

Testing Policy Document

CAPSTONE 2025 - Demo 4

Project name: PortfolioPortal

Group name: Ctrl Freaks



Name and Surname	Student Number
Angelique Breedt	u23542838
Eric Booyens	u05127824
Nabegh Muhra	u23661268
Keegan Walker	u22693760
Christopher Yoko	u22857941

Table of Contents

1	Introduction	Page 2
2	Objectives of Testing	Page 3
3	Scope of Testing	Page 3
4	Testing Strategy	Page 4
5	Unit Testing	Page 4-10
6	Integration Testing	Page 11-15
7	Non-functional Requirement Testing	Page 16-22
8	End-To-End Testing	Page 23
9	Testing Standards & Best Practices	Page 22
10	Test Environment	Page 24
11	Defect & Risk Management	Page 24-25
12	Entry and Exit Criteria	Page 25-26
13	Continuous Testing and CI/CD	Page 26

1. Introduction

The purpose of this Testing Policy is to establish a standardised framework for testing activities within **Portfolio Portal**. Testing ensures that the application meets functional and non-functional requirements, maintains high quality, and provides users with a reliable and engaging 3D portfolio experience.

As the Portfolio Portal transforms traditional CVs into immersive, interactive websites, the system must perform accurately, consistently, and securely across various environments and devices. This document defines the objectives, scope, responsibilities, standards, tools, and methodologies for testing throughout the project lifecycle.

2. Objectives of Testing

- To verify that the system meets all functional requirements (e.g., CV upload, text extraction, 3D template generation).
 - To validate non-functional requirements, including performance, usability, availability, and security.
 - To detect and resolve defects early to minimise risk and development costs.
 - To ensure consistent quality across all releases.
 - To build stakeholder confidence in the reliability of *Portfolio Portal*.
-

3. Scope of Testing

The following components of the system are in Scope:

- CV upload functionality (file formats, OCR extraction).
 - Template rendering in 3D environments.
 - User interactions with generated portfolio sites.
 - Compatibility across browsers (Chrome, Firefox, Edge, Safari) and devices (desktop, mobile, tablet).
 - API integration (CV parsing, template selection).
 - Security features (data privacy, file upload validation).
-

4. Testing Strategy

- Levels of Testing:
 - *Unit Testing*: Validate individual functions (e.g., CV text extraction, template loader).
 - *Integration Testing*: Ensure modules (e.g., CV parsing → template selection → rendering) work together.
 - *E2ETesting*: Validate complete workflow from CV upload to portfolio generation.
 - *User Acceptance Testing (UAT)*: Involve stakeholders to ensure business goals are met.
 - Types of Testing:
 - *Functional Testing*: Confirm features behave as expected.
 - *Performance Testing*: Stress/load testing of portfolio rendering.
 - *Usability Testing*: Ensure intuitive navigation and accessibility.
 - *Regression Testing*: Ensure updates don't break existing functionality.
-

5. Unit Testing

Purpose:

Unit testing is aimed at verifying the correctness of individual components or modules of a system in isolation, typically at the function or class level. The purpose of unit tests is to ensure that each small piece of code behaves as expected, producing the correct output for given inputs and handling edge cases properly. By testing these building blocks independently, developers can quickly identify and fix bugs early in the development cycle, before they propagate into larger system issues. Unit testing also improves code maintainability and reliability, making it easier to refactor or extend the system with confidence.

Tools used: Jest (<https://jestjs.io/>)

Run the tests running `npm test` in the server directory.

Where tests can be found: `server/tests/unit/tests`

For the `sectionizer.test.js`:

(https://github.com/COS301-SE-2025/Portfolio-Portal/blob/main/server/tests/unit/_tests_/sectionizer.test.js)

The functions that we did unit testing on for the `sectionizer.test.js` code are as follows:

- `extractPersonallInfo(lines, ocrName)` — pulls name, email, phone, LinkedIn, website, address, and a top-of-CV summary paragraph, while stripping contact lines from the remaining text.
- `extractReferencesFirst(lines)` — detects References / Referees blocks, returns those lines, and removes them from the remaining body to prevent leakage into other sections.
- `extractSectionByHeader(lines, key)` — generic section slicer used by Experience/Education (and others) to capture lines until the next header.
- `extractSimpleBlocks(lines)` — builds simple section buckets (experience, education, skills, etc.) from headers and content lines.
- `processCV(ocr)` — end-to-end composition: given the OCR output (name, remainingCV), produces normalised structured data: `personal_info`, `experience`, `education`, `skills`, `languages`, `projects`, `certifications`, `references`.

Some of the high level logic that the tests cover are as follow:

Personal info:

- Labeled and unlabeled extraction (email/phone), LinkedIn normalisation, website URL filtering (ignores GitHub for website; ignores tech tokens like “node.js”), address proximity at top of document, fallback name logic (OCR name vs labeled), and summary paragraph without/with explicit header (excludes contact lines).
- References:
 - Detects “References”/“Referees”/“Reference”, collects until next header or EOF, removes from remaining text so numbers/emails don’t leak into other sections.
- Sections:
 - Experience/Education slicing by multiple header variants (“Work Experience”, “Professional Experience”, “Academic Background”, “Qualifications”, etc.) and EOF behavior.
- Languages vs Skills:
 - “English (Fluent)”/“Afrikaans (Native)” siphoned out of skills into languages.

- De-duplication & noise removal:
 - Deduplicates repeated skills; scrubs standalone name tokens (“AVA”, “REYNOLDS”) from content blocks.
- End-to-end samples:
 - Validates realistic CVs for Ava Reynolds, Brian Park, Daniel Brooks, Alex Omari to ensure consistent structure and key content presence.

Run the sectionize.test.js code as follow:

```
npx jest tests/unit/__tests__/sectionizer.test.js
```

Get the coverage by running:

```
npx jest --coverage --collectCoverageFrom "app/utils/sectionizer.js"
```

The above must be run in the server directory (cd server).

