Benchmarking AI Factories on MeluXina Supercomputer

A Unified Platform for Deploying, Monitoring, and Benchmarking AI Factories

EUMaster4HPC Student Challenge 2025

Tommaso Crippa
Edoardo Leali
Emanuele Caruso

12 January 2026

Abstract

This report presents our implementation for the EUMaster4HPC Student Challenge 2025, where we were tasked to build a unified framework designed to streamline the deployment, monitoring, and benchmarking of AI inference services on High-Performance Computing clusters. The framework addresses the challenges of managing containerized AI workloads in SLURM-based environments. AI-Factories provides three integrated modules (Server, Monitor, and Client) that work together through an automatic service discovery mechanism. The framework enables researchers and engineers to deploy inference servers, monitor performance metrics via Prometheus, and execute standardized benchmarks with minimal manual configuration. This report details the system architecture, design decisions, implementation specifics, and provides guidance on deploying the framework on the MeluXina supercomputer.

Contents

1 Introduction

1.1 The Rise of AI Factories

The artificial intelligence landscape is undergoing a fundamental transformation with the emergence of AI Factories, large-scale inference platforms designed to serve AI models at production scale. Unlike traditional research-focused deployments, AI Factories represent the industrialization of AI, where models must deliver consistent, low-latency responses to thousands of concurrent users while maintaining cost efficiency and reliability.

This shift from experimentation to production has been driven by multiple interconnected factors. The explosion of **foundation models**, including large language models, vision models, and multimodal systems, has created unprecedented demand for specialized inference infrastructure capable of handling production workloads. Organizations across industries are moving beyond proof-of-concept deployments and adopting AI as **mission-critical technology**, requiring robust infrastructure with enterprise-grade reliability. The **scale requirements** have grown dramatically, with single inference endpoints now expected to serve millions of requests daily. At the same time, there is mounting pressure to optimize **GPU utilization** and reduce inference costs, as infrastructure budgets become increasingly constrained. These factors combined have transformed AI from a research curiosity into a production engineering problem.

1.2 Why HPC Clusters Need Better Benchmarking

HPC clusters are a perfect fit for AI Factories, but they are not used enough. Cloud providers dominate AI inference today, yet HPC systems have clear benefits. **GPU accelerators** such as NVIDIA A100 and H100 provide state-of-the-art performance with fast interconnections between nodes. **University infrastructure** eliminates the hourly charges typical of cloud services, making long-term deployments more economical. Inference workloads can run close to training data and other research activities, reducing data movement and latency. Most importantly, **SLURM** provides precise control over CPU, memory, and GPU allocation, allowing researchers to optimize resource usage for their specific needs.

However, using HPC for AI Factories creates problems that cloud platforms do not have. **SLURM complexity** arises from running inference services as batch jobs instead of traditional always-on containerized deployments. **Dynamic allocation** means services are assigned to different compute nodes each time a job is submitted. There is no built-in DNS or load balancing mechanism for dynamically allocated services, forcing users to manually track endpoints. **Monitoring tools** like Prometheus were designed for static cloud environments and do not naturally handle ephemeral HPC jobs. Additionally, **Apptainer** security models differ fundamentally from Docker and Kubernetes paradigms, requiring different approaches to container management and resource isolation.

Current benchmarking tools do not work well for HPC. They have three fundamental problems. First, **cloud-centric tooling** such as Locust, K6, and wrk expect fixed HTTP endpoints and do not integrate with SLURM schedulers. These tools cannot handle the dynamic nature of HPC job allocation or coordinate monitoring across temporary compute nodes that are created and destroyed with each job submission. Second, the current approach requires **manual setup processes** including copying job IDs, extracting node names, and configuring Prometheus targets. These error-prone steps introduce variability and make results unreliable, causing a **reproducibility crisis** where most published benchmarks cannot be repeated by other researchers due to undocumented configuration details. Third, **incomplete metrics** focus exclusively on latency and throughput but ignore HPC-specific concerns such as GPU cache utilization, memory bandwidth saturation, scheduler overhead, and multi-tenant interference from other users. Without these specialized metrics, researchers cannot effectively optimize inference workloads for HPC characteristics and cannot understand how their systems perform under realistic production conditions.

