Test Plan Report

Cloud Cost Intelligence Platform

Michael, Ishan, Sean, Bryana, Tony

CMSC 495 Computer Science Capstone

Instructor: Lynda Metallo

February 10, 2026

Table of Contents

1. Purpose

This Test Plan outlines the testing strategy, scope, resources, schedule, and test cases for the Cloud Cost Intelligence Platform developed by The Code Collective for the CMSC 495 Capstone Project.

The Cloud Cost Intelligence Platform is a web dashboard that brings together cloud spending data from AWS and Azure. It helps organizations find waste, track budgets, and manage cloud costs. The system uses simulated data to show how the concept works.

This document follows the IEEE 829 standard for software test documentation. It covers all required testing types, including unit, integration, functionality, performance, stress, and acceptance testing.

2. Scope and Objectives

2.1 Testing Scope

This test plan includes the types of testing listed in the course rubric.

- **Unit Testing:** Individual function and module verification
- **Integration Testing:** Component interaction and data flow verification
- **Functionality Testing:** Feature-level behavior validation against requirements
- **Performance Testing:** Response time and throughput measurement
- **Stress Testing:** System behavior under peak and overload conditions
- **Acceptance Testing:** End-to-end user workflow validation

2.2 Testing Objectives

1. Ensure that every functional requirement is implemented accurately.
2. Validate data flows correctly from the database to the frontend display
3. Ensure the system meets performance requirements under normal load
4. Identify defects before final delivery
5. Confirm user workflows are intuitive and complete

3. Testing Approach

3.1 Unit Testing Approach

Unit tests verify individual functions and modules in isolation. We use Pytest for Python backend testing and Jest for JavaScript frontend testing. Each unit test focuses on a single function with mocked dependencies to ensure isolation.

3.2 Integration Testing Approach

Integration tests verify that components work together correctly. We test the connections between frontend, backend API, and database using Docker containers to simulate the production environment. Focus areas include API contracts, data transformation, and cross-boundary error handling.

3.3 Functionality Testing Approach

Functionality tests verify that features meet user requirements. We use automated testing with Playwright to validate each feature against the Project Design document requirements. Tests are executed in a browser against the running application.

3.4 Performance Testing Approach

Performance tests measure response times and throughput under normal conditions. We use Apache Bench (ab) or similar tools to generate controlled load and measure API response times. Browser developer tools measure frontend rendering performance.

3.5 Stress Testing Approach

Stress tests verify system behavior under peak load and failure conditions. We simulate concurrent users and monitor errors, degradation, and recovery. Tests identify breaking points and verify graceful degradation.

3.6 Acceptance Testing Approach

Acceptance tests validate complete user workflows end-to-end. Automated testing with Playwright follows realistic user scenarios to confirm the system meets business objectives. Tests are performed from the user's perspective without knowledge of internal implementation. Acceptance test cases are implemented as End-to-End tests (TC-E2E-004 through TC-E2E-FR08) using Playwright, validating complete user workflows against functional requirements FR-01 through FR-08.

3.7 Automated Testing Strategy

The project employs a comprehensive automated testing approach using industry-standard frameworks:

Test Frameworks:

Framework	Version	Purpose
Jest	v29.7	JavaScript unit testing for frontend business logic
Pytest	v9.0	Python unit, integration, and functionality testing
Playwright	v0.7.2	Browser-based end-to-end testing

Test Pyramid Distribution:

Layer	Count	Percentage	Purpose
Unit Tests	102	70%	Function-level verification
Integration Tests	8	5%	Component interaction
Functionality Tests	6	4%	Feature Validation
E2E Tests	18	12%	Full workflow validation
API Tests	8	5%	Endpoint verification
Performance Tests	4	3%	Response time validation
Stress Tests	2	1%	Load handling verification
Total	**148**	**100%**	

Automation Coverage:

- All unit tests are fully automated
- All integration tests are fully automated
- All E2E tests are fully automated
- Performance and stress tests use Apache Bench and manual verification

Execution Commands:

```
bash

# JavaScript Unit Tests (94 tests)
npm test --prefix src/frontend

# Python Tests – requires Docker (54 tests)
TEST_API_URL=http://localhost:5000/api/v1 pytest tests/ -v
```

Continuous Integration:
With each commit, tests run automatically thanks to GitHub repository integration. Test results are logged and tracked to detect regression.

