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)

January 2025

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.

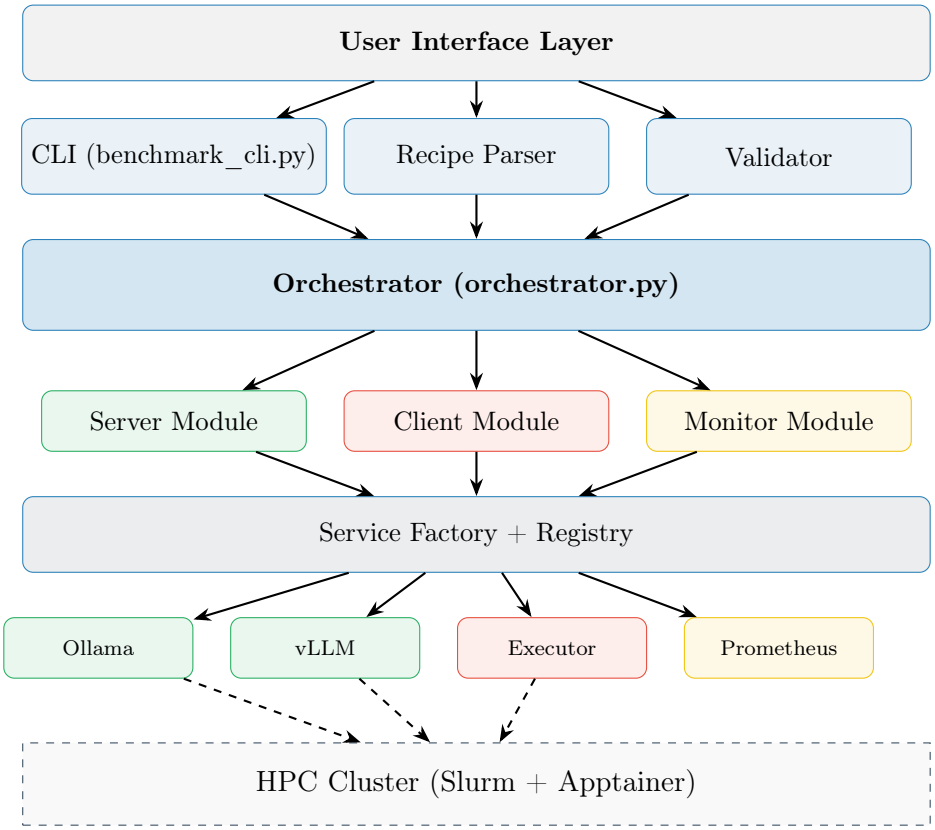


Figure 1: High-level system architecture

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:

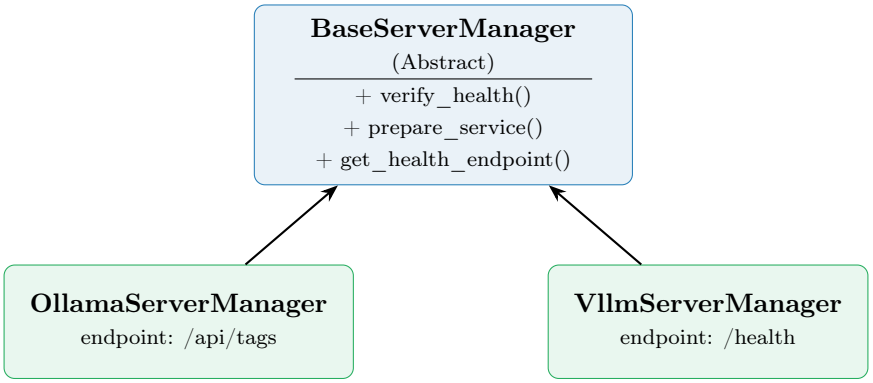


Figure 2: Template Method Pattern in Server Managers

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:

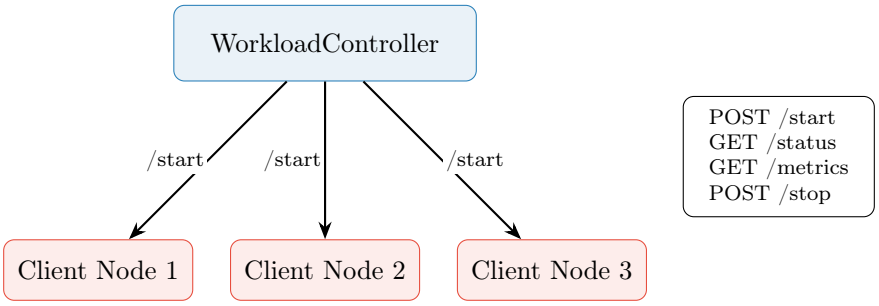


Figure 3: Workload Controller to Client communication

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

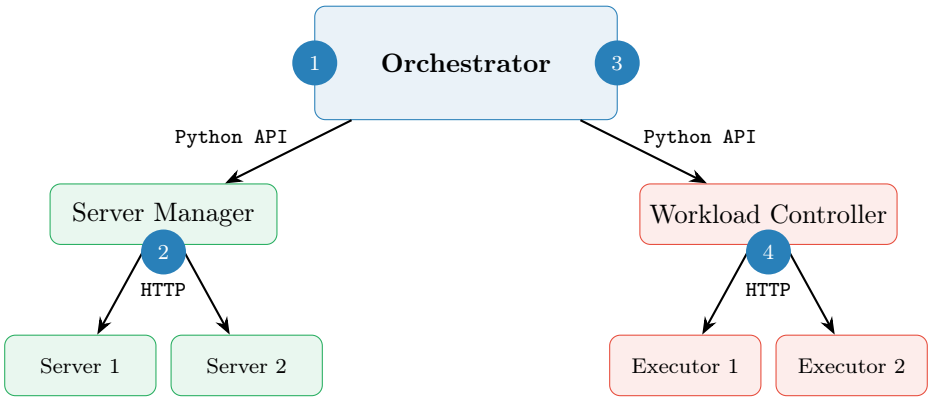


Figure 4: Control plane communication flow

3.3 Data Plane Communication

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

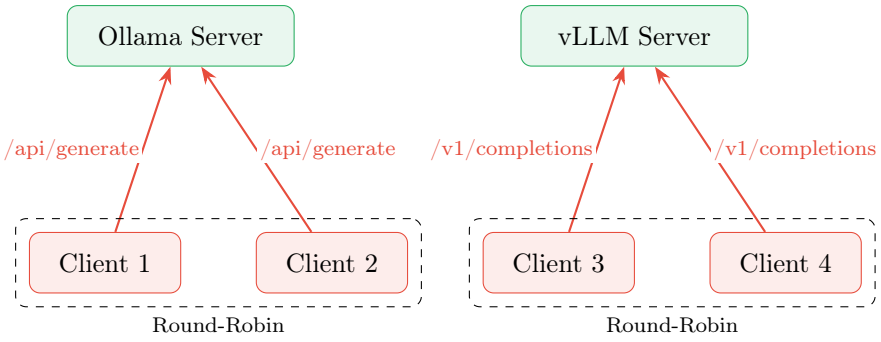


Figure 5: Data plane: Client to Server inference requests

3.4 Metrics Collection Pipeline

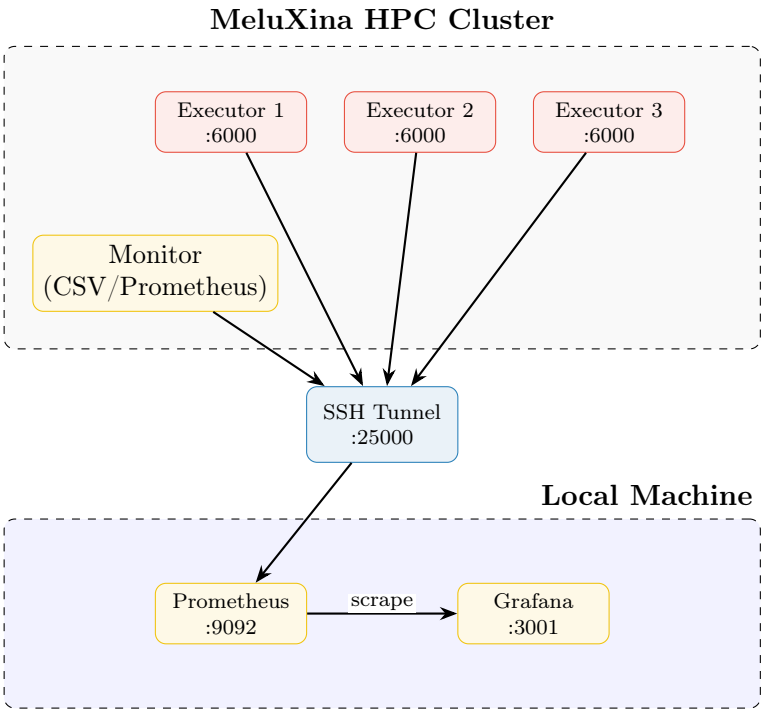


Figure 6: Metrics collection pipeline with SSH tunneling

3.5 Message Sequence Diagram

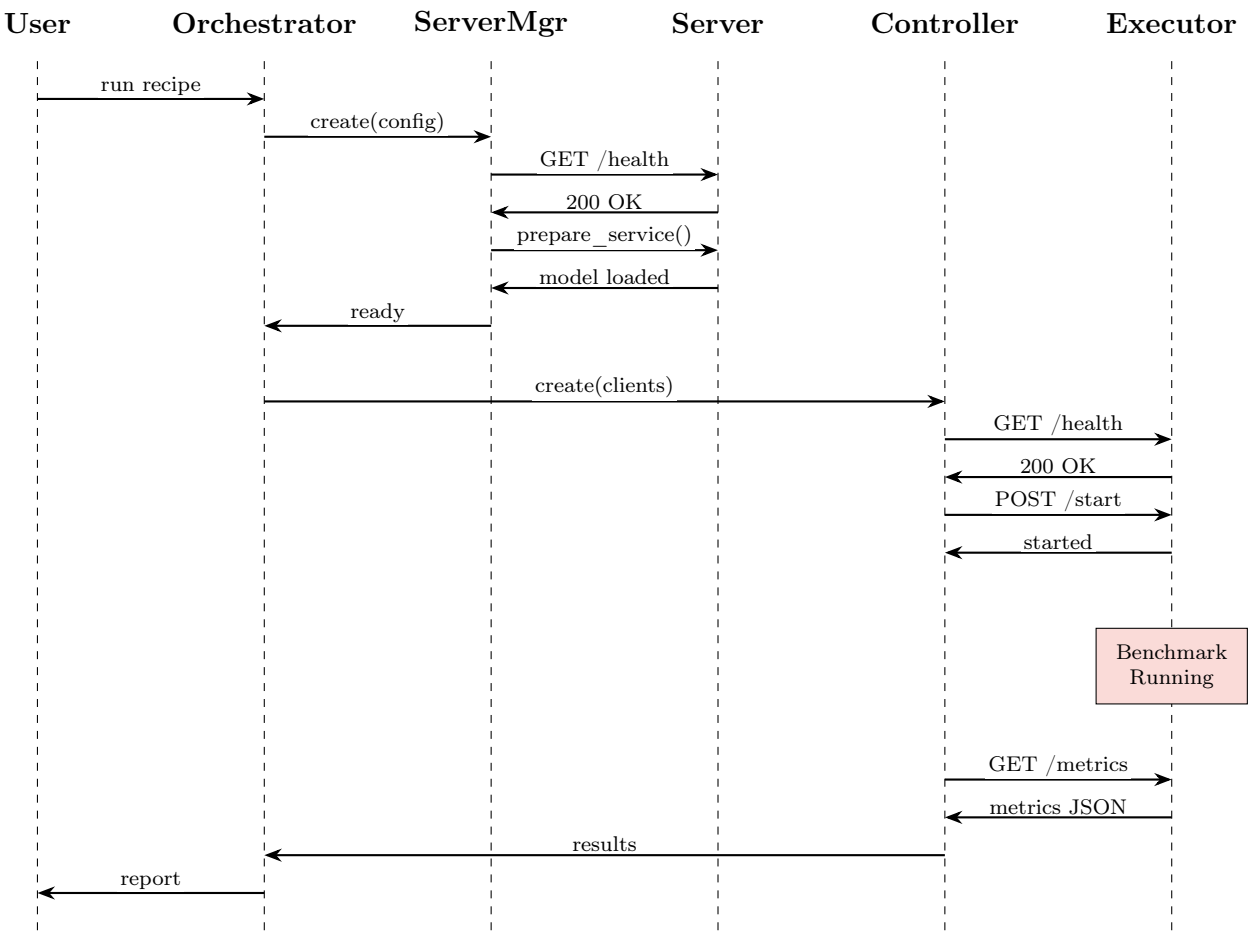


Figure 7: Message sequence for benchmark execution

4 Execution Flow

4.1 Seven-Phase Execution Model

The benchmark execution follows a structured seven-phase model:

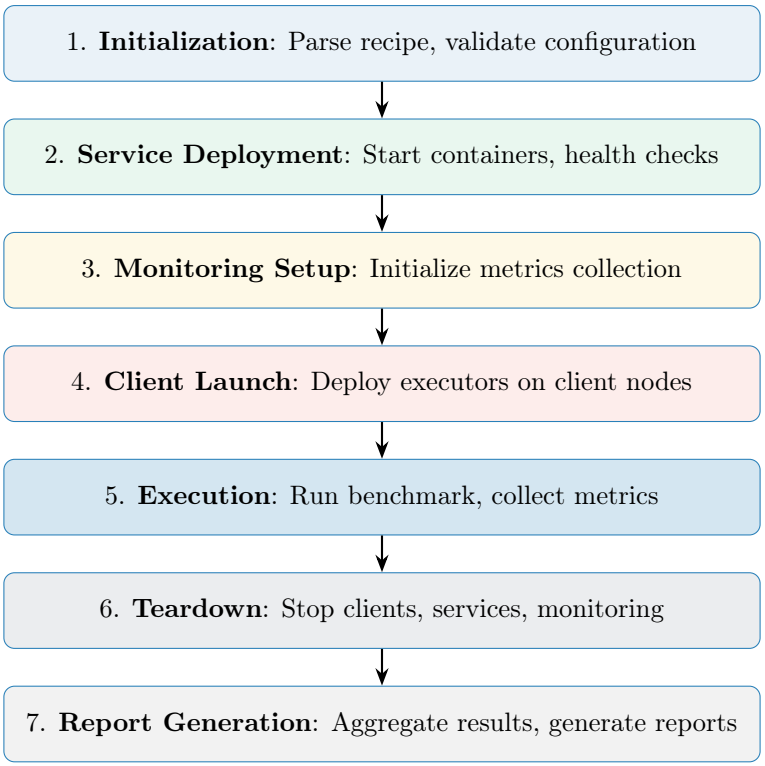


Figure 8: Seven-phase execution model

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

This phase sets up the system to track what's happening during the benchmark. It prepares both the HPC cluster and personal laptop to collect performance data:

1. **Start Monitoring**: Turn on the monitor tool that will watch CPU, GPU, and memory usage
2. **Choose What to Track**: Decide which metrics to collect (CPU usage, GPU usage, memory, etc.)

3. **Connect to Pushgateway**: Set up connection to Prometheus Pushgateway (stores the data)
4. **Create Folders**: Make directories to save the collected data as CSV files
5. **Run Monitor in Background**: Start monitoring that runs continuously and takes a measurement every 1 second by default
6. **Check Grafana Dashboard**: Make sure the dashboard on your laptop is ready to see the data

Main Settings:

- `monitor_interval`: How often to check the system (example: every 1 second)
- `prometheus_push_interval`: How often to send data to storage (example: every 15 seconds)
- `pushgateway_url`: Where to send the data (example: `http://mel2109:9091`)
- `output_file`: File name to save results (example: `benchmark_metrics.csv`)

Parts of the Monitoring System:

- **Prometheus on Your Laptop** (port 9092): Collects data every 15 seconds from the HPC cluster
- **Grafana on Your Laptop** (port 3001): Shows graphs and charts of the performance data
- **Pushgateway on HPC**: A storage box that receives metrics from the cluster
- **SSH Tunnels**: Secret paths that let your laptop see the HPC metrics safely