HPC-based AI Factories need a unified framework that simplifies SLURM integration while preserving access to important HPC features. The framework must provide automatic service discovery in dynamic environments where node assignments change with each deployment. It should collect HPC-specific monitoring metrics beyond standard cloud benchmarks. The system must make benchmarks reproducible through declarative configuration files that can be version-controlled and shared among researchers. Finally, it needs to bridge the gap between cloud AI tools and HPC infrastructure, bringing modern AI development practices to supercomputers while respecting the unique constraints and opportunities that HPC environments offer.

1.3 Project Goals

AI-Factories was developed to meet several key objectives. The framework provides a **unified command-line interface** that handles server deployment, monitoring, and benchmarking through a single consistent tool. It **eliminates manual endpoint management** through automatic service discovery, removing the need to manually track and configure service addresses. The system enables **recipe-based configuration** that makes deployments reproducible and shareable across teams. It fully **supports GPU-accelerated inference servers** on HPC clusters, taking advantage of supercomputer hardware for AI workloads. **Prometheus monitoring is integrated** with automatic target resolution, so that metric collection is set up without manual configuration. Finally, the framework **facilitates performance benchmarking** with configurable workload patterns, allowing researchers to test their systems under diverse conditions.

1.4 Target Platform: MeluXina

MeluXina is Luxembourg's national supercomputer, featuring AMD EPYC processors and NVIDIA A100 GPUs. The framework is specifically designed to work with the **SLURM workload manager** for job scheduling, which is the standard scheduler on HPC systems. It uses **Apptainer** (formerly Singularity) for secure containerization of inference workloads. The system integrates with the **module system** for managing software environments across different compute nodes. Finally, it takes advantage of **shared scratch storage** available on supercomputers for storing container images and benchmark data.

2 System Architecture

The AI-Factories framework follows a modular architecture with three core modules and a central service discovery mechanism. Figure 1 illustrates the high-level system design.

