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

Supervisor: Dr Farouk Mansouri

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. Our application 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

Our application 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 project follows a modular architecture with three core modules and a central service discovery mechanism. Figure 1 illustrates the high-level system design.

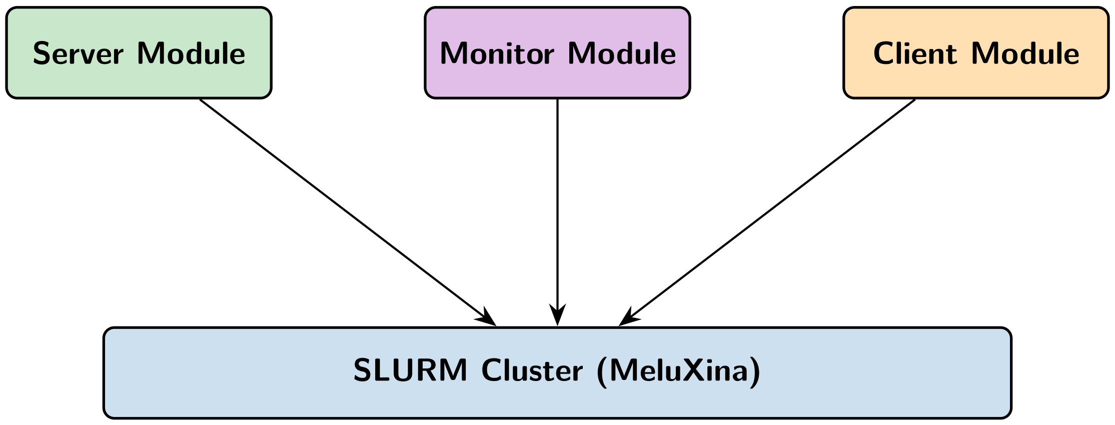

Figure 1: High-Level Architecture

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:

```
root/
|-- 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 Server Module

The Server Module is responsible for deploying AI inference services as SLURM jobs using containerized applications. It provides automated deployment, lifecycle management, and service discovery integration.

3.1 Module Architecture

The server module follows a layered architecture with five core classes working together to manage the complete deployment lifecycle. Figure 2 illustrates the relationships between these components.

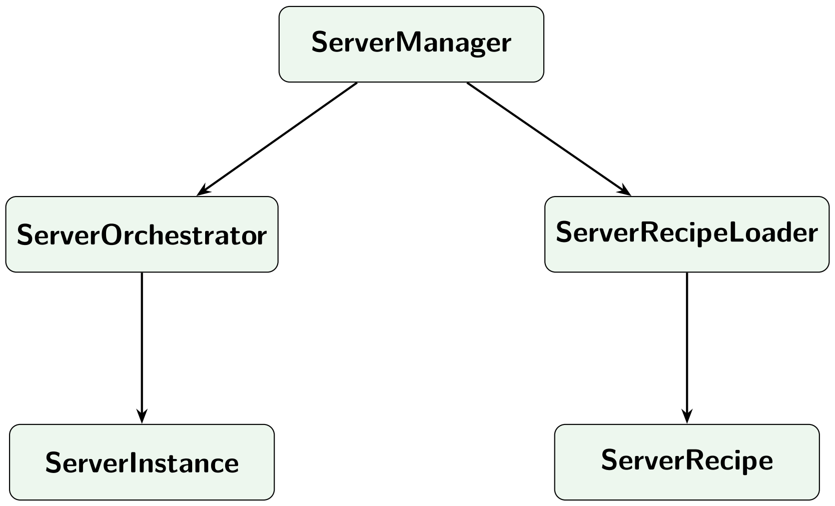

Figure 2: Server Module Class Structure

3.2 Class Descriptions

3.2.1 ServerManager

The `ServerManager` class serves as the primary interface for all server operations, coordinating recipe loading, job submission, and instance lifecycle management. This class implements the high-level business logic while delegating infrastructure concerns to the orchestrator.

The manager maintains a collection of active server instances indexed by unique names. When deploying a server, it loads the recipe specification, creates a `ServerInstance` object, submits the job through the orchestrator, and waits for SLURM to assign a compute node. Once the node assignment is complete, the manager writes discovery information to the shared filesystem, enabling clients and monitors to automatically locate the service. The synchronous waiting behavior ensures that discovery data is accurate before the deployment operation returns to the user.

Key methods include `run(recipe_name, count)` which deploys one or more instances of a server recipe, `stop(name)` which cancels a running deployment and clears discovery information, `stop_all()` which terminates all managed instances, and `collect_status()` which queries SLURM for current job states and updates discovery data for running services.

3.2.2 ServerOrchestrator

The `ServerOrchestrator` class handles all direct interaction with the SLURM workload manager, encapsulating platform-specific details and providing a clean abstraction for the manager. This separation enables the framework to support alternative schedulers in the future without modifying business logic.

The orchestrator dynamically generates SBATCH scripts based on recipe specifications, incorporating resource requirements, environment variables, and startup commands. It reads SLURM configuration from environment variables and YAML files, supporting flexible deployment across different HPC systems. The generated scripts include comprehensive logging that records job metadata, node assignments, and execution status.

Core responsibilities include `submit(instance, recipe)` which builds and submits SBATCH scripts returning job IDs, `stop(job_id)` which cancels running jobs using `scancel`, and `status(job_id)` which queries job state using `squeue` and parses output to determine if jobs are pending, running, or completed. The orchestrator maps SLURM status strings to internal `ServerStatus` enum values, providing a consistent state model across the framework.

