Team number:

Feinan Guo, Jiahui Huang, Ruxin Ma, Shiyu Zhao, Xuran Chen

Project Title:

Project LS1

AI-Driven Warehouse Packing and Visualization Solutions

Table of Contents

1. Introduction

1.1 Project Overview

Our project introduces **PackPilot**, a warehouse management platform featuring advanced placement algorithm and intuitive visualisation interfaces to improve warehouse packing efficiency. The system integrates a React-based frontend and a Flask-based backend, fully deployed via *Docker*. It supports multiple user roles (e.g., manager and worker) and provides simplified item and task operations through a RESTful API design.

1.2 Scope of the test plan document

This document defines the comprehensive testing strategy for PackPilot, covering frontend and backend quality assurance through unit, integration, and release testing. The plan includes detailed test cases designed for both technical team members and external stakeholders to understand testing objectives, methodologies, and success criteria. All test cases are documented in the appendix with clear descriptions, inputs, expected outcomes, and traceability information.

2. Test strategy

2.1 Test scope

The testing focus of this project includes the following four aspects:

- **Authentication and Authorization**: Validates registration, login, role-based page redirection, and permission restrictions.
- **Item and Task Management**: Testing manager capabilities for adding, editing, and deleting items, form interactions and data validation.
- **2D/3D Visualization**: Validation of current and historical item placement displays, rendering performance and interactions for users with worker permissions.
- **API Communication**: Ensuring robust frontend-backend communication, proper data handling, and comprehensive exception management.

2.2 Testing report

Test cases are documented in a structured format (see **Appendix A, Table 1-9**), using a structured format to ensure traceability, review, and retesting. Records include test description, input, expected output, actual output, responsible person, and reviewer. Unified format for easy comparison of results and problem location.

2.3 Test assumptions

The testing framework assumes: *Docker* environment is properly configured with all services accessible via docker-compose; pre-configured test accounts exist for Manager and Worker roles; API interfaces remain stable during testing; testing tools (*ESLint* for JavaScript linting, *Jest* for JavaScript testing, React Testing Library for React component testing, *Pytest* for Python testing, Mock Service Worker (MSW) for API mocking) are operational; *JSON Web Token* (JWT) authentication is implemented with functional token generation and validation; test database supports isolation through reset capabilities; 2D/3D visualization components render consistently across target browsers including Chrome (v90+), Firefox (v88+), and Safari (v14+); all key dependencies are version-locked in *package.json* and *requirements.txt* to ensure consistent testing environments; test data is deterministic and reproducible across test runs through seeded databases and fixed mock responses.

3. Test execution

3.1 Defect testing

Defect testing will be conducted continuously throughout the development process to ensure early detection and resolution of common types of software errors.

What will be tested:

The testing will cover syntax, logic, runtime, semantic, and compilation errors across both frontend and backend components.

How it will be tested:

For **syntax errors**, *ESLint* is already configured on the frontend to automatically detect violations, while *Pylint* will be introduced to perform similar checks on backend Python code. To address **logic errors**, we plan to expand the frontend test suite using *Jest* and add *unit tests* on the backend using Pytest. **Runtime errors** will be handled by introducing error boundary components in the frontend and enhancing exception handling in the backend. For **semantic errors**, strict type checking with TypeScript is applied in the frontend, and type annotations will be implemented in Python backend code. For **compilation errors**, the frontend will utilize the TypeScript compiler for type validation, while the backend, though not requiring compilation, will employ *mypy* for static type checking in Python.

3.2 Unit testing

Unit testing verifies the correctness and robustness of individual modules including components, services, database utilities, and authentication handlers in isolation before system-level integration. The goal is to identify defects early, ensure logic correctness, and facilitate maintainability through automated, repeatable tests.

What will be tested:

Frontend development is based on React and TypeScript, and the following components will be the primary focus:

- **React Components** (e.g., login forms): Tested for correct rendering, props handling, and interaction responses (e.g., button clicks, form submissions).
- **Custom Hooks and Services**: Logic encapsulated in hooks and service functions (e.g., API requests) will be tested in isolation using mocked data.
- **State Management and Routing**: React contexts and route guards will be tested to ensure appropriate navigation and role-based access control.

