Phase II Report

Cloud Cost Intelligence Platform

Michael, Ishan, Sean, Bryana, Tony

CMSC 495 Computer Science Capstone

Instructor: Lynda Metallo

February 17, 2026

Table of Contents

1. Executive Summary

Phase II of the Cloud Cost Intelligence Platform is shifting from a prototype to a full application. It builds on Phase I's database and API foundation. This phase combines features from all 8 functional requirements in our project plan. It also enables logging and fixes defects, labeled DEF-001 to DEF-021. Additionally, the automated test suite grows from 58 to 143 tests, reaching a 100% pass rate to prevent regression during updates.

Key achievements in Phase II include a cost analysis engine. This engine offers rule-based waste detection and optimization recommendations (FR-04, FR-05). The team revamped the CSV/PDF export system. It now generates invoices and an analyst-grade workbook with raw data columns (FR-06). The team set adjustable budget thresholds using a PATCH API endpoint (FR-07). They also provided interactive resource metrics with detailed use breakdowns (FR-08). The frontend response transformer layer fixed a critical DECIMAL-to-string type mismatch. This fixed NaN errors in dashboard calculations. Costs now display with accuracy.

The platform has complete integration. Browser requests go through an nginx reverse proxy to the Flask backend. This backend queries Azure SQL Server and returns data via a standardized REST API. Docker containerization ensures teams in PST, EST, and CET have consistent deployment. Development has changed from direct commits to a branching and PR workflow. Now, six pull requests need code review before merging. The project repository has 7,533 lines of source code in 32 files. It includes 143 automated tests that use pytest, Jest, and Playwright frameworks.

2. Project Setup

2.1 Development Environment

The development environment remains consistent with Phase I, with refinements to the Docker configuration and deployment workflow

Component	Technology	Version
Frontend	HTML/CSS/JavaScript + Chart.js	Chart.js 4.4.7
Backend	Python / Flask	Python 3.12 / Flask 3.1
Database	Microsoft SQL Server (Azure SQL)	Azure SQL Serverless Gen5
Containerization	Docker + nginx reverse proxy	Docker 29.2.0
Version Control	Git / GitHub	N/A
Testing	Pytest, Jest, Playwright	Pytest 8.x, Jest 29.7