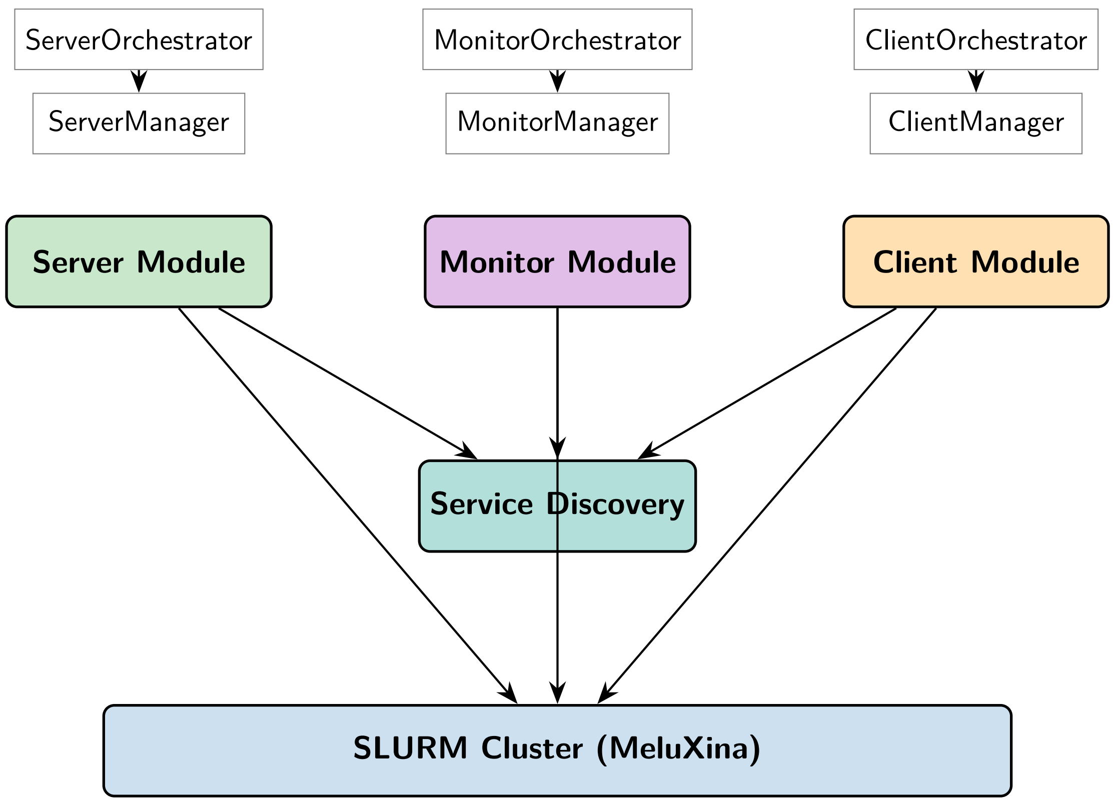

Figure 1: AI-Factories High-Level Architecture

2.1 Architectural Principles

The framework is built on several key architectural principles that guide its design and implementation. **Separation of Concerns** ensures that each module handles a specific domain (deployment, monitoring, benchmarking) with clear interfaces, making the codebase maintainable and easier to extend. **Recipe-Based Configuration** means that all operations are driven by YAML recipes rather than imperative code, enabling reproducibility and version control of deployments. The **Manager-Orchestrator Pattern** is used throughout the framework, where each module has a Manager class that handles business logic and an Orchestrator class that interacts directly with SLURM, providing clean separation between domain logic and infrastructure details. **Automatic Discovery** allows services to register their endpoints automatically as they deploy, eliminating manual configuration propagation and reducing operational overhead.

2.2 Directory Structure

The framework organizes code and configuration as follows:

```
AI-Factories/
|-- src/
|   |-- server/          # Server deployment module
|   |-- monitor/         # Prometheus monitoring module
|   |-- client/          # Benchmark client module
|   |-- discover.py      # Service discovery mechanism
|   |-- list_services.py # CLI for listing services
|   +-- clear_services.py
|-- recipes/
```

```
|    |-- servers/          # Server deployment recipes
|    |-- monitors/         # Monitor configuration recipes
|    +-- clients/          # Benchmark workload recipes
|-- config/
|    +-- slurm.yml         # SLURM configuration defaults
|-- logs/                  # Runtime logs and state
+-- results/               # Benchmark output files
```

Listing 1: Project Directory Structure

3 Service Discovery Mechanism

One of the key innovations in AI-Factories is the automatic service discovery system. This mechanism solves the fundamental challenge of dynamic endpoint management in SLURM environments.

3.1 The Discovery Problem

When submitting jobs to SLURM, the scheduler assigns compute nodes dynamically. This creates a challenge: how do clients and monitors find the server they need to connect to? Traditional approaches require:

1. Waiting for job allocation
2. Querying SLURM for the assigned node
3. Manually copying the endpoint to other configurations

3.2 Solution: File-Based Service Registry

AI-Factories implements a lightweight, file-based service registry stored in the user's home directory:

```
~/.ubenchai/discover/
|-- vllm.json
|-- ollama.json
+-- chromadb.json
```

Each service file contains the connection information:

```json
{
  "job_id": "3757043",
  "recipe_name": "vllm-server",
  "instance_id": "a1b2c3d4-...",
  "node": "mel2013",
  "ports": [8000]
}
```

Listing 2: Discovery File Format

3.3 Discovery Workflow

Figure 2 shows the discovery workflow in action.