The Flask backend exposes RESTful APIs and encapsulates business logic, database operations, and role-based access control.

- **API Endpoints**: Each route (e.g., for authentication, item, task, container) will be tested for method correctness, response validity, and error handling.
- **Database Operations**: CRUD functions for key models will be tested using an isolated test database to verify data manipulation.
- **Authentication Logic**: JWT generation, validation, and role-based access will be tested to ensure secure authorization.

How it will be tested:

- **Frontend**: *Jest* and React Testing Library for component rendering and interaction simulation, *MSW* for API call mocking, snapshot testing for UI consistency, and isolated testing of custom hooks with test wrapper components
- **Backend**: *Pytest* fixtures for HTTP client simulation, test database with transaction rollbacks for isolation, *JWT* token generation and validation testing, model instantiation with test data for CRUD validation, and mock objects for external service dependencies

Pass Criteria: Unit tests must achieve a minimum of **80%** statement coverage for both frontend and backend code. All test cases must pass without errors, and critical functionality (authentication, data persistence, API endpoints) must achieve **95%** coverage.

3.3 Integration and release testing

What will be tested:

Integration testing validates interactions between frontend, backend, and supporting services within the *Dockerized* environment, including API communication for user authentication and CRUD operations, cross-component state synchronization between React contexts, backend service collaborations such as triggering database updates, database transaction integrity during concurrent operations, and Docker container networking with proper environment variable configuration. **Release testing** focuses on deployment readiness through *Docker* build process validation, service health monitoring, complete user journey scenarios from login to task creation, system performance under simulated loads, *JWT* security mechanisms including token expiration and encryption, and API documentation accuracy verification.

How it will be tested:

Integration testing employs *Pytest* with requests library to simulate end-to-end workflows ensuring correct data flow and error handling across the system, dedicated test database instances to validate transactional integrity and data consistency during concurrent operations, *docker-compose* deployment testing to confirm network connectivity between services and proper configuration management, and automated workflow testing for complete user scenarios. **Release testing** utilizes Docker build validation (`docker-compose build`) to ensure frontend static assets compile correctly and backend dependencies install without conflicts, service health monitoring (`docker-compose up -d`) to confirm containers start and remain operational, *Cypress* for end-to-end user scenario automation simulating real workflows in the deployed environment, *JMeter* for performance testing to measure response times under simulated user loads and identify bottlenecks, *JWT* security validation for token handling including expiration and encryption verification, and automated API documentation verification to ensure current and accurate reflection of system capabilities and interfaces.

To ensure system recovery from failed deployments, the testing framework includes automated **rollback** procedures using *Docker* image versioning and `docker-compose down/up` commands, database backup and restore procedures before major deployments to verify critical functionality before production release.

4. Testing plan schedule

The testing approach follows a structured **five-phase** methodology spanning weeks 6-12 of the development cycle. Beginning with initial setup and tool configuration, the process progresses through continuous defect testing, comprehensive unit testing of frontend and backend components, integration testing for system connectivity, and concludes with final release testing including performance and security validation. Detailed schedule information is provided in **Appendix B, Table 10.**

5. Risks

The potential testing risks may as follows:

- **Incomplete Test Coverage**: Edge cases and error scenarios may be inadequately tested, potentially allowing undetected bugs to reach production.
- **API Communication Failures**: Data format mismatches, response timing issues, or access control problems could disrupt frontend-backend communication.
- **Environment Inconsistencies**: Differences between local, staging, and containerized environments may cause behavioural variations during testing.
- **Limited Real-World Simulation**: Artificial test data and unrealistic user flows may fail to capture actual usage patterns, leading to post-deployment failures.

