HPC Benchmark Toolkit

Technical Report

A Comprehensive Framework for Benchmarking
LLM Inference Services on HPC Clusters

Target Platform:	MeluXina HPC Cluster
Container Runtime:	Apptainer/Singularity
Scheduler:	Slurm
Services:	Ollama, vLLM (Single & Distributed)

December 2024

Contents

1 Introduction

1.1 Project Overview

The HPC Benchmark Toolkit is a production-ready framework designed for benchmarking Large Language Model (LLM) inference services on High-Performance Computing (HPC) clusters. The toolkit addresses the critical need for reproducible, scalable, and observable benchmarking in enterprise AI deployments.

1.1.1 Motivation

Modern AI deployments require careful performance characterization before production rollout. Key challenges include:

- **Reproducibility**: Experiments must be repeatable with identical configurations
- **Scalability**: Benchmarks must scale from single-node to multi-node distributed setups
- **Observability**: Real-time metrics are essential for performance analysis
- **HPC Integration**: Seamless integration with Slurm and containerized workloads

1.1.2 Key Features

Feature	Description
Multi-Service Support	Ollama, vLLM (single and distributed), extensible architecture
Distributed Benchmarking	Multi-node server/client orchestration with Ray
Real-Time Monitoring	Prometheus/Grafana integration with live dashboards
Recipe-Driven	YAML configurations for reproducible experiments
HPC-Optimized	Slurm integration, Apptainer container support
Comprehensive Metrics	Latency (p50/p90/p99), throughput, resource utilization

Table 1: Key features of the HPC Benchmark Toolkit

1.2 Supported Services

The toolkit currently supports the following inference services:

Service	Description	Default Port	API Type
Ollama	Local LLM inference server	11434	REST
vLLM	High-throughput LLM serving	8000	OpenAI-compatible
vLLM Distributed	Multi-node vLLM with Ray	8000	OpenAI-compatible
Dummy	Template for custom services	5000	REST

Table 2: Supported inference services

2 System Architecture

2.1 High-Level Architecture

The system follows a modular architecture with clear separation of concerns. Figure 1 illustrates the high-level component structure.

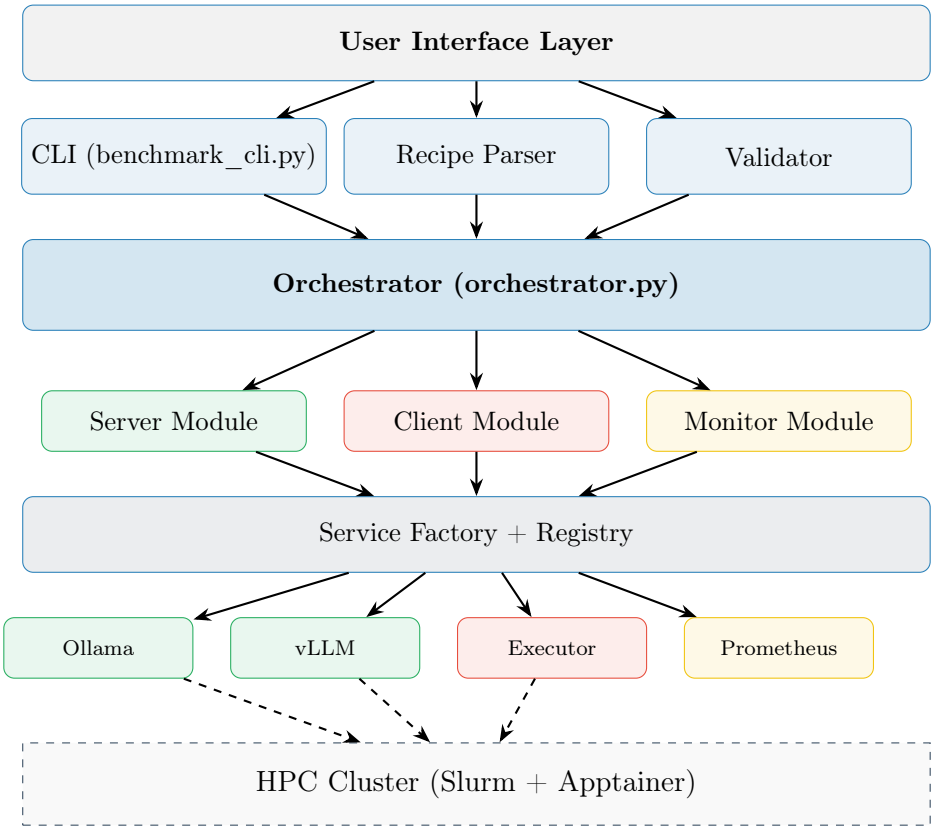

Figure 1: High-level system architecture

2.2 Design Patterns

The toolkit employs several software design patterns to ensure maintainability and extensibility:

2.2.1 Factory Pattern

The `ServiceFactory` class provides dynamic component instantiation based on service type:

```
class ServiceFactory:
    _registry = {}

    @classmethod
    def register_service(cls, name, server_cls, controller_cls, executor_cls):
        cls._registry[name] = {
            'server': server_cls,
            'controller': controller_cls,
            'executor': executor_cls
        }

    @classmethod
    def create_server_manager(cls, service_name, config):
        return cls._registry[service_name]['server'](config)
```