Below are screenshots of the sectionizer.test.js after running it:

```

PASS tests/unit/__tests__/sectionizer.test.js
  ✓ siphons language lines out of skills into languages (2 ms)
  ✓ skills are deduplicated where appropriate (1 ms)
  ✓ standalone name lines are removed from the body
  ✓ captures address from alternative labels (residential/physical)
  ✓ extracts a top summary paragraph without an explicit header (1 ms)
  ✓ does not include contact lines in summary
  ✓ keeps references intact and removes them from remaining (1 ms)
  extractPersonalInfo
    ✓ extracts labeled name (fallback), email, phone, linkedin, website, address, and about description (5 ms)
    ✓ keeps OCR-provided name even if a labeled name exists (2 ms)
    ✓ address via proximity heuristic near contact lines (top window), no explicit label
    ✓ does not pick address when it only appears deep in the document (beyond top window) (1 ms)
    ✓ linkedin extracted and first non-LinkedIn non-GitHub URL becomes website; GitHub is ignored
    ✓ handles unlabeled email/phone and no links gracefully (1 ms)
    ✓ falls back to labeled name when OCR name is empty
  extractReferencesFirst
    ✓ captures lines under 'References' until next header and removes them from remaining (2 ms)
    ✓ works with alternate header keyword 'Referees' (1 ms)
    ✓ captures until end-of-file when there is no subsequent header
    ✓ returns empty when no references header is present (1 ms)
  extractExperience
    ✓ extracts lines under Experience header until next header
    ✓ works with alternate header keyword 'Work Experience' (1 ms)
    ✓ captures until end-of-file when there is no subsequent header
    ✓ extractSimpleBlocks returns only the experience block for its section (1 ms)
  extractEducation
    ✓ extracts lines under Education header until next header
    ✓ works with alternate header keyword 'Academic Background' (1 ms)
    ✓ works with alternate header keyword 'Qualifications'
    ✓ captures until end-of-file when there is no subsequent header
    ✓ extractSimpleBlocks returns the education block alongside others (1 ms)
  phones vs IDs
    ✓ does not treat SA-style ID as phone when a real phone is present
    ✓ does not produce a phone if only an ID number is present
    ✓ reference phone numbers do not leak into personal_info.phone (3 ms)
  email restoration and URL filtering
    ✓ restores broken emails with 'e' and single-space formats
    ✓ does not treat tech tokens like node.js as a website
  processCV end-to-end on OCR sample
    ✓ returns expected structure and content (20 ms)
  processCV end-to-end on OCR sample (Brian Park)
    ✓ returns expected structure and content (10 ms)
  processCV end-to-end on OCR sample (Daniel Brooks)
    ✓ returns expected structure and content (7 ms)
  processCV end-to-end on OCR sample (Alex Omari)
    ✓ with explicit About header (preferred: minimal change, matches current extractor) (6 ms)

Test Suites: 1 passed, 1 total
Tests:       36 passed, 36 total

```

Below is a screenshot of the coverage:

```
PASS tests/unit/__tests__/social.test.js
PASS tests/unit/__tests__/sectionizer.test.js
PASS tests/integration/__tests__/social.test.js (18.317 s)
• Console

console.log
  Found 3 existing users for testing

  at log (tests/integration/__tests__/social.test.js:77:15)

-----|-----|-----|-----|-----|-----
File    | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-----|-----|-----|-----|-----|-----
All files | 85.66   | 68.71    | 95.23    | 91.88    |
sectionizer.js | 85.66   | 68.71    | 95.23    | 91.88    | 119-120,126-137,191-196,203-204,244-245,264,353-354,472,486,715-716,825-827,880,890-896
-----|-----|-----|-----|-----|-----

Test Suites: 3 passed, 3 total
Tests:       75 passed, 75 total
Snapshots:   0 total
Time:        18.585 s, estimated 19 s
Ran all test suites.
```

For the `social.test.js`: The functions that we did unit testing on for the `social.test.js` code are as follows:

(https://github.com/COS301-SE-2025/Portfolio-Portal/blob/main/server/tests/unit/__tests__/social.test.js)

- Route validation logic — validates required fields (`userId`, `targetUserId`, `action`) for follow/like operations, ensures users cannot follow/like themselves, and confirms valid action types.
- GET `/users` endpoint logic — fetches users with CV data by querying the `cv_data` table first, then retrieving corresponding user profiles, handles empty/null data gracefully.
- GET `/interactions/:userId` endpoint logic — retrieves user interactions (follows/likes) from the database, returns an empty array when no interactions are found.
- POST `/follow` endpoint logic — processes follow/unfollow actions, prevents duplicate follows, updates follower counts, and handles database constraint violations.
- POST `/like` endpoint logic — processes like/unlike actions, prevents duplicate likes, updates like counts, and handles database constraint violations.
- Supabase integration mocking — mocks database client with query chaining (`select`, `insert`, `update`, `delete`), handles SQL function calls, simulates database responses and errors.

Some of the high-level logic that the tests cover are as follows:

Route validation:

- Required field validation (userId, targetUserId, action present), self-interaction prevention (users cannot follow/like themselves), action type validation (follow/unfollow, like/unlike only), missing field detection and error handling.

Database operations:

- CV data querying with null/empty handling, user profile retrieval with filtering, interaction logging with duplicate prevention, follower/like count updates with atomic operations.

Error handling:

- Database connection errors, unique constraint violations (duplicate follows/likes), missing data scenarios, invalid input validation, empty result set handling.

Supabase mocking:

- Query chain mocking (from().select().eq().in().order()), insert/update/delete operation mocking, SQL function call simulation, error response simulation with specific error codes.

Social features:

- Follow/unfollow workflow validation, like/unlike workflow validation, interaction history tracking, user discovery through CV data filtering.

Data integrity:

- Prevents self-follows and self-likes, handles duplicate interaction attempts gracefully, maintains accurate follower and like counts, ensures clean data relationships.

Run the social.test.js code as follow: `npx jest tests/unit/__tests__/social.test.js`

Get the coverage by running: `npx jest --coverage --collectCoverageFrom "app/routes/social.routes.js"`

The above must be run in the server directory (cd server).

