Cloud Computing (UE22CS351B)
Mini Project

Serverless Function Execution Platform

Documentation

Team Number : 6
Team Members :

Name	**SRN**
Mohammed Ashfaq Ali	PES2UG22CS316
Nishant Mazumder	PES2UG22CS367
Mir Abbas Hussain	PES2UG22CS312
Mohit Prasad Singh	PES2UG22CS320

Project Overview

The Serverless Function Platform is a system designed to emulate AWS Lambda functionality, allowing users to deploy and execute code on-demand without managing server infrastructure. This platform supports Python and JavaScript functions, executes them in isolated environments, applies resource constraints, and provides comprehensive metrics and monitoring.

Design Architecture

The platform follows a modular architecture with the following key components:

- **API Server**: Implemented with FastAPI to handle function management and execution requests
- **Execution Engine**: Supports two virtualization technologies (Docker and gVisor)
- **Container Pool**: Pre-warms containers for improved performance
- **Metrics Manager**: Collects and aggregates execution metrics
- **Web UI**: Provides a dashboard for function management and monitoring

Week 1: Project Setup and Core Infrastructure

Task 1: Project Planning and Environment Setup

Design Decisions:

- Used a layered architecture to separate concerns and improve maintainability
- Implemented a factory pattern for runtime selection (Docker/gVisor)
- Selected FastAPI for the backend API due to its performance and async support
- Chose Docker as the primary virtualization technology for its widespread adoption and robust tooling

Implementation:

- Created project directory structure with clear separation of backend, frontend, and supporting components
- Set up GitHub repository with CI/CD pipeline through GitHub Actions
- Configured project dependencies in requirements.txt
- Developed environment configuration for local development and testing

Task 2 : Backend API Foundation

Design Decisions:

- Used FastAPI for creating RESTful API endpoints
- Implemented an in-memory data store for function metadata as a simple solution for prototype development
- Defined a clear interface for function execution that abstracts away the underlying virtualization technology

Implementation:

- Created basic API server in main.py
- Implemented CRUD operations for function management:
  - POST /functions - Create/update functions
  - GET /functions - List all stored functions
  - GET /functions/{function_id} - Retrieve a specific function
  - DELETE /functions/{function_id} - Delete a function
- Defined Pydantic models for request/response validation

Function Storage Schema:

```
Function {
    id: string              # Unique identifier
    name: string            # Display name
    code: string            # Function code
    function_name: string   # Entry point function name (default: handler)
    description: string     # Optional function description
    language: enum          # "python" or "javascript"
    timeout: int            # Maximum execution time in seconds
    memory: int             # Memory allocation in MB
    created_at: timestamp   # Creation timestamp
    updated_at: timestamp   # Last update timestamp
}
```

Task 3 : First Virtualization Technology (Docker)

Design Decisions:

- Created separate Docker images for Python and JavaScript runtimes
- Implemented function execution through container isolation
- Used volume mounting for securely injecting function code
- Applied resource constraints (CPU, memory, networking) for security and fair resource allocation

Implementation:

- Developed DockerRuntime class in docker_runtime.py
- Created Docker images for both Python and JavaScript with appropriate runtime handlers
- Added timeout enforcement mechanism to prevent long-running functions
- Implemented secure code injection through temporary files and volume mounting
- Added error handling and result parsing

Week 2 : Enhanced Execution and Second Virtualization Technology

Task 1 : Execution Engine Improvements

Design Decisions:

- Implemented a container pool to reduce cold start latency
- Used a factory pattern to abstract runtime selection
- Created a queue for container management

Implementation:

- Developed `ContainerPool` class in container_pool.py
- Implemented container pre-warming mechanism

- Added container reuse logic to improve performance
- Created metrics tracking for warm vs. cold starts

Task 2 : Second Virtualization Technology (gVisor)

Design Decisions:

- Selected gVisor as the second virtualization technology for its enhanced security while maintaining compatibility with Docker images
- Implemented a fallback mechanism to Docker if gVisor is not available
- Used the same container images for both technologies to ensure consistent execution

Implementation:

- Created `GVisorRuntime` class in runtime_factory.py
- Added detection for gVisor availability
- Implemented runtime switching logic based on user preference and availability
- Conducted performance comparisons between Docker and gVisor runtimes

Task 3 : Metrics Collection

Design Decisions:

- Implemented a central metrics manager to collect and aggregate function execution data
- Designed a thread-safe approach for metrics collection
- Created flexible metrics aggregation by runtime, language, and function