2.2 Phase II Architecture

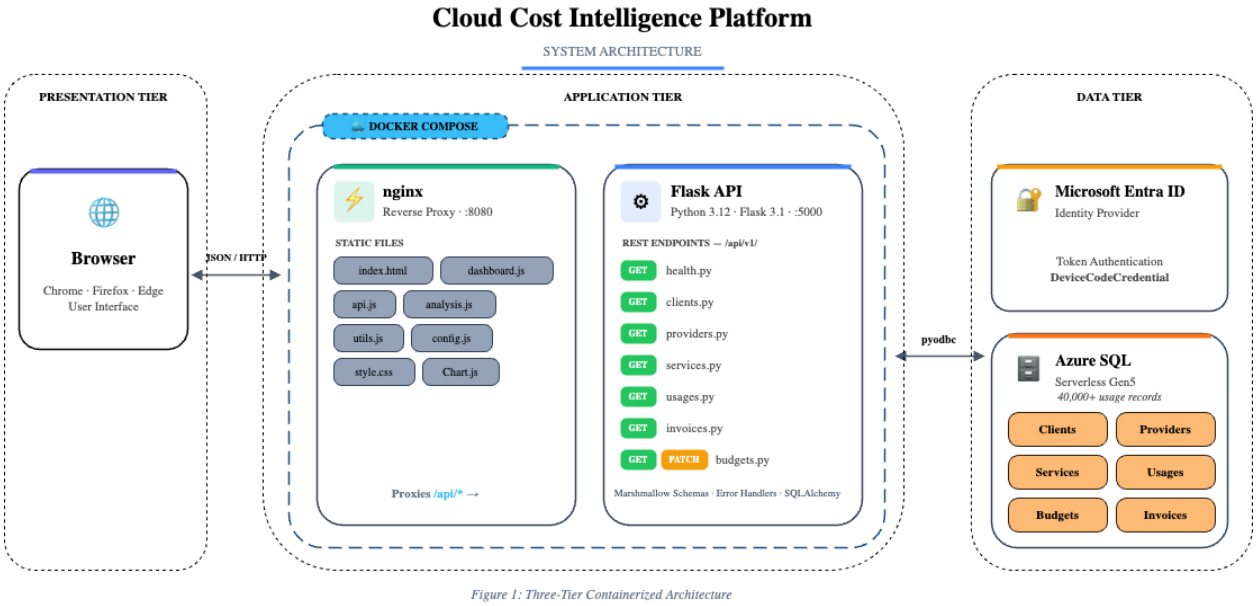

Figure 1: Three-Tier Containerized Architecture

Figure 1: System Architecture (Phase II) from Draw.io

2.3 Repository Structure

```
CMSC495-CloudCost/
├── src/
│   ├── backend/
│   │   ├── wsgi.py                    # App entry point
│   │   ├── __init__.py                # App factory, CORS, DB init
│   │   ├── config.py                  # Environment configuration
│   │   ├── db/engine.py               # Azure SQL connection pooling
│   │   ├── db/session.py              # Session management
│   │   ├── routes/v1/                 # REST endpoints (7 resource files)
│   │   │   ├── budgets.py             # Budget CRUD + PATCH threshold
│   │   │   ├── clients.py             # Client + nested resources
│   │   │   ├── health.py              # Health + DB checks
│   │   │   ├── invoices.py            # Invoice listing
│   │   │   ├── providers.py           # Provider + nested services
│   │   │   ├── services.py            # Service + nested usages
│   │   │   └── usages.py              # Usage records + date filter
│   │   ├── api_http/                  # Responses, schemas, errors
│   │   └── Dockerfile
│   ├── frontend/
│   │   ├── index.html                 # Dashboard UI (9 sections)
│   │   ├── js/config.js               # API URLs, provider mappings
│   │   ├── js/api.js                  # Data access + transformer
│   │   ├── js/analysis.js             # Cost analysis engine
│   │   ├── js/dashboard.js            # App logic, state, rendering
│   │   ├── js/utils.js                # Pure utility functions
│   │   ├── css/style.css              # Centralized styling
│   │   ├── __tests__/                 # Jest suites (94 tests)
│   │   └── Dockerfile
│   └── database/
│       ├── schema.md                  # Table definitions
│       ├── seed_usages.sql            # 40K+ usage records
│       ├── generate_invoices.sql      # Invoice generation
│       └── ERD.png                    # Entity relationship diagram
├── tests/                             # 49 pytest test files
├── docs/                              # Project documentation
├── docker-compose.yml
└── README.md
```

2.4 Key Phase II Environment Changes

Phase II improved the development environment in three key areas.

We first set up Docker networking. This allows us to serve frontend static files and proxy /api/* requests through nginx. All of this runs on a single origin at port 8080. This change fixed CORS issues without needing backend modifications.

Second, we added a 150-line response transformer layer to api.js. This layer addressed the DECIMAL-to-string type mismatch between Azure SQL, pyodbc, and JavaScript. It fixed NaN errors in all dashboard calculations. This allows for accurate cost displays across the app, with no changes needed on the backend.

The team set up a branching/PR workflow. Now, we need to conduct a code review before merging into main. This led to six merged pull requests during Phase II. We used Git assume-unchanged on local config files. This helps prevent accidental commits of credentials.

3. Core Functionality

3.1 Feature Implementation Status

Feature	FR	Phase I	Phase II	Owner
View Cost Dashboard	FR-01	DB population	AWS/ Azure/ GCP	Frontend
View Spending Trends	FR-02	Mock data only	Full dataset	Frontend
Filter by Provider/Service	FR-03	Started w/out function	Cascading filters	Front/back
View Waste Alerts	FR-04	Started	Rule-based engine	PM/front
View Recommendations	FR-05	Started	Phased roadmap	PM/front
Export Reports	FR-06	Started	20-column workbook	Frontend
Set Budget Thresholds	FR-07	Started	PATCH API	Front/back
View Resource Metrics	FR-08	Not started	Utilization stats	Frontend