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
```

```
47
48 artifacts:
49   containers_dir: "/path/to/containers/"
50   service:
51     path: "ollama_latest.sif"
52   python:
53     path: "python_3_12_3_v2.sif"
54
55 binds:
56   - "/project/.ollama:/root/.ollama:rw"
57   - "/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:

```
1 workload:
2   batch: [1, 4, 8]            # 3 values
3   concurrency: [1, 8, 32]     # 3 values
4   prompt_len: [128, 512]      # 2 values
5   # 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:

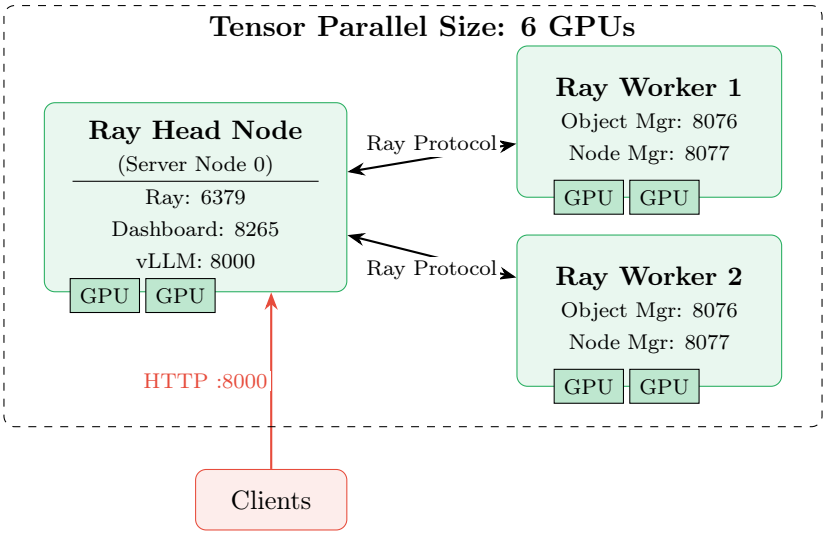


Figure 9: Ray cluster architecture for distributed vLLM

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

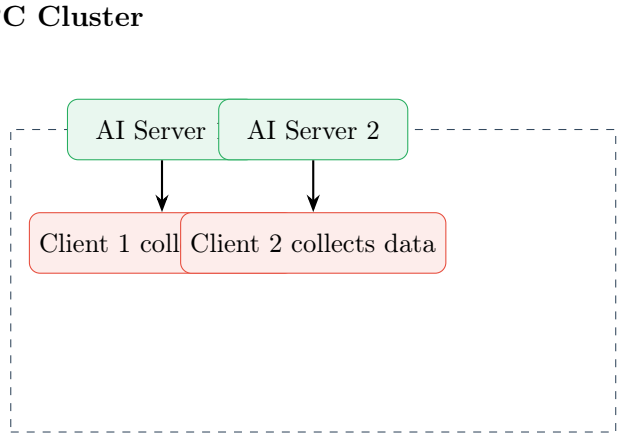
The monitoring system watches what happens during the benchmark and shows you the results. It has two main parts: one on the HPC cluster that collects data, and one on your laptop that shows graphs of this data.

7.1 How Monitoring Works

Simple Overview:

1. Client computers on HPC measure CPU, GPU, and memory usage every second
2. These measurements are stored and sent to a storage server
3. Your laptop connects to the HPC cluster through a secure tunnel
4. Your laptop collects these measurements every 15 seconds
5. Your laptop shows you graphs of all this data in Grafana

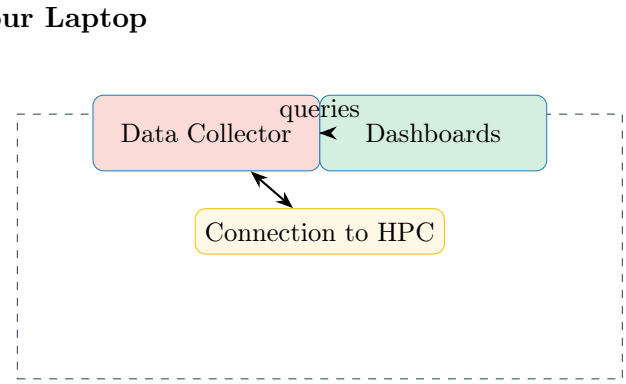
7.2 Data Collection on HPC



On the HPC cluster, each client computer has a small Python program that:

1. Measures system performance (CPU, GPU, memory) every 1 second
2. Stores these measurements in memory
3. Provides them through a web endpoint on port 6000

7.3 Data Viewing on Your Laptop



On the laptop, there are two programs:

1. **Prometheus** (on port 9092): A program that collects data from HPC every 15 seconds through the secure tunnel
2. **Grafana** (on port 3001): A program that creates nice graphs from the collected data

7.4 Step-by-Step: From HPC to Your Dashboard

Step 1: HPC Client Collects Data

On client computers on HPC, a Python monitor continuously runs:

```
# Data is available here on HPC (inside the cluster)
http://client-node:6000/metrics/prometheus

# The data looks like:
ollama_throughput_rps 578           # 578 requests per second
ollama_workload_running 1           # 1 = workload is running
ollama_request_latency_seconds 0.015  # Each request takes 15ms
```

Step 2: Create a Tunnel to HPC

Since your laptop cannot directly see HPC, a secure tunnel is created:

```
# Example: Create tunnel from your port 25000 to HPC client-node port 6000
ssh -N -L 25000:client-node:6000 meluxina &

# Now you can see HPC data from your laptop at:
http://localhost:25000/metrics/prometheus
```

Step 3: Prometheus Reads the Data

A program called Prometheus on your laptop reads from this tunnel:

```
# Prometheus configuration file tells it where to look
# It checks every 15 seconds
scrape_configs:
  - job_name: 'ollama'
    static_configs:
      - targets: ['localhost:25000', 'localhost:25001']
```

Step 4: Grafana Shows You the Graphs

Grafana reads the stored data from Prometheus and shows nice graphs:

- A number showing if the workload is running (0 means stopped, 1 means running)
- A line graph showing requests per second over time

- A line graph showing response time over time
- Counters showing total successes and failures

7.5 What the Dashboards Show

Ollama Performance Dashboard
This shows performance of the Ollama AI model:

- **Is it Running?**: Shows 0 (stopped) or 1 (running)
- **Total Requests**: How many AI requests have been made (goes up over time)
- **Speed per Computer**: Shows requests/sec on each computer (line graph)
- **Total Speed**: Shows all requests/sec combined (big number)
- **Response Time**: How long each request takes (line graph in milliseconds)
- **Success vs Failure**: Comparison of successful vs failed requests (bar chart)
- **Error Percentage**: What percent of requests failed (line graph)

vLLM Performance Dashboard
Same metrics of the Ollama but for the vLLM AI model, with extra panels:

- **Tokens per Request**: How many words the AI generates per request
- **Cache Usage**: How full the GPU cache is (as a percentage)

7.6 SSH Tunneling for Metrics

- **Ollama Dashboard**: 13 panels monitoring Ollama workload metrics
  - Workload Status (gauge: 0=idle, 1=running)
  - Requests Total (counter)
  - Throughput by Host (timeseries)
  - Total Throughput (stat panel in RPS)
  - Request Latency (timeseries in ms)
  - Success vs Failed Requests (bar chart)
  - Error Rate Over Time (percentage)
  - Plus additional CPU/GPU/Memory panels
- **vLLM Dashboard**: Same layout with vLLM-specific metrics
  - vllm_workload_running
  - vllm_throughput_rps
  - vllm_request_latency_seconds
  - vllm_tokens_per_request
  - vllm_cache_usage

Example Grafana Queries:

```
# Current workload status
ollama_workload_running

# Throughput aggregated across hosts
sum(ollama_throughput_rps)

# Latency p95
histogram_quantile(0.95, ollama_request_latency_seconds)

# Error rate
100 * (sum(rate(ollama_errors_total[5m])) / sum(rate(ollama_requests_total[5m]))
    )
```

7.7 SSH Tunneling for Metrics Access

1. **Find client nodes from job output**:

```
ssh meluxina "scontrol show job JOBID | grep StdOut"
ssh meluxina "cat /path/to/logs/*.out" | grep "Client nodes"
```

2. **Open SSH tunnels**:

```
# Ollama clients
ssh -N -L 25000:mel2120:6000 meluxina &
ssh -N -L 25001:mel2148:6000 meluxina &

# vLLM clients
ssh -N -L 25002:mel2142:6000 meluxina &
ssh -N -L 25003:mel2185:6000 meluxina &
```

3. **Verify metrics endpoint**:

```
curl http://localhost:25000/metrics/prometheus | head -5
```

4. **Access Grafana**:

```
open http://localhost:3001  # admin / admin
```

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

9.1 Overview

The logging system provides comprehensive log aggregation and management across distributed HPC benchmark executions. It collects, aggregates, and stores logs from multiple nodes in a centralized, structured format suitable for analysis and debugging.

9.1.1 Design Goals

The logging system was designed with the following objectives:

- **Multi-Node Aggregation**: Collect logs from distributed server and client nodes into centralized files
- **Structured Output**: Provide both human-readable and machine-parseable (JSON) log formats
- **Extensibility**: Support multiple log collection implementations through abstract interfaces
- **Real-Time Collection**: Capture logs continuously during benchmark execution
- **Minimal Overhead**: Use lightweight file tailing to avoid performance impact

9.2 Architecture

The logging system employs the Strategy pattern to provide a flexible, extensible architecture. Figure 10 illustrates the high-level architecture.

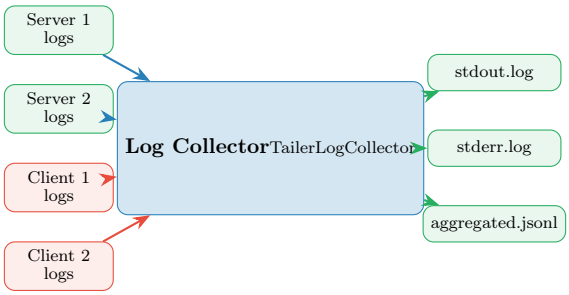


Figure 10: Logging system architecture

9.2.1 Strategy Pattern Implementation

The logging system uses the Strategy pattern to allow different log collection implementations while maintaining a consistent interface. Figure 11 shows the class hierarchy.

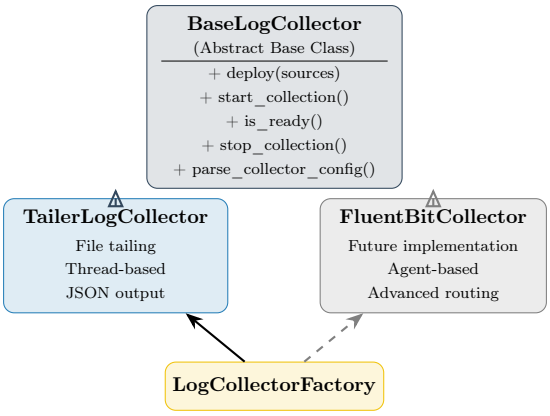


Figure 11: Strategy pattern in logging system

9.3 Core Components

9.3.1 LogSource Dataclass

The `LogSource` dataclass provides a structured representation of a single log source:

```
@dataclass
class LogSource:
    """
    Represents a single source of logs to collect.

    Attributes:
        node: Node hostname where logs are generated
        component: Type of component ("server", "client", "monitor")
        container_name: Name/ID of the container producing logs
    """
    node: str
    component: str
    container_name: str
```

Listing 7: LogSource dataclass definition

Example usage:

```
# Create a log source for a server node
log_source = LogSource(
    node="mel2120",
    component="server",
    container_name="ollama_0"
)

# Create a log source for a client node
client_source = LogSource(
    node="mel2148",
    component="client",
    container_name="python_executor_1"
)
```

Listing 8: LogSource example

9.3.2 BaseLogCollector Interface

The `BaseLogCollector` abstract class defines the interface that all log collector implementations must follow:

```python
class BaseLogCollector(ABC):
    """Abstract base class for managing log collection operations."""

    @abstractmethod
    def deploy(self, sources: List[LogSource]) -> bool:
        """
        Deploy log collection infrastructure.