3.2.3 ServerInstance

The `ServerInstance` class represents a single deployed server with runtime state tracking. Each instance receives a unique UUID identifier on creation and maintains a reference to its SLURM job ID (orchestrator handle). The instance tracks its lifecycle status through the `ServerStatus` enumeration, which includes states such as SUBMITTED, STARTING, RUNNING, COMPLETED, FAILED, and CANCELED.

Instances store deployment metadata including the assigned compute node, exposed ports, creation timestamp, and completion timestamp. This information is essential for service discovery and debugging.

3.2.4 ServerRecipe

The `ServerRecipe` class represents a parsed and validated server deployment specification loaded from YAML files. Each recipe encapsulates all information needed to deploy a service, including the service startup command, port mappings, environment variables, working directory, and resource requirements.

The recipe validates its configuration during initialization, ensuring that required fields are present and that port numbers fall within valid ranges. It provides convenient property accessors for common configuration elements such as `resources` which returns the orchestration resource dictionary, `env` which returns environment variables, `working_directory` which specifies where the service executes, and `ports` which lists exposed network ports.

The class method `from_yaml(yaml_path)` implements the factory pattern for loading recipes from disk, reading YAML files, parsing their contents, and constructing validated recipe objects. This approach centralizes parsing logic and ensures consistent validation across all recipe loads.

3.2.5 ServerRecipeLoader

The `ServerRecipeLoader` class manages recipe discovery, loading, and caching from the recipes directory. It maintains an in-memory cache of loaded recipes to avoid repeated file I/O and parsing overhead. The loader scans the configured recipe directory for YAML files with `.yml` or `.yaml` extensions.

Key methods include `load_recipe(name)` which locates, parses, and caches a recipe by name, `list_available_recipes()` which returns all recipe names found in the directory, and `get_recipe_info(name)` which returns summary information about a recipe without fully loading it. The loader also provides `create_recipe_template(name)` which generates a starter recipe file with example configuration, helping users create new recipes quickly.

3.3 Server Recipe Specification

Server recipes define all aspects of a deployment:

```yaml
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 2: vLLM Server Recipe Example

3.4 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 `~/.aibenchmark/discover/vllm.json`
7. Instance is registered and status is returned to user

4 Client Module

The Client Module executes benchmarks against deployed AI services, collecting performance metrics and generating comprehensive reports. It supports multiple workload patterns and provides detailed latency and throughput measurements.

4.1 Module Architecture

The client module consists of six core classes that work together to execute benchmarks as SLURM jobs. Figure 3 shows the architectural relationships.