3.2 Phase II Dashboard:

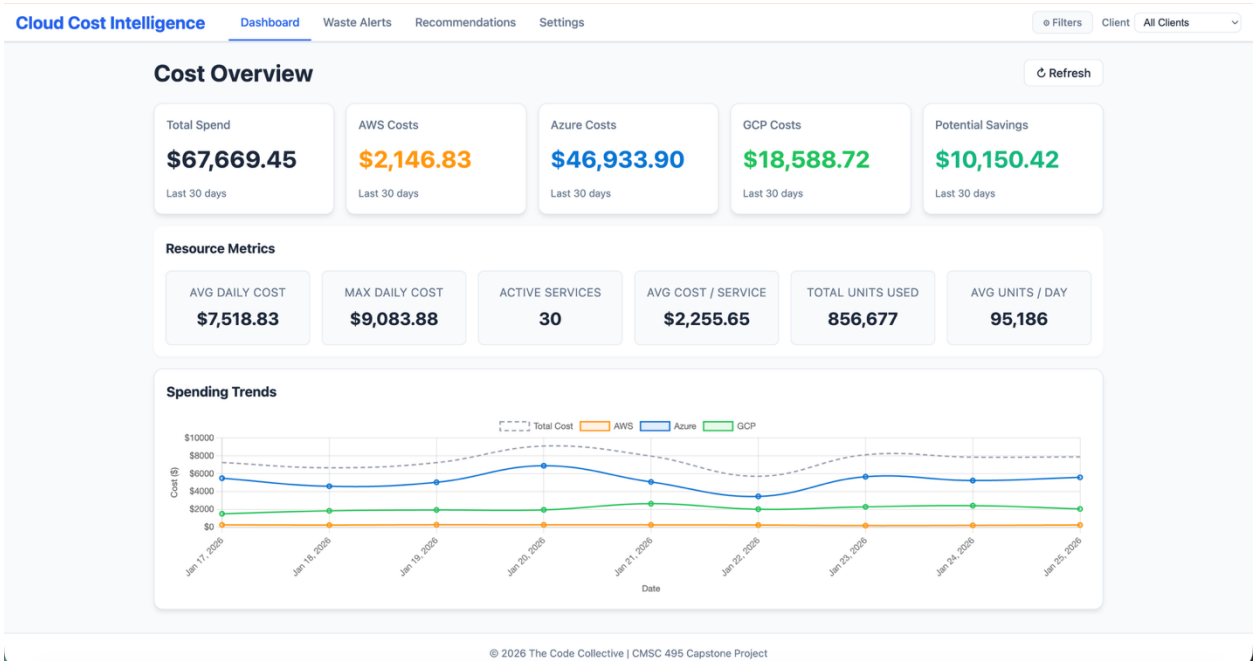

Figure 2: Phase II Dashboard — All 8 Functional Requirements Implemented

3.3 Algorithm and Design Decisions

Waste Detection Engine (analysis.js, 687 lines): Calculates utilization ratios (usage_quantity / service capacity) per record, classifying resources below 20% as waste and 20–50% as rightsizing candidates. Results group by service type with monthly waste cost and severity levels (Critical, High, Medium, Low) based on dollar impact. This rule-based approach produces actionable results from existing usage data without external ML dependencies.

Optimization Recommendations: A template map keyed by service_type and action (e.g., 'Object Storage' + 'lifecycle') provides pre-written guidance with estimated savings rates, implementation effort (hours, cost, downtime), and phased rollout plans. Dollar amounts are computed at render time by multiplying current cost by the template's savings rate.

Data Aggregation and Filtering: The frontend fetches the full dataset (50,000 usage records) and performs client-side aggregation by date and provider for trend charts. A global filter drawer in the navbar persists cascading provider/service/client filters across all views, with budget UI elements hidden when "All Clients" is selected. Both CSV and PDF exports respect active filters and date range.

3.4 Defect Resolution

Phase II tracked 21 defects through GitHub Issues (DEF-001 through DEF-021). Twenty were resolved; one remains open as a performance enhancement (DEF-015: initial load optimization). Key fixes included the DECIMAL-to-string type mismatch causing NaN errors, resolved via a 150-line response transformer in api.js; provider ID misalignment between frontend constants and database values 2001/2002/2003; and export filtering to sync CSV/PDF output with active dashboard filters. The complete defect log with root cause analysis is maintained in GitHub Issues.