Where tests can be found: server/tests/unit/__tests__/social.test.js

Below are screenshots of the social.test.js after running it:

```
keegan@Keegan-PC: /mnt/c/Users/Walker_PC/Desktop/3rd Year/COS301/Capstone_Demo4/Repo/Portfolio-Portal/server$ npx jest tests/unit/__tests__/social.test.js
PASS tests/unit/__tests__/social.test.js
  Social Routes
    GET /test
      ✓ should return test message (5 ms)
    GET /users
      ✓ should return users with CV data successfully (1 ms)
      ✓ should handle empty CV data
      ✓ should handle null CV data
      ✓ should handle CV data query error (1 ms)
    GET /interactions/:userId
      ✓ should return user interactions successfully (1 ms)
      ✓ should return empty array when no interactions found (1 ms)
      ✓ should handle database error (1 ms)
    POST /follow
      ✓ should validate follow data successfully (1 ms)
      ✓ should detect missing fields (1 ms)
      ✓ should detect self-follow attempt (1 ms)
      ✓ should detect invalid action
      ✓ should handle successful follow operation
      ✓ should handle duplicate follow attempt (2 ms)
    POST /like
      ✓ should validate like data successfully (1 ms)
      ✓ should detect missing fields (1 ms)
      ✓ should detect self-like attempt
      ✓ should detect invalid action (1 ms)
      ✓ should handle successful like operation
      ✓ should handle duplicate like attempt (1 ms)
  Supabase Integration
    ✓ should mock Supabase client correctly (1 ms)
    ✓ should mock database queries correctly (3 ms)
    ✓ should handle SQL function calls (2 ms)

Test Suites: 1 passed, 1 total
Tests: 23 passed, 23 total
Snapshots: 0 total
Time: 0.976 s, estimated 3 s
Ran all test suites matching /tests/unit/__tests__/social.test.js/i.
```

6. Integration Testing

Purpose:

Integration testing focuses on verifying that different modules, components, or services of a system work together correctly when combined. Its purpose is to detect issues that may arise from interactions between units, such as mismatched data formats, incorrect API calls, or communication failures between subsystems. Unlike unit testing, which isolates individual pieces of code, integration testing ensures that the flow of data and control across multiple parts of the application is seamless and reliable. This type of testing helps confirm that the system's components integrate as intended, reducing the risk of errors when the full application is executed in real-world scenarios.

Tools used:

- Postman

Used Postman to manually test the correctness and integration of the backend services and the OCR scanner logic.

Where tests can be found:

- Postman

The Postman tests can be run to test the correctness of the OCR scanner and the text extraction. To do this one will have to follow these steps:

Step 1 — Log in and get a token

1. Open Postman.
2. Set the method to POST.

Enter the request URL:

`http://localhost:5050/api/users/login`

- 3.
4. Go to the Body tab.
5. Select raw.
6. Choose JSON.

Paste valid credentials, e.g:

```
{  
  "email": "valid_email",  
  "password": "password"  
}
```

- 7.
8. Click send.
9. Copy the token (excluding quotations)

Step 2 — Upload a CV

1. Open Postman.
2. Set the method to POST.

Enter the request URL:

`http://localhost:5050/api/ocr/upload`

- 3.
4. Go to the Header tab.
5. Add a key Authorisation, wiht the value:

`Bearer <paste-your-token-here>`

6. Go to the Body tab.
7. Select form-data.
8. Add a new field,

key: `cv`

Type: `file`

Value: your file that you want to upload

9. Click send

An example of the response after a CV is uploaded will look as follows:

The screenshot displays a REST client interface with a POST request to `http://localhost:5050/api/ocr/upload`. The request is configured with the `form-data` body type. A single form entry is visible with the key `cv` and a file named `Ava Reynolds CV (demo3 - so...`. The response status is `200 OK`, and the body is shown in JSON format.

Request Details:

- Method: POST
- URL: `http://localhost:5050/api/ocr/upload`
- Body Type: `form-data`
- Form Entry: `cv` (File) with value `Ava Reynolds CV (demo3 - so...`

Response Details:

- Status: 200 OK
- Time: 1.94 s
- Size: 2.14 KB

Response Body (JSON):

```
1 {
2   "success": true,
3   "data": {
4     "personal_info": {
5       "name": "AVA REYNOLDS",
6       "description": "Innovative developer with a passion for artificial intelligence, quantum
7         computing, and space exploration technologies. With over 9 years of experience in software
8         development, data science, and cloud architecture, Ava has contributed to cutting- edge
9         projects in AI-driven satellite navigation and astronomical data processing.",
10      "email": "avareynolds@gmail.com",
11      "phone": "123-456-7890",
12      "address": "123 Anywhere St, Any City",
13      "linkedin": "",
14      "website": ""
15    }
16  }
17 }
```

For the socialIntegration.test.js: The functions that we did integration testing on for the socialIntegration.test.js code are as follows:

(https://github.com/COS301-SE-2025/Portfolio-Portal/blob/main/server/tests/integration/_tests_/social.test.js)

- Database user queries — fetches users with CV data from actual Supabase database, retrieves public/private user profiles with proper filtering, handles empty result sets gracefully.
- User interaction operations — creates, reads, and deletes follow/like interactions in database, maintains referential integrity with foreign key constraints, handles duplicate interaction prevention.
- Follow workflow integration — inserts follow interactions into user_interactions table, increments/decrements follower counts atomically, removes follow relationships and updates counts consistently.
- Like workflow integration — creates like interactions with proper user relationships, updates likes_received counters accurately, handles unlike operations with count adjustments.
- Concurrent operation handling — processes multiple simultaneous follow/like operations, maintains data consistency during concurrent updates, prevents race conditions in counter updates.
- Database constraint validation — enforces foreign key constraints on user relationships, validates unique constraints for duplicate interactions, handles constraint violations gracefully.

Some of the high level logic that the tests cover are as follow:

Database connectivity:

- Real Supabase connection using environment variables, actual table queries with proper error handling, data retrieval with filtering and pagination, foreign key relationship validation across tables.