Listing 1: ServiceFactory implementation pattern

2.2.2 Template Method Pattern

Base classes define algorithm skeletons while subclasses provide specific implementations:

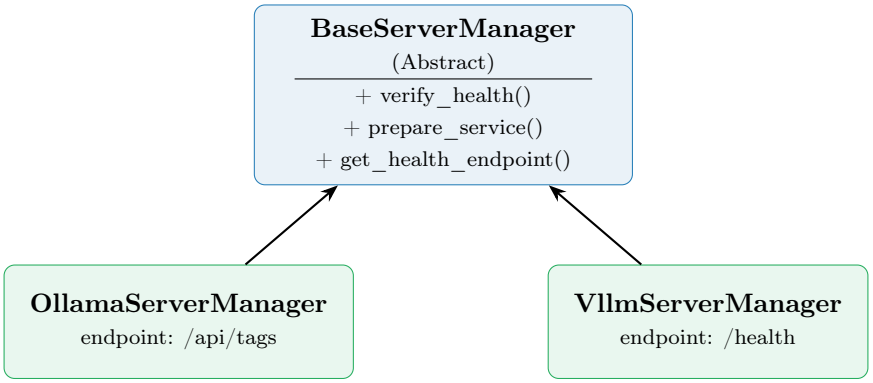

Figure 2: Template Method Pattern in Server Managers

2.3 Component Details

2.3.1 Server Managers

Server managers handle the lifecycle of inference services:

- **Health Checking**: Polls service endpoints to verify readiness
- **Service Preparation**: Loads models, initializes resources
- **Configuration Parsing**: Extracts service-specific settings from recipes

Manager	Health Endpoint	Port	Special Features
OllamaServerManager	/api/tags	11434	Model pulling via /api/pull
VllmServerManager	/health	8000	Ray cluster integration
DummyServerManager	/health	5000	Template implementation

Table 3: Server Manager implementations

2.3.2 Workload Controllers

Controllers coordinate workload execution across client nodes:

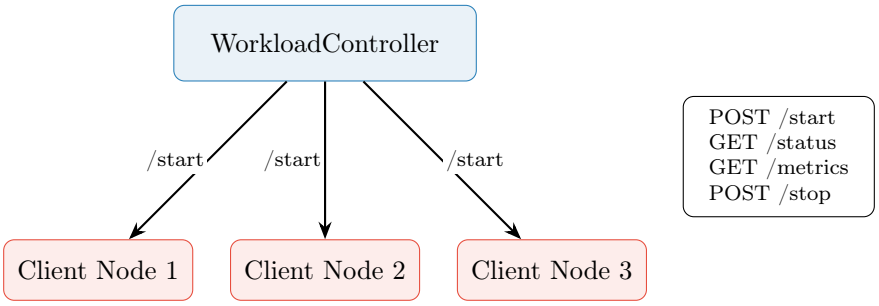

Figure 3: Workload Controller to Client communication

2.3.3 Workload Executors

Executors run on client nodes as Flask servers, executing the actual benchmark workload:

```
# Flask endpoints exposed by each executor
GET  /health               # Check executor status
POST /start                # Start workload with JSON config
GET  /status               # Get current workload status
GET  /metrics              # Fetch collected metrics
GET  /metrics/prometheus   # Prometheus-compatible format
POST /stop                 # Stop workload execution
```

Listing 2: Workload Executor REST API

3 Communication Architecture

3.1 Overview

The toolkit employs a hybrid communication architecture combining:

- **REST/HTTP**: Control plane communication between orchestrator and components
- **Service APIs**: Data plane communication between clients and inference servers
- **Prometheus Push/Pull**: Metrics collection and aggregation

3.2 Control Plane Communication

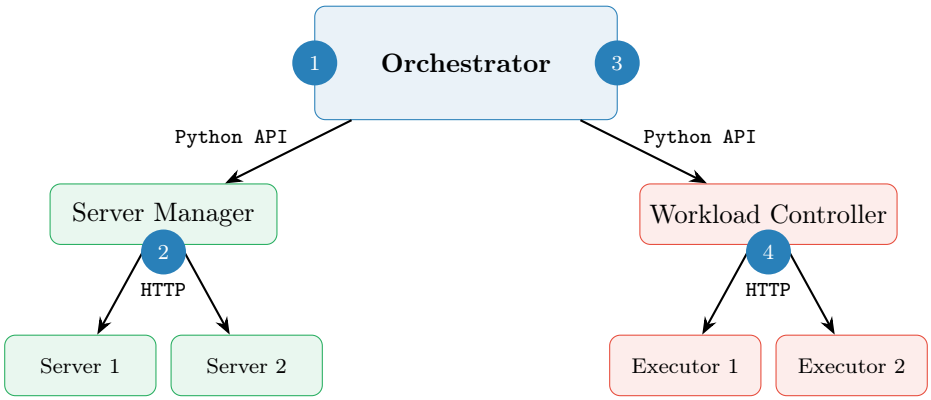

Figure 4: Control plane communication flow

3.3 Data Plane Communication

During benchmark execution, clients send inference requests directly to servers:

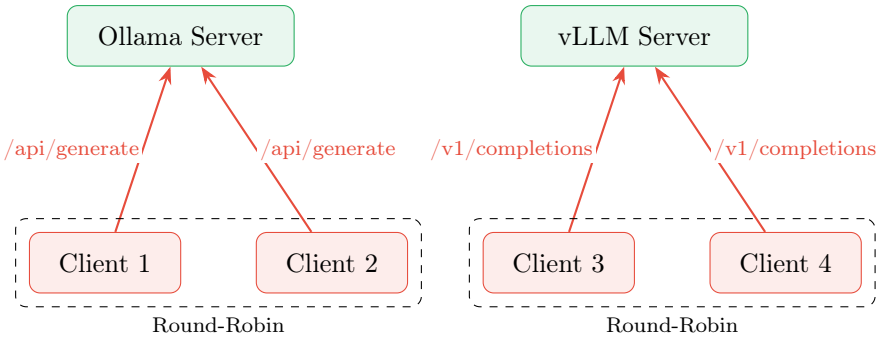

Figure 5: Data plane: Client to Server inference requests

3.4 Metrics Collection Pipeline

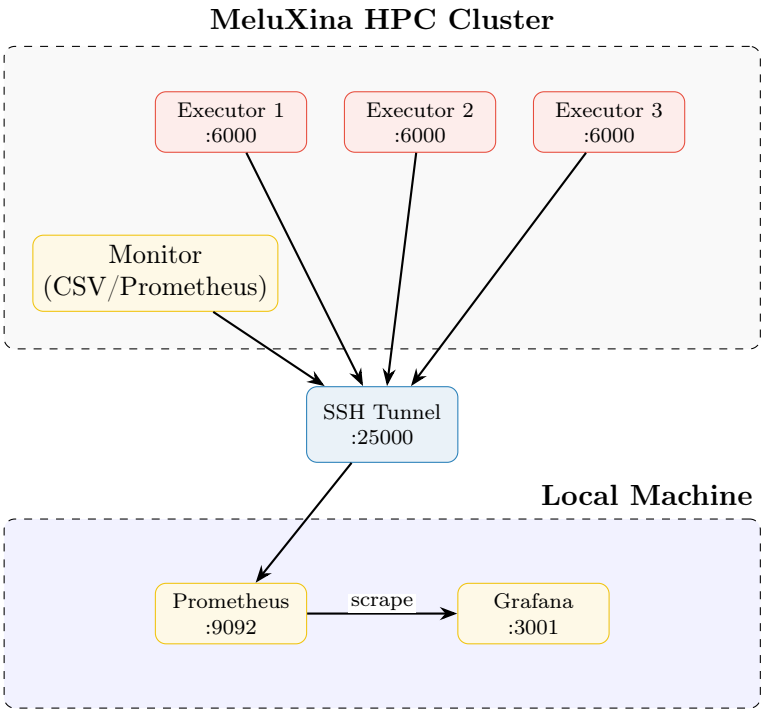

Figure 6: Metrics collection pipeline with SSH tunneling

3.5 Message Sequence Diagram

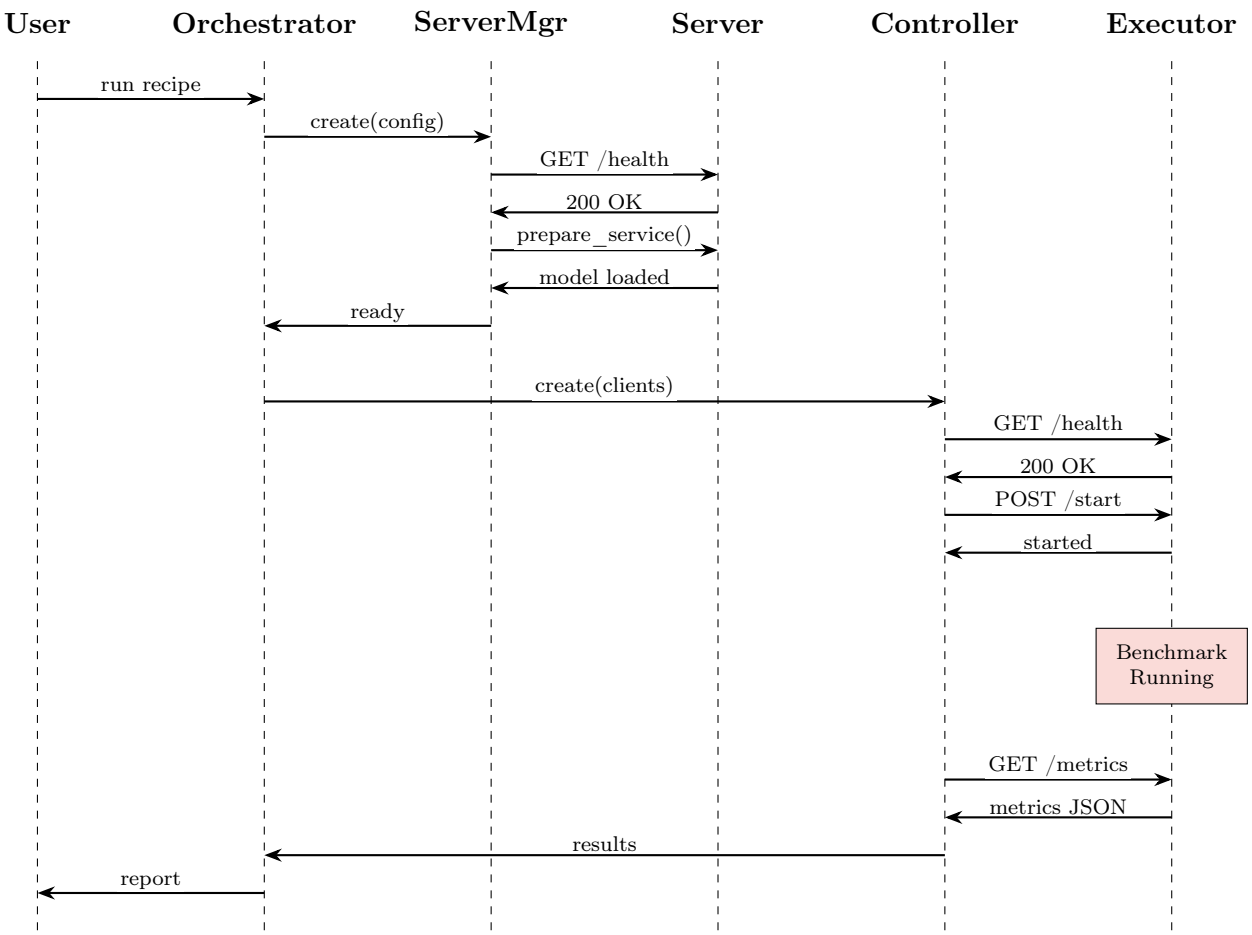

Figure 7: Message sequence for benchmark execution

4 Execution Flow

4.1 Seven-Phase Execution Model

The benchmark execution follows a structured seven-phase model:

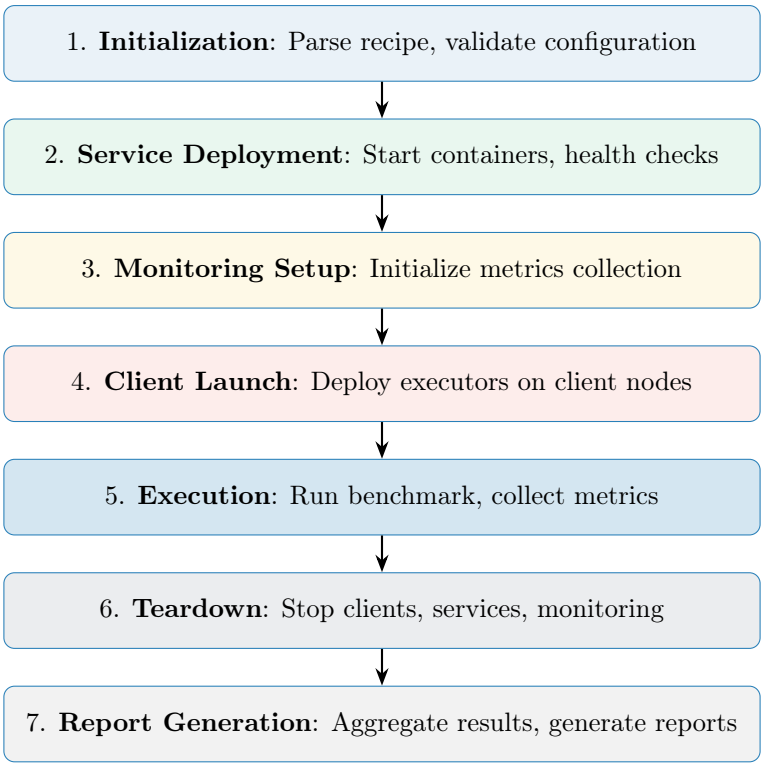

Figure 8: Seven-phase execution model

4.2 Detailed Phase Descriptions

4.2.1 Phase 1: Initialization

1. Load and parse YAML recipe file
2. Validate against JSON schema (`schemas/recipe-format.yaml`)
3. Expand parameter sweeps (Cartesian product)
4. Initialize logging and output directories

4.2.2 Phase 2: Service Deployment

1. Create ServerManager via ServiceFactory
2. Build endpoint list from server node hostnames
3. Poll health check endpoints with configurable timeout
4. Execute service preparation (model loading)

4.2.3 Phase 3: Monitoring Setup

PLACEHOLDER:

Giovanni's Section Please describe the monitoring setup phase in detail:

- Monitor initialization process
- Prometheus integration details
- Pushgateway configuration
- Grafana dashboard setup
- SSH tunnel requirements

4.2.4 Phase 4: Client Launch

1. Create WorkloadController via ServiceFactory
2. Verify executor health on all client nodes
3. Distribute workload configuration
4. Initialize thread pools on each executor

4.2.5 Phase 5: Execution

1. Warmup period (configurable duration)
2. Main benchmark execution
3. Continuous metrics collection
4. Real-time Prometheus scraping