Appendix

Appendix A

Table 1 **Defect Testing** details

Test description	Input	Expected output	Current output	Responsible	Reviewer
Verify ESLint catches syntax errors in frontend	JavaScript file with missing semicolon	ESLint error: "Missing semicolon"	Same as expected	Jiahui Huang	Ruxin Ma
Test Pylint detects backend syntax issues	Python file with incorrect indentation	Pylint error: "Indentation error"	Same as expected	Xuran Chen	Feinan Guo
Validate TypeScript compilation errors	TypeScript file with type mismatch	tsc error: "Type 'string' is not assignable to type 'number'"	Same as expected	Shiyu Zhao	Ruxin Ma
Check runtime error boundary handling.	Component throws unhandled exception	Error boundary displays fallback UI	Same as expected	Shiyu Zhao	Ruxin Ma
Test mypy static type checking	Python function with incorrect type annotation	mypy error: "Argument has incompatible type"	Same as expected	Xuran Chen	Feinan Guo
Test Type: Defect Testing		Date: April 8 – 11, Week 6			

Table 2 **Unit Testing** – Login API details

Test description	Input	Expected output	Current output
Check if a user can log in with valid credentials.	Valid username, password, and role (e.g., manager or worker).	System responds "Login successful" Returns security token (JWT) Shows user role (admin/staff) Redirects to appropriate page based on role	Same as expected
Verify the response when login credentials are incorrect.	Invalid username, password, or role.	System rejects login Error message: "Incorrect username, password, or role" No security token provided	Same as expected
Ensure the system correctly hashes and verifies user passwords.	A plain-text password.	The hashed password is different from the plain-text password The hash verification function confirms the plain-text password matches the stored hash	Same as expected

Test if a request with a valid token can access a protected endpoint.	API request with a valid JWT token in Authorization header.	User sees restricted content System responds normally	Same as expected
Test if the system rejects access when the token is invalid.	API request with an invalid or expired JWT token.	System blocks access Error: "Invalid token!"	Same as expected
Check that access is denied when no token is provided.	API request with no security token.	System blocks access Error: "Token is missing!"	Same as expected
Check if the backend can successfully connect to and disconnect from the database.	Call the database connection and closing functions.	Database session is created Session is closed without error	Same as expected
Verify that the correct user data can be fetched from the database.	A known username and role.	A record with matching username, hashed password, and role	Same as expected
Test the full flow from login to accessing a protected endpoint.	Valid login request Use returned token to request protected resource	Login response includes a valid token Second request with token returns protected data	Same as expected
Test if the system enforces minimum password complexity.	Passwords with varying complexity (e.g., "123", "Pass123!").	Weak passwords are rejected	Same as expected
Check that the system accepts only properly formatted usernames.	Usernames that are too short/long or contain illegal characters.	Usernames outside the allowed length (5–20 characters) are rejected Usernames with special characters are rejected	Same as expected
Test the full flow from login to accessing a protected endpoint.	Valid login request Use returned token to request protected resource	Login response includes a valid token Second request with token returns protected data	Same as expected
Test type: Unit Testing Date: May 2 – 10, Week 7 - 8		Responsible: Feinan Guo Reviewer: Xuran Chen, Ruxin Ma	

Table 3 **Unit Testing** – Container/Task/Item API details

Test description	Input	Expected output	Current output
Container API			
Check if a manager can add a container with valid token and payload.	Valid JWT token with Manager role and JSON body containing length, width, height, label.	System responds “Container added” Returns container_id Status: is “success”	Same as expected
Test if a non-manager is denied access to add a container.	JWT token with Worker role.	System blocks access Status: “error”	Same as expected