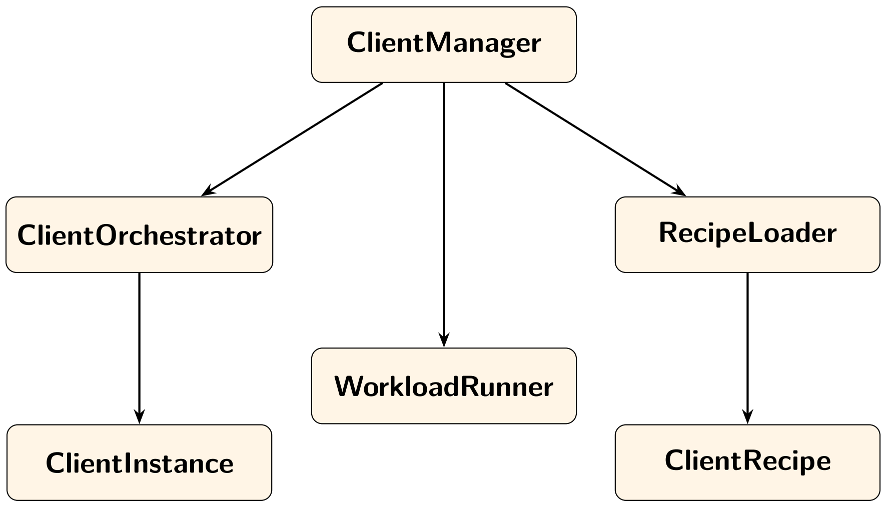

Figure 3: Client Module Class Structure

4.2 Class Descriptions

4.2.1 ClientManager

The `ClientManager` class serves as the primary interface for benchmark operations, coordinating recipe loading, service discovery, and benchmark execution. This manager maintains a collection of active benchmark runs and provides methods for executing single or multiple benchmark iterations.

The manager integrates with the service discovery mechanism to automatically locate target services. When a benchmark recipe specifies a service name, the manager reads discovery information to determine the current endpoint, eliminating manual endpoint configuration. This integration enables seamless benchmarking workflows where users can deploy servers and immediately run benchmarks against them without copying connection details.

Key methods include `run_bench(name, runs)` which loads a recipe, discovers the target endpoint, and executes one or more benchmark runs, `add_client(name, config)` which manually registers a client configuration, `remove_client(name)` which unregisters a client and cancels running benchmarks, `stop_all()` which terminates all active benchmark jobs, and `collect_metrics()` which gathers performance metrics from all client instances for analysis.

4.2.2 ClientOrchestrator

The `ClientOrchestrator` class handles SLURM integration for benchmark jobs, generating SBATCH scripts and managing job submission. Similar to the server orchestrator, this class encapsulates scheduler-specific logic and provides a clean abstraction for the manager.

The orchestrator builds batch scripts that invoke the workload runner with appropriate parameters derived from the recipe configuration. It handles both closed-loop and open-loop workload patterns, constructing command-line arguments for duration, concurrency, think time, and request payloads. The orchestrator also manages resource allocation, ensuring benchmark jobs receive sufficient CPU and memory to generate their target workload without becoming resource-constrained.

Core responsibilities include `submit(run, recipe, target_endpoint)` which generates and submits benchmark jobs, `stop(job_id)` which cancels running benchmarks using `scancel`, `status(job_id)` which queries SLURM for job state, and `_build_workload_command()` which constructs the Python invocation for the workload runner with all necessary parameters. The orchestrator also handles payload generation from dataset specifications, supporting both synthetic datasets defined inline and custom datasets loaded from files.

4.2.3 ClientInstance

The `ClientInstance` class represents a single benchmark run with state tracking and metrics collection. Each instance receives a unique identifier and maintains a reference to its SLURM job ID. The instance tracks its lifecycle through the `RunStatus` enumeration with states including SUBMITTED, RUNNING, COMPLETED, FAILED, and CANCELED.

Instances store benchmark results as they complete, including total requests, success counts, error counts, latency measurements, and throughput calculations. The class provides lifecycle methods including `start()` which transitions to RUNNING state, `stop()` which cancels the benchmark, and `update_status(new_status)` which transitions the instance through states.

Metrics methods support result reporting and analysis. The `get_metrics()` method returns a comprehensive dictionary including run identifier, recipe name, status, duration, endpoint, and all collected performance metrics. The `to_dict()` method provides JSON serialization for persistence and logging.

4.2.4 ClientRecipe

The `ClientRecipe` class represents a parsed benchmark specification loaded from YAML files. Each recipe defines the target service, workload pattern, dataset configuration, resource requirements, and output settings. The recipe validates its configuration during initialization, checking for required fields and ensuring that parameters fall within valid ranges.

The recipe supports both service discovery and explicit endpoint specification. When a `service_name` field is present, the framework attempts service discovery before falling back to explicit endpoints. This flexibility enables both automatic and manual configuration workflows.

Key properties include `target` which specifies protocol and endpoint configuration, `workload` which defines pattern, duration, concurrency, and think time, `dataset` which configures request payloads including prompt length, token limits, and model parameters, `orchestration` which specifies SLURM resources, and `output` which controls metrics collection and result destination.

The class method `from_yaml(yaml_path)` implements recipe loading from disk, parsing YAML content and constructing validated recipe objects. The `validate()` method ensures configuration correctness before benchmarks execute.

4.2.5 WorkloadRunner

The `WorkloadRunner` class implements the actual benchmark execution logic, generating HTTP requests and measuring performance. This class runs directly on SLURM compute nodes as part of benchmark jobs, making requests to target services and collecting latency data.

