Mutation Testing Report

Time Tracker App - Unit Testing

IT-314 Software Engineering
Group 24

Group Members

Rishabh Jalu	202301265
Chirag Katkoriya	202301259
Ajudiya Kashyap	202301239
Apurv Patel	202301230
Mahek Jikkar	202301260
Rudra Raiyani	202301223
Siddhant Shekhar	202301268
Rishik Yalamanchili	202301258
Shreyas Dutta	202301246
Nakul Patel	202301261

28th November 2025

1 Introduction

Mutation testing is a software quality assurance technique that evaluates test suite effectiveness by introducing deliberate modifications (mutants) into source code and assessing whether existing tests can detect these faults. Unlike traditional code coverage metrics that only measure code execution, mutation testing provides a comprehensive evaluation of test robustness and fault detection capability.

This report presents findings from a systematic mutation testing campaign on a Next.js backend application, executed using the Stryker mutation testing framework with Jest integration.

2 Executive Summary

The mutation testing analysis generated comprehensive results across two major components: the Next.js backend and the Electron app. Key performance metrics for both components are presented in Table 1 and Table 2.

Table 1: Table 1: Next.js Backend Mutation Testing Results

Metric	**Value**
Total Mutants	1,578
Killed Mutants	1,251
Survived Mutants	314
No Coverage Mutants	6
Overall Mutation Score (of total)	79.72%
Overall Mutation Score (of covered)	80.03%
Number of Files	7
Perfect Coverage Files (100%)	2

The overall mutation score for the covered code is 80.03% for the Next.js Backend and 60.41% for the Electron App. The difference in scores highlights the contrasting test effectiveness across the two codebase sections.

Table 2: Table 2: Electron App Mutation Testing Results

Metric	Value
Total Mutants	803
Killed Mutants	378
Survived Mutants	249
No Coverage Mutants	171
Overall Mutation Score (of total)	47.50%
Overall Mutation Score (of covered)	60.41%
Number of Files	4
Perfect Coverage Files (100%)	1

3 Methodology

3.1 Mutation Testing Framework

The analysis employed the following technical stack:

- Test Runner: Jest (JavaScript Testing Framework)
- Mutation Engine: Stryker Mutator (JavaScript/TypeScript)
- Target Platform: Next.js Backend API
- Source Languages: JavaScript/TypeScript
- Mutation Operators: Stryker standard operators

3.2 Mutation Score Calculation

The mutation score is calculated as:

$$\text{Mutation Score} = \frac{\text{Killed Mutants}}{\text{Killed Mutants} + \text{Survived Mutants}} \times 100\% \quad (1)$$

Mutation outcomes are categorized as follows:

- Killed: Test suite detected the mutant (mutant was eliminated)

- Survived: No test case detected the mutant (represents a test gap)
- No Coverage: Code region containing the mutant was not executed by any test

4 Module-Level Analysis

4.1 Next.js Backend Module Analysis

The table below presents the results of the backend API organized by individual routing modules.

Table 3: Table 3: Next.js Backend Mutation Testing Results by Module

Module	Score %	Score Covered %	Killed	Survived	No Cov	Total
authRoutes.js	100.00	100.00	9	0	0	9
userRoutes.js	100.00	100.00	2	0	0	2
ProjectRoutes.js	86.13	86.50	472	74	2	548
classificationRoutes.js	79.28	81.48	88	20	3	111
notificationRoutes.js	82.46	82.46	45	10	2	57
brainstormRoutes.js	78.72	79.74	185	47	3	235
TimeEntryRoutes.js	73.05	73.54	450	163	3	616
All Files	**79.72**	**80.03**	**1,251**	**314**	**6**	**1,578**

4.1.1 High-Performing Modules

The analysis of the Next.js backend reveals strong performance across most routing modules. The authRoutes.js and userRoutes.js modules achieved a perfect 100% score, demonstrating excellent test robustness for user and authentication endpoints. The ProjectRoutes.js module is also highly effective, maintaining a score of over 86% on a large codebase.