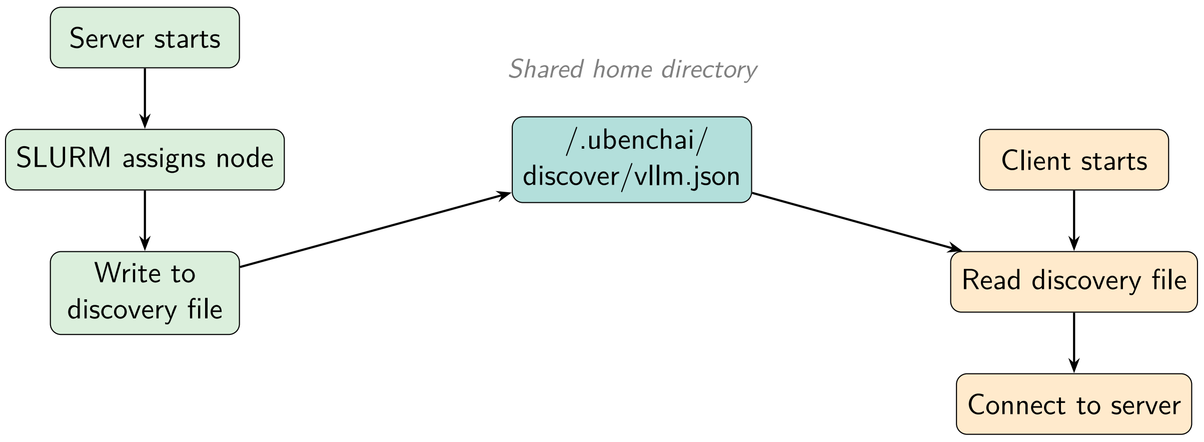

Figure 2: Service Discovery Workflow

3.4 Service Name Linking

All recipes for the same service share a common `service_name` field:

```yaml
# recipes/servers/vllm-server.yaml
name: vllm-server
service_name: vllm  # Discovery key

# recipes/monitors/vllm-monitor.yml
name: vllm-monitor
service_name: vllm  # Same key

# recipes/clients/vllm-simple-test.yaml
name: vllm-simple-test
service_name: vllm  # Same key
```

Listing 3: Service Name Consistency Across Recipes

This design allows the client and monitor to automatically find the server without any manual endpoint configuration.

4 Server Module

The Server Module is responsible for deploying AI inference services as SLURM jobs using containerized applications.

4.1 Module Architecture

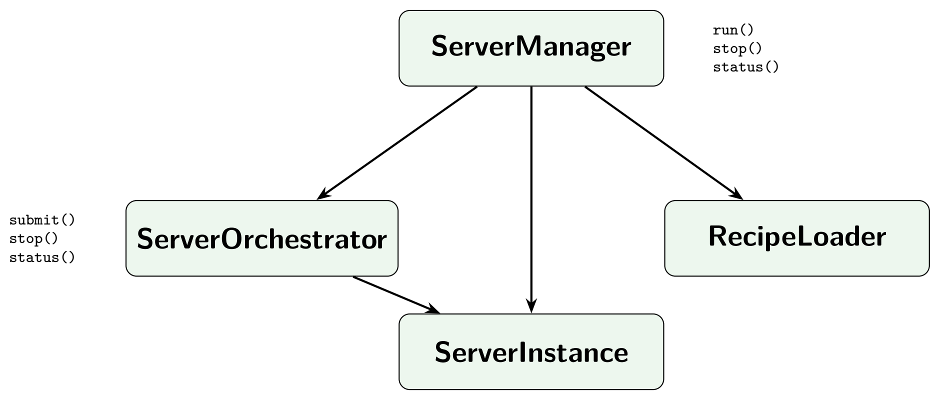

Figure 3: Server Module Class Structure

4.1.1 ServerManager

The `ServerManager` class serves as the primary interface for server operations. It coordinates recipe loading, job submission, and instance lifecycle management.