The runner supports two workload patterns. **Closed-loop mode** simulates realistic user behavior by maintaining a fixed number of concurrent users with think time between requests. Each user makes requests sequentially, waiting for responses before issuing the next request. This pattern tests how systems behave under user-driven load. **Open-loop mode** generates requests at a constant rate regardless of response times, useful for stress testing and finding maximum throughput limits. This pattern can drive systems beyond their capacity, revealing failure modes and queueing behavior.

For each request, the runner measures end-to-end latency including network time, records HTTP status codes, and tracks success or failure. After the benchmark completes, the runner computes aggregate statistics including total requests, success count, error count, average latency, minimum latency, maximum latency, and throughput in requests per second. These metrics are written to JSON files for analysis.

Core methods include `run()` which executes the benchmark and returns metrics, `_run_closed_loop()` which implements concurrent user simulation, `_run_open_loop()` which implements fixed-rate request generation, `_make_request()` which executes individual HTTP requests with error handling, and `_compute_metrics(total_duration)` which calculates aggregate performance statistics from collected measurements.

4.2.6 RecipeLoader

The `RecipeLoader` class manages client recipe discovery, loading, and caching. It scans the recipes directory for YAML files and maintains an in-memory cache to avoid repeated parsing. The loader provides `load_recipe(name)` for loading specific recipes, `list_available_recipes()` for discovering all available benchmarks, and caching mechanisms for performance optimization.

4.3 Workload Patterns

The framework supports two workload patterns:

4.3.1 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
```

4.3.2 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
```

4.4 Collected Metrics

Table 1: 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

5 Monitor Module

The Monitor Module deploys Prometheus monitoring stacks to collect metrics from running AI services. It provides automatic target discovery, metrics export capabilities, and service-specific metric selection.

5.1 Module Architecture

The monitor module implements a sophisticated monitoring infrastructure with eight core classes managing the complete Prometheus lifecycle. Figure 4 illustrates component relationships.

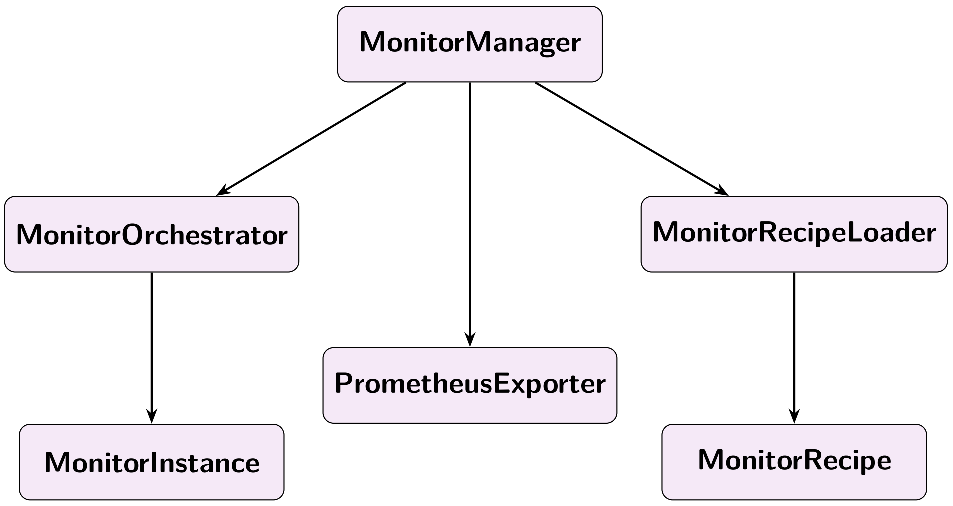

Figure 4: Monitor Module Class Structure

5.2 Class Descriptions

5.2.1 MonitorManager

The `MonitorManager` class orchestrates the complete monitoring lifecycle, managing Prometheus deployments, target resolution, and metrics export operations. This manager maintains persistent state across sessions by saving instance information to JSON files in the logs directory, enabling recovery after framework restarts.

The manager integrates deeply with the service discovery mechanism. When starting a monitor, it reads discovery information for target services and automatically resolves their endpoints. This eliminates manual configuration of Prometheus scrape targets and ensures monitoring adapts dynamically as services are deployed or redeployed to different compute nodes.

Key methods include `start_monitor(recipe_name, target_job_ids, metadata)` which deploys a Prometheus instance with automatic target resolution, `stop_monitor(monitor_id)` which terminates a monitoring stack and cancels all component jobs, `list_running_monitors()` which returns currently active monitors, `export_metrics(monitor_id, format, export_type)` which exports collected metrics to JSON or CSV formats, and `_resolve_targets(recipe, targets, job_ids)` which converts service names to actual host:port endpoints using discovery data.