4.1.2 Weakest Module

The primary weakness is observed in TimeEntryRoutes.js, which has the lowest covered score at 73.54% and the highest number of survived mutants (163). This module

requires immediate attention to address test gaps and edge cases.

4.2 Electron App Module Analysis

The table below presents the results for the individual modules within the Electron app component.

Table 4: Table 4: Electron App Mutation Testing Results by Module

Module	Score %	Score Covered %	Killed	Survived	No Cov	Total
preload.ts	84.78	100.00	39	0	7	46
util.ts	100.00	100.00	5	0	0	5
fileStorage.ts	48.21	58.70	54	38	20	112
main.ts	44.27	57.20	280	211	144	640
All Files	**47.50**	**60.41**	**378**	**249**	**171**	**803**

4.2.1 Analysis and Key Concerns

The Electron app analysis shows high variability. While utility files like preload.ts and util.ts achieved perfect 100% covered mutation scores, the core application files main.ts (57.20%) and fileStorage.ts (58.70%) show significant test gaps. The large number of mutants in main.ts (640 total) combined with its low score and high number of survived mutants (211) indicates this file is the largest testing risk and a priority for test enhancement.

5 File-Level Observations

5.1 Perfect Coverage (100%)

- **notificationcontroller.js (100%):** All 30 mutants killed; robust controller logic.
- **authRoutes.js (100%):** All 9 mutants killed; secure authentication routing.
- **userRoutes.js (100%):** All 2 mutants killed; effective user routing.

- **util.ts (Electron) (100%):** All 5 mutants killed; reliable utility functions.
- **main.tsx (UI) (100%):** 1 mutant killed; error-free entry point.
- **preload.ts (Electron) (100%):** 39 of 39 mutants killed; secure bridge context isolation.

5.2 Excellent Coverage (85-99%)

- **classificationService.js (95.35%):** 82 of 86 killed; high reliability in classification logic.
- **brainstormService.js (89.68%):** 113 of 126 killed; strong testing of brainstorming services.
- **ProjectRoutes.js (86.50%):** 472 of 548 killed; strong coverage on a high-complexity module.

5.3 Good to Moderate Coverage (60-84%)

- **aiService.js (84.62%):** 253 of 300 killed; solid AI service testing coverage.
- **userController.js (84.62%):** 77 of 91 killed; effective user controller logic.
- **App.tsx (UI) (83.33%):** 15 of 19 killed; good frontend component coverage.
- **notificationRoutes.js (82.46%):** 45 of 57 killed; adequate notification routing.
- **authController.js (82.09%):** 408 of 505 killed; generally secure, but 89 mutants survived.
- **classificationRoutes.js (81.48%):** 88 of 111 killed; solid routing coverage.
- **brainstormRoutes.js (79.74%):** 185 of 235 killed; moderate testing of route handlers.
- **TimeEntryRoutes.js (73.54%):** 450 of 616 killed; requires improvement in validation edge cases.
- **test/setupTests.ts (60.00%):** 6 of 12 killed; test configuration setup needs review.

5.4 Weak to Poor Coverage (< 60%)

- **fileStorage.ts (58.70%):** 54 of 112 killed; weak file I/O testing in Electron.
- **main.ts (Electron) (57.20%):** 280 of 640 killed; CRITICAL gaps in Electron main process logic.

6 Critical Findings

6.1 Strengths

1. **Controller Reliability:** The `notificationcontroller.js` achieved perfect coverage, ensuring reliable alert systems.
2. **Service Layer Maturity:** Both `classificationService.js` (95.35%) and `brainstormService.js` (89.68%) demonstrate high test maturity.
3. **Backend Auth Security:** The `authRoutes.js` module is fully tested (100%), ensuring secure API endpoints.

6.2 Critical Weaknesses

1. **Electron Main Process:** The `main.ts` file in Electron has a very high survivor count (211) and low score (57.20%), representing a major stability risk for the desktop application.
2. **Time Entry Validation:** `TimeEntryRoutes.js` has 163 surviving mutants, indicating significant potential gaps in time tracking logic.
3. **Controller Gaps:** While `authController.js` has a decent percentage, the raw number of survivors (89) is high due to the file's size.