4. Features to be Tested

The following features will be tested according to the project requirements:

Feature ID	Feature Name	Test Types
F-001	Multi-cloud cost data aggregation (AWS + Azure)	Unit, Integration, Functionality, Acceptance
F-002	Spending trend visualization	Integration, Functionality, Acceptance
F-003	Waste identification (unused/idle resources)	Functionality, Acceptance
F-004	Daily budget tracking dashboard	Functionality, Acceptance
F-005	Client filtering and selection	Integration, Functionality, Acceptance
F-006	API endpoints for cost data	Unit, integration, Performance
F-007	Database queries and data retrieval	Unit, Integration, Performance
F-008	System under load	Performance, Stress

5. Features Not to be Tested

The following features are excluded from the testing scope:

- Real cloud provider API integration (using simulated data)
- User authentication/authorization (not implemented for proof-of-concept)
- PDF export functionality (CSV export only for Phase I)
- Third-party library internal functionality
- Browser compatibility beyond Chrome/Firefox
- Mobile device responsiveness

6. Test Environment

6.1 Hardware Requirements

Component	Specification
Development Machine	Minimum 8GB RAM, 4-core CPU, 50GB available storage
Network	Internet connection for Azure SQL Database access

6.2 Software Requirements

Component	Version/Details
Operating System	Windows 10/11, macOS, or Linux (Ubuntu 20.04+)
Docker	Docker Desktop 4.x with Docker Compose
Python	Python 3.9+
Node.js	Node.js 18+ (for Jest testing)
Web Browser	Chrome 100+ or Firefox 100+
Database	Azure SQL Database (serverless tier)
Testing Frameworks	Pytest 9.x, Jest 29.x, Playwright 0.7.2

6.3 Environment Configuration

1. Clone the repository from GitHub: https://github.com/michaelallenus217-lang/CMSC495-CloudCost
2. Configure environment variables for the database connection
3. Run `docker compose up --build -d` to start containers
4. Complete Azure authentication (device code flow)
5. Seed database with test data
6. Verify frontend is accessible at http://localhost:8080
7. Verify backend API accessible at http://localhost:5000

7. Test Data Preparation

7.1 Test Data Requirements

Data Type	Quantity	Purpose
Client records	5 clients	Filter testing
Provider records	2 providers (AWS + Azure)	Multi-cloud testing
Service records	30 services	Service-level testing
Usage records	100+ entries	Cost aggregation testing
Budget configurations	1 per client	Budget tracking testing

7.2 Data Seeding Process

1. Database is pre-seeded by Database Lead (Tony)
2. Verify data loaded: Run validation queries against Azure SQL
3. Record baseline values for comparison during testing:
    a. AWS Total: $3.30
    b. Azure Total: $20.00
    c. Google Cloud Total: $12.00
    d. Grand Total: $35.30

Note: Test data includes a Google Cloud provider record to validate the platform's extensibility for future multi-cloud expansion. The core feature set targets AWS and Azure per project requirements.

8. Suspension and Resumption Criteria

8.1 Suspension Criteria

Testing will be suspended when any of the following conditions occur:

- Critical defect prevents further testing (system crash, data corruption)
- Test environment becomes unavailable (Azure SQL outage, Docker failure)
- More than 50% of test cases fail in a single test type
- Test data corrupted or invalid

8.2 Resumption Criteria

Testing will resume when:

- Critical defects are resolved and verified
- Test environment is restored and validated
- Test data is reset to known good state
- QA Lead approves resumption

9. Acceptance Criteria

The system will be considered ready for release when the following criteria are met:

1. 95% of unit tests pass
2. 95% of integration tests pass
3. 90% of functionality tests pass (no critical failures)
4. Performance tests meet response time thresholds
5. Stress tests show graceful degradation (no crashes)
6. 90% of acceptance test workflows are completed successfully
7. All critical and high-priority defects are resolved
8. Team consensus that the system meets requirements