The manager also implements state persistence through `_save_state()` and `_load_state()` methods, writing instance collections to disk and restoring them on initialization. This ensures monitoring continuity across framework restarts.

5.2.2 MonitorOrchestrator

The `MonitorOrchestrator` class handles SLURM integration for Prometheus deployments, managing container execution via Apptainer. Unlike the server and client orchestrators which run custom scripts, the monitor orchestrator deploys containerized Prometheus instances with dynamically generated configuration files.

The orchestrator generates Prometheus configuration YAML from recipe specifications, creating scrape configurations for each target service. It sets up bind mounts that expose configuration and data directories to the Prometheus container, enabling persistent metric storage and runtime configuration updates. The orchestrator also manages Apptainer module loading and container image caching.

Core responsibilities include `deploy_prometheus(config, targets, config_dir, data_dir)` which generates configuration, creates SBATCH scripts, and submits Prometheus jobs, `stop_component(job_id)` which cancels running monitoring components,

The orchestrator reads SLURM configuration from YAML files and environment variables, supporting flexible deployment across different HPC systems. It handles resource specification for monitoring workloads, which typically require fewer resources than inference servers but need sufficient CPU and memory for metric scraping and storage.

5.2.3 PrometheusExporter

The `PrometheusExporter` class provides metrics export functionality, querying Prometheus HTTP APIs and formatting results as JSON or CSV. This class implements service-specific metric selection, automatically choosing appropriate metrics based on the monitored service type.

The exporter maintains registries of common system metrics (CPU, memory, network) and service-specific metrics for vLLM, Ollama, and other AI frameworks. When exporting metrics, it selects the appropriate metric set based on the service name, ensuring relevant data is collected without overwhelming users with unnecessary metrics.

Key methods include `query_instant(query)` which executes immediate PromQL queries returning current metric values, `query_range(query, start, end, step)` which executes time-range queries for historical data, `export_instant_metrics(output_file, format, service_name)` which exports current metric snapshots, `export_range_metrics(output_fil`
`format, start, end, service_name)` which exports time-series data, and `get_queries_for_service`
which returns appropriate metric queries for a specific service.

The exporter supports both JSON and CSV output formats. JSON format provides structured data suitable for programmatic analysis, while CSV format enables easy import into spreadsheet applications and data analysis tools. The CSV exporter includes metadata headers documenting Prometheus URL, time range, and scrape interval.

5.2.4 MonitorInstance

The `MonitorInstance` class represents a deployed monitoring stack with state tracking and component management. Each instance maintains references to its deployed components (Prometheus, exporters, alertmanagers), tracks their job IDs and endpoints, and stores resolved target mappings.

Instances support serialization to and from dictionaries, enabling persistent storage and state recovery. The class tracks creation time, current status (PENDING, STARTING, RUNNING, STOPPING, STOPPED, ERROR), and metadata about the monitoring deployment.

5.2.5 MonitorRecipe

The `MonitorRecipe` class represents a parsed monitoring specification loaded from YAML files. Each recipe defines target services to monitor, Prometheus configuration including scrape intervals and retention time, and resource requirements for the monitoring stack.

The recipe implements validation logic ensuring target specifications are complete and Prometheus configuration is valid. It provides the `to_prometheus_config(resolved_targets)` method which generates Prometheus YAML configuration from recipe specifications and resolved endpoint mappings.

Key properties include `targets` which lists `TargetService` objects specifying what to monitor, `prometheus` which contains `PrometheusConfig` with scrape interval, retention time, port, and resource settings, `service_name` which links to service discovery, and `name` which uniquely identifies the recipe.

The class method `from_yaml(path)` implements recipe loading, parsing YAML content, constructing target and Prometheus configuration objects, and validating the complete specification before returning the recipe.

5.2.6 TargetService

The `TargetService` class represents a single service to be monitored, encapsulating configuration for Prometheus scrape targets. Each target specifies a service name for discovery, optional explicit endpoint, metrics path (typically /metrics), and port number.

The class provides validation ensuring either a service name or explicit endpoint is specified, enabling both automatic discovery and manual configuration workflows. Targets support optional job ID tracking for monitoring specific SLURM deployments.

5.2.7 PrometheusConfig

The `PrometheusConfig` class encapsulates all configuration for a Prometheus deployment, including container image, scrape interval, retention time, exposed port, SLURM partition, and resource requirements. This class provides validation for port numbers and enables configuration serialization.

