Testbed Design Specification: Robust-ML-Testbed

Version: 1.0 **Date:** 2025-05-27

Table of Contents

Testbed Design Specification: Robust-ML-Testbed 1
1. Introduction 1
2. System Architecture 1
2.1 Component Details 1
2.2 Data Flow 1
3. Module Descriptions 2
3.1. Frontend (Web UI) 2
3.2. Node.js Backend (API Gateway/File Handler) 3
3.3. Flask Backend (ML Processing) 3
3.4. MySQL Database 3
3.5. Docker Environment 4
4. Technology Stack 4
5. Design Rationale and Alternatives 4
5.1. Architecture: Microservices (Multi-Container) Approach 4
5.2. Database: MySQL 5
5.3. Deployment: Docker & Docker Compose 5
5.4. File Handling: Shared Docker Volume 5

1. Introduction

This document outlines the design specifications for the Robust-ML-Testbed project. The goal of this project is to create a platform for evaluating the robustness of machine learning models against various adversarial attacks. The system allows users to upload their models and datasets, configure and run attack simulations, and view the results, ultimately generating a comprehensive assessment report.

2. System Architecture

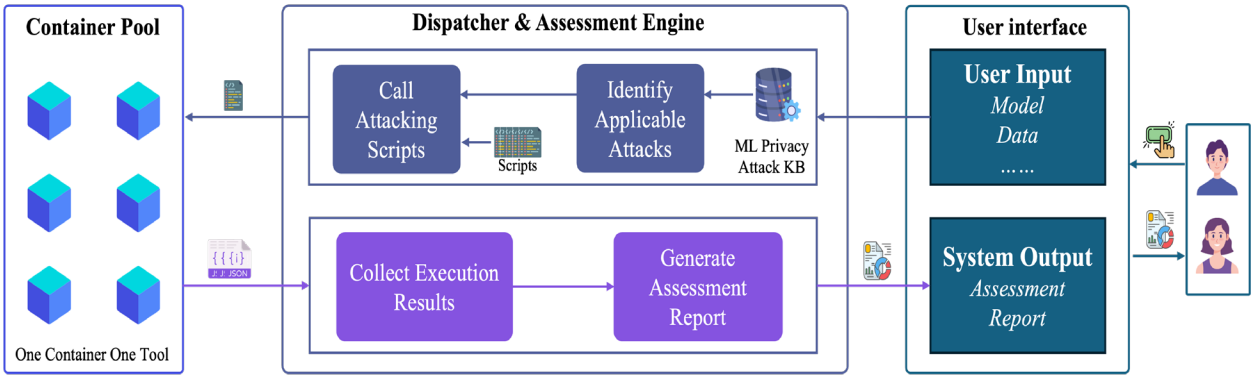

Fig.1 System Architecture Diagram

2.1 Component Details

The system employs a multi-container architecture orchestrated by Docker Compose, comprising the following main components:

1. **Frontend (Web UI):** The user interface for interacting with the testbed.
2. **Node.js Backend (API Gateway/File Handler):** Handles client requests, manages file uploads, identifies applicable attacks based on the knowledge base, and orchestrates calls to the ML Flask Backend.
3. **Flask Backend (ML Processing):** Executes the core machine learning model loading, attack simulations, and analysis using corresponding Python scripts.
4. **MySQL Database(ML Privacy Attack DB):** Structured knowledge base about systematically cataloging attacks and defense methods, which is used by the **Dispatcher & Assessment Engine** component.
5. **Docker Environment:** Provides the containerized infrastructure for deployment and inter-service communication, including different ML tools with pre-installed libraries and environment.