10. Defect Management Section

10.1 Defect Tracking

Defects are tracked in GitHub Issues (Repository: https://github.com/michaelallenus217-lang/CMSC495-CloudCost/issues):

Level	Definition	Response Time
Critical	System crash, data loss, security breach	Immediate
High	Feature unusable, no workaround	Within 24 hours
Medium	Feature impaired, workaround exists	Within 72 hours
Low	Cosmetic, minor inconvenience	Next sprint

Defect Lifecycle:

1. **Discovered** - Logged in GitHub Issues
2. **Triaged** - Assigned severity and owner
3. **In Progress** - Developer is working fix
4. **Fixed** - Code committed
5. V**erified** - QA confirms resolution
6. **Closed** - Merged to main

10.2 Known Defects

Defect ID	Severity	Description	Status	Owner
DEF-001	Medium	API limit caused incomplete service mapping in tests	Resolved	Michael
DEF-002	High	Frontend displays NaN for cost totals	Fixed (PR #7)	Ishan
DEF-003	Low	Duplicate entry – consolidated to DEF-002	Closed	Michael
DEF-004	Medium	CSV export uses 30-day filter when dashboard shows 90-day	Fixed (PR #7)	Michael
DEF-005	Medium	Costs do not match dashboard and CSV	Open	TBD
DEF-006	Low	CSV export missing header information	Open	TBD
DEF-007	Medium	Chart dates one day off due to timezone	Fixed (PR #7)	Michael

11. Schedule

11.1 Testing Phases

Phase	Activities	Dates	Status
Unit Testing	JavaScript and Python unit tests	Jan 27 – Feb 7	Complete
Integration Testing	API and database connectivity tests	Feb 3 – Feb 7	Complete
Functionality Testing	FR-01 through FR-08 validation	Feb 5 – Feb 7	Complete
End-to-End Testing	Full workflow browser automation	Feb 6 – Feb 7	Complete
Regression Testing	Full suite after defect fixes	Feb 7 – Feb 10	InProgress
User Acceptance	Final demo and validation	Feb 17 – Feb 21	Scheduled

11.2 Key Milestones

Milestone	Due Date	Status
Test Plan Document Submission	February 10, 2026	In Progress
All Automated Tests Passing	February 7, 2026	Complete (143/143)
DEF-002 Resolution (NaN Display)	February 9, 2026	Complete (PR #7)
Phase II Source Code	February 17, 2026	Scheduled
User Guide	February 17, 2026	Scheduled
Final Report	February 24, 2026	Scheduled

12. Roles and Responsibilities

Name	Role	Responsibilities
Michael	Test Lead	• Develop and maintain automated test suite (143 tests) • Define test strategy and standards • Track defects and coordinate resolution • Review test results and approve releases • Ensure rubric compliance for deliverables
Ishan	UI/UX Testing	• Validate frontend functionality against FR requirements • Test cross-browser compatibility • Verify responsive design and accessibility
Sean	API Testing	• Validate all API endpoints return correct data • Test error handling and edge cases • Verify JSON response structure
Tony	Data Validation	• Verify database schema integrity • Validate data relationships and constraints • Ensure seed data accuracy
Bryana	Test Documentation	• Document test cases and procedures • Maintain test execution records • Prepare test summary reports • Archive test artifacts

13. Test Cases

This section contains all test cases organized by testing type. Each test case includes a unique ID, purpose, preconditions, resources, detailed steps, and expected results.

TC-U-001 | Unit Test | ✓ PASS

Verify database connection function establishes connection to Azure SQL Database
REQ-DB-001: System shall connect to Azure SQL Database

Preconditions:
1. Azure SQL Database is running
2. Connection string in environment
3. Valid database credentials

Resources:
1. Python 3.x with pydodbc library
2. Azure SQL Database instance
3. Docker containers running

Test Steps:
1. Call /health/db endpoint – Status 200
2. Verify response contains db status – db: 'connected'
3. Verify response contains status – status: 'ok'
4. Confirm connection released – No errors

Expected Results:
HTTP Status 200, Response: {db: 'connected', status: 'ok'}

TC-U-003 | Unit Test | ✓ PASS

Verify cost calculation function correctly sums provider spending
REQ-CALC-001: System shall calculate total cloud spending