4.2.6 Phase 6: Teardown

1. Send stop signals to all executors
2. Collect final metrics
3. Stop monitoring processes
4. Clean up resources

4.2.7 Phase 7: Report Generation

1. Aggregate metrics from all clients
2. Calculate statistics (mean, p50, p90, p99)
3. Generate CSV/JSON output files
4. Create summary report

5 Recipe Configuration System

5.1 Recipe Structure

Recipes are YAML files that declaratively define benchmark configurations:

```yaml
scenario: "experiment-name"
partition: "gpu"
account: "p200981"
qos: "default"

orchestration:
  mode: "slurm"
  total_nodes: 5
  node_allocation:
    servers:
      nodes: 2
    clients:
      nodes: 2
      clients_per_node: 10
    monitors:
      nodes: 1
  job_config:
    time_limit: "02:00:00"
    exclusive: true

resources:
  servers:
    gpus: 2
    cpus_per_task: 1
    mem_gb: 32
  clients:
    gpus: 0
    cpus_per_task: 2
    mem_gb: 16

workload:
  component: "inference"
  service: "ollama"
  duration: "2m"
  warmup: "1m"
  model: "llama2"
  clients_per_node: 10

servers:
  health_check:
    enabled: true
    timeout: 300
    interval: 5
    endpoint: "/api/tags"
  service_config:
    gpu_layers: 0

artifacts:
  containers_dir: "/path/to/containers/"
  service:
    path: "ollama_latest.sif"
  python:
    path: "python_3_12_3_v2.sif"

binds:
  - "/project/.ollama:/root/.ollama:rw"
  - "/project/scratch:/scratch:rw"
```

Listing 3: Complete recipe structure

5.2 Recipe Validation

PLACEHOLDER:

Laura's Section Please describe the recipe validation system in detail:

- JSON Schema validation implementation
- Validation rules and error messages
- Custom validators
- Schema versioning
- Interactive validation mode

5.3 Parameter Sweeps

The recipe system supports automatic parameter expansion:

```
workload:
  batch: [1, 4, 8]            # 3 values
  concurrency: [1, 8, 32]     # 3 values
  prompt_len: [128, 512]      # 2 values
  # Total trials: 3 x 3 x 2 = 18
```

Listing 4: Parameter sweep configuration

6 Distributed Benchmarking with Ray

6.1 Ray Cluster Architecture

For distributed vLLM deployments, the toolkit integrates with Ray for tensor and pipeline parallelism:

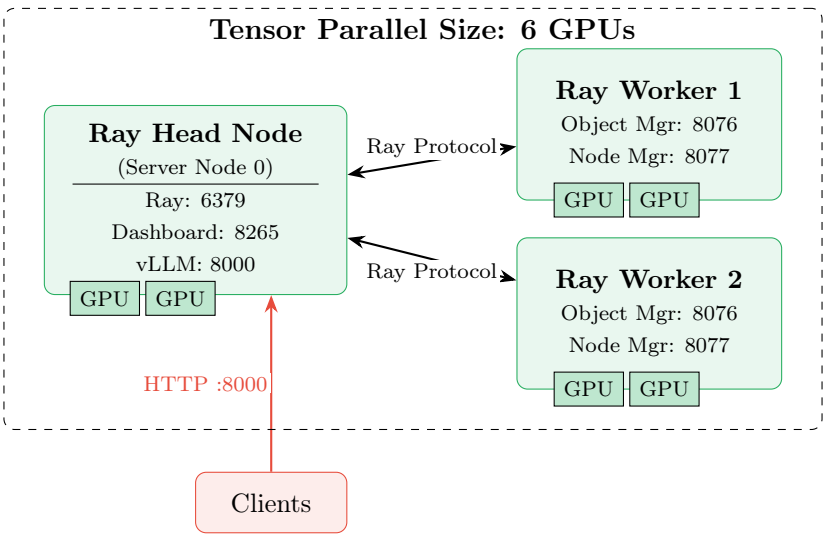

Figure 9: Ray cluster architecture for distributed vLLM

6.2 RayClusterManager

The `RayClusterManager` class handles Ray cluster lifecycle:

```python
class RayClusterManager:
    def start_head_node(self, port: int = 6379) -> bool:
        """Initialize Ray head node with ray start --head"""
        cmd = f"ray start --head --port={port}"
        # Execute and verify

    def connect_worker(self, head_address: str) -> bool:
        """Connect worker to existing Ray cluster"""
        cmd = f"ray start --address={head_address}"
        # Execute and verify

    def get_head_ip(self) -> str:
        """Auto-detect local IP for Ray communication"""
        # Network interface detection
```

Listing 5: RayClusterManager key methods

6.3 Distributed vLLM Configuration

```yaml
servers:
  service_config:
    distributed:
      enabled: true
      backend: "ray"
      tensor_parallel_size: 4
      pipeline_parallel_size: 1
      ray:
        dashboard_port: 8265
        object_manager_port: 8076
        node_manager_port: 8077
        num_cpus_per_node: 4
        num_gpus_per_node: 2
    max_model_len: 2048
    gpu_memory_utilization: 0.7
```

Listing 6: Distributed vLLM recipe configuration

7 Monitoring System

PLACEHOLDER:

Giovanni's Section Please provide comprehensive documentation of the monitoring system:

7.1 Monitoring Architecture

- Overall monitoring design
- Component interactions
- Data flow diagrams

7.2 Monitor Module Implementation

- monitor.py implementation details
- Metrics collection logic
- GPU/CPU/RAM monitoring
- Sampling intervals

7.3 Prometheus Integration

- Pushgateway setup
- Metrics format
- Scrape configuration
- Label management

7.4 Grafana Dashboards

- Dashboard design
- Panel configurations
- Query examples
- Alert setup (if any)

7.5 SSH Tunneling for Metrics

- Tunnel setup process
- Port forwarding configuration
- Troubleshooting guide

Please include:

- TikZ diagrams for architecture
- Code snippets for key implementations
- Configuration examples
- Screenshots or diagram of Grafana dashboard

8 Benchmarking Implementation

PLACEHOLDER:

Laura's Section Please provide comprehensive documentation of the benchmarking system:

8.1 Workload Executor Implementation

- Thread pool management
- Request generation
- Latency measurement
- Error handling

8.2 Ollama Benchmarking

- HellaSwag dataset integration
- Request format
- Response parsing
- Metrics collection

8.3 vLLM Benchmarking

- OpenAI-compatible API usage
- Streaming vs non-streaming
- Token counting
- Throughput calculation

8.4 Metrics Collection

- Latency percentiles (p50/p90/p99)
- Throughput (requests/second, tokens/second)
- Error rates
- Resource utilization

8.5 Load Patterns

- Constant load
- Poisson distribution
- Burst patterns

Please include:

- Code snippets for key implementations
- Diagrams showing request flow
- Example metrics output
- Performance considerations

9 Logging System

PLACEHOLDER:

Giulia's Section Please provide comprehensive documentation of the logging system:

9.1 Logging Architecture

- Overall logging design
- Log sources and destinations
- Aggregation strategy

9.2 BaseLogCollector

- Abstract interface design
- LogSource dataclass
- Method specifications

9.3 TailerLogCollector

- File tailing implementation
- Remote node log collection
- Log aggregation

9.4 Log Categories

- Application logs
- System logs (Slurm)
- Benchmark logs
- Infrastructure logs

9.5 Log Format and Storage

- Structured logging (JSON)
- Timestamps and correlation IDs
- Storage organization
- Retention policies

Please include:

- TikZ diagrams for log flow
- Code snippets for key implementations
- Example log entries
- Integration with other components

10 CLI and User Interface

10.1 benchmark_cli.py

The main CLI provides three primary commands:

Command	Arguments	Description
`list`	–	Display all available recipes with details
`create`	–	Interactive wizard for recipe creation
`run`	`-recipe PATH`	Deploy and run a benchmark recipe

Table 4: CLI commands

10.2 Orchestrator Arguments

```
python3 orchestrator.py \
  --server-nodes NODE [NODE ...]      # Required: Server hostnames
  --client-nodes NODE [NODE ...]      # Required: Client hostnames
  --workload-config-file PATH         # Required: Recipe file
  [--server-port PORT]                # Default: 11434/8000
  [--client-port PORT]                # Default: 5000
  [--timeout SECONDS]                 # Default: 600
  [--enable-monitoring]               # Enable metrics
  [--pushgateway-node NODE]           # For Prometheus
  [--monitor-interval SECONDS]        # Default: 5
  [--monitor-output PATH]             # Output file
```

Listing 7: Orchestrator command-line arguments

10.3 Interactive Recipe Creation

The CLI guides users through recipe creation:

1. Service selection (Ollama, vLLM, vLLM Distributed)
2. Scenario configuration (name, partition, account)
3. Node allocation (servers, clients, monitors)
4. Resource requirements (GPUs, CPUs, memory)
5. Workload parameters (model, duration, clients)
6. Container paths and bind mounts

11 Slurm Integration

11.1 SBATCH Script Generation

The toolkit generates Slurm batch scripts from recipes:

```
#!/bin/bash
#SBATCH --job-name=ollama-benchmark
#SBATCH --partition=gpu
#SBATCH --account=p200981
#SBATCH --nodes=5
#SBATCH --time=02:00:00
```

```
#SBATCH --exclusive

# Load modules
module load Apptainer

# Get node list
NODES=($(scontrol show hostnames $SLURM_JOB_NODELIST))
SERVER_NODES="${NODES[0]} ${NODES[1]}"
CLIENT_NODES="${NODES[2]} ${NODES[3]}"
ORCHESTRATOR_NODE="${NODES[4]}"

# Start servers
for node in $SERVER_NODES; do
    srun --nodes=1 --nodelist=$node \
        apptainer run --nv ollama.sif &
done

# Wait for servers
sleep 30

# Start client executors
for node in $CLIENT_NODES; do
    srun --nodes=1 --nodelist=$node \
        apptainer exec python.sif \
        python3 executor.py --port 6000 &
done

# Run orchestrator
python3 orchestrator.py \
    --server-nodes $SERVER_NODES \
    --client-nodes $CLIENT_NODES \
    --workload-config-file recipe.yaml
```

