration
requests>=2.31.0       # HTTP client
numpy>=1.24.0          # Numerical operations
pandas>=2.0.0          # Data analysis
psycopg2-binary>=2.9.0  # PostgreSQL
boto3>=1.28.0          # S3 client
chromadb>=0.4.0        # ChromaDB client
faiss-cpu>=1.7.4       # Faiss
weaviate-client>=3.24.0  # Weaviate
```

```
prometheus-client>=0.18.0  # Metrics export
rich>=13.0.0           # CLI output
matplotlib>=3.7.0      # Visualization
```

Listing 7.1: Core dependencies

7.3 Recipe Configuration System

The YAML-based recipe system enables reproducible benchmark configurations:

```
benchmark:
  name: "AI Factory Comprehensive Benchmark"
  description: "Full-scale benchmarking of AI components"
  duration: 3600

global:
  log_level: INFO
  metrics_db: metrics.db
  reports_dir: reports
  pushgateway_url: http://localhost:9091

services:
  # LLM Inference
  - service_name: ollama-llama2
    service_type: ollama
    container_image: docker://ollama/ollama
    port: 11434
    client_count: 5
    requests_per_second: 10
    duration: 300
    model: llama2
    slurm:
      partition: gpu
      time: "01:00:00"
      gres: gpu:1

  # Vector Database
  - service_name: chromadb
    service_type: vectordb
    backend: chromadb
    port: 8000
    dimension: 384
    operation_mix:
      insert: 0.3
      search: 0.5
      update: 0.1
```

```
      delete: 0.1
```

Listing 7.2: Comprehensive recipe example

7.4 Installation and Setup

7.4.1 HPC Installation (MeluXina)

```
# Connect to MeluXina
ssh <username>@login.lxp.lu -p 8822

# Navigate to project directory
cd $PROJECT/p200981

# Load required modules
module purge
module add Apptainer
module load Python/3.12.3-GCCcore-13.3.0

# Clone repository
git clone https://github.com/Valegrl/Team6_EUMASTER4HPC2526.git
cd Team6_EUMASTER4HPC2526

# Create virtual environment
python -m venv venv
source venv/bin/activate
pip install -r requirements.txt

# Pull service containers
./scripts/pull_containers.sh
```

Listing 7.3: Installation steps for MeluXina

7.5 Usage Examples

7.5.1 Running Benchmarks

```
# Basic execution
python src/main.py --recipe recipe.yml

# With Pushgateway for monitoring
python src/main.py --recipe recipe.yml \
  --pushgateway http://mel2110:9091

# Parallel execution with custom workers
```

```
python src/main.py --recipe recipe.yml --max-workers 10

# Sequential execution for debugging
python src/main.py --recipe recipe.yml --sequential
```

Listing 7.4: Benchmark execution examples

7.5.2 SLURM Batch Execution

```
# Submit benchmark job
sbatch scripts/run_benchmark.sh

# Monitor job status
squeue -u $USER

# View job output
tail -f logs/benchmark_JOBID.out
```

Listing 7.5: SLURM job submission

7.5.3 Interactive Session

```
# Request interactive session
salloc --partition=gpu --account=p200981 \
       --time=01:00:00 --nodes=1

# Load modules and run
module add Apptainer
module load Python/3.12.3-GCCcore-13.3.0
source venv/bin/activate
python src/main.py --recipe recipe.yml
```

Listing 7.6: Interactive SLURM session

Chapter 8

Benchmark Methodology and Results

8.1 Benchmark Methodology

The framework follows established benchmarking best practices:

8.1.1 Warm-up Phase

Each benchmark begins with a warm-up period to:

- Initialize service caches
- Establish connection pools
- Load models into GPU memory
- Stabilize system performance

8.1.2 Statistical Rigor

To ensure meaningful results:

- Multiple runs (3-5) are recommended for averaging
- Sufficiently long durations (300-600 seconds) capture variance
- Control variables by changing one parameter at a time
- Report confidence intervals when appropriate

8.2 Performance Metrics

8.2.1 Latency Metrics

- **Average Duration:** Mean request processing time
- **Median Duration (P50):** 50th percentile latency
- **P90/P95/P99:** Tail latency percentiles
- **Min/Max:** Extreme values for outlier detection
- **Standard Deviation:** Latency consistency measure

8.2.2 Throughput Metrics

- **Requests per Second:** Overall throughput rate
- **Total Requests:** Volume of operations completed
- **Duration:** Total benchmark runtime

8.2.3 Reliability Metrics

- **Success Rate:** Percentage of successful requests
- **Failed Requests:** Count of failed operations
- **Error Classification:** Types of failures encountered

8.3 Result Analysis

8.3.1 Querying Metrics Database

```
-- Summary statistics by service
SELECT
    service_name,
    COUNT(*) as total,
    SUM(CASE WHEN success = 1 THEN 1 ELSE 0 END) as successful,
    AVG(request_duration) as avg_duration,
    MIN(request_duration) as min_duration,
    MAX(request_duration) as max_duration
FROM metrics
WHERE benchmark_id = 'benchmark_1234567890'
GROUP BY service_name;

-- Percentile calculation
```

```
SELECT
    service_name,
    request_duration
FROM metrics
WHERE success = 1
ORDER BY request_duration;
```

Listing 8.1: SQLite query examples

8.3.2 Comparing Benchmarks