3.5 Code Modularity

Backend (16 source files, 1,028 lines): Single responsibility per route file. Shared error handlers in api_http/. SQLAlchemy engine and session management isolated in db/. Marshmallow schemas for input validation.

Frontend (5 source files, 3,487 lines): API calls isolated in api.js (590 lines), analysis engine in analysis.js (687 lines), DOM manipulation in dashboard.js (1,853 lines), pure functions in utils.js (267 lines), configuration in config.js (90 lines). No cross-layer dependencies.

Database (6 files): Normalized schema (6 tables) with foreign key relationships. No business logic in database layer.

4. Documentation

4.1 Code Comments

All source files include module-level docstrings (backend) or file-header comments (frontend) with author, description, and creation date. Inline comments explain non-obvious logic, SQL queries, and authentication flows.

Component	Files	Approach
Frontend (JS)	5	File headers + function docs + inline comments for complex logic
Backend (Python)	16	Module docstrings + inline for SQL and auth logic
Database (SQL)	3	schema.md documentation + seed script comments
Tests (pytest)	49	Docstrings describing requirement and expected behavior
Tests (Jest)	2	Describe/it blocks with requirement mapping

4.2 README

The project README.md includes project description, technology stack, installation instructions, environment variables, Docker commands, test execution instructions, API reference, and team roles.

4.3 External Documentation

- /docs/01_Project_Plan.pdf — Timeline, milestones, and risk assessment
- /docs/02_Project_Design.pdf — Phase I architecture and requirements
- /docs/03_Phase_I_Report.pdf — Phase I source code project report
- /docs/04_Test_Plan_Report.pdf — Test plan report with 148 test cases
- /docs/Testing_and_QA_Plan.md — Test strategy and acceptance criteria (IEEE 829 aligned)
- /src/database/schema.md — Table definitions with column types and relationships
- /src/database/ERD.png — Entity relationship diagram
- Meeting notes: 5 weekly sync summaries (January 18 through February 14)
- /tests/screenshots/ — Visual regression reference images
- /tests/MANUAL_TESTING.pdf — Manual testing log with 50 test cases across sessions

4.4 Defect Tracking

All defects tracked through GitHub Issues (DEF-001 through DEF-021) with standardized fields: description, root cause, severity, assigned owner, affected functional requirement, and resolution. Issues linked to pull requests for traceability.

5. Unit Testing

5.1 Testing Framework

Backend tests use pytest with live Azure SQL connections. Frontend tests use Jest 29.7.0 with jsdom for DOM simulation. End-to-end tests use Playwright for browser automation. All tests execute via command line with no manual intervention required.

5.2 Test Summary

Test Category	Framework	Count	Pass	Fail
Backend Unit Tests	pytest	8	8	0
Frontend Unit Tests	Jest	94	94	0
Integration Tests	pytest	8	8	0
E2E / FR Tests	pytest	18	18	0
Functional Tests	pytest	6	6	0
API Resource Tests	pytest	9	9	0
Manual Tests	Browser/DevTools	50	50	0
TOTAL		**193**	**193**	**0**

5.3 Test Coverage by Requirement

End-to-end tests validate each functional requirement against live API data: FR-01 (dashboard total cost > 0), FR-02 (trend chart usage dates present), FR-03 (provider/service filter reduces dataset), FR-04 (waste alerts flag low utilization), FR-05 (recommendations display rightsizing options), FR-06 (CSV/PDF exports match filtered data), FR-07 (budget PATCH returns updated threshold), and FR-08 (resource metrics calculate avg/max/min). Integration tests verify API response structure against database state. Frontend unit tests cover the response transformer, utility functions, and cost calculations with 94 test cases across 2 Jest suites executing in 0.543 seconds.