Preconditions:
1. Database connection available
2. Services mapped to providers in database
3. Usage records exist with cost data

Resources:
1. Python 3.x with pytest
2. requests library
3. Test data fixtures

Test Steps:
1. GET /providers – 3 providers returned
2. GET /services?limit=100 – 30 services mapped
3. Calculate totals by provider – AWS $3.30, Azure $20.00, GCP $12.00
4. Verify sum equals grand total – $35.30 = $35.30 ✓

Expected Results:
All provider costs calculated correctly, sum matches grand total

TC-I-002 | Integration Test | ✓ PASS

Verify backend API connects to and retrieves data from Azure SQL Database
REQ-INT-002: Backend shall retrieve data from database

Preconditions:
1. Docker containers running (frontend, backend, database)
2. Backend accessible at localhost:5000
3. Database populated with test data

Resources:
1. Docker with docker-compose
2. Python Flask backend
3. Azure SQL Database

Test Steps:
1. Start Docker containers – All services running
2. GET /health/db – Status 200, db: connected
3. GET /clients – Client records returned from DB
4. Verify data matches seed values released – Data integrity confirmed

Expected Results:
Backend successfully queries Azure SQL, returns valid JSON with client data

TC-F-001 | Functionality Test | ✓ PASS

Verify cost breakdown by provider displays accurate totals (FR-01)
FR-01: Display costs from AWS and Azure on single dashboard

Preconditions:
1. All API endpoints accessible
2. Services linked to providers
3. Usage data exists in database

Resources:
1. pytest framework
2. requests library
3. Running backend server

Test Steps:
1. GET /providers – 3 providers retrieved
2. GET /services with provider mapping – Services linked to providers
3. GET /usages – Usage costs retrieved
4. Calculate cost per provider – Totals match expected values

Expected Results:
Provider costs calculated: AWS $3.30, Azure $20.00, Google Cloud $12.00

TC-E2E-001 | End-toEnd Test | ✓ PASS

Verify dashboard page loads and displays all required UI elements
FR-01: Dashboard shall display costs within 3 seconds

Preconditions:
1. Docker containers running (frontend, backend, database)
2. Test data seeded in database
3. Browser automation configured

Resources:
1. Playwright v0.7.2
2. Chromium browser
3. http://localhost:8080

Test Steps
1. Launch Chromium browser – Browser opens
2. Navigate to http://localhost:8080 – Page loads < 3 sec
3. Verify cost summary cards visible – Cards displayed
4. Verify trend chart rendered – Chart.js canvas present
5. Check for JavaScript errors – No console errors

Expected Results:
Dashboard loads in 1.2s, all UI elements render correctly, no JS errors

TC-E2E-FR01 | End-to-End Test | ✓ PASS

Verify dashboard cost values match database values (FR-01 validation)
FR-01: Dashboard shall display accurate provider cost totals

Preconditions:
1. Docker containers running
2. Database contains known cost values
3. API accessible at localhost:5000

Resources:
1. Playwright v0.7.2
2. Chromium browser
3. requests library for API validation

Test Steps
1. Query API for services (limit=100) – All 30 services retrieved
2. Calculate expected provider totals – Sums computed from API
3. Load dashboard in browser – Page renders
4. Extract displayed cost values – Values captured from DOM
5. Compare frontend vs API values – Values match

Expected Results:
Frontend displays same totals as API: Total $35.30, all providers accurate

Full Test Suite Summary

Complete automated test execution results

Total Tests: 148
Passed: 148
Failed: 0
Pass Rate: 100%

Test Breakdown: by type

- JavaScript Unit Tests (Jest): 94 passed
- Python Unit Tests (TC-U): 8 passed
- Integration Tests (TC-I): 8 passed
- Functionality Tests (TC-F): 6 passed
- End-to-End Tests (TC-E2E): 18 passed
- API Endpoint Tests: 8 passed
- Performance Tests: 4 passed
- Stress Tests: 2 passed

Appendix: Test Case Summary

Summary: 148 total tests | 148 passed | 0 failed | 100% pass rate

JavaScript Unit Tests (Jest) — Tests:

Test ID	Description	Requirement	Status
TC-U-JS-001	formatCurrency formats positive numbers	FR-08	Pass
TC-U-JS-002	formatCurrency handles zero	FR-08	Pass
TC-U-JS-003	formatCurrency handles null/undefined	FR-08	Pass
TC-U-JS-004	formatCurrency handles large numbers	FR-08	Pass
TC-U-JS-005	formatCurrency rounds to 2 decimals	FR-08	Pass
TC-U-JS-006	formatPercentage formats decimals	FR-08	Pass
TC-U-JS-007	formatPercentage handles zero	FR-08	Pass
TC-U-JS-008	formatPercentage handles null	FR-08	Pass
TC-U-JS-009	formatDate formats ISO strings	FR-02	Pass
TC-U-JS-010	formatDate handles invalid dates	FR-02	Pass
TC-U-JS-011	debounce delays execution	NFR-01	Pass
TC-U-JS-012	debounce cancels previous calls	NFR-01	Pass
TC-U-JS-013	validateEmail accepts valid emails	FR-07	Pass
TC-U-JS-014	validateEmail rejects invalid emails	FR-07	Pass
TC-U-JS-015	sanitizeInput removes HTML tags	NFR-02	Pass
TC-U-JS-016	sanitizeInput handles empty strings	NFR-02	Pass
TC-U-JS-017	calculateSavings computes correctly	FR-05	Pass
TC-U-JS-018	calculateSavings handles zero cost	FR-05	Pass
TC-U-JS-019	groupByProvider groups services	FR-01	Pass
TC-U-JS-020	calculateTotalCost sums all costs	FR-01	Pass
TC-U-JS-021	calculateTotalCost handles empty array	FR-01	Pass
TC-U-JS-022	calculateTotalCost handles null values	FR-01	Pass
TC-U-JS-023	filterByProvider filters correctly	FR-03	Pass
TC-U-JS-024	filterByProvider handles no matches	FR-03	Pass
TC-U-JS-025	filterByDateRange filters by dates	FR-03	Pass
TC-U-JS-026	filterByDateRange handles edge dates	FR-03	Pass
TC-U-JS-027	sortByDate sorts ascending	FR-02	Pass
TC-U-JS-028	sortByDate sorts descending	FR-02	Pass
TC-U-JS-029	aggregateByMonth groups monthly	FR-02	Pass
TC-U-JS-030	detectWaste flags low utilization	FR-04	Pass
TC-U-JS-031	detectWaste threshold at 30%	FR-04	Pass
TC-U-JS-032	detectWaste handles missing data	FR-04	Pass
TC-U-JS-033	generateRecommendations creates suggestions	FR-05	Pass
TC-U-JS-034	generateRecommendations min 3 items	FR-05	Pass
TC-U-JS-035	calculatePotentialSavings estimates	FR-05	Pass
TC-U-JS-036	validateBudgetInput checks ranges	FR-07	Pass
TC-U-JS-037	validateBudgetInput rejects negatives	FR-07	Pass
TC-U-JS-038	fetchFromApi handles success	REQ-API	Pass
TC-U-JS-039	fetchFromApi handles 404 errors	REQ-API	Pass