User data management:

- CV data querying with null/undefined handling, public profile filtering with privacy controls, user profile retrieval with complete field sets, empty result handling for users without data.

Social interaction workflows:

- Follow creation with database persistence, follower count incrementation with atomic updates, interaction removal with count decrementation, duplicate follow prevention with unique constraints.

Like system integration:

- Like interaction logging with proper relationships, likes_received counter management, unlike functionality with count restoration, duplicate like constraint enforcement.

Data consistency:

- Concurrent user operations without conflicts, counter increment/decrement accuracy, foreign key constraint enforcement, and transaction-like behavior for related operations.

Edge case handling:

- Zero follower count boundary conditions, non-existent user foreign key violations, duplicate interaction constraint violations, concurrent operation conflict resolution.

End-to-end flow validation:

- Complete social workflow from follow to like to unfollow, multi-step interaction verification with database state, counter accuracy throughout operation lifecycle, cleanup and restoration of original state.

Error handling:

- Database connection failures, constraint violation responses (23505, 23503), missing data scenarios, and invalid foreign key references.

Run the socialIntegration.test.js code as follow: `npx jest tests/integration/__tests__/socialIntegration.test.js`

Get the coverage by running: `npx jest --coverage --collectCoverageFrom "app/routes/social.routes.js"`

The above must be run in the server directory (cd server).

Where tests can be found: server/tests/integration/**tests**/socialIntegration.test.js

An example of the response:

```
keegan@Keegan-PC: /mnt/c/Users/Walker PC/Desktop/3rd Year/C05301/Capstone_Demo4/Repo/Portfolio-Portal/server$ npx jest tests/integration/__tests__/social.test.js
console.log
  Found 3 existing users for testing

    at log (tests/integration/__tests__/social.test.js:77:15)

PASS tests/integration/__tests__/social.test.js (20.581 s)
  Social API Integration Tests
    Database Integration - Users with CV Data
      ✓ should fetch users with CV data from actual database (338 ms)
      ✓ should fetch public users from actual database (508 ms)
      ✓ should query private profiles correctly (379 ms)
    Database Integration - User Interactions
      ✓ should fetch user interactions from actual database (688 ms)
      ✓ should return empty array for user with no interactions (900 ms)
    Database Integration - Follow Operations
      ✓ should create follow interaction in database (1015 ms)
      ✓ should increment follower count in database (1593 ms)
      ✓ should remove follow interaction from database (939 ms)
      ✓ should handle duplicate follow constraint (900 ms)
    Database Integration - Like Operations
      ✓ should create like interaction in database (926 ms)
      ✓ should increment likes count in database (1593 ms)
      ✓ should handle duplicate like constraint (709 ms)
  End-to-End Database Flow
    ✓ should complete a full social interaction flow in database (5042 ms)
  Data Consistency and Edge Cases
    ✓ should maintain data consistency during concurrent operations (956 ms)
    ✓ should handle follower count edge cases (2239 ms)
    ✓ should verify database constraints are working (316 ms)

Test Suites: 1 passed, 1 total
Tests: 16 passed, 16 total
```

7. Non-Functional Testing

Purpose:

Non-functional testing focuses on evaluating aspects of a system that define how well it performs rather than what it does. The purpose of non-functional tests is to ensure that the application meets quality attributes such as performance, scalability, reliability, usability, security, and maintainability. For example, these tests can measure how quickly the system responds under heavy load, how secure it is against unauthorised access, or how user-friendly the interface feels. Unlike functional testing, which checks if features work correctly, non-functional testing validates whether the system can handle real-world demands and provide a smooth, efficient, and dependable user experience.

Types of non-functional tests:

- Usability
- Performance
- Availability
- Scalability
- Maintainability

Tools used:

- Google Forms for Usability
- Cypress (<https://www.cypress.io/>) for Performance of OCR CV Scanner (CV processing/portfolio generation time)

Where tests can be found:

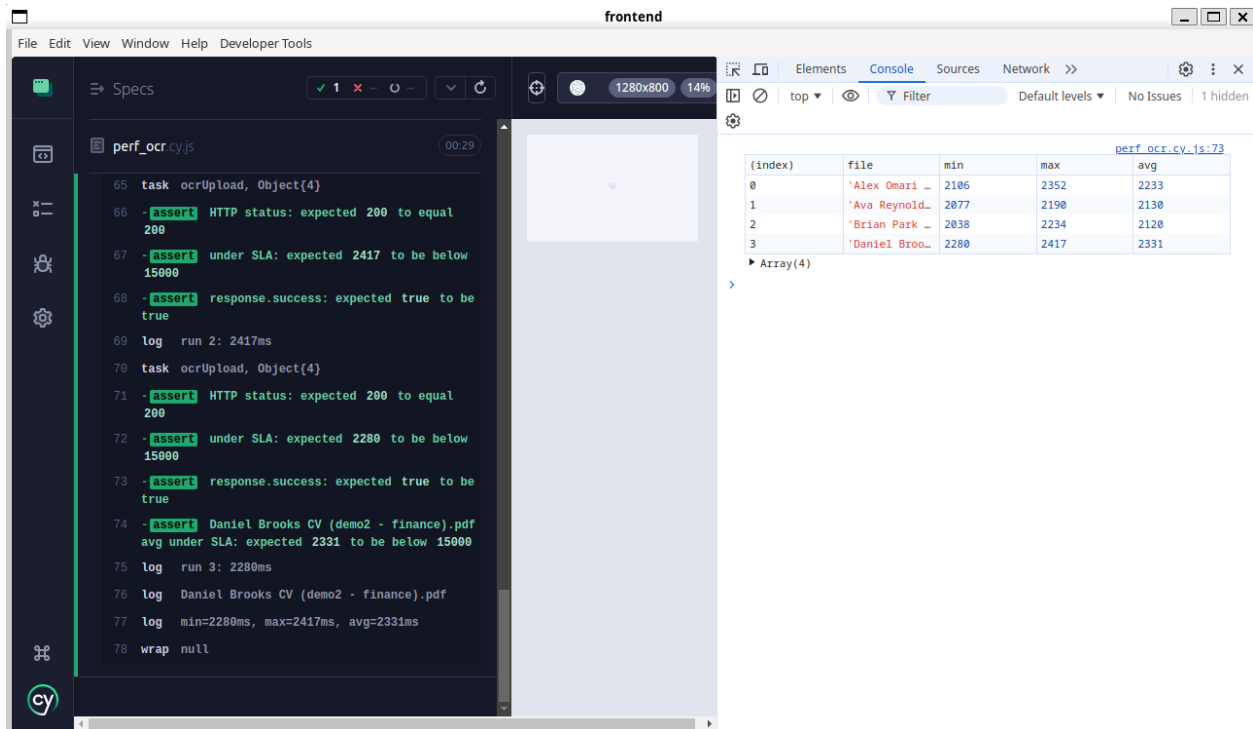
- Google form link for Usability test:
<https://docs.google.com/forms/d/e/1FAIpQLSf7iFTOtMTGfQd17pkCjx21ouwu1e5npW4eH8Yz-1FazwjV3A/vi ewform?usp=preview>
- The Cypress performance testing can be found at the directory:
frontend/cypress/integrationTests/perf_ocr.cy.js

