Test Plan and Results

Testing Strategy

Our testing strategy follows the unit testing and component testing paradigm using Jest as the primary testing framework. We adopted the following approach:

1. **Unit Testing**
    - **Purpose**: To verify the correctness of individual utility functions and small modules in isolation.
    - **Tools Used**: `Jest`
    - **Examples**:
        - Validating input sanitization and formatting logic.
        - Testing pure functions such as date conversions, calculations, or filtering logic
    - **Benefit**: Catches bugs early in core logic before they propagate into UI or business flow.
2. **Component Testing**
    - **Purpose**: To test **React components** in isolation, including UI rendering and user interaction.
    - **Approach:**
        - Components were rendered in a simulated DOM.
        - User events (e.g., clicks, typing, selecting dropdowns) were triggered using `fireEvent` and `userEvent`.
        - Assertions verified the visual state, presence of elements, and DOM changes (e.g., modals, lists, loading spinners).
    - **Example**: Ensuring a "Sign in with Google" button correctly triggers the auth flow and updates the UI.
3. **Component Testing**
    - **Purpose**: To isolate components and avoid real side effects from external services.
    - **Mocked:**
        - **API calls**: Using jest.mock() to simulate resolved and rejected promises.
        - **Firebase methods** (e.g., get, set, signInWithPopup): Mocked to simulate auth, reads, and writes without hitting the network.

    - **Benefit**: Tests become **deterministic, fast, and safe**, and can simulate failures like network errors.

4. **Edge Case & Negative Testing**
    - **Purpose**: To ensure the application behaves reliably in unexpected or extreme conditions.
    - **Tested Scenarios:**
        - Empty or malformed inputs.
        - Slow network or failed fetch requests.
        - Conditional UI states (e.g., when data is `null`, `undefined`, or loading).
        - Invalid user behaviour (e.g., clicking disabled buttons, submitting without input).
    - **Outcome**: Improved application robustness and graceful error handling.
5. **Code Coverage & Reporting**
    - **Coverage Tool:** Jest
    - **Upload Target**: Codecov for aggregated visual coverage reports.
    - **Strategy**:
        - Ensure key business logic and user flows were covered.
        - Added additional tests where coverage was low (e.g., fallback components, modal behaviours).
        - Compared local coverage (shown in GitHub Actions) with **Codecov reports** to identify discrepancies.
    - **Common Discrepancy Reasons:** Path mismatches, config inconsistencies, or files not covered due to mocking or exclusions.

Test Environment

- **Testing Framework**: Jest
- **Component Testing**: React Testing Library
- **Mocking Tools**: jest.fn(), jest.mock()

Test Cases

Description	Input	Expected Output	Status
Renders all hero slider images on Home page	Render <Home /> via renderWithRouter()	All 8 hero images with alt text matching /Hero/i	✅ Passed

		should be found and rendered	
Renders category cards with correct text on Home page	Render <Home /> via renderWithRouter()	Text elements: "Explore Categories", "Post and Apply for Jobs", "Manage Tasks & Milestones", and "Secure Milestone-Based Payment" should appear in the document	✅ Passed
Handles fetch error in AdminReports component	Mock API call (get) to reject with error 'Failed to fetch users'; render <AdminReports />	global.alert should be called with 'Error fetching users:' and the error message	✅ Passed
Writes user to database if user does not exist (SignUp flow)	Render <SignUp />; simulate "Client" selection; click "Sign in with Google" button	set (mocked DB write function) should be called to write new user to the database	✅ Passed
Filters categories based on search term matching name	Render <Client /> in a router; dismiss welcome message; enter 'jobs' into search input	Only categories with names matching 'jobs' are displayed; others are filtered out	✅ Passed
Displays current and previous jobs based on job status	Render <ConTasksClients />	Text based on current job and previous job should both appear in the document	✅ Passed

Additional test cases were added to increase code coverage and test logic branching, error boundaries, and conditional rendering.