Test ID	Description	Requirement	Status
TC-U-JS-040	fetchFromApi handles network errors	REQ-API	Pass
TC-U-JS-041	fetchFromApi handles timeout	REQ-API	Pass
TC-U-JS-042	prepareChartData formats for Chart.js	FR-02	Pass
TC-U-JS-043	prepareChartData handles empty data	FR-02	Pass
TC-U-JS-044	getChartColors returns provider colors	FR-02	Pass
TC-U-JS-045	createTrendDataset builds dataset	FR-02	Pass
TC-U-JS-046	createTrendDataset multi-provider	FR-02	Pass
TC-U-JS-047	calculateTrendLine computes slope	FR-02	Pass
TC-U-JS-048	formatChartTooltip formats currency	FR-02	Pass
TC-U-JS-049	generateChartLabels creates months	FR-02	Pass
TC-U-JS-050	updateChartData refreshes chart	FR-02	Pass
TC-U-JS-051	updateCostCard updates DOM	FR-01	Pass
TC-U-JS-052	updateCostCard handles null	FR-01	Pass
TC-U-JS-053	renderWasteAlerts creates rows	FR-04	Pass
TC-U-JS-054	renderWasteAlerts sorts by savings	FR-04	Pass
TC-U-JS-055	renderWasteAlerts handles empty	FR-04	Pass
TC-U-JS-056	applyFilters updates dashboard	FR-03	Pass
TC-U-JS-057	applyFilters clears filters	FR-03	Pass
TC-U-JS-058	refreshDashboard reloads data	FR-01	Pass
TC-U-JS-059	handleClientChange updates view	FR-06	Pass
TC-U-JS-060	populateDropdowns fills selects	FR-03	Pass
TC-U-JS-061	exportToCSV generates valid CSV	FR-08	Pass
TC-U-JS-062	exportToCSV escapes commas	FR-08	Pass
TC-U-JS-063	exportToCSV handles special chars	FR-08	Pass
TC-U-JS-064	exportToCSV includes headers	FR-08	Pass
TC-U-JS-065	exportToPDF generates document	FR-08	Pass
TC-U-JS-066	exportToPDF includes branding	FR-08	Pass
TC-U-JS-067	exportToPDF paginates correctly	FR-08	Pass
TC-U-JS-068	downloadFile triggers download	FR-08	Pass
TC-U-JS-069	formatExportData prepares data	FR-08	Pass
TC-U-JS-070	getExportFilename generates name	FR-08	Pass
TC-U-JS-071	loadSettings retrieves budgets	FR-07	Pass
TC-U-JS-072	loadSettings handles API error	FR-07	Pass
TC-U-JS-073	saveBudget sends POST request	FR-07	Pass
TC-U-JS-074	saveBudget validates input	FR-07	Pass
TC-U-JS-075	updateThreshold modifies alert	FR-07	Pass
TC-U-JS-076	toggleAlerts enables/disables	FR-07	Pass
TC-U-JS-077	resetSettings restores defaults	FR-07	Pass
TC-U-JS-078	validateSettingsForm checks all	FR-07	Pass
TC-U-JS-079	checkBudgetAlerts compares limits	FR-07	Pass
TC-U-JS-080	checkBudgetAlerts triggers at 90%	FR-07	Pass
TC-U-JS-081	displayAlert shows notification	FR-07	Pass
TC-U-JS-082	dismissAlert removes notification	FR-07	Pass
TC-U-JS-083	getAlertSeverity returns level	FR-07	Pass

Test ID	Description	Requirement	Status
TC-U-JS-084	formatAlertMessage creates text	FR-07	Pass
TC-U-JS-085	navigateTo changes page	NFR-01	Pass
TC-U-JS-086	highlightActiveNav updates menu	NFR-01	Pass
TC-U-JS-087	handleBackButton manages history	NFR-01	Pass
TC-U-JS-088	getApiUrl builds correct URL	REQ-API	Pass
TC-U-JS-089	getApiUrl adds query params	REQ-API	Pass
TC-U-JS-090	API_CONFIG has required keys	REQ-API	Pass
TC-U-JS-091	ENDPOINTS defines all routes	REQ-API	Pass
TC-U-JS-092	CHART_COLORS has provider colors	FR-02	Pass
TC-U-JS-093	WASTE_THRESHOLDS defines limits	FR-04	Pass
TC-U-JS-094	DEFAULT_LIMIT is reasonable	REQ-API	Pass

Python Unit Tests (pytest) — 8 Tests:

Test ID	Description	Requirement	Status
TC-U-001	Database connection establishes	REQ-DB-001	Pass
TC-U-002	Cost data retrieval returns structure	REQ-API-001	Pass
TC-U-003	Cost calculation sums providers	REQ-CALC-001	Pass
TC-U-004	Invalid requests return errors	REQ-VAL-001	Pass
TC-U-005	Invoice structure validates	REQ-API-002	Pass
TC-U-006	Budget structure validates	REQ-API-003	Pass
TC-U-007	Service-provider link exists	REQ-DB-002	Pass
TC-U-008	Client structure validates	REQ-API-004	Pass