Performance Test:

(<https://github.com/COS301-SE-2025/Portfolio-Portal/tree/main/frontend/cypress/integrationTests>)

The test firstly logs in via the POST API login endpoint. It then sets the CV data, which are 4 CV PDFs for the purpose of this testing, each with varying formats. The testing then processes these CVs via the OCR scanner and it records the time it takes for the OCR scanner to process each CV. It then iterates over this 3 times. It then takes the minimum time it took for processing, the maximum time, and the average time over these three runs. The goal is to have these processing times to all be lower than 5

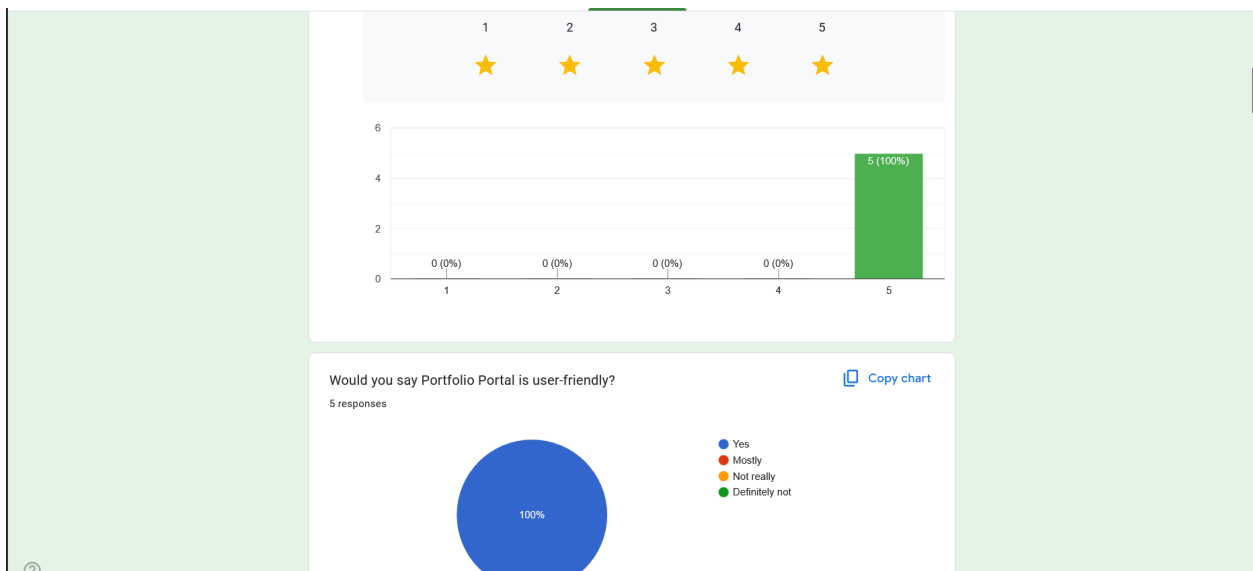
seconds. (5000 ms). Below is a screenshot after running the performance testing with Cypress.