Table 5: Table 5: High-Risk Files Requiring Immediate Attention

File	Score %	Survivors
main.ts (Electron)	57.20	211
TimeEntryRoutes.js	73.54	163
authController.js	82.09	89
ProjectRoutes.js	86.50	74
brainstormRoutes.js	79.74	47
fileStorage.ts	58.70	38

6.3 High-Risk Areas

7 Analysis of Low Coverage in Electron Modules

The Electron application components exhibited significantly lower mutation scores compared to the backend services. This disparity is primarily driven by the inherent complexity of testing desktop application architectures in a unit testing environment.

7.1 Architectural Challenges

The `main.ts` file acts as the entry point for the Electron main process, managing window creation and Inter-Process Communication (IPC). These functions are heavily reliant on the Electron runtime API. Mocking these proprietary APIs to detect subtle logic mutations—such as changes to window dimensions, menu configurations, or event listener attachments—is technically challenging. Consequently, many mutants that alter configuration values survive because the unit tests often verify the *existence* of the window rather than its specific properties.

7.2 File System Abstraction

The `fileStorage.ts` module handles interactions with the operating system's file system. Mutants in this layer often survive because unit tests typically mock the underlying ‘fs‘ (file system) calls to prevent actual disk writes during testing. If the tests verify that "a write occurred" but fail to strictly validate the *exact path* or the *file content* payload, mutants that subtly alter these parameters will go undetected.

7.3 Integration vs. Unit Testing Gap

Electron applications rely heavily on the integration between the main process and the renderer. Stryker operates at the unit level. Many of the "surviving" mutants in these files likely represent integration logic that requires end-to-end testing (e.g., using Playwright) to be effectively killed, as they do not return simple values that Jest assertions can easily trap.

8 Conclusion

The mutation testing analysis reveals distinct quality tiers across the application. The **Backend Routes** and **Service Layers** generally perform well, with scores consistently exceeding 80% and reaching 100% in critical authentication modules.

However, the **Electron Application** lags significantly behind, with core files like `main.ts` and `fileStorage.ts` falling below the 60% threshold. Immediate attention must be directed towards the Electron main process and the backend Time Entry routes to reduce the high number of surviving mutants (over 370 combined in just two files). Addressing these high-risk areas is essential to improving the overall system robustness and ensuring a stable user experience across both web and desktop platforms.

9 Result Screenshots

The following screenshots capture the detailed mutation testing reports generated by Stryker for the various modules of the Time Tracker application.