Implementation:

- Developed `MetricsManager` class in metrics_manager.py
- Added execution tracking for response times, status, and resource usage
- Implemented aggregation mechanisms for different dimensions (runtime, language)

- Created API endpoints to expose metrics data to the frontend

Week 3 : Enhanced Execution and Second Virtualization Technology

Task 1 : Basic Frontend

Design Decisions :

- Used HTML/CSS/JavaScript for a lightweight frontend that can be served directly from the FastAPI server
- Implemented an interface with separate spaces for function execution, management, and metrics
- Designed responsive layouts to work across different device sizes

Implementation:

- Created index.html as the main application entry point and implemented function creation and execution panels
- Added dynamic content loading through JavaScript
- Created function selection and management interfaces

Task 2 : Monitoring Dashboard

Design Decisions:

- Used simple yet effective visualizations for metrics display
- Implemented real-time metrics updates using AJAX
- Created both summary and detailed metrics views

Implementation:

- Developed dashboard UI in index.html with metrics tabs

- Created visualization components for execution statistics
- Implemented system status displays
- Added detailed execution history tables

Task 3 : Integration and Polishing

Design Decisions:

- Focused on robust error handling throughout the application
- Added comprehensive testing to ensure reliability
- Created detailed documentation for users and developers

Implementation:

- Integrated all components into a cohesive system
- Conducted end-to-end testing with test scripts
- Added static file serving for web UI
- Created comprehensive documentation
- Performed bug fixes and optimization

Technical Challenges and Solutions

Challenge 1: gVisor Setup and Compatibility Issues

Issue: Setting up gVisor proved challenging due to compatibility issues with different environments. Initial attempts to set up gVisor in Windows Subsystem for Linux (WSL) weren't successful despite having Docker WSL integration enabled.

Investigation: After examining the gVisor documentation more closely, it was determined that gVisor requires specific kernel features and configurations that weren't fully available in the WSL environment.
Additionally, attempts on existing Linux VMs also faced compatibility issues due to conflicts with existing Docker configurations and kernel parameters.

Solution: A clean Linux virtual machine was provisioned specifically for gVisor testing. The installation process was followed exactly as documented in the official gVisor documentation, ensuring all dependencies were properly installed from scratch. To accommodate users without gVisor access, a fallback mechanism was implemented to transparently use standard Docker when gVisor is unavailable, allowing the platform to function across different environments while still leveraging gVisor's security benefits where available.

Challenge 2: Container Pool Reliability Issues

Issue: During testing, the container pool mechanism exhibited stability issues. Containers in the pool would sometimes be in an unusable state when retrieved for function execution, resulting in execution failures. This led to a pattern where the system continuously created new containers rather than reusing existing ones, defeating the purpose of the container pool.

Investigation: Through extensive logging and debugging, several issues were identified:

1. Containers weren't given sufficient time to fully initialize before being added to the pool
2. The health checking mechanism wasn't robust enough to detect improperly initialized containers
3. Container status wasn't being properly verified before execution attempts

Solution:

1. Increased container initialization time from 3 to 6 seconds to ensure complete readiness
2. Implemented a more robust health checking system that verifies container status more thoroughly
3. Added an automatic container recovery mechanism that attempts to restart containers in bad states
4. Implemented a tiered verification approach that checks container status at multiple points in the execution flow
5. Enhanced error handling to properly remove problematic containers from the pool and create new ones when needed

Performance Considerations

Following optimizations were made in the system :

- **Reduced cold start times** through container pooling
- **Resource efficiency** by applying appropriate constraints to containers
- **Scalability** through a modular design that separates concerns
- **Security** by isolating function execution in containers with limited privileges

Performance testing showed that warm starts were approximately 60-70% faster than cold starts, demonstrating the effectiveness of the container pooling mechanism.