5.4 Phase I vs Phase II Test Growth

Metric	Phase I	Phase II	Growth
Total Test Cases	58	143	+85 (+147%)
Backend Tests	8	49	+41 (+513%)
Frontend Tests	50	94	+44 (+88%)
Integration/E2E	0	26	+26 (new)
Manual Test Cases	0	50	+50 (new)
Defects Tracked	0	21	+21 (new)
Defects Resolved	0	20	20/21 (95%)
Pass Rate	**100%**	**100%**	**Maintained**

6. Integration Status

6.1 System Architecture

Three-tier containerized architecture: Browser connects to nginx reverse proxy (port 8080) which serves static frontend files and proxies /api/* requests to Flask backend. Backend authenticates to Azure SQL via Microsoft Entra ID device code flow. Docker Compose orchestrates both containers.

6.2 Integration Milestones

Milestone	Target	Actual	Status
Database schema finalized	January 24	January 24	Complete
Backend API endpoints functional	January 31	January 31	Complete
Frontend connected to backend	February 7	February 7	Complete
CORS resolved via nginx proxy	February 7	February 7	Complete
Critical defects resolved	February 14	February 15	Complete (20/21)
End-to-end feature integration	February 14	February 15	Complete (8/8 FRs)
Phase II test suite passing	February 16	February 15	143/143 passing

6.3 Pull Request Workflow

Phase II adopted a branching/PR workflow with required code review. All merges to main require at least one reviewer approval.

PR	Title	Branch	Merged
#7	Fix DEF-002/004: NaN, provider IDs, export date	fix/def-004-export-date-sync	February 8
#10	API Structure Refactor	API-Structure-Refactor	February 9
#20	Compact UI for demo	feature/dashboard-layout	February 14
#22	Fix DEF-014: Provider filter IDs	fix/def-014-provider-filter-ids	February 14
#30	Global Filters, FR-08 Metrics, Chart Polish	feature/navbar-filter-drawer	February 14
#31	FR-04/05/06/07: Waste, Recs, Export, Budget	feature/export-reports	February 15

7. Team Contributions

7.1 Role-Based Contributions

Role	Member	Phase II Contributions	Key Deliverables
PM / Tech Lead	Michael Allen	Defect triage (DEF-001–021), testing, analysis engine, JSDoc pass, PR coordination	Test Plan, analysis.js (687 lines), dashboard.js (+1000 lines), 20 defect fixes
Frontend Lead	Ishan Akhouri	Dashboard UI, Chart.js, response transformer, filters, exports, navbar drawer	api.js transformer, dashboard.js core, PDF/CSV export, filter system
Backend Lead	Sean Kellner	API endpoints, schema validation, error handling, PATCH endpoint, nested routes	7 route files, Marshmallow schemas, API refactor, budget PATCH
Database Dev	Tony Arista	Schema design, seed data, Azure SQL config, ERD, backup/restore	40K+ seed records, invoice generation, ERD, identity properties
Testing/QA	Bryana Henderson	Test case design, QA documentation, test specifications	Testing_and_QA_Plan.md, test case specifications

7.2 Git Activity Summary

As of February 16 Total: 70 commits across 10 branches. Phase II (since February 1): 53 commits. All merges to main require PR review with at least one approval.

Team Member	Total Commits	Phase II (Feb)	Primary Branches
Michael Allen	45	39	main, feature/export-reports, navbar-filter-drawer, dashboard-layout
Ishan Akhouri	11	9	frontend, fix/def-004
Sean Kellner	10	4	main, API-Structure-Refactor
Tony Arista	4	1	main, MS Azure management
Bryana Henderson	0	0	Documentation via Teams

8. Repository Access

GitHub: https://github.com/michaelallenus217-lang/CMSC495-CloudCost

Public repository. No authentication required for instructor access.

Branches:

- main — Production-ready, stable Phase II deliverables
- frontend — Frontend development (dashboard, response transformer)
- API-Structure-Refactor — Backend API restructuring
- feature/dashboard-layout — CSS layout improvements
- feature/navbar-filter-drawer — Global filter drawer, FR-08 metrics
- feature/export-reports — FR-04/05/06/07 implementation
- fix/def-004-export-date-sync — Export date sync fix
- fix/def-014-provider-filter-ids — Provider ID corrections
- experiment/summary-endpoint — Backend summary endpoint prototype