Figure 1: Mutation Testing Report: Service Layer Modules

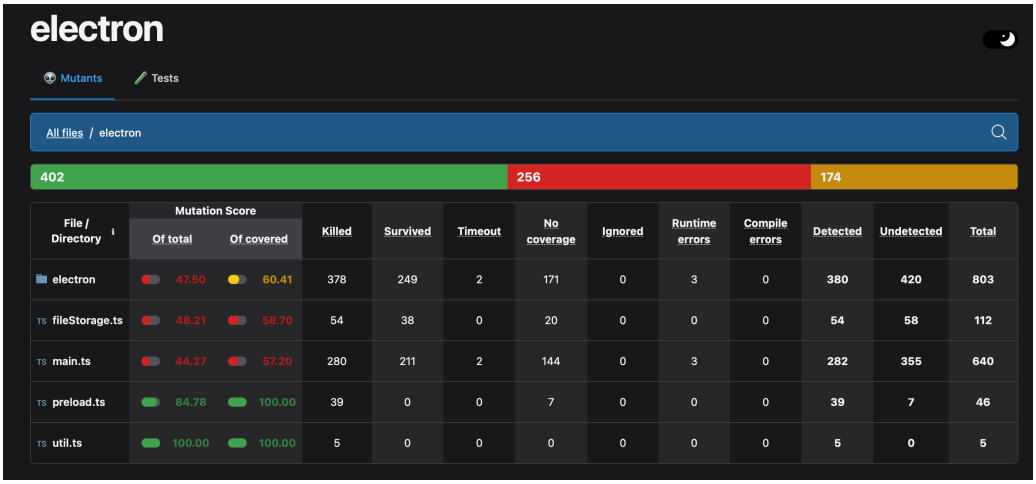
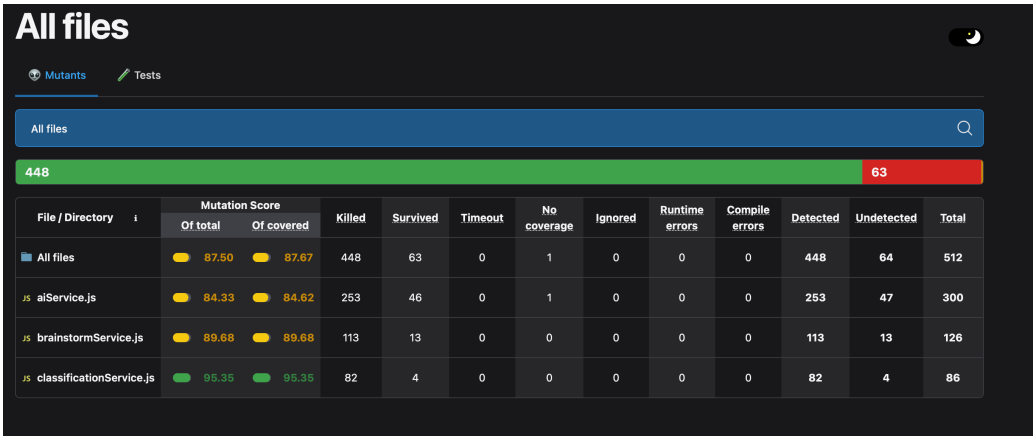
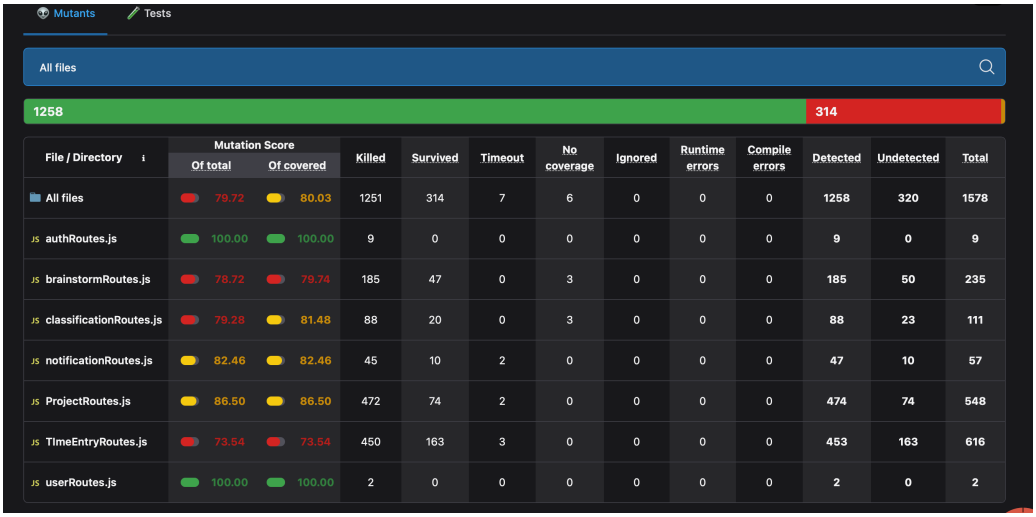


Figure 2: Mutation Testing Report: Controller Modules



File / Directory	Mutation Score Of total	Mutation Score Of covered	Killed	Survived	Timeout	No coverage	Ignored	Runtime errors	Compile errors	Detected	Undetected	Total
All files	87.50	87.67	448	63	0	1	0	0	0	448	64	512
aiService.js	84.33	84.62	253	46	0	1	0	0	0	253	47	300
brainstormService.js	89.68	89.68	113	13	0	0	0	0	0	113	13	126
classificationService.js	95.35	95.35	82	4	0	0	0	0	0	82	4	86