Key responsibilities:

- Loading and validating server recipes
- Creating `ServerInstance` objects for each deployment
- Coordinating with the orchestrator for SLURM submissions
- Waiting for node assignment and updating discovery information
- Managing the collection of active server instances

4.1.2 ServerOrchestrator

The `ServerOrchestrator` handles direct interaction with SLURM:

- Building SBATCH scripts from recipe specifications
- Submitting jobs using the `sbatch` command
- Querying job status with `squeue`
- Canceling jobs with `scancel`

The orchestrator generates batch scripts dynamically based on recipe resources:

```
#!/bin/bash -l

#SBATCH --job-name=vllm-server_a1b2c3d4
#SBATCH --time=04:00:00
#SBATCH --partition=gpu
#SBATCH --account=p200981
#SBATCH --nodes=1
#SBATCH --cpus-per-task=8
#SBATCH --mem=64G
```

```
#SBATCH --gres=gpu:1

# Recipe-defined startup commands
module load Apptainer/1.3.6-GCCcore-13.3.0
apptainer exec $IMAGE python -m vllm.entrypoints.openai.api_server ...
```

Listing 4: Generated SBATCH Script Structure

4.1.3 ServerInstance

Each deployment creates a `ServerInstance` object tracking:

- Unique instance identifier
- SLURM job ID (orchestrator handle)
- Current status (SUBMITTED, STARTING, RUNNING, etc.)
- Assigned node and ports
- Recipe reference

4.2 Server Recipe Specification

Server recipes define all aspects of a deployment:

```
name: vllm-server
service_name: vllm
description: vLLM inference server for MeluXina

service:
  command: |
    #!/bin/bash
    module load Apptainer/1.3.6-GCCcore-13.3.0

    IMAGE=$CONTAINER_DIR/vllm-openai_v0.5.4.sif

    apptainer exec --nv $IMAGE \
      python -m vllm.entrypoints.openai.api_server \
        --model facebook/opt-125m \
        --host 0.0.0.0 \
        --port 8000

  working_dir: .
  ports:
    - 8000

orchestration:
  resources:
    cpu_cores: 8
    memory_gb: 64
    gpu_count: 1
    partition: gpu
    time_limit: "04:00:00"
```

Listing 5: vLLM Server Recipe Example

4.3 Deployment Workflow

The server deployment follows this sequence:

1. User executes: `python -m src.server run -recipe vllm-server`
2. `ServerManager` loads the recipe from `recipes/servers/`
3. A new `ServerInstance` is created with a unique ID
4. `ServerOrchestrator` generates and submits the SBATCH script
5. Manager polls SLURM until the job is running and node is assigned
6. Discovery information is written to `~/.ubenchai/discover/vllm.json`
7. Instance is registered and status is returned to user

5 Monitor Module

The Monitor Module deploys Prometheus monitoring stacks to collect metrics from running AI services.

5.1 Module Architecture

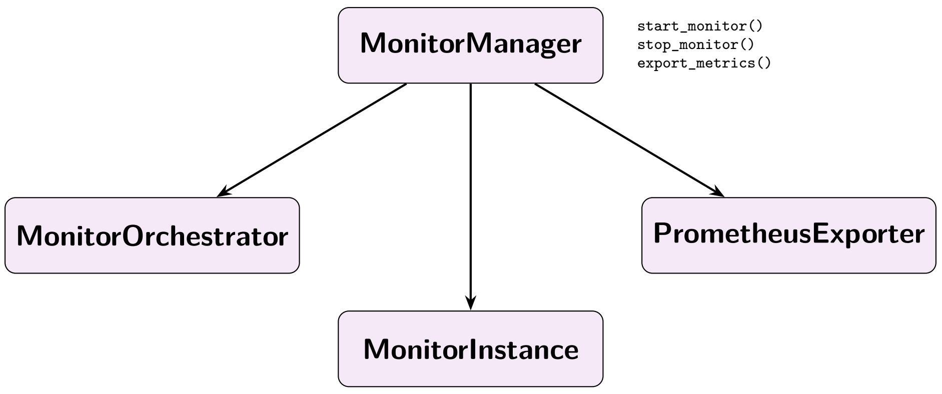

Figure 4: Monitor Module Class Structure

5.1.1 MonitorManager

The `MonitorManager` orchestrates the complete monitoring lifecycle:

- Loading monitor recipes
- Resolving target endpoints through service discovery
- Generating Prometheus configuration files
- Deploying Prometheus as a SLURM job
- Managing monitor state persistence
- Coordinating metrics export

5.1.2 MonitorOrchestrator

The `MonitorOrchestrator` handles Prometheus deployment:

- Building SBATCH scripts for Prometheus containers
- Managing Apptainer container lifecycle
- Generating `prometheus.yml` configuration
- Setting up bind mounts for configuration and data directories

5.1.3 PrometheusExporter

The `PrometheusExporter` class provides metrics export functionality:

- Querying Prometheus HTTP API for instant and range queries
- Service-specific metric selection (e.g., vLLM metrics)
- Export to JSON and CSV formats
- Common system metrics (CPU, memory, HTTP)

5.2 Target Resolution

The monitor module automatically resolves targets using service discovery:

```python
def _resolve_targets(self, recipe, targets, job_ids):
    resolved = {}

    # Use service discovery
    service_name = recipe.service_name
    discover_info = read_discover_info(service_name)

    node = discover_info.get("node")
    ports = discover_info.get("ports", [])

    for target in targets:
        port = target.port
        resolved[target.name] = f"{node}:{port}"

    return resolved
```

Listing 6: Target Resolution Logic

5.3 Prometheus Configuration Generation

The orchestrator dynamically generates Prometheus configuration:

```yaml
global:
  scrape_interval: 15s
  evaluation_interval: 15s

scrape_configs:
  - job_name: vllm-metrics
    static_configs:
      - targets: ["mel2013:8000"]
    metrics_path: /metrics
    scrape_interval: 15s
```

Listing 7: Generated prometheus.yml

5.4 Metrics Export

The framework supports exporting collected metrics to portable formats:

```
# Export instant metrics snapshot
python -m src.monitor export --id abc123 --format json

# Export time-range metrics to CSV
python -m src.monitor export --id abc123 --format csv --type range \
    --start "2025-12-02T10:00:00Z" --end "2025-12-02T11:00:00Z"
```

Listing 8: Metrics Export Commands

Service-specific metrics are automatically selected based on the monitored service:

Table 1: vLLM-Specific Prometheus Metrics

Metric	**Description**
`vllm_requests_total`	Total number of requests
`vllm_request_duration_seconds`	Request duration
`vllm_time_to_first_token_seconds`	Time to first token
`vllm_num_requests_running`	Running requests
`vllm_gpu_cache_usage_perc`	GPU cache usage

6 Client Module

The Client Module executes benchmarks against deployed AI services, collecting performance metrics and generating reports.

6.1 Module Architecture

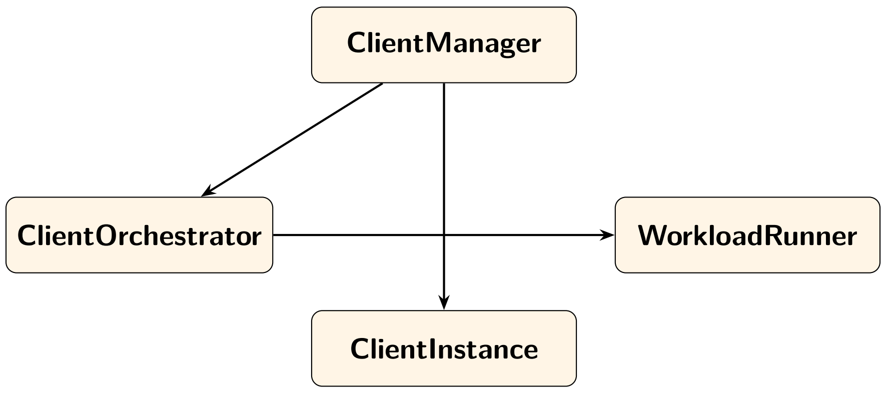

Figure 5: Client Module Class Structure

6.2 Workload Patterns

The framework supports two workload patterns:

6.2.1 Closed-Loop Pattern

Simulates real users with think time between requests:

```yaml
workload:
  pattern: closed-loop
  concurrent_users: 10
  think_time_ms: 500      # 500ms delay between requests
  requests_per_user: 100
  duration_seconds: 300
```

6.2.2 Open-Loop Pattern

Generates requests at a fixed rate regardless of response times:

```yaml
workload:
  pattern: open-loop
  requests_per_second: 50
  duration_seconds: 300
```

6.3 WorkloadRunner

The `WorkloadRunner` class executes the actual benchmark:

- Makes HTTP requests to the target endpoint
- Measures latency for each request
- Tracks success/failure rates
- Computes aggregate statistics

6.4 Collected Metrics

Table 2: Benchmark Output Metrics

Metric	**Description**
`total_requests`	Total requests made
`successes`	Successful requests (2xx status)
`errors`	Failed requests
`avg_latency_ms`	Average latency
`min_latency_ms`	Minimum latency
`max_latency_ms`	Maximum latency
`throughput_req_per_sec`	Requests per second

7 Recipe System

The recipe system provides a declarative approach to defining deployments, making configurations version-controllable and reproducible.

7.1 Recipe Types

Table 3: Recipe Types and Locations

Type	Directory	Purpose
Server	`recipes/servers/`	AI service deployment
Monitor	`recipes/monitors/`	Prometheus monitoring
Client	`recipes/clients/`	Benchmark workloads

7.2 Recipe Linking

All recipes for the same service share the `service_name` field, enabling automatic discovery and connection between components. This design choice was made to:

- Eliminate manual endpoint configuration
- Enable one-command deployment workflows
- Reduce human error in multi-component setups

7.3 Resource Specification

Resources are specified consistently across all recipe types:

```
orchestration:
  resources:
    partition: gpu          # SLURM partition
    cpu_cores: 8            # CPUs per task
    memory_gb: 64           # Memory allocation
    gpu_count: 1            # GPU count (0 for CPU-only)
    time_limit: "04:00:00"  # Job time limit
```

8 Design Decisions

This section documents key design decisions made during development and their rationale.

8.1 File-Based Service Discovery

Decision: Use JSON files in the user's home directory for service discovery instead of a centralized database or service mesh.

Rationale:

- **Simplicity**: No additional infrastructure required

- **Reliability**: Works with shared filesystems standard in HPC environments
- **User Isolation**: Each user has their own discovery namespace
- **Debugging**: Easy to inspect and manually modify if needed

8.2 Manager-Orchestrator Pattern

Decision: Separate business logic (Manager) from SLURM interaction (Orchestrator).
Rationale:

- **Testability**: Managers can be tested without SLURM
- **Portability**: Orchestrators can be swapped for other schedulers
- **Maintainability**: Clear separation of concerns

8.3 Synchronous Node Wait

Decision: Block during server deployment until SLURM assigns a node.
Rationale:

- Ensures discovery information is complete before returning
- Provides immediate feedback on deployment success
- Simplifies client/monitor workflows that depend on server endpoints

8.4 Recipe-Based Configuration

Decision: Use YAML recipes for all configurations instead of CLI flags.
Rationale:

- **Reproducibility**: Recipes can be version-controlled
- **Complexity Management**: Complex configurations are readable
- **Sharing**: Teams can share standardized configurations

8.5 Prometheus for Monitoring

Decision: Use Prometheus as the monitoring backend.
Rationale:

- Industry standard for metrics collection
- Native support in vLLM and many AI frameworks
- Pull-based model works well with SLURM job discovery
- Rich query language (PromQL) for analysis

9 Getting Started

This section provides step-by-step instructions for deploying AI-Factories on MeluXina.