		Message: “Forbidden”	
Test if an invalid or expired token is rejected.	Expired or malformed JWT in Authorization header.	System blocks access Message: “Invalid token”	Same as expected
Test if request is rejected when required fields are missing.	JSON body with missing length, width, or height.	System returns “Validation failed” Status: “error”	Same as expected
Item API			
Check if a manager can successfully add an item.	Valid JWT token with Manager role and JSON body: length, width, height, orientation, remarks.	System responds “Item added” Returns item_id Status: “success”	Same as expected
Check if system blocks non-manager roles.	JWT token from a Worker.	System blocks access Status: “error” Message: “Forbidden”	Same as expected
Check validation when required item fields are missing.	Missing length, width, or height in body.	System returns 400 Status:: “error” Message: “Validation failed”	Same as expected
Test if backend correctly stores optional fields.	Include remarks and orientation.	Optional fields saved in DB Response contains the submitted data	Same as expected
Task API			
Check if a manager can assign a task.	Valid JWT token with Manager role, and valid JSON including task_name, container_id, assigned_to.	System responds “Task assigned” Returns task_id Status: “success”	Same as expected
Test if worker role is blocked from assigning a task.	JWT token with Worker role.	System blocks access Status: “error” Message: “Forbidden”	Same as expected
Test task creation with missing required fields.	JSON missing assigned_to or container_id.	System returns 400 Status: “error” Message: “Validation failed”	Same as expected
Ensure assigned tasks are linked to manager.	Submit request as manager1.	Task in DB has manager_name = manager1 Appears in /task_history	Same as expected
Test type: Unit Testing Date: May 7 – 17, Week 8 - 9		Responsible: Xuran Chen Reviewer: Feinan Guo, Ruxin Ma	

Table 4 **Unit Testing** –Algorithm API details

Test description	Input	Expected output	Current output
Check if algorithm returns placement result for valid input	JSON with valid container dimensions and box list (including is_fragile)	Status: “success” Return: result with box id, positions x, y, z, and dimension	
Check response on missing container or boxes	Request body missing container or boxes field	Status: “error” Message: “No data provided” or “Validation error: str(ve)” or “Unexpected error: str(e)”	
Check optimization with large number of boxes	30+ boxes with various sizes	Response completes within expected time and returns placements	
Check cost value is returned	Valid input data	Response includes “cost” field (float)	
Check if algorithm avoids overlapping boxes	Boxes with potential overlaps	Returned placements show no overlaps between any two boxes	
Check if fragile box is not stacked with another box on top	Include a fragile box and another box that could stack on it	Output shows fragile box has no other box above it	
Check if small box is not placed near edge	A small box with volume < threshold (10) placed close to wall	Result avoids placing small boxes on edge or applied penalty	
Test type: Unit Testing Date: Week 11 (planned)		Responsible: Jiabao Ye Reviewer: Feinan Guo, Jiahui Huang	

Table 5 **Unit Testing** – Login UI details

Test description	Input	Expected output	Current output
Test login form rendering	Navigate to login page	Form displays username, password fields and submit button	Same as expected
Validate form input validation	Empty username/password submission	Error messages display for required fields	Same as expected
Test successful login navigation	Valid credentials submission	User redirected to appropriate dashboard	Same as expected
Verify error message display	Invalid credentials submission	Error message "Invalid credentials" displayed	Same as expected
Test token invalid or logout	Click “log out” button or auth expire	Redirect to log in page	
Test type: Unit Testing Date: May 7 – 17, Week 8 - 9		Responsible: Shiyu Zhao Reviewer: Ruxin Ma, Feinan Guo	

Table 6 **Unit Testing** – Manager Dashboard UI details

Test description	Input	Expected output	Current output
Test manager dashboard rendering	Manager user login	Dashboard displays navigation, item management, task overview	Same as expected
Validate item creation form	Click "Add Item" button	Item creation form modal appears	Same as expected
Test item list display	Navigate to items lists section	List of items with edit/delete options displayed	Same as expected
Verify task assignment interface	Click "Assign Task" button	Task assignment form with worker selection available	Same as expected
Test the workflow of creating a task	Select items, work, input container size and submit form	The selected items removed from current item list; task history display the new task	Same as expected
Verify task history interface	Click "Task History" button	List of created tasks, sorted by the created time	Same as expected
Test edit operation of item list	Click edit button	Display the "add item" popup with history data	Same as expected
Test delete operation of item list	Click delete button	Remove the item from the item list	Same as expected
Test type: Unit Testing Date: May 5 – 23, Week 7 - 10		Responsible: Ruxin Ma Reviewer: JiaHui Huang, Feinan Guo	