```
# Compare multiple runs
for report in reports/benchmark_*_report.json; do
    echo "=== $(basename $report) ==="
    jq '.summary.success_rate, .timing.avg_duration' $report
done
```

Listing 8.2: Comparing multiple benchmark runs

8.4 Expected Performance Characteristics

Based on the architecture and typical HPC deployments, expected performance characteristics include:

Table 8.1: Expected performance ranges by service type

Service Type	**Latency (P50)**	**Throughput**	**GPU Impact**
LLM Inference (Ollama)	100-500ms	10-50 req/s	5-10×
LLM Inference (vLLM)	50-200ms	20-100 req/s	10-20×
Vector DB (ChromaDB)	5-50ms	100-500 req/s	2-5×
Vector DB (Faiss)	1-10ms	1000+ req/s	5-10×
PostgreSQL	1-10ms	500-2000 req/s	Minimal
S3 Storage	10-100ms	100-500 req/s	N/A
File Storage	0.1-10ms	1000+ req/s	N/A

Chapter 9

Conclusion and Future Work

9.1 Summary of Achievements

This project has successfully delivered a comprehensive AI Factory Benchmarking Framework with the following key achievements:

1. **Unified Framework:** A single, cohesive framework supporting 10+ AI infrastructure services
2. **Production-Grade Monitoring:** Integration with Prometheus and Grafana for real-time visualization
3. **HPC Integration:** Seamless SLURM orchestration and Apptainer containerization
4. **Reproducible Benchmarks:** YAML-driven configuration for exact reproduction of experiments
5. **GPU Acceleration:** Support for NVIDIA GPUs across all service types
6. **Parallel Execution:** ThreadPoolExecutor enables up to 10× faster benchmarking
7. **Comprehensive Reporting:** Detailed statistical analysis with percentile calculations

9.2 Lessons Learned

Key insights gained during development:

- **Container Compatibility:** Apptainer provides excellent Docker compatibility for HPC environments

- **Endpoint Discovery:** Dynamic service endpoint discovery is essential for SLURM-based deployments
- **Monitoring Architecture:** Compute node-based monitoring eliminates SSH tunneling complexity for metrics collection
- **Concurrency Benefits:** Thread-based parallelism proved effective for I/O-bound benchmark workloads
- **Configuration Management:** YAML-based recipes greatly simplify reproducibility

9.3 Future Work

Several enhancements are planned for future development:

Multi-Node Benchmarking: Extend support for distributed benchmarks across multiple compute nodes

Advanced Triton Models: Add pre-configured model repositories for Triton Inference Server

Scaling Analysis: Implement automated scaling studies with varying client counts and node configurations

EU AI Factory Integration: Align with emerging European AI Factory standards and requirements

Machine Learning Workloads: Add benchmarks for training workloads alongside inference

Network Analysis: Include network bandwidth and latency measurements between services

Cost Modeling: Integrate resource utilization with cost estimation for cloud deployments

9.4 Acknowledgments

We thank Dr. Farouk Mansouri of LuxProvide for his supervision and guidance throughout this project. We also acknowledge LuxProvide for providing access to the MeluXina supercomputer and the EUMASTER4HPC program for the opportunity to work on this challenge.

Appendix A

Apptainer Container Recipes

The framework includes Apptainer definition files for building custom containers when needed:

- `chromadb.def` – ChromaDB vector database
- `minio.def` – MinIO S3-compatible storage
- `ollama.def` – Ollama LLM inference
- `postgres.def` – PostgreSQL database
- `triton.def` – NVIDIA Triton Inference Server
- `vllm.def` – vLLM high-performance inference
- `weaviate.def` – Weaviate vector search

Appendix B

Grafana Dashboard Configuration

The pre-configured Grafana dashboard is available at:

`monitoring/grafana/dashboards/ai-factory-benchmark.json`

Dashboard panels include:

- Request Rate (RPS) time series
- Success Rate gauge
- Response Time graph with percentiles
- Total Requests stat
- Failed Requests stat
- Requests by Service pie chart
- Success vs Failure stacked bar

Appendix C

Prometheus Configuration

Prometheus configuration is located at:

`monitoring/prometheus/prometheus.yml`

The configuration includes:

- Scrape configurations for Pushgateway
- Alerting rules in `alerts.yml`
- Retention settings

Appendix D

Troubleshooting Guide

D.1 Common Issues

D.1.1 Docker Not Found on HPC

Issue: "docker: command not found" on MeluXina

Solution: Docker is not available on HPC systems. Use Apptainer instead:

```
module add Apptainer
apptainer run container.sif
```

D.1.2 Job Stays in PENDING

Solution:

```
# Check reason
squeue -u $USER --start

# Check partition availability
sinfo -p gpu
```

D.1.3 Service Connection Failed

Solution:

```
# Check service logs
grep -i error logs/benchmark_*.log

# Verify container is running
singularity instance list
```

D.1.4 Out of Memory

Solution:

```
# Request more memory
sbatch --mem=64G scripts/run_benchmark.sh
```

Bibliography

[1] LuxProvide. MeluXina Documentation. https://docs.lxp.lu/

[2] Prometheus Authors. Prometheus - Monitoring system & time series database. https://prometheus.io/

[3] Grafana Labs. Grafana - The open observability platform. https://grafana.com/

[4] SchedMD. Slurm Workload Manager. https://slurm.schedmd.com/

[5] Apptainer Community. Apptainer Documentation. https://apptainer.org/docs/

[6] Ollama. Ollama - Get up and running with large language models locally. https://ollama.ai/

[7] vLLM Team. vLLM: Easy, Fast, and Cheap LLM Serving. https://vllm.ai/

[8] NVIDIA. NVIDIA Triton Inference Server. https://developer.nvidia.com/nvidia-triton-inference-server

[9] Chroma. ChromaDB - AI-native open-source embedding database. https://www.trychroma.com/