        Args:
            sources: List of log sources to collect from

        Returns:
            True if deployment succeeded, False otherwise
        """
        pass

    @abstractmethod
    def start_collection(self) -> bool:
        """
        Start collecting logs from all deployed sources.

        Returns:
            True if collection started successfully
        """
        pass

    @abstractmethod
    def is_ready(self) -> bool:
        """
        Check if log collector is ready to collect logs.

        Returns:
            True if ready, False otherwise
        """
        pass

    @abstractmethod
    def stop_collection(self) -> Dict[str, Any]:
        """
        Stop log collection and finalize outputs.

        Returns:
            Dictionary with collection summary/metadata
        """
        pass
```

Listing 9: BaseLogCollector abstract methods

9.4 TailerLogCollector Implementation

The `TailerLogCollector` provides a lightweight, thread-based implementation that tails log files and aggregates them into centralized outputs.

9.4.1 Architecture

The TailerLogCollector operates in five distinct phases:

1. **Deploy Phase**: Create output files (stdout.log, stderr.log, aggregated.jsonl)
2. **Start Phase**: Launch one thread per log source, create "loggers_ready" flag

3. **Tail Phase**: Each thread monitors its log file, reading new lines continuously
4. **Process Phase**: Add timestamp, node, and component metadata to each log line
5. **Stop Phase**: Stop threads, close files, return summary with line counts and file paths

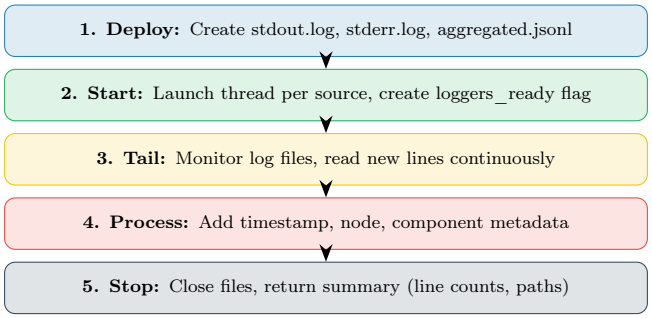


Figure 12: TailerLogCollector workflow

9.4.2 Configuration

TailerLogCollector is configured via the recipe YAML file:

```yaml
logging:
  type: "tailer"
  create_jsonl: true
  flush_interval: 5

  outputs:
    stdout: "stdout.log"
    stderr: "stderr.log"
    aggregated: "aggregated.jsonl"
```

Listing 10: TailerLogCollector configuration in recipe

Configuration parameters:

Parameter	Default	Description
type	"tailer"	Collector implementation type
create_jsonl	true	Enable structured JSON output
flush_interval	5	Buffer flush interval in seconds
outputs.stdout	"stdout.log"	Path for stdout aggregation
outputs.stderr	"stderr.log"	Path for stderr aggregation
outputs.aggregated	"aggregated.jsonl"	Path for JSON Lines output

Table 4: TailerLogCollector configuration parameters

9.4.3 Implementation Details

Thread-Based Tailing:

```python
def _tail_log_file(self, source: LogSource, log_file: Path):
    """Tail a log file and write to aggregated outputs."""
    print(f"Starting tail for {log_file}")

    # Wait for file to exist
    while not log_file.exists() and not self.stop_event.is_set():
        time.sleep(1)
```

```
    if self.stop_event.is_set():
        return

    try:
        with open(log_file, 'r') as f:
            while not self.stop_event.is_set():
                line = f.readline()

                if line:
                    self._process_log_line(source, line.rstrip('\n'))
                else:
                    time.sleep(0.1)

    except Exception as e:
        print(f"Error tailing {log_file}: {e}")
```

Listing 11: Thread-based log tailing implementation

Log Line Processing:

```
def _process_log_line(self, source: LogSource, line: str):
    """Process a single log line from a source."""
    try:
        timestamp = datetime.utcnow().isoformat() + 'Z'

        # Write to aggregated stdout with metadata
        log_entry = f"[{timestamp}] [{source.node}] [{source.component}] {line}\
    n"
        self.stdout_handle.write(log_entry)

        # Write to structured .jsonl
        if self.create_jsonl and self.jsonl_handle:
            jsonl_entry = {
                "timestamp": timestamp,
                "node": source.node,
                "component": source.component,
                "message": line
            }
            self.jsonl_handle.write(json.dumps(jsonl_entry) + '\n')

    except Exception as e:
        print(f"Error processing log line: {e}")
```

Listing 12: Log line processing with metadata

9.5 Log Output Formats

The logging system produces three output files, each serving a different purpose:

9.5.1 stdout.log - Plain Text with Metadata

Human-readable format with timestamp, node, and component prefixes:

```
[2026-01-12T14:32:15Z] [mel2120] [server] Model loaded successfully
[2026-01-12T14:32:16Z] [mel2148] [client] Started 100 worker threads
[2026-01-12T14:32:17Z] [mel2120] [server] Ready to accept connections
[2026-01-12T14:32:18Z] [mel2148] [client] Benchmark warmup phase started
```

Listing 13: stdout.log example

9.5.2 stderr.log - Error Messages

Contains error messages and warnings (typically empty for successful runs):

```
[2026-01-12T14:35:20Z] [mel2148] [client] Connection timeout to server
[2026-01-12T14:35:21Z] [mel2148] [client] Retrying connection (attempt 2/3)
```

Listing 14: stderr.log example

9.5.3 aggregated.jsonl - Structured JSON Lines

Machine-parseable format for automated analysis:

```
{"timestamp": "2026-01-12T14:32:15Z", "node": "mel2120", "component": "server",
    "message": "Model loaded successfully"}
{"timestamp": "2026-01-12T14:32:16Z", "node": "mel2148", "component": "client",
    "message": "Started 100 worker threads"}
{"timestamp": "2026-01-12T14:32:17Z", "node": "mel2120", "component": "server",
    "message": "Ready to accept connections"}
```

Listing 15: aggregated.jsonl example

9.6 Multi-Node Log Aggregation

One of the key challenges in HPC benchmarking is collecting logs from distributed nodes and merging them into a coherent timeline. The logging system achieves this through timestamp-based aggregation.

9.6.1 Aggregation Strategy

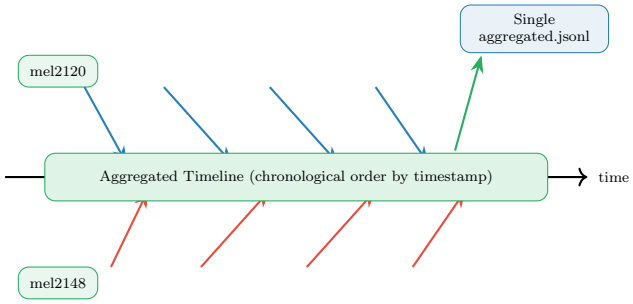


Figure 13: Multi-node log aggregation with timestamps

Each thread writes logs with timestamps, enabling natural chronological merging. The aggregated files maintain temporal order across all nodes without requiring complex synchronization.

9.7 Integration with Orchestrator

The logging system integrates seamlessly into the seven-phase benchmark execution model:

Phase	Logging Action	Description
1. Initialization	–	Logging system is initialized
2. Service Deployment	–	Log sources are identified
3. Monitoring Setup	**Deploy & Start**	Log collector deploys and starts collection
4. Client Launch	–	Client logs begin flowing
5. Execution	**Active Collection**	Continuous log capture during benchmark
6. Teardown	**Stop & Finalize**	Collection stops, files closed, summary returned
7. Report Generation	–	Logs available for analysis

Table 5: Logging system integration with execution phases

9.7.1 Orchestrator Integration Code

```python
# Phase 3: Monitoring Setup - Deploy logging
log_sources = [
    LogSource(node=node, component="server", container_name=f"ollama_{i}")
    for i, node in enumerate(server_nodes)
]
log_sources += [
    LogSource(node=node, component="client", container_name=f"client_{i}")
    for i, node in enumerate(client_nodes)
]

log_collector = LogCollectorFactory.create(
    collector_type="tailer",
    config=logging_config,
    output_dir=experiment_dir
)

# Deploy and start collection
log_collector.deploy(log_sources)
log_collector.start_collection()

# Wait for ready flag
while not log_collector.is_ready():
    time.sleep(1)

# ... benchmark execution ...

# Phase 6: Teardown - Stop logging
summary = log_collector.stop_collection()
print(f"Collected {summary['stdout_lines']} log lines")
```

Listing 16: Logging integration in orchestrator

9.8 Validation and Testing

A comprehensive validation script ensures logging correctness across distributed executions.

9.8.1 Automated Tests

The `validate_logging.sh` script performs 16 automated tests:

Test Category	Tests Performed
File Existence	Verify stdout.log, stderr.log, aggregated.jsonl exist
File Size	Verify files have content (>0 bytes)
Content Validation	Check line counts (>50 lines expected)
Format Validation	Verify timestamp format, JSON validity
Multi-Node Aggregation	Verify logs from multiple nodes present
Component Coverage	Verify both server and client logs present
Line Count Consistency	Compare stdout and jsonl line counts

Table 6: Automated logging validation tests

9.8.2 Validation Script Example

```
#!/bin/bash
# Run validation on experiment directory
./validate_logging.sh experiments/logging-test_20251228_123456

# Output example:
# [1] Testing: stdout.log exists... PASS
# [2] Testing: stderr.log exists... PASS
# [3] Testing: aggregated.jsonl exists... PASS
# [4] Testing: loggers_ready flag exists... PASS
# [5] Testing: stdout.log has content... PASS
# [6] Testing: aggregated.jsonl has content... PASS
# [7] Testing: stdout.log line count (>50 expected)...
#     Lines: 1247
#     PASS
# [8] Testing: aggregated.jsonl line count (>50 expected)...
#     Lines: 1247
#     PASS
# ...
# Total tests: 16
# Passed: 16
# Failed: 0
# ALL TESTS PASSED!
```

Listing 17: Validation script usage

9.9 Log Analysis Tools

The structured JSON format enables automated log analysis:

```
import json
from collections import Counter
from datetime import datetime

# Analyze log distribution
def analyze_logs(jsonl_file):
    with open(jsonl_file) as f:
        logs = [json.loads(line) for line in f]

    # Count by node
    node_counts = Counter(log['node'] for log in logs)

    # Count by component
    component_counts = Counter(log['component'] for log in logs)

    # Timeline analysis
    timestamps = [datetime.fromisoformat(log['timestamp'].rstrip('Z'))
```

```
                    for log in logs]
    duration = (max(timestamps) - min(timestamps)).total_seconds()

    return {
        'total_logs': len(logs),
        'nodes': dict(node_counts),
        'components': dict(component_counts),
        'duration_seconds': duration,
        'logs_per_second': len(logs) / duration if duration > 0 else 0
    }