Listing 8: Generated SBATCH script structure

11.2 Resource Allocation

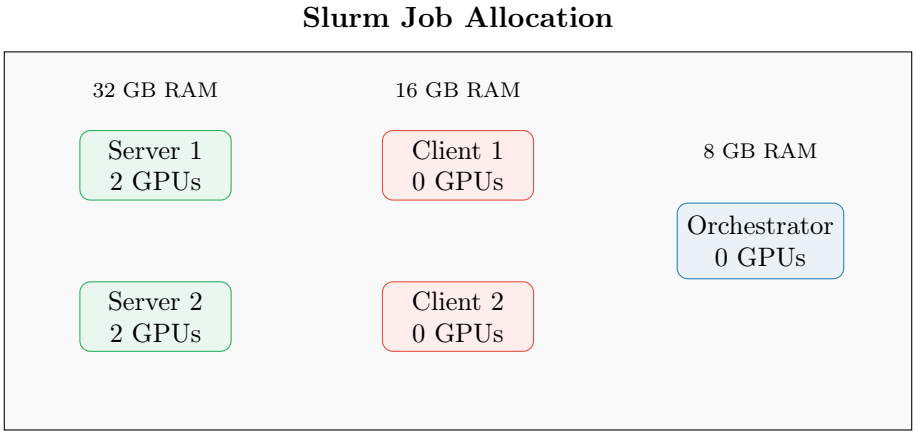

Figure 10: Slurm resource allocation example

12 Extensibility

12.1 Adding a New Service

To add a new inference service, implement four components:

1. **Server Manager**: Handle service lifecycle
2. **Workload Controller**: Coordinate clients
3. **Workload Executor**: Execute benchmarks
4. **Service Registration**: Register with factory

Files to Create/Modify

```
servers/myservice_server_manager.py
controller/myservice_workload_controller.py
executor/myservice_workload_executor.py
service_registry.py (update)
```

12.2 Custom Metrics

Extend the Monitor class for custom metrics:

```python
from prometheus_client import Gauge

class CustomMonitor(Monitor):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.custom_metric = Gauge(
            'custom_metric',
            'Description of custom metric'
        )

    def collect_custom(self):
        value = self._get_custom_value()
        self.custom_metric.set(value)
```

Listing 9: Custom metrics extension

13 Deployment and Operations

13.1 Prerequisites

Component	Requirement	Version
Python	Runtime	3.6+
Slurm	Job scheduler	Any
Apptainer	Container runtime	1.0+
Docker	Local monitoring	20.10+

Table 5: System prerequisites

13.2 Python Dependencies

```
pip install flask requests psutil prometheus_client pyyaml
```

Listing 10: Required Python packages

13.3 Container Setup

```
module load Apptainer

# Pull Ollama container
apptainer pull docker://ollama/ollama:latest

# Pull vLLM container
apptainer pull docker://vllm/vllm-openai:latest

# Pull Python container for clients
apptainer pull docker://python:3.12.3-slim
```

Listing 11: Building containers on MeluXina

14 Performance Considerations

PLACEHOLDER:

Performance Analysis - Pending Laura's Benchmarking Results This section will be completed once Laura's benchmarking work is finalized. It will include:

14.1 Bottleneck Analysis

- Latency breakdown for LLM inference
- GPU compute vs memory transfer analysis
- Network overhead measurements
- Tokenization performance impact

14.2 Optimization Recommendations

- Warmup period requirements
- Batch size tuning guidelines
- Tensor parallelism configuration
- Memory utilization optimization
- Client concurrency tuning

14.3 Scaling Analysis

- Single-node vs multi-node performance
- Strong scaling efficiency
- Weak scaling characteristics
- Resource utilization patterns

Please provide:

- Benchmark results from Ollama and vLLM tests
- Performance charts and graphs
- Latency distributions (p50/p90/p99)
- Throughput measurements
- Resource utilization data
- Optimization recommendations based on findings

15 Troubleshooting Guide

15.1 Common Issues

Issue	Symptom	Solution
Server health check fails	Timeout during startup	Verify port accessibility, check container logs
Client cannot connect	Connection refused	Check executor is running, verify port
No metrics in Grafana	Empty dashboard	Verify SSH tunnel, check Prometheus targets
Ray cluster fails	Workers not connecting	Check network ports, verify head IP

Table 6: Common issues and solutions

15.2 Debugging Commands

```
# Check server health
curl http://server-node:11434/api/tags

# Verify executor
curl http://client-node:6000/health

# Check Prometheus targets
curl http://localhost:9092/api/v1/targets

# View Ray cluster status
ssh head-node "ray status"
```

Listing 12: Useful debugging commands

16 Project Structure

```
hpc-benchmark-toolkit/
+-- src/
|   +-- benchmark/
|   |   +-- orchestrator.py
|   |   +-- service_factory.py
|   |   +-- service_registry.py
|   |   +-- servers/
|   |   |   +-- base_server_manager.py
|   |   |   +-- ollama_server_manager.py
|   |   |   +-- vllm_server_manager.py
|   |   |   +-- ray_cluster_manager.py
|   |   +-- workload/
|   |   |   +-- controller/
|   |   |   +-- executor/
|   |   +-- logging/
|   +-- benchmark_cli.py
|   +-- monitor/
|       +-- monitor.py
+-- monitoring/
|   +-- docker-compose.yml
|   +-- prometheus.yml
|   +-- grafana/
+-- schemas/
|   +-- recipe-format.yaml
+-- docs/
```

```
+-- diagrams/
```

Listing 13: Complete project structure

17 Division of Work

This section documents the contributions of each team member to the project.

17.1 Alberto Finardi

Role: System Architect and Service Implementation
Contributions:

- **Core Architecture Design**
  - Designed the overall system architecture
  - Implemented the Service Factory pattern
  - Created the modular component structure
- **Server Management**
  - Implemented `BaseServerManager` abstract class
  - Developed `OllamaServerManager` with health checks and model pulling
  - Developed `VllmServerManager` with distributed support
  - Created `RayClusterManager` for distributed vLLM
- **Workload Control**
  - Designed `BaseWorkloadController` interface
  - Implemented service-specific controllers
  - Developed HTTP-based client coordination
- **Workload Execution**
  - Created `BaseWorkloadExecutor` Flask server
  - Implemented REST API endpoints
  - Developed thread pool management
- **CLI Development**
  - Developed `benchmark_cli.py`
  - Implemented interactive recipe creation wizard
  - Created deployment and job submission logic
- **Orchestration**
  - Implemented main orchestrator logic
  - Developed seven-phase execution model
  - Created Slurm sbatch script generation
- **Documentation**
  - Wrote comprehensive README.md

  - Created API Reference documentation
  - Developed Developer Guide
  - Authored this technical report

- **Integration**
  - Integrated all components
  - Implemented service registry
  - Coordinated team contributions

17.2 Giovanni

Role: Monitoring

PLACEHOLDER:

Giovanni's Contributions Please list your specific contributions:

- **Monitor Module**
  - Implementation details of monitor.py
  - GPU/CPU/RAM metrics collection
  - CSV output generation
- **Prometheus Integration**
  - Pushgateway setup and configuration
  - Metrics format design
  - Prometheus client integration
- **Grafana Setup**
  - Docker Compose configuration
  - Dashboard design and implementation
  - Panel and query configuration
- **Documentation**
  - QUICKSTART.md guide
  - Monitoring section of this report

17.3 Laura

Role: Validation & Benchmarking

PLACEHOLDER:

Laura's Contributions Please list your specific contributions:

- **Recipe Validation**
  - JSON Schema design (recipe-format.yaml)
  - Validation logic implementation
  - Custom validators
  - Error message formatting
- **Benchmarking Logic**
  - Workload executor implementations
  - Request generation logic
  - Latency measurement
  - Metrics aggregation
- **Load Patterns**
  - Constant load implementation
  - Poisson distribution
  - Burst patterns
- **Documentation**
  - Recipe guide contributions
  - Benchmarking section of this report

17.4 Giulia

Role: Logging

PLACEHOLDER:

Giulia's Contributions Please list your specific contributions:

- **Logging Architecture**
  - Log collector design
  - Aggregation strategy
- **BaseLogCollector**
  - Abstract interface implementation
  - LogSource dataclass design
- **TailerLogCollector**
  - File tailing implementation
  - Remote log collection
- **Log Management**
  - Structured logging format
  - Storage organization
  - Correlation ID implementation
- **Documentation**
  - Logging section of this report

17.5 Contribution Summary

Team Member	Primary Area	Key Deliverables
Alberto Finardi	Architecture & Core	Orchestrator, Server Managers, Controllers, Executors, CLI, Documentation
Giovanni	Monitoring	Monitor module, Prometheus/-Grafana, Dashboards
Laura	Validation & Benchmarking	Recipe validation, Workload execution, Metrics
Giulia	Logging	Log collectors, Aggregation, Storage

Table 7: Team contribution summary

18 Conclusion

The HPC Benchmark Toolkit provides a comprehensive solution for benchmarking LLM inference services on HPC clusters. Key achievements include:

- **Modular Architecture**: Clean separation of concerns with factory pattern
- **Multi-Service Support**: Ollama, vLLM (single and distributed)
- **Real-Time Monitoring**: Full Prometheus/Grafana integration

- **Reproducibility**: YAML-based recipe system
- **Extensibility**: Easy addition of new services
- **HPC Integration**: Native Slurm and Apptainer support

18.1 Future Work

Potential enhancements include:

- Support for additional inference services (TensorRT, Triton)
- Kubernetes orchestration mode
- Automated performance regression testing
- Interactive web dashboard
- Multi-cluster support

A Port Reference

Port	Service	Description
11434	Ollama	Default Ollama API
8000	vLLM	Default vLLM API
5000/6000	Executor	Workload executor Flask
9091	Pushgateway	Prometheus Pushgateway
9092	Prometheus	Prometheus server
3001	Grafana	Grafana dashboard
6379	Ray	Ray cluster communication
8265	Ray Dashboard	Ray monitoring
8076	Ray Object Manager	Ray object store
8077	Ray Node Manager	Ray node management

Table 8: Complete port reference

B Environment Variables

Variable	Description	Default
PYTHONPATH	Python module path	–
MLUX_USER	MeluXina username	–
MLUX_ACCOUNT	MeluXina project account	–
MLUX_KEY	Path to SSH key	~/.ssh/id_ed25519_mlux

Table 9: Environment variables