Table 7 **Unit Testing** –Worker Dashboard UI details

Test description	Input	Expected output	Current output
Test worker dashboard rendering	Worker user login	Dashboard displays assigned tasks and 2D/3D views	Same as expected
Test whether the 3D model can be displayed on the webpage	Three.js model & React	The 3D model is properly displayed on the webpage	Same as expected
Test the interactivity between the 3D model and the webpage, including zooming and panning	Interaction between mouse dragging, scroll wheel, and the canvas	The 3D model can be zoomed, panned, and have its view switched based on mouse operations	Same as expected
Test whether objects can trigger click events	Mouse click, three.js items	Objects can be successfully clicked and highlighted	Same as expected
Test whether the webpage UI can interact with the 3D model	Mock item database, container data	Objects from the database can be successfully added to or remove from the canvas via button clicks	Same as expected
Test the next operation	Click "Next" button	The packing view can display next item with correct position	Same as expected
Test the prev operation	Click "Prev" button	The packing view can display prev item, highlight the prev item position, hidden other item	Same as expected

Verify the different status of tasks	Worker user login	If no task in progress, display “no task” massage If at least on task left, display “click next to start” massage	
Test type: Unit Testing Date: May 2 – 22, Week 7 - 10		Responsible: Jiahui Huang Reviewer: Ruxin Ma, Jiabao Ye	

Table 8 **Integration Testing** details

Test description	Input	Expected output	Current output	Responsible	Reviewer
Test Docker container build	Execute docker-compose build	Frontend and backend containers build successfully	Same as expected	Shiyu Zhao	Feinan Guo
Validate service startup	Execute docker-compose up	All services start without errors	Same as expected		
Test inter-service communication	Frontend makes API call to backend	Request reaches backend and response received	Same as expected		
Verify environment variables	Check container configuration	Environment variables loaded correctly	Same as expected		
Validate frontend-backend API communication	Frontend sends login request to backend	Backend responds with valid JWT token		Ruxin Ma	Feinan Guo
Test database transaction integrity	Concurrent user registration requests	All registrations succeed without data corruption		Xuran Chen	Feinan Guo
Verify Docker container networking	Start all services via docker-compose	All containers communicate successfully		Shiyu Zhao	Feinan Guo
Test React context state synchronization	User login updates multiple components	All components reflect authentication state		Jiahui Huang	Ruxin Ma
Validate end-to-end user workflow	Complete user journey from login to task creation	Workflow completes without errors		Feinan Guo	Ruxin Ma
Test type: Integration Testing		Date: Week 11 (planned)			

Table 9 **Release Testing** details

Test description	Input	Expected output	Current output	Responsible	Reviewer
Validate Docker build process	Execute docker-compose build	All services build without errors		Feinan Guo	Ruxin Ma
Test service health checks	Execute docker-compose up -d	All containers start and remain operational		Shiyu Zhao	Feinan Guo
Performance testing with JMeter	Simulate 50 concurrent users	Response time < 2 seconds		Ruxin Ma	Feinan Guo
End-to-end testing with Cypress	Login-to-task-creation workflow	Complete workflow executes successfully		Jiahui Huang	Ruxin Ma
JWT security validation	Test token expiration and encryption	Expired tokens rejected; encryption verified		Xuran Chen	Feinan Guo
API documentation verification	Compare docs with actual endpoints	Documentation matches implementation		Xuran Chen	Feinan Guo
Test type: Release Testing		Date: Week 12 (planned)			

Appendix B

Table 10 Testing Schedule

Phase	Timeframe	Testing Activities
1. Initial Setup	Week 6	- Set up ESLint, Pylint, Pytest, Jest, React Testing Library, and three.js - Define testing conventions
2. Continuous Defect Testing	Week 7–Week 10	- Syntax & logic error detection in frontend and backend - Runtime & semantic testing - Linting & type checking - other basic function testing
3. Unit Testing	Week 8–Week 10	- Manager dashboard UI tests - Worker dashboard UI tests - Login & register page UI tests - Backend API and DB model tests - 3D/2D Visualize & interactive tests
4. Integration Testing	Week 11	- Validate frontend ↔ backend communication - Verify DB transactions & data flow - Docker service connectivity
5. Release Testing (Final)	Week 12	- Full testing - Build validation - Performance & Security Testing