# Example output:
# {
#   'total_logs': 1247,
#   'nodes': {'mel2120': 623, 'mel2148': 624},
#   'components': {'server': 623, 'client': 624},
#   'duration_seconds': 125.3,
#   'logs_per_second': 9.95
# }
```

Listing 18: Log analysis example

9.10 Future Enhancements

Potential enhancements to the logging system include:

- **FluentBitCollector**: Agent-based log collection with advanced routing capabilities
- **Log Compression**: Automatic compression of log files to save storage space
- **Real-Time Streaming**: Stream logs to centralized logging services (e.g., Elasticsearch)
- **Log Filtering**: Filter logs by severity level or component type
- **Automatic Alerting**: Trigger alerts based on error patterns in logs
- **Log Retention Policies**: Automatic archiving or deletion of old logs

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 7: 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 19: 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 20: Generated SBATCH script structure

11.2 Resource Allocation

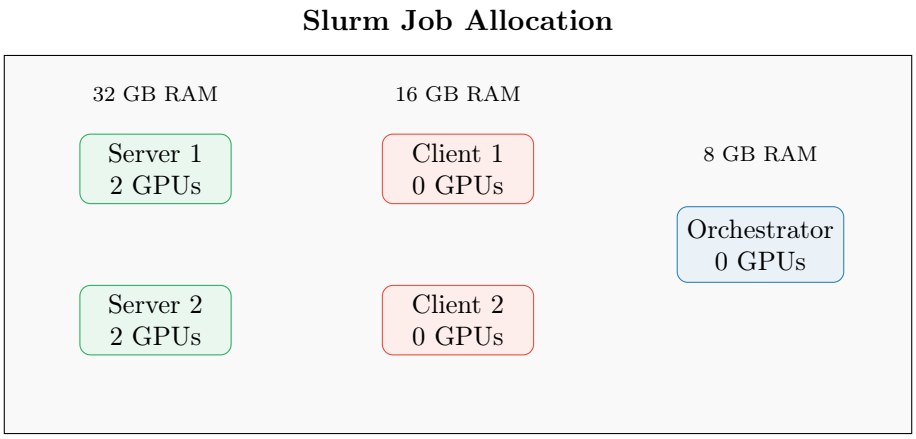


Figure 14: Slurm resource allocation example

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 21: 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 8: System prerequisites

13.2 Python Dependencies

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

Listing 22: 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 23: 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 9: 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 24: 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 25: 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, Infrastructure, and Core Implementation
Contributions:

- **Core Infrastructure**
  - Designed and implemented the overall system architecture
  - Created the Service Factory pattern for extensibility
  - Built the modular component structure
  - Implemented service registry for dynamic component loading
- **Server Management**
  - Implemented `BaseServerManager` abstract class
  - Developed `OllamaServerManager` with health checks and model pulling
  - Developed `VllmServerManager` with distributed support
  - Created `RayClusterManager` for distributed vLLM deployments
- **Workload Controllers**
  - Designed `BaseWorkloadController` interface
  - Implemented service-specific controllers (Ollama, vLLM)
  - Developed HTTP-based client coordination protocol
- **Workload Executors**
  - Created `BaseWorkloadExecutor` Flask server framework
  - Implemented REST API endpoints for workload management
  - Developed thread pool management for concurrent requests
  - Built metrics collection infrastructure
- **Benchmarking Framework**
  - Implemented core benchmarking logic
  - Developed request generation and load patterns
  - Created latency measurement and metrics aggregation
  - Built Ollama and vLLM specific benchmark implementations
- **CLI Development**
  - Developed `benchmark_cli.py` with three main commands
  - Implemented interactive recipe creation wizard

  - Created deployment and job submission logic
  - Built recipe listing and management features

- **Orchestration**
  - Implemented main orchestrator logic (`orchestrator.py`)
  - Developed seven-phase execution model
  - Created Slurm sbatch script generation
  - Built node allocation and resource management

- **Basic Logging Infrastructure**
  - Set up initial logging framework
  - Implemented basic log output and formatting
  - Created log directory structure

- **Documentation**
  - Wrote comprehensive README.md
  - Created API Reference documentation
  - Developed Developer Guide
  - Authored this technical report

- **Integration**
  - Integrated all team contributions
  - Coordinated component interfaces
  - Performed system testing and debugging

17.2 Giovanni

Role: Monitoring Integration
Contributions:

- **Prometheus Integration**
  - Configured Prometheus for metrics collection
  - Set up Docker container for Prometheus server

- **Grafana Dashboards**
  - Created Grafana dashboards for Ollama and vLLM metrics visualization
  - Configured Docker container for Grafana

17.3 Laura

Role: Recipe Validation and Benchmark Execution
Contributions:

- **Recipe Validation System**
  - Designed JSON Schema for recipe validation (`schemas/recipe-format.yaml`)
  - Implemented validation logic and error handling

  - Created custom validators for service-specific configurations
  - Developed user-friendly error message formatting
  - Built schema versioning support

- **Benchmark Execution**
  - Ran and validated Ollama benchmarks on MeluXina
  - Ran and validated vLLM benchmarks (single and distributed)
  - Tested parameter sweep configurations
  - Verified metrics collection and reporting

- **Documentation**
  - Recipe validation section of this report
  - Benchmarking implementation section of this report

17.4 Giulia

Role: Logging System
Contributions:

- **Logging Architecture**
  - Designed overall logging architecture
  - Defined log sources and destinations
  - Created aggregation strategy for distributed logs

- **BaseLogCollector**
  - Implemented abstract interface for log collection
  - Designed `LogSource` dataclass
  - Defined method specifications for collectors

- **TailerLogCollector**
  - Implemented file tailing for real-time log collection
  - Built remote node log collection via SSH
  - Created log aggregation from multiple sources

- **Log Management**
  - Implemented structured logging format (JSON)
  - Added timestamps and correlation IDs
  - Organized storage structure
  - Defined retention policies

- **Documentation**
  - Logging system section of this report

17.5 Contribution Summary

Team Member	Primary Area	Key Deliverables
Alberto Finardi	Infrastructure & Core	Orchestrator, Server Managers, Controllers, Executors, Benchmarks, CLI, Basic Logging, Documentation
Giovanni	Monitoring	Prometheus/Grafana integration, Dashboards
Laura	Validation	Recipe validation, Benchmark execution
Giulia	Logging	Log collectors, Aggregation

Table 10: 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 11: 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 12: Environment variables