Integration Tests — 8 Tests:

Test ID	Description	Requirement	Status
TC-I-001	Health endpoint responds	REQ-INT-001	Pass
TC-I-002	Database connectivity verified	REQ-INT-002	Pass
TC-I-003	Chart data endpoint works	FR-02	Pass
TC-I-004	Client filter returns data	FR-06	Pass
TC-I-005	Invoice-client integrity	REQ-INT-003	Pass
TC-I-006	Service-provider integrity	REQ-INT-004	Pass
TC-I-007	Usage relationships valid	REQ-INT-005	Pass
TC-I-008	Count consistency across endpoints	REQ-INT-006	Pass

Functionality Tests — 6 Tests:

Test ID	Description	Requirement	Status
TC-F-001	Cost by provider displays (FR-01)	FR-01	Pass
TC-F-002	Spending trends calculate (FR-02)	FR-02	Pass
TC-F-003	Filter capability works (FR-03)	FR-03	Pass

Test ID	Description	Requirement	Status
TC-F-004	Budget thresholds trigger (FR-07)	FR-07	Pass
TC-F-005	Usage metrics accurate (FR-08)	FR-08	Pass
TC-F-006	Multi-client support (FR-06)	FR-06	Pass

End-to-End Test (Playwright) — 18 Tests:

Test ID	Description	Requirement	Status
TC-E2E-001	Dashboard loads < 3 seconds	FR-01	Pass
TC-E2E-002	Charts render correctly	FR-02	Pass
TC-E2E-003	Client filter updates view	FR-06	Pass
TC-E2E-004	Full workflow completes	ALL	Pass
TC-E2E-005	Navigation between pages	NFR-01	Pass
TC-E2E-006	Cost cards display values	FR-01	Pass
TC-E2E-007	Filter controls function	FR-03	Pass
TC-E2E-008	Waste alerts display	FR-04	Pass
TC-E2E-009	Trend chart renders	FR-02	Pass
TC-E2E-010	Settings page loads	FR-07	Pass
TC-E2E-011	Client selection integrity	FR-06	Pass
TC-E2E-FR01	FR-01 cost accuracy validated	FR-01	Pass
TC-E2E-FR02	FR-02 trend data validated	FR-02	Pass
TC-E2E-FR03	FR-03 filter functionality	FR-03	Pass
TC-E2E-FR04	FR-04 waste alerts display	FR-04	Pass
TC-E2E-FR05	FR-05 recommendations display	FR-05	Pass
TC-E2E-FR07	FR-07 budget settings work	FR-07	Pass
TC-E2E-FR08	FR-08 metrics accuracy	FR-08	Pass

API Endpoint Tests — 8 Tests:

Test ID	Description	Requirement	Status
API-001	/health returns ok	REQ-API	Pass
API-002	/health/db returns connected	REQ-DB	Pass
API-003	/clients returns list	FR-06	Pass
API-004	/providers returns list	FR-01	Pass
API-005	/services returns list	FR-01	Pass
API-006	/usages returns list	FR-01	Pass
API-007	/budgets returns list	FR-07	Pass
API-008	/invoices returns list	FR-08	Pass

Performance Tests — 4 Tests:

Test ID	Description	Threshold	Result	Status
TC-P-001	API/health Response time	< 500ms	89ms	Pass
TC-P-002	API/usages response time	< 500ms	245ms	Pass
TC-P-003	Dashboard initial load	< 3s	1.2s	Pass
TC-P-004	Chart rendering time	< 2s	0.8s	Pass

Stress Tests — 2 Tests:

Test ID	Description	Condition	Result	Status
TC-S-001	API under moderate load	50 concurrent requests	No errors, avg 890ms	Pass
TC-S-002	API under peak load	100 concurrent requests	2% error rate, graceful degradation	Pass

TEAM APPROVAL

By submitting this document, all team members confirm agreement with this Test Plan.

Team Member	Role	Date
Ishan	Frontend Lead	10 FEB 2026
Michael	Project Manager	10 FEB 2026
Bryana	Test/ QA Lead	10 FEB 2026
Sean	Backend Lead	10 FEB 2026
Tony	Database Developer	10 FEB 2026