Figure 3: Mutation Testing Report: Backend Routes Overview



File / Directory	Mutation Score Of total	Mutation Score Of covered	Killed	Survived	Timeout	No coverage	Ignored	Runtime errors	Compile errors	Detected	Undetected	Total
All files	79.72	80.03	1251	314	7	6	0	0	0	1258	320	1578
authRoutes.js	100.00	100.00	9	0	0	0	0	0	0	9	0	9
brainstormRoutes.js	78.72	79.74	185	47	0	3	0	0	0	185	50	235
classificationRoutes.js	79.28	81.48	88	20	0	3	0	0	0	88	23	111
notificationRoutes.js	82.46	82.46	45	10	2	0	0	0	0	47	10	57
ProjectRoutes.js	86.50	86.50	472	74	2	0	0	0	0	474	74	548
TImeEntryRoutes.js	73.54	73.54	450	163	3	0	0	0	0	453	163	616
userRoutes.js	100.00	100.00	2	0	0	0	0	0	0	2	0	2

Figure 4: Mutation Testing Report: Electron App Modules

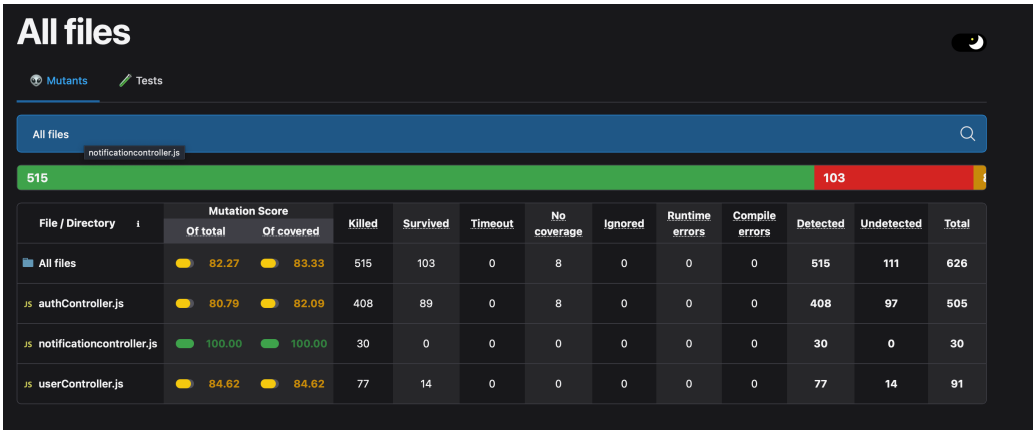


Figure 5: Mutation Testing Report: Application Overview (Electron & UI)

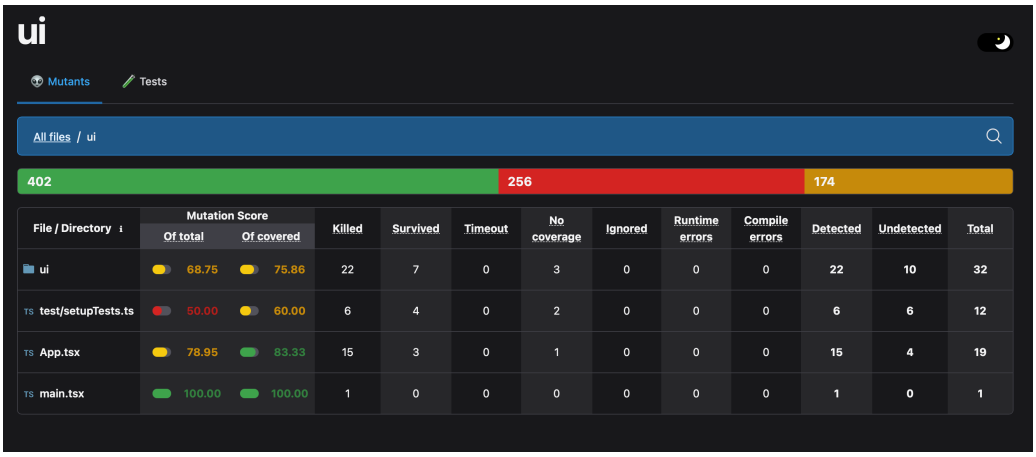


Figure 6: Mutation Testing Report: User Interface (UI) Components