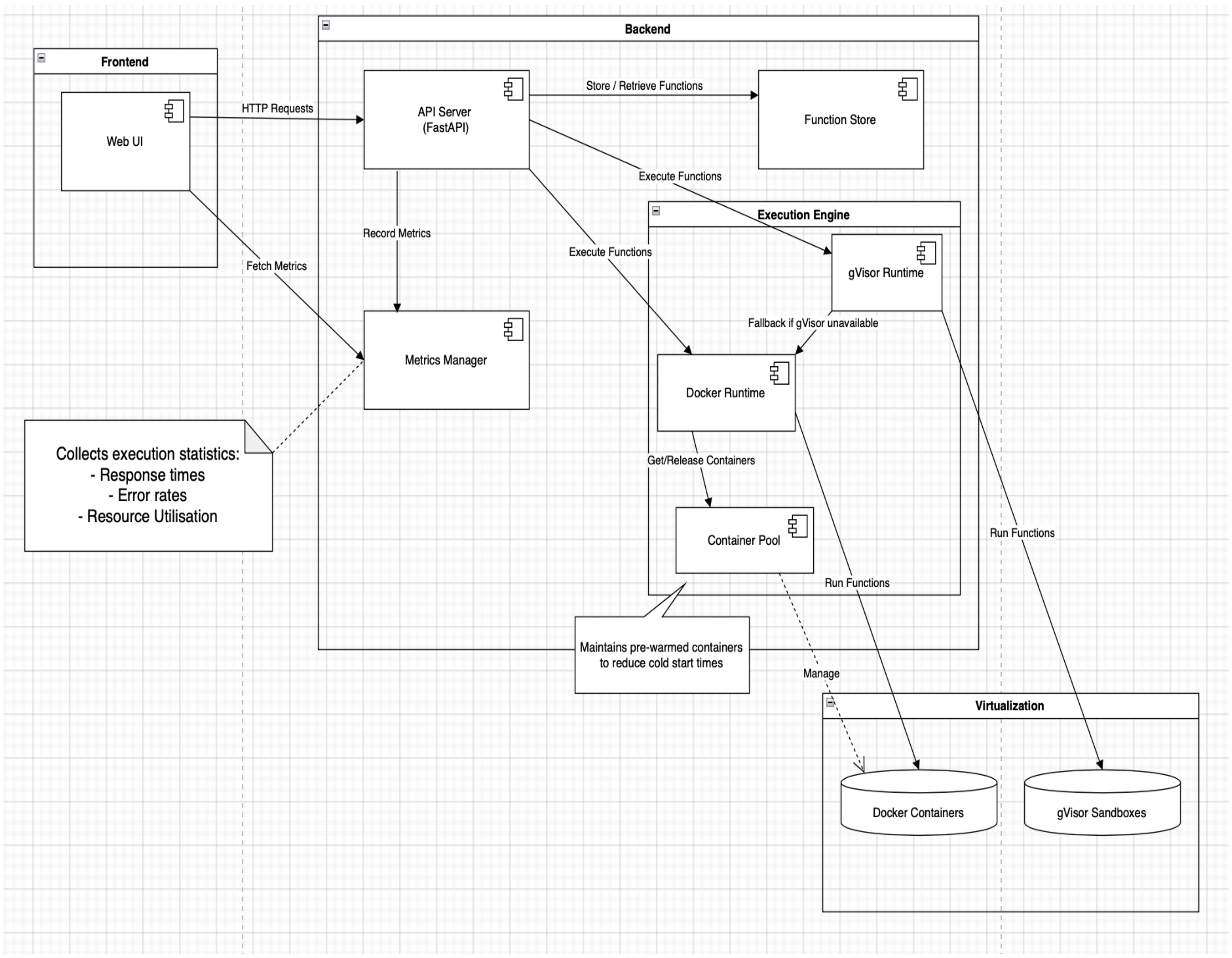
Future Enhancements

1. **Database Integration**: Replace in-memory storage with a persistent database
2. **Authentication and Multi-tenancy**: Add user authentication and function isolation
3. **Advanced Resource Management**: Implement auto-scaling based on demand
4. **Function Versioning**: Add support for multiple versions of functions
5. **Additional Language Support**: Extend support to other programming languages

Conclusion

The Serverless Function Platform successfully meets the project requirements, providing a functional
serverless execution environment with multiple runtime options, comprehensive monitoring, and an intuitive user interface. The modular architecture ensures extensibility, while the implementation of container pooling and gVisor integration demonstrates advanced concepts in serverless computing.

System Design


Frontend
Web UI
HTTP Requests
Fetch Metrics
Backend
API Server (FastAPI)
Store / Retrieve Functions
Function Store
Execute Functions
Record Metrics
Execute Functions
Execution Engine
gVisor Runtime
Fallback if gVisor unavailable
Docker Runtime
Metrics Manager
Collects execution statistics:
- Response times
- Error rates
- Resource Utilisation
Get/Release Containers
Container Pool
Run Functions
Run Functions
Maintains pre-warmed containers to reduce cold start times
Manage
Virtualization
Docker Containers
gVisor Sandboxes