Default values provide sensible monitoring defaults: 15-second scrape intervals for responsive metric collection, 24-hour retention for reasonable storage requirements, and 2 CPU cores with 4GB memory for efficient operation without excessive resource consumption.

5.2.8 MonitorRecipeLoader

The `MonitorRecipeLoader` class manages monitor recipe discovery, loading, and caching from the recipes directory. Similar to other loaders, it maintains an in-memory cache and provides methods for listing available recipes, loading specific recipes, and managing recipe metadata.

5.3 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 3: Target Resolution Logic

5.4 Prometheus Configuration Generation

The orchestrator dynamically generates Prometheus configuration:

```
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 4: Generated prometheus.yml

5.5 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 5: Metrics Export Commands

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

Table 2: 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 Recipe System

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

6.1 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

6.2 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

6.3 Resource Specification

Resources are specified consistently across all recipe types:

```yaml
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
```

6.4 Creating Custom Recipes

The framework provides a flexible recipe system that allows users to define custom deployments for any AI service. Creating recipes follows a structured approach with three main recipe types that must work together.

6.4.1 Server Recipe Structure

Server recipes define how to deploy an AI inference service on the HPC cluster. Each server recipe must specify the **service startup command**, which is a bash script executed when the SLURM job starts. This command typically loads required modules (such as Apptainer), prepares the container environment, and launches the inference server. The recipe also defines **port mappings** that specify which network ports the service exposes. These ports are automatically registered in the discovery system when the job starts running. **Environment variables** can be configured to pass runtime parameters to the containerized service. The **working directory** determines where the service executes, which is important for finding model files and configuration data.

Resource requirements must be carefully specified based on the service needs. For GPU-accelerated inference servers, the recipe must request GPU resources through the `gpu_count` parameter and use the `gpu` partition. CPU-only services should set `gpu_count` to zero and use the `cpu` partition. Memory allocation should account for model size plus inference batch processing overhead. Time limits prevent jobs from running indefinitely and should be set based on expected workload duration.

6.4.2 Monitor Recipe Structure

Monitor recipes configure Prometheus deployments for collecting metrics from running services. The recipe specifies **target endpoints** that Prometheus should scrape for metrics. These targets are automatically resolved using the service discovery mechanism, eliminating manual endpoint configuration. The **scrape interval** determines how frequently Prometheus collects metrics, with shorter intervals providing more granular data at the cost of storage space. **Retention time** controls how long metrics are stored before being deleted, balancing data availability with disk usage. The **metrics path** specifies the HTTP endpoint where the service exposes Prometheus-format metrics, typically `/metrics` for most services.

Monitor recipes require fewer resources than inference servers since Prometheus is primarily I/O bound rather than compute intensive. A typical monitoring deployment uses two CPU cores and four gigabytes of memory, which suffices for scraping dozens of targets at 15-second intervals.

6.4.3 Client Recipe Structure

Client recipes define benchmark workloads that test service performance under controlled conditions. The recipe specifies the **workload pattern**, which can be either closed-loop or open-loop. **Closed-loop patterns** simulate realistic user behavior by maintaining a fixed number of concurrent users with think time between requests. This pattern is useful for understanding how systems behave under user-driven load. **Open-loop patterns** generate requests at a constant rate regardless of response times, which is useful for stress testing and finding maximum throughput limits.

The **dataset configuration** defines the request payloads sent to the service. For synthetic datasets, parameters like model name, prompt length, and token limits are specified directly in the recipe. For custom datasets, the recipe can reference external files containing pre-generated requests. **Output metrics** determine which performance indicators are collected during the benchmark, including latency percentiles, throughput, error rates, and success rates.

6.4.4 Naming Conventions and Best Practices

Recipe names should follow consistent patterns to maintain clarity across the framework. Server recipes use the format `service-server.yaml`, monitor recipes use `service-monitor.yml`, and client recipes use `service-testtype.yaml`. The `service_name` field must be identical across all recipes for the same service to enable automatic discovery. This field should be short, lowercase, and use hyphens to separate words.

Port selection requires careful consideration to avoid conflicts on shared compute nodes. The framework recommends using ports in the 8000 to 9000 range, which are typically available on HPC clusters. Services can also use dynamic port allocation by setting the port to zero in the startup script, though this requires parsing log files to determine the actual assigned port.

Resource allocation should be conservative to maximize job scheduling success while providing adequate performance. Over-requesting resources causes jobs to wait longer in the queue, while under-requesting can lead to out-of-memory errors or poor performance. GPU services typically require 64 gigabytes or more of memory due to model size and batch processing needs, while CPU services can often run with 16 gigabytes.

7 Service Discovery Mechanism

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

7.1 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

7.2 Solution: File-Based Service Registry

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

```
~/.aibenchmark/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 6: Discovery File Format

7.3 Discovery Workflow

Figure 5 shows the discovery workflow in action.