2.2 Data Flow

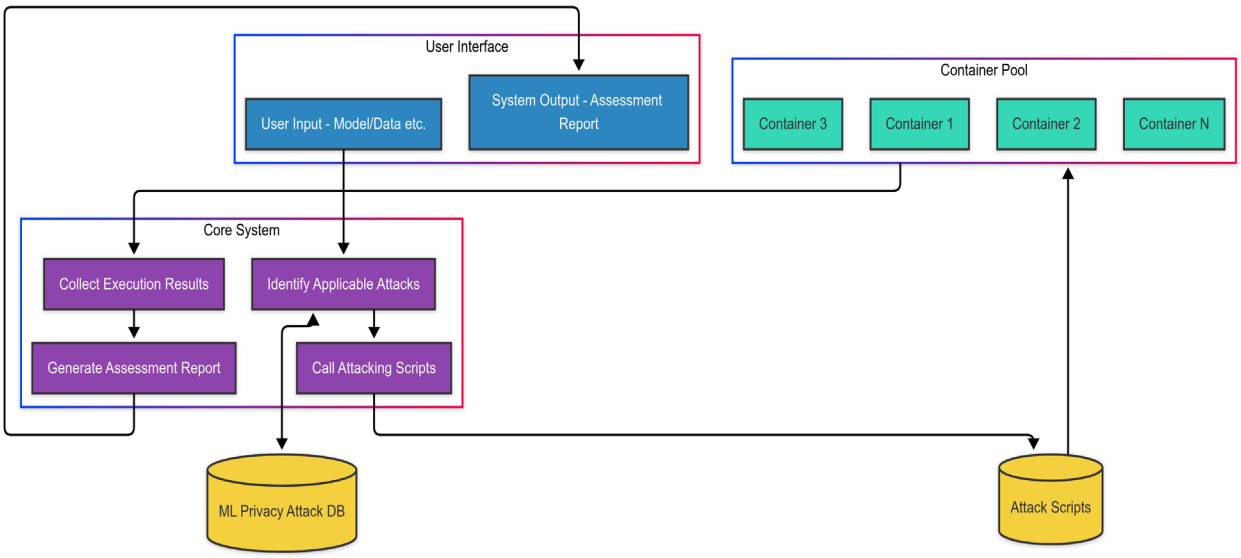

Fig.2 Data Flow Diagram

1. User interacts with the Frontend (e.g., uploads model/data, configures test).
2. Frontend sends requests to the Node.js Backend API.
3. Node.js Backend handles file uploads, storing them in a persistent volume accessible by other services. It validates requests and forwards processing tasks (like attack execution) to the Flask Backend API.
4. Flask Backend retrieves necessary files (model, data) from the shared volume, performs the ML computations (loading models, running attacks using libraries like TensorFlow Privacy), and writes result files to the shared volume.
5. Node.js Backend may query the MySQL Database or retrieve result files to send back status or final results to the Frontend.
6. Frontend displays results or generated reports to the user.

3. Module Descriptions

3.1. Frontend (Web UI)

- **Directory:** public/, src/ (based on typical structures, confirm actual location)
- **Functionality:**
  - Provides a user interface for model and dataset upload.
  - Allows configuration of robustness tests/attacks.
  - Initiates test execution requests.
  - Displays test progress and results.
  - Facilitates report generation requests.
  - Handles user authentication/session management (if implemented).
- **Interfaces:**
  - Interacts with users via a web browser.
  - Calls RESTful APIs provided by the Node.js Backend (e.g., /api/upload, /api/start_test, /api/results, /api/generate_report).
- **Dependencies:** Node.js Backend API.

3.2. Node.js Backend (API Gateway/File Handler)

- **File:** server.js, potentially others.
- **Functionality:**
  - Acts as the primary API endpoint for the Frontend.
  - Manages HTTP requests and responses.
  - Handles file uploads securely, saving them to a persistent Docker volume.
  - Validates user inputs and requests.
  - Orchestrates the workflow by calling the Flask Backend for ML tasks.
  - Interacts with the MySQL database for storing/retrieving metadata, job status, or results.
  - May handle report generation logic or coordination.
- **Interfaces:**
  - Provides RESTful APIs for the Frontend.
  - Calls RESTful APIs provided by the Flask Backend (e.g., /process, /attack).
  - Connects to the MySQL Database via a database driver.
  - Reads/writes to the shared Docker volume for file access.
- **Dependencies:** Flask Backend API, MySQL Database, Shared Docker Volume.

3.3. Flask Backend (ML Processing)

- **Directory:** flask_backend/
- **Functionality:**
  - Provides API endpoints for ML-specific tasks requested by the Node.js Backend.
  - Loads ML models (e.g., TensorFlow, PyTorch) from the shared volume.
  - Loads datasets from the shared volume.
  - Implements and executes various adversarial attack algorithms (potentially using libraries like tensorflow-privacy, ART, etc.).
  - Performs robustness evaluations and analysis.
  - Stores detailed results in the MySQL database or writes output files (e.g., logs, attacked datasets) to the shared volume (attack_results/ seems relevant).
- **Interfaces:**
  - Provides RESTful APIs for the Node.js Backend.
  - Reads models and datasets from the shared Docker volume.
  - Writes results/logs to the shared Docker volume (attack_results/, user_models/, train_dataset/, test_dataset/).
  - Connects to the MySQL Database.
- **Dependencies:** Python ML libraries (TensorFlow, tensorflow-privacy, Flask, etc.), MySQL Database, Shared Docker Volume.

3.4. MySQL Database

- **Configuration:** docker-compose.yml, init_dump.sql, init.sql
- **Functionality:**
  - Persistently stores structured data.
  - Potential schemas: User accounts, uploaded file metadata (path, user, timestamp), test configurations, attack parameters, execution status, detailed results metrics.
  - Initialized using init_dump.sql or init.sql.