9.1 Prerequisites

- SSH access to MeluXina cluster
- SLURM account (e.g., p200981)
- Python 3.8+ environment

9.2 Initial Setup

```bash
# 1. Connect to MeluXina
ssh <username>@meluxina.lxp.lu

# 2. Request an interactive session
salloc -A p200981 -t 02:00:00 -q dev

# 3. Navigate to the repository
cd /path/to/AI-Factories

# 4. Run the setup script
./setup.sh

# 5. Configure your SLURM account
export SLURM_ACCOUNT=p200981
```

Listing 9: MeluXina Setup Commands

9.3 Basic Workflow

```bash
# Deploy a vLLM server (auto-registers endpoint)
python -m src.server run --recipe vllm-server
# Output: vllm-server:a1b2c3d4 -> JobID 3757043

# Verify service discovery
python -m src.list_services
# Output: vllm: node=mel2013, ports=[8000]

# Start Prometheus monitoring (auto-discovers server)
python -m src.monitor start --recipe vllm-monitor
# Output: Monitor ID: xyz789, Prometheus: http://mel2014:9090

# Run benchmark (auto-discovers server)
python -m src.client run --recipe vllm-simple-test
# Output: Results saved to ./results/

# Cleanup
python -m src.monitor stop-all
python -m src.server stop-all
python -m src.clear_services
```

Listing 10: Complete Deployment Workflow

9.4 Accessing Prometheus UI

To access the Prometheus web interface from your local machine:

```
# From your local machine (new terminal)
ssh -L 9090:mel2014:9090 <username>@meluxina.lxp.lu

# Open in browser
# http://localhost:9090
```

Listing 11: SSH Tunnel for Prometheus

10 Benchmarking Results

This section is a placeholder for benchmarking results that will be completed in a future iteration of this report.

10.1 Experimental Setup

TODO: Document the experimental configuration including:

- Hardware specifications (GPU model, memory, etc.)
- Model configurations tested
- Workload parameters
- Duration of experiments

10.2 Performance Metrics

TODO: Present collected metrics including:

- Throughput (requests/second)
- Latency distribution (p50, p95, p99)
- Time to first token
- GPU utilization
- Memory consumption

10.3 Comparative Analysis

TODO: Compare performance across:

- Different model sizes
- Varying concurrency levels
- Open-loop vs. closed-loop patterns

10.4 Scalability Evaluation

TODO: Analyze scaling behavior with:

- Increasing number of concurrent users
- Different batch sizes
- Multi-GPU configurations (if applicable)

11 Future Improvements

Based on development experience and user feedback, the following improvements are planned:

11.1 Short-Term Enhancements

- **Multi-Node Server Deployment**: Support for distributed inference across multiple nodes using tensor parallelism.
- **Grafana Integration**: Add Grafana dashboards for visualization alongside Prometheus.
- **Alerting**: Configure Prometheus alerting for service health monitoring.
- **Result Aggregation**: Automatic aggregation and comparison of benchmark runs.

11.2 Long-Term Goals

- **Web Dashboard**: Browser-based UI for monitoring and management.
- **Additional Backends**: Support for other inference servers (TensorRT-LLM, text-generation-inference).
- **CI/CD Integration**: Automated deployment and testing pipelines.
- **Cost Tracking**: Integration with SLURM accounting for resource usage reports.
- **Auto-Scaling**: Dynamic scaling based on request queue depth.

12 Conclusion

AI-Factories provides a comprehensive solution for deploying, monitoring, and benchmarking AI inference services on HPC clusters. The framework addresses key challenges in managing containerized AI workloads in SLURM environments through:

- A modular architecture with clear separation of concerns
- Automatic service discovery that eliminates manual endpoint management
- Recipe-based configuration for reproducible deployments
- Integrated Prometheus monitoring with automatic target resolution

- Flexible benchmarking capabilities with configurable workload patterns

The framework has been successfully deployed and tested on the MeluXina supercomputer, demonstrating its effectiveness for managing AI inference workloads in production HPC environments.

References

1. vLLM: Easy, Fast, and Cheap LLM Serving. https://github.com/vllm-project/vllm
2. Prometheus Monitoring System. https://prometheus.io/
3. Apptainer (Singularity) Container Platform. https://apptainer.org/
4. SLURM Workload Manager. https://slurm.schedmd.com/
5. MeluXina Supercomputer. https://luxprovide.lu/