System Documentation

Instructions for the Teaching Assistant

1. Clone the repository and ensure Docker and Docker Compose are installed
2. Navigate to the project directory
3. Run docker-compose build to build the services
4. Run docker-compose up -d to start the system
5. Access the application at http://localhost:8197
6. Default credentials: user/test@123

Development Platform Details

- Operating System: Linux-based system
- Docker version: 24.0.0 or higher
- Docker Compose version: 2.20.0 or higher

CI/CD Pipeline Description

Version Management

- GitLab for version control
- Main branch protected, code in project branch for it
- Commit messages follow conventional commits format

Building Tools

- Docker for containerization
- Docker Compose for multi-container orchestration
- Python 3.9 for service1
- Node.js 16 for service2
- Nginx for reverse proxy

Testing

Tools:

- pytest for Python service testing
- GitLab CI for pipeline execution
- Nginx configuration testing

Test Cases:

1. **test_state_management_feature**
    - Tests basic state management (GET and PUT /state).
    - Verifies state transitions (e.g., to RUNNING).
2. **test_request_handling_in_paused_state**
    - Tests request handling in PAUSED state.
    - Ensures requests are blocked (HTTP 503).
3. **test_run_log_format**
    - Tests the /run-log endpoint.
    - Verifies log format: YYYY-MM-DDTHH.MM:SS.sssZ: OLD_STATE->NEW_STATE.
4. **test_request_endpoint**
    - Tests the /request endpoint.
    - Verifies response format (text/plain) and system info (e.g., Disk Space, IP).
5. **test_shutdown_state (Commented Out because it actually shuts down containers)**
    - Tests SHUTDOWN state behavior.
    - Verifies requests are blocked and services become unavailable.

Packing

- Docker images built for each service
- Multi-stage builds for optimized image size
- Shared network configuration
- Volume mounts for persistence

Deployment

1. Pipeline Stages:

```
stages:
 - build
 - test
 - deploy
```

2. Deployment Process:

  - Docker images built with no-cache option
  - Services started with docker-compose, locally

Example Pipeline Runs

Successful Pipeline Run

```
$ gitlab-ci pipeline success

Running pipeline...

✓ build:service1

✓ build:service2

✓ test:integration

✓ deploy:staging

Pipeline succeeded
```

Failed Pipeline Run

```
$ gitlab-ci pipeline failure

Running pipeline...

✓ build:service1

✓ build:service2

✗ test:integration

  Error: Test case "state_management" failed

  - Expected state: RUNNING

  - Actual state: INIT

Pipeline failed
```

Reflections

Main Learnings

1. Microservices Architecture
    - Service isolation and communication
    - State management across distributed systems
    - Load balancing considerations
2. CI/CD Implementation
    - Pipeline configuration
    - Test automation
    - Deployment strategies
3. Docker Best Practices
    - Multi-stage builds
    - Network configuration

Difficulties Encountered

1. State Management Complexity
    - Handling distributed state
    - Race conditions in state transitions
    - Test case reliability
2. Service Communication
    - Initial setup of inter-service communication
    - Debugging network issues
    - Health check timing
3. Pipeline Configuration
    - GitLab CI configuration
    - Test environment setup
    - Pipeline performance optimization

Problem

The start.sh script, which works perfectly fine in a local Unix environment, fails to execute correctly in the **Windows PowerShell GitLab Runner** environment. The issues observed are:

1. **Syntax Errors**:

   - Errors like : not found | /start.sh: 3: and Syntax error: word unexpected (expecting "do") occur because the script is written for a Unix shell (#!/bin/sh), but the GitLab Runner uses **Windows PowerShell**, which is incompatible with Unix shell scripts.

2. **Behavior with Commented Script**:
   - When the start.sh script is **commented out**, the containers shuts down correctly apart from the nginx one because it doesn't attempt to execute the script.
   - When the start.sh script is **enabled**, the nginx container fails unexpectedly because the script either doesn't run or doesn't handle the logic correctly in the Windows environment.

3. **Local vs. Remote CI Runner**:
   - The script works fine **locally** because the local environment uses a Unix shell (e.g., Bash).
   - The script fails in the **remote CI runner** because it uses Windows PowerShell, which is incompatible with Unix shell scripts.

Improvement Suggestions

1. State Management
   - Consider using a dedicated state store (e.g., Redis)
   - Implement event-driven architecture
   - Add more robust error handling
2. Testing
   - Add more unit tests
   - Improve test isolation
   - Implement contract testing
3. Monitoring
   - Add centralized logging
   - Implement metrics collection
   - Enhanced error tracking

Effort Estimation

Total Hours: 60

Breakdown:

- Initial Setup and Planning: 7 hours
- Service Implementation: 30 hours
- CI/CD Pipeline: 8 hours
- Testing: 10 hours
- Documentation: 5 hours