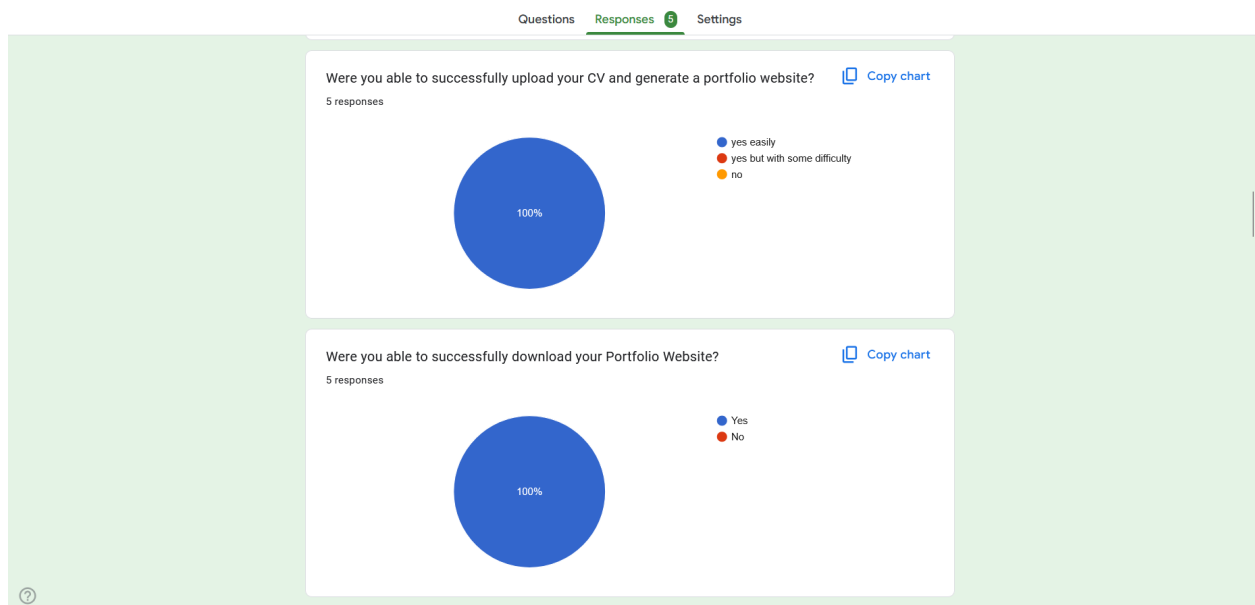


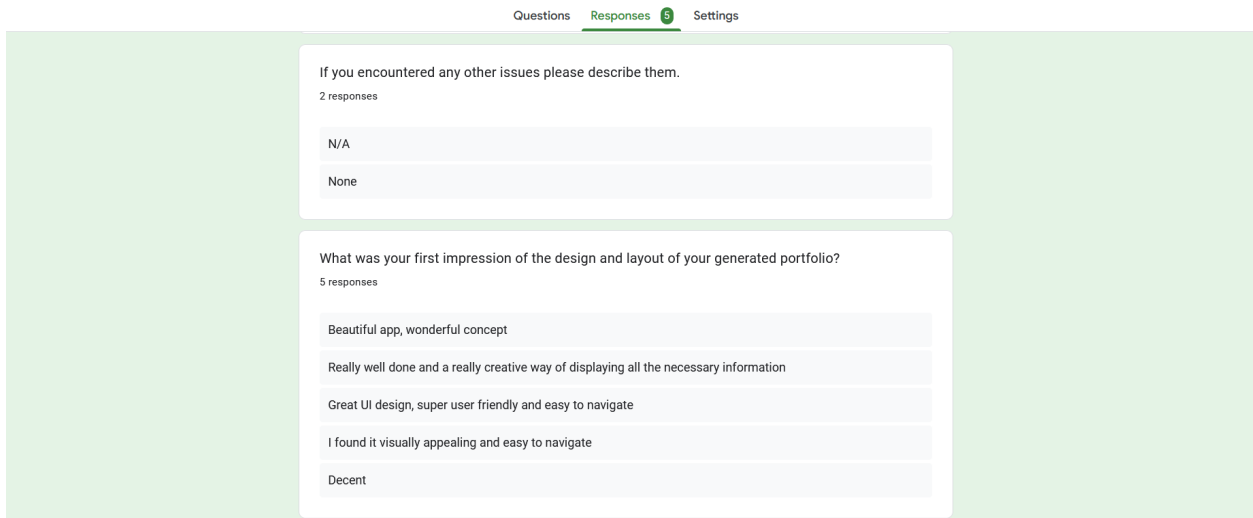
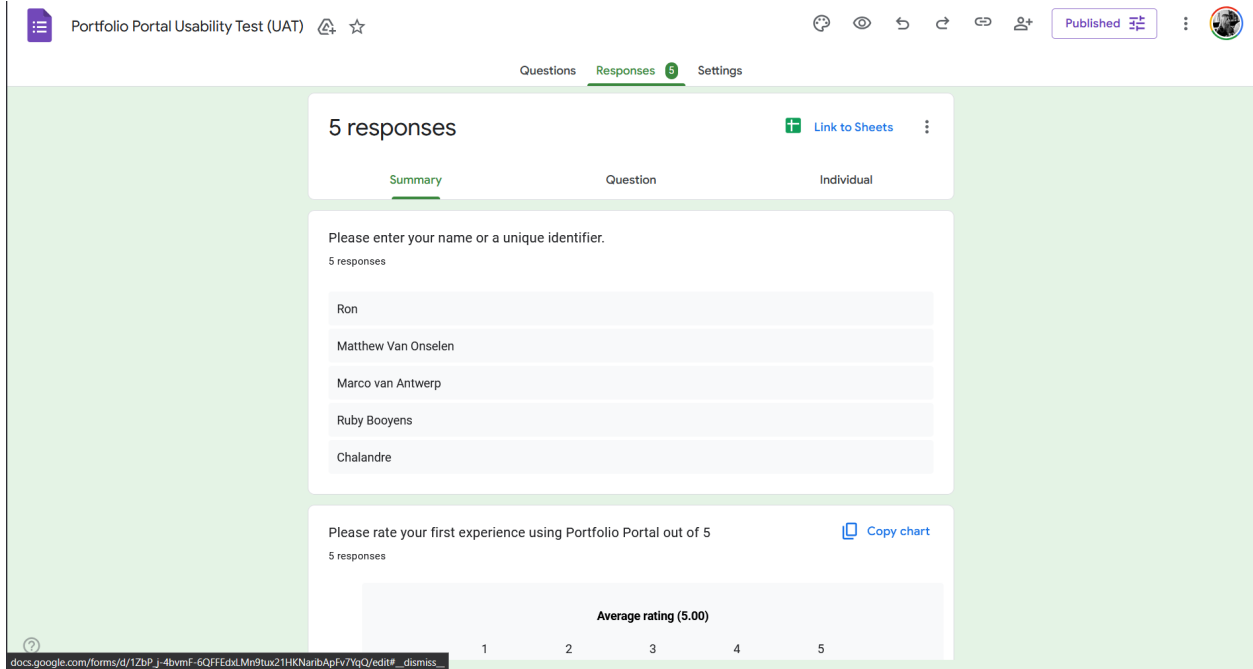
Google Form Usability Test:

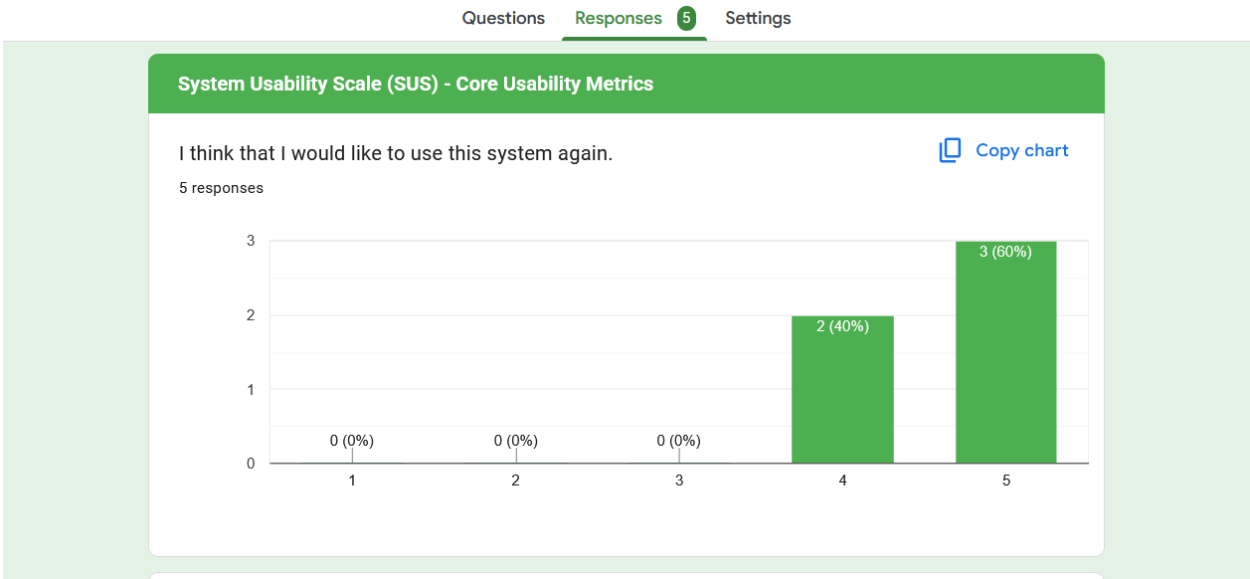
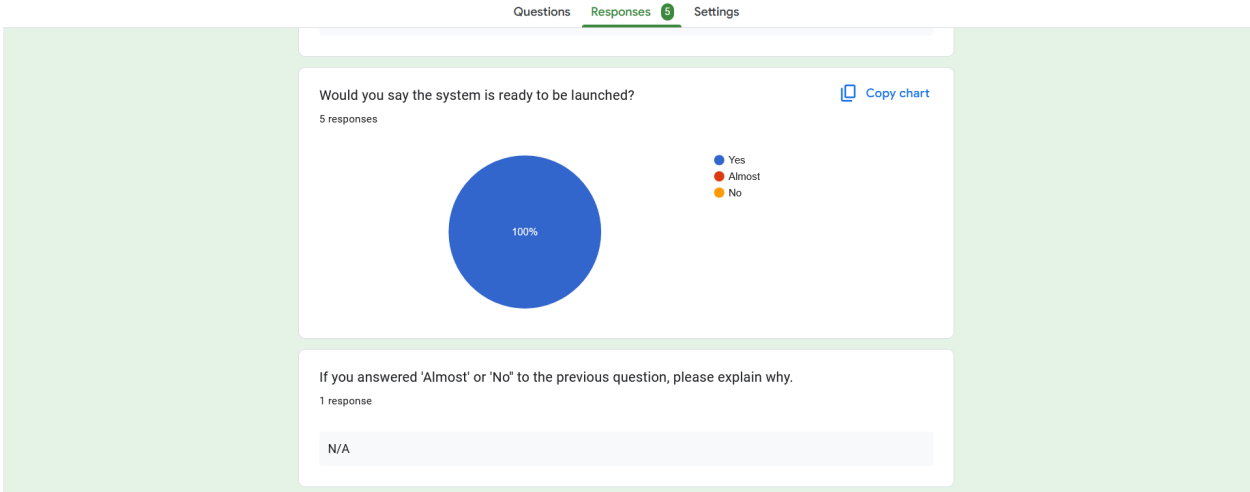
(<https://docs.google.com/forms/d/e/1FAIpQLSf7iFTOtMTGfQd17pkCjx21ouwu1e5npW4eH8Yz-1FazwiV3A/viewform?usp=preview>)

The **Usability** test google form displayed an average rating of 5 stars after sending to the first 5 reviewers (when asked to rate their initial experience using the system).





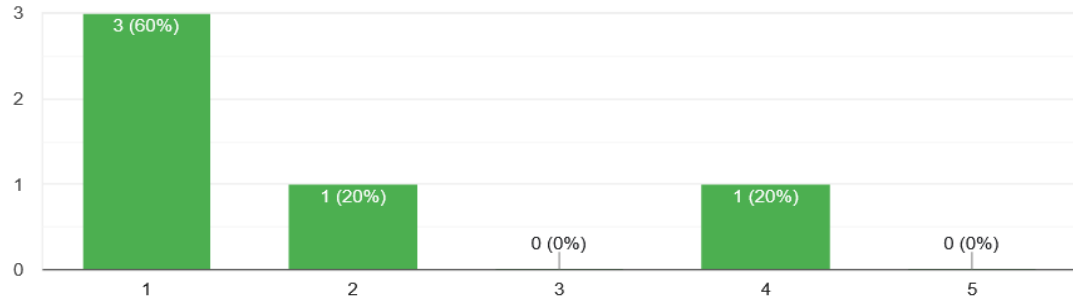




I found the system unnecessarily complex.

[Copy chart](#)

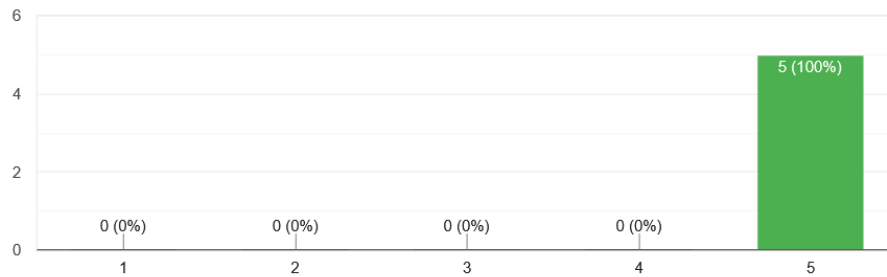
5 responses



I would imagine that most people would learn to use this system very quickly.

[Copy chart](#)

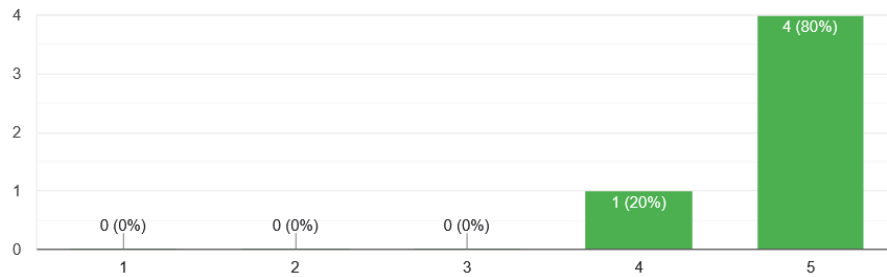
5 responses



I felt very confident using the system.

[Copy chart](#)

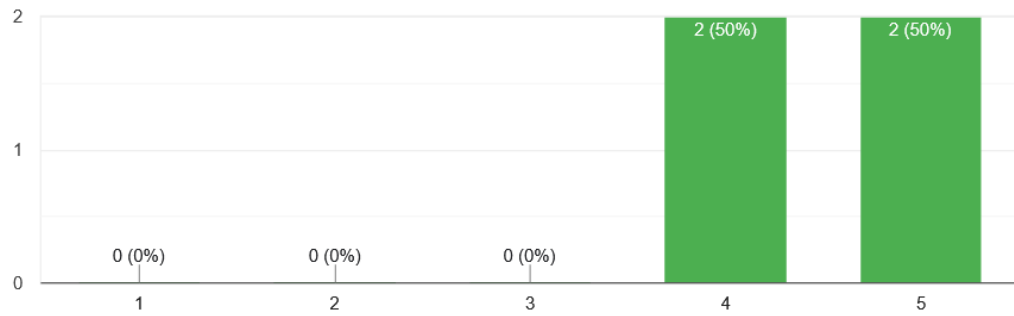
5 responses



If you tested out the fun, interactive versions of any of the 5 templates, would you say that the navigation was intuitive?

[Copy chart](#)

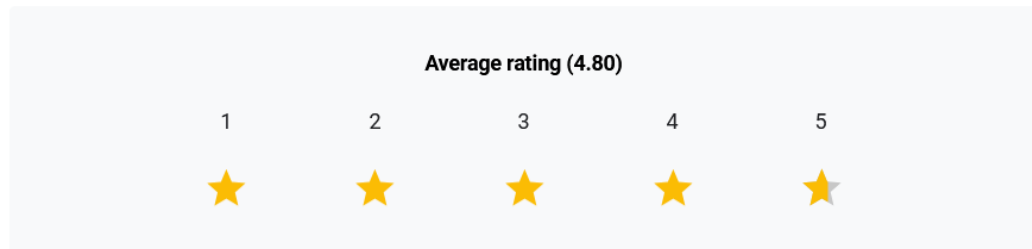
4 responses



Please give your overall rating of the platform

[Copy chart](#)

5 responses



An overall rating of 4.8 stars was calculated from the Usability test.

8. End-To-End Testing

Purpose:

End-To-End (e2e) Testing is performed to validate the complete flow of an application from start to finish, ensuring that all components of the system work together as expected. The purpose of E2E testing is to simulate real-world user scenarios, covering interactions across the frontend, backend, database, APIs, and external services. By testing the system as a whole, it helps identify integration issues, broken workflows, or unexpected behavior that might not surface in unit or integration tests. Ultimately, E2E testing provides confidence that the application delivers a seamless and reliable experience to users in a production-like environment.

Tools used:

The E2E testing was done using the Cypress automated testing tool.

Where tests can be found:

In the frontend of the system directory under "Cypress"

Portfolio-Portal/frontend/Cypress/

9. Testing Standards & Best Practices

- Follow IEEE 829 (Software Test Documentation Standard).
 - Apply Test-Driven Development (TDD) where feasible.
 - Automate repetitive tests (using Jest, Cypress, Selenium).
 - Maintain traceability between requirements, test cases, and defects.
-

10. Test Environment

- Hardware: Standard desktop & mobile configurations.
- Software:
 - Browsers (latest stable versions).
 - Backend: Node.js.
 - Database: Supabase
 - Cloud hosting: Railway
- Tools:
 - Unit testing: Jest (<https://jestjs.io/>)
 - End-to-end: Cypress (<https://www.cypress.io/>)

11. Defect & Risk Management

Defect Management:

- All defects to be logged in GitHub Issues.
- Each defect will include:
 - Unique ID, severity, priority.
 - Steps to reproduce, screenshots/logs.
 - Assigned developer, status (open, in-progress, fixed, verified).
- Prioritisation:

- *Critical*: Blocking issues (e.g., file upload crash).
- *High*: Major feature broken (e.g., 3D template not rendering).
- *Medium*: Usability or minor logic errors.
- *Low*: Cosmetic/UI issues.

Risk Management:

Risks:

- OCR inaccuracies leading to wrong data extraction.
- Browser incompatibility issues.
- High rendering load causing performance bottlenecks.
- Mitigation:
Run compatibility tests early.
 - Use fallback solutions for OCR errors.

Optimise 3D rendering pipeline.

12. Entry and Exit Criteria

Entry Criteria:

- Requirements are finalised.
- Development environment stable.
- Test cases prepared.

Exit Criteria:

- All critical and high defects resolved
- Test coverage $\geq 80\%$.
- UAT approval obtained.

13. Continuous Testing & CI/CD

- Integrate automated tests into CI/CD pipeline (GitHub Actions).
 - Ensure regression tests run on every build.
 - Block deployments on failed critical test cases.
-