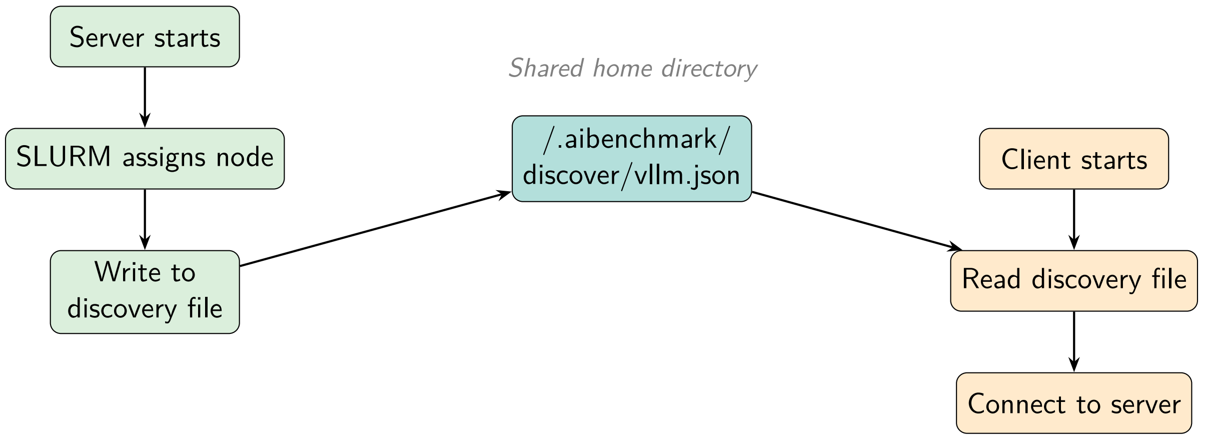

Figure 5: Service Discovery Workflow

7.4 Service Name Linking

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

```
# 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 7: Service Name Consistency Across Recipes

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

8 Getting Started

This section provides step-by-step instructions for deploying our application on MeluXina.

8.1 Initial Setup