- **Interfaces:** Standard SQL interface accessible via TCP/IP (port 3306 mapped in Docker).
- **Dependencies:** None (acts as a stateful service).

3.5. Docker Environment

- **Configuration:** Dockerfile, Dockerfile.web, Dockerfile.python, Dockerfile.mysql, docker-compose.yml
- **Functionality:**
    - Defines the build process for individual service images (Node.js, Flask, potentially Frontend).
    - Orchestrates the multi-container application startup and networking using docker-compose.
    - Manages persistent data storage using Docker volumes (e.g., for MySQL data, uploaded files, results).
    - Defines inter-service communication channels (Docker network).
    - Specifies environment variables for configuration.
- **Interfaces:** Docker CLI, Docker Compose CLI.
- **Dependencies:** Base images (e.g., node, python, mysql), application source code, configuration files.

4. Technology Stack

- **Frontend:** HTML, CSS, JavaScript (Potentially a framework like React/Vue/Angular - check package.json and src/ for specifics)
- **Backend (API Gateway):** Node.js (Check package.json for version), Express.js (Likely, check package.json)
- **Backend (ML Processing):** Python (Check Dockerfile.python or requirements.txt in flask_backend/ for version), Flask, TensorFlow, TensorFlow Privacy (based on README), other ML/data science libraries (Pandas, NumPy, Scikit-learn - check requirements).
- **Database:** MySQL 8.0
- **Containerization:** Docker, Docker Compose
- **Version Control:** Git
- **Development Environment:** (Specify OS, IDEs like VS Code, etc., used during development)

5. Design Rationale and Alternatives

5.1. Architecture: Microservices (Multi-Container) Approach

- **Rationale:**
    - **Separation of Concerns:** Isolates UI logic (Frontend), API handling/orchestration (Node.js), and intensive computation (Flask/Python).
    - **Technology Suitability:** Leverages Node.js's strengths in asynchronous I/O for handling web requests and file uploads, and Python's rich ecosystem for machine learning.
    - **Scalability:** Allows independent scaling of services (e.g., add more Flask workers if ML processing is the bottleneck).

    - **Maintainability:** Smaller, focused codebases for each service are easier to manage and update.
- **Alternatives Considered:**
    - **Monolithic Backend:** Using only Node.js or only Python/Flask for all backend tasks. This simplifies deployment initially but can lead to a less modular and potentially less performant system, especially mixing heavy computation with web request handling. Python might struggle with highly concurrent I/O compared to Node.js, while Node.js lacks the extensive native ML libraries of Python.

5.2. Database: MySQL

- **Rationale:**
    - **Structured Data:** Suitable for storing relational data like user info, file metadata, test configurations, and potentially structured results.
    - **Maturity & Stability:** Widely used, well-documented, and robust relational database.
    - **Transaction Support:** Ensures data integrity for critical operations.
    - **Existing Schema:** Project already includes init_dump.sql, indicating prior use and defined structure.
- **Alternatives Considered:**
    - **NoSQL (e.g., MongoDB):** Could be suitable for less structured result logs, but might be less ideal for managing relations between users, files, and tests. Might require schema redesign.
    - **PostgreSQL:** Another strong relational database, similar capabilities to MySQL. Choice might be based on team familiarity or specific feature needs.

5.3. Deployment: Docker & Docker Compose

- **Rationale:**
    - **Environment Consistency:** Ensures the application runs the same way across development, testing, and production environments.
    - **Dependency Management:** Packages application code and all dependencies (OS libraries, language runtimes) into isolated containers.
    - **Simplified Deployment:** docker-compose up provides an easy way to start the entire multi-container application.
    - **Isolation:** Services run in separate containers, minimizing conflicts.
- **Alternatives Considered:**
    - **Manual Installation:** Installing Node.js, Python, MySQL, and all libraries directly on the host machine. This is prone to version conflicts, difficult to replicate, and harder to manage.
    - **Virtual Machines:** Provides isolation but with higher resource overhead compared to containers.

5.4. File Handling: Shared Docker Volume

- **Rationale:**
    - **Persistence:** Ensures uploaded files and generated results are not lost when containers restart.

    - **Inter-service Access:** Allows the Node.js backend (receiving uploads) and the Flask backend (processing files) to access the same data easily.
- **Alternatives Considered:**
    - **Cloud Storage (e.g., AWS S3, Azure Blob Storage):** More scalable and robust for large amounts of data or distributed deployments, but adds external dependency and potentially cost.
    - **Database Storage (BLOBs):** Storing files directly in the database is generally not recommended for large files due to performance implications and database bloating.