Phase I Report

Cloud Cost Intelligence Platform

Michael, Ishan, Sean, Bryana, Tony

CMSC 495 Computer Science Capstone

Instructor: Lynda Metallo

February 3, 2026

Table of Contents

1. Executive Summary

Phase I development is ahead of schedule. Backend API is operational with 7 endpoints serving live data from Azure SQL (40,000 usage records). The frontend dashboard is complete with all 9 GUI components. Integration testing successful via Docker with nginx reverse proxy. All 9 unit tests passing. No critical blockers.

Phase I development is ahead of schedule... No critical blockers. **Source code repository: https://github.com/michaelallenus217-lang/CMSC495-CloudCost**

2. Project Setup

Cloud Cost Intelligence Platform – System Architecture

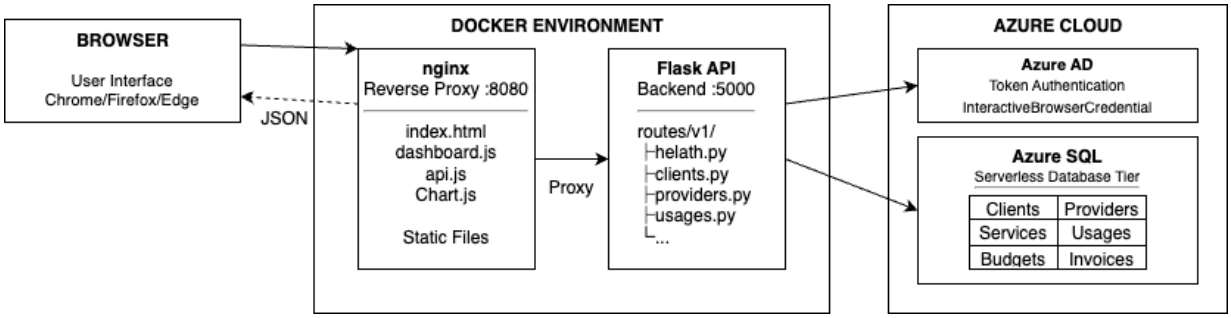

Figure 1: System Architecture

2.1 Development Environment

Component	Details
IDE/Editor	VS Code
Python Version	Python 3.11
Node.js Version	N/A (No framework used)
Database	Azure SQL Server (CMSC495-cloud-cost.database.windows.net)
Version Control	Git, GitHub
OS/Platform	macOS, Windows (varies by team member)
Containerization	Docker with nginx reverse proxy

2.2 Libraries and Dependencies

Backend Dependencies:

Library	Version	Purpose
Flask	3.x	Web framework
pyodbc	5.x	SQL Server connectivity
SQLAlchemy	2.x	ORM and database engine
Azure-identity	1.x	Azure AD authentication
Python-dotenv	1.x	Environment variable management
Flask-cors	4.x	Cross-origin request handling

Frontend Dependencies:

Library	Version	Purpose
Chart.js	4.x	Data visualization
Standard JS	1.x	Environment variable management

2.3 Project Structure

Tree Screenshot

Figure 2: Project Directory Structure

2.4 Version Control Integration

Team uses GitHub with main branch for stable code. Direct commits to main for documentation; feature work coordinated via team chat before pushing. Commit messages follow format: "Add___ - brief description". All team members have push access to the repository.

Repository URL: https://github.com/michaelallenus217-lang/CMSC495-CloudCost

3. Core Functionality

3.1 Features Implemented

Feature	Description	Status
Database Schema	6 tables deployed to Azure SQL (Clients, Providers, Services, Usages, Budgets, Invoices)	Complete
Test Data	40,000 usage records, 10 clients, 3 providers, 10 services	Complete
Backend API	Flask endpoints for all tables with GET operations	Complete
Azure AD Auth	Token-based authentication for database connection	Complete
Frontend Dashboard	Cost overview, filters, spending trends chart	Complete
Mock Data Fallback	Frontend gracefully falls back to mock data is API unavailable	Complete
Docker Integration	Frontend and backend containerized with nginx reverse proxy	Complete

Database Schema (ERD):

Entity tables and their relationships:

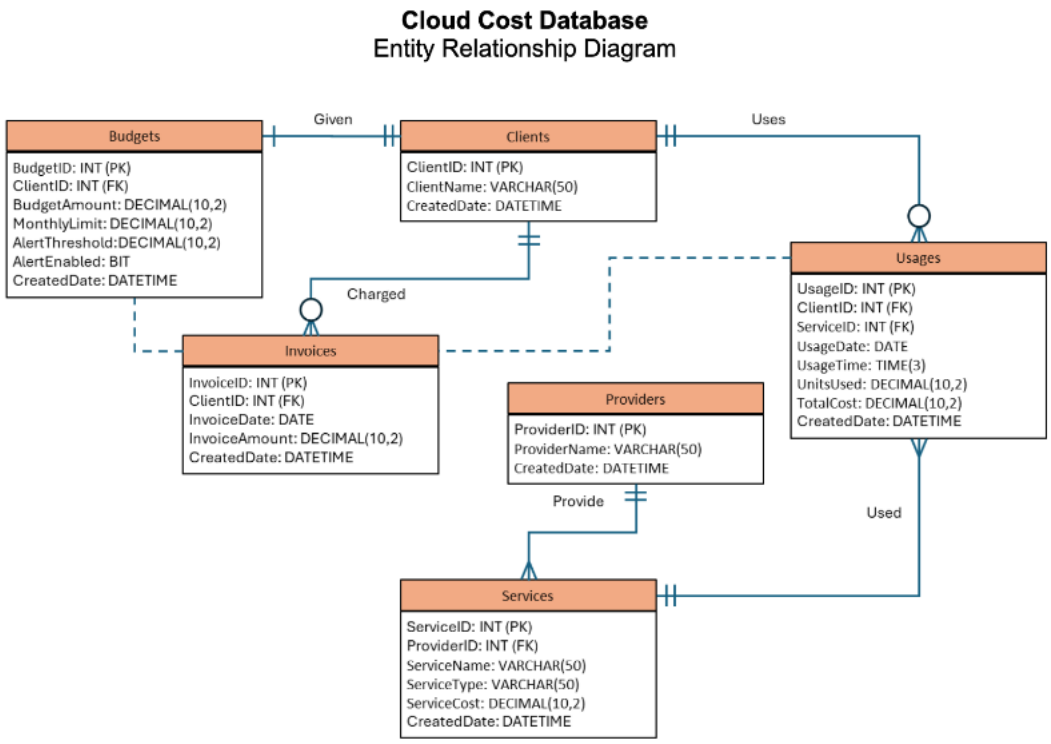

Figure 3: Entity Relationship Diagram

Frontend Dashboard:

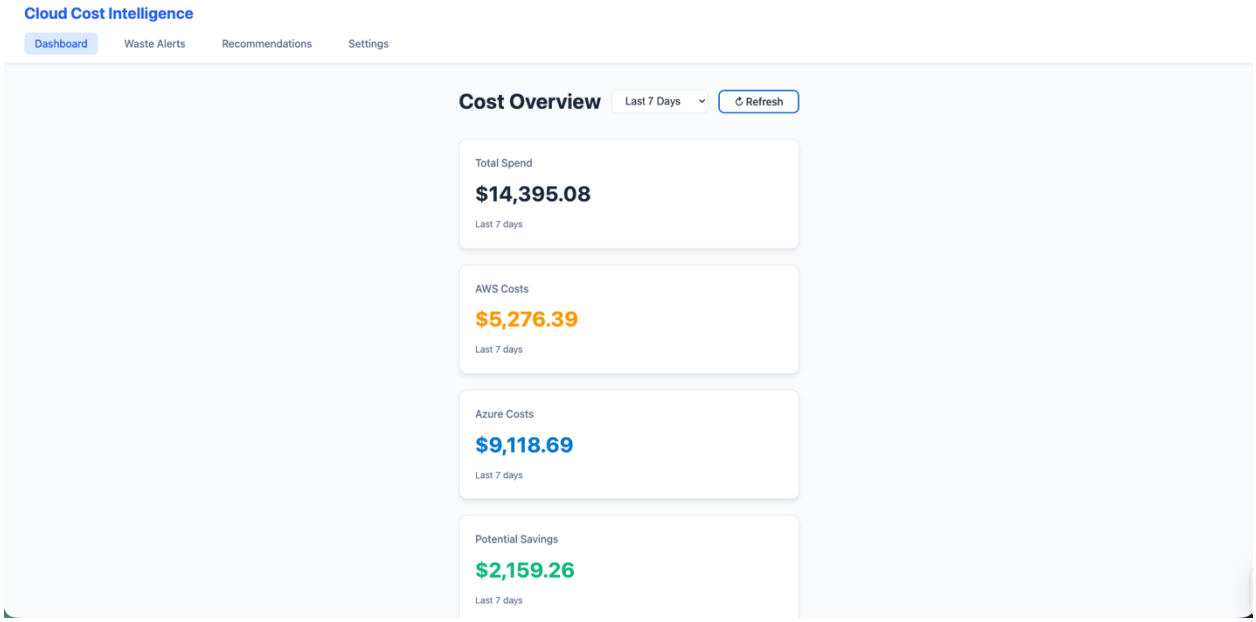

Figure 4: Frontend Dashboard

3.2 Requirements Traceability

Requirement	Phase I Implementation
Display cloud costs by provider	/api/v1/providers, /api/v1/usages endpoints
Support AWS and Azure	3 providers in database (AWS, Azure, GCP)
Show spending trends	Chart.js trend visualization in dashboard
Filter by client/service	Filter dropdowns in frontend UI

3.3 Algorithm and Design Decisions

Cost data is stored at the transaction level with UsageDate, UsageTime, and TotalCost fields. This allows flexible aggregation (daily, weekly, monthly) at query time rather than pre-aggregating. The backend uses raw SQL queries via SQLAlchemy for simplicity and direct control over Azure SQL performance.

3.4 Modular Code Organization

Backend follows a route-based structure. Each table has its own route file (clients.py, providers.py, etc.) under `routes/v1/`. Database connection logic is isolated in `db/engine.py` and `db/session.py`. Frontend separates concerns across api.js (API calls), dashboard.js (UI logic), config.js (settings), and utils.js (helpers).

3.5 Code Samples

3 key code snippets demonstrating core functionality.

Database connection (Azure AD auth):

```
(.venv) m4llen@Michaels-MacBook-Air src % head -40 ~/Documents/CMSC495-CloudCost/src/backend/db/engine.py
import os
import struct
from azure.identity import InteractiveBrowserCredential, DefaultAzureCredential
from sqlalchemy import create_engine
from sqlalchemy.engine import Engine
from sqlalchemy.pool