```
# 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/repo
```

```

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

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

Listing 8: MeluXina Setup Commands

8.2 Basic Workflow

```
# 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 9: Complete Deployment Workflow

8.3 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 10: SSH Tunnel for Prometheus

9 Benchmarking Results

This section presents the results of comprehensive benchmarking experiments conducted on the MeluXina supercomputer using the project. Four distinct test scenarios were designed to evaluate vLLM inference performance under various workload conditions.

9.1 Test Scenarios

Four benchmark recipes were designed to evaluate different aspects of inference performance of vllm. The **simple test** provided a baseline measurement with 3 concurrent users, 50-token prompts, 20-token generation limits, and 500 milliseconds think time over a 2-minute duration. This test validated basic functionality and established performance baselines. The **stress test** increased load substantially with 20 concurrent users, 256-token prompts, 128-token generation, and 100 milliseconds think time over 10 minutes. This configuration represented realistic production workloads for conversational AI applications.

The **high-throughput test** focused on maximum request-per-second capacity by using 50 concurrent users with very short prompts (20 tokens) and minimal generation (10 tokens) with only 50 milliseconds think time. This test evaluated scheduler efficiency and batching capabilities under high request rates. The **long-context test** evaluated memory scaling and attention mechanism performance with large inputs, using 5 concurrent users with 2048-token prompts and 512-token generation over 10 minutes. Lower concurrency was necessary due to the memory-intensive nature of processing large contexts.

9.2 Performance Results

Benchmark results revealed distinct performance characteristics across the four test scenarios. Table 4 summarizes key metrics for each test configuration.

Table 4: Benchmark Results Summary

Test	**Throughput (req/s)**	**Avg Latency (ms)**	**Max Latency (ms)**	**Requests**
Simple Test	1.81	52.30	100.97	218
Stress Test	2.65	276.59	507.29	1593
High-Throughput	12.02	33.10	5037.83	5000
Long-Context	0.69	945.93	1422.34	416

The **simple test baseline** achieved 1.81 requests per second with average latency of 52.30 milliseconds and maximum latency under 101 milliseconds, demonstrating excellent consistency. All 218 requests completed successfully with zero errors. The low latency reflects the minimal computational requirements of the small prompt and generation lengths combined with light concurrent load.

The **stress test** processed 1593 requests over 10 minutes at 2.65 requests per second with average latency of 276.59 milliseconds. The increased latency compared to the simple test reflects the substantially larger prompt sizes (256 tokens versus 50 tokens) and longer generation lengths (128 tokens versus 20 tokens). Maximum latency remained reasonable at 507.29 milliseconds, indicating good queue management even under sustained concurrent load of 20 users.

The **high-throughput test** achieved the highest request rate at 12.02 requests per second, processing 5000 total requests with remarkably low average latency of 33.10 milliseconds. This result demonstrates vLLM's efficient batching capabilities when handling many short requests. However, the maximum latency spiked to 5037.83 milliseconds, revealing occasional queueing delays when the system became saturated. This behavior is expected under extreme load and validates the importance of monitoring tail latency in production deployments.

The **long-context test** exhibited the lowest throughput at 0.69 requests per second and highest average latency at 945.93 milliseconds, which directly reflects the computational cost of processing 2048-token prompts and generating 512-token outputs. Despite the heavy workload, maximum latency remained under 1.5 seconds and all 416 requests completed successfully, demonstrating that the framework can reliably handle memory-intensive inference tasks.

9.3 Comparative Analysis

Analyzing performance across test scenarios reveals how workload characteristics affect inference efficiency. **Throughput scaling** showed clear correlation with request size, where the high-throughput test (short requests) achieved 17.4 times higher throughput than the long-context test (large requests) despite having 10 times more concurrent users. This demonstrates that request size, not just concurrency, determines system capacity. **Latency scaling** followed expected patterns, with average latency increasing proportionally to prompt length plus generation length. The stress test with medium-sized requests showed 5.3 times higher latency than the simple test, while the long-context test showed 18 times higher latency.

Queue management effectiveness varied by scenario. The simple and stress tests maintained consistent latencies with maximum latencies only 1.9 to 2.0 times higher than average latencies. However, the high-throughput test showed maximum latency 152 times higher than average, indicating that under saturation conditions, some requests experience significant queueing delays while most requests are processed quickly. This bimodal distribution is characteristic of systems operating at capacity limits. **Reliability** remained excellent across all scenarios, with 100% success rates and zero errors even under extreme load conditions totaling 7,227 completed requests.

9.4 Time-Series Analysis

Prometheus monitoring data collected during the stress test provides insight into system behavior over time. Data shows average generation throughput measured in tokens per second. After an initial ramp-up period, throughput stabilized around 300 tokens per second, demonstrating consistent performance throughout the 10-minute test duration. The relatively flat profile indicates efficient resource utilization without degradation over time.

GPU cache usage percentage exhibited periodic fluctuations between near-zero and 0.014% as shown in monitoring data, which reflects vLLM's dynamic cache management for the CPU-based deployment. Since the test ran on CPUs without GPUs, these values represent KV-cache operations in system memory rather than GPU memory. The low absolute values confirm that the opt-125m model with specified context lengths did not stress memory capacity.

Request queue depths remained low throughout testing, with running requests typically between 1 and 3 concurrent operations and waiting requests occasionally spiking to 5 during burst periods. These patterns confirm that the 8-core CPU allocation provided sufficient compute capacity to process incoming requests without building large backlogs. CPU utilization grew linearly from approximately 44 seconds to 345 seconds of accumulated CPU time over the test duration, indicating efficient CPU usage at roughly 75% utilization across the allocated cores.

10 Future Improvements

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

10.1 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.

10.2 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.

11 Conclusion

Our application 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, and 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. Comprehensive benchmarking experiments validated the framework's capabilities across diverse workload scenarios, processing over 7,200 inference requests with 100% success rates. Performance results showed that the framework efficiently handles workloads ranging from simple validation tests to high-throughput scenarios processing 12 requests per second, as well as memory-intensive long-context inference with multi-kilobyte prompts.

Time-series monitoring data confirmed consistent throughput and resource utilization throughout extended test runs, demonstrating the framework's reliability for sustained production workloads. The declarative recipe system proved effective for defining reproducible benchmarks with varying concurrency levels, request sizes, and workload patterns. Automatic service discovery successfully eliminated manual endpoint configuration across all test scenarios, enabling seamless coordination between server, monitor, and client modules.

Future work will focus on expanding the framework's capabilities to support multi-node distributed inference, additional inference backends beyond vLLM, and enhanced visualization through Grafana integration. The framework provides a solid foundation for research groups and HPC centers seeking to deploy AI inference services with robust monitoring and standardized benchmarking capabilities.