Code Coverage

Jest was configured to collect code coverage using jest –coverage

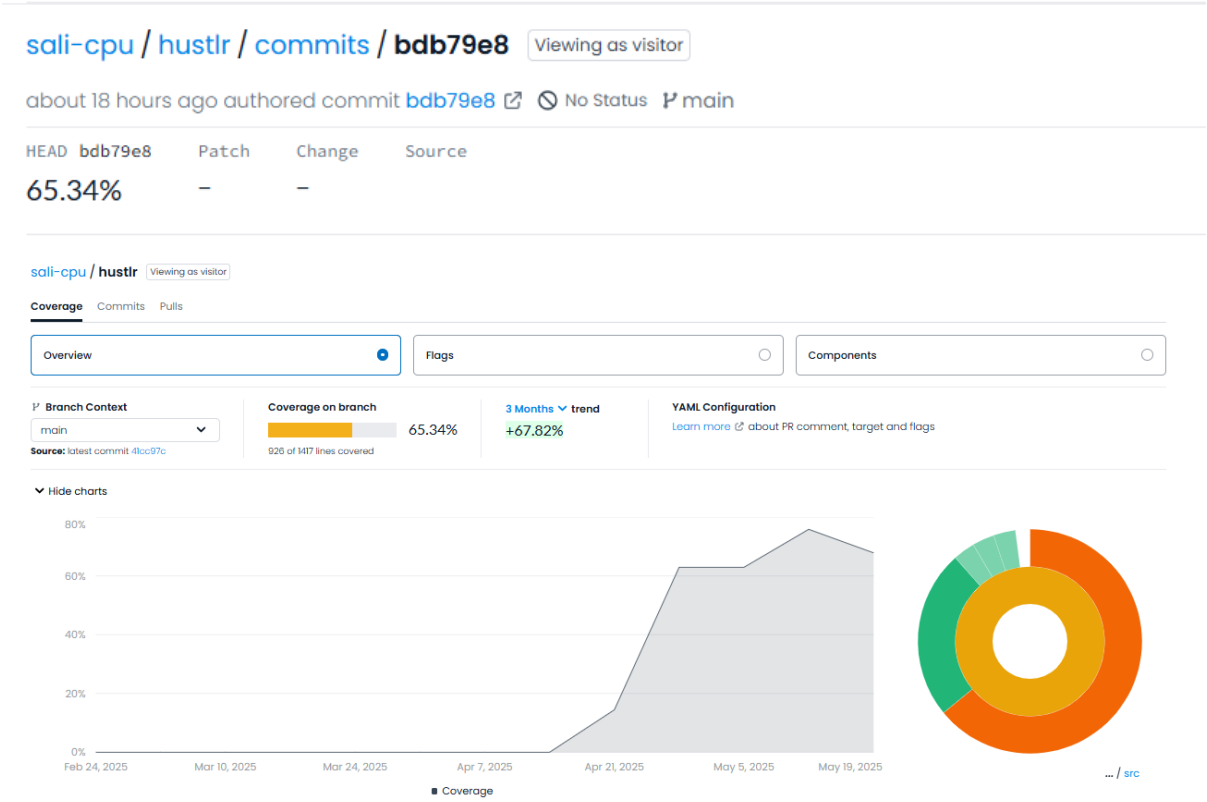
Coverage Report Summary

Metric	**Coverage**
Statements	64.67
Branch	51.76
Functions	63.52

Lines	65.26

Sample Screenshots

The code was also uploaded to **Codecov** and it showed the following results. The code coverage percentage on Github was different from the one uploaded to Github. GitHub Actions might show only line coverage (e.g., from Jest), while Codecov could factor in branch, statement, or function coverage, leading to variations. Codecov also analyses all source files, even those not explicitly tested, while Github only show tested files. For example, files excluded by .gitignore, jest.config.js, or nyc.config.js may not be uniformly applied.



File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	64.67	51.76	63.52	65.26	
src	100	100	100	100	
App.js	100	100	100	100	
firebaseConfig.js	100	100	100	100	
index.js	100	100	100	100	
src/components	92.53	85.71	88.88	93.44	
Footer.js	100	100	100	100	
FooterClient.js	100	100	100	100	
Header.js	100	100	100	100	
HeaderAdmin.js	100	100	100	100	
HeaderClient.js	100	87.5	100	100	21
HeaderFreelancer.js	70.58	62.5	57.14	73.33	29-35
RouteTracker.js	100	100	100	100	
src/pages	62.44	50.16	61.12	63.08	

```
Snapshot Summary
 › 1 snapshot written from 1 test suite.

Test Suites: 29 failed, 6 passed, 35 total
Tests:       121 failed, 128 passed, 249 total
Snapshots:   1 written, 1 total
Time:        24.367 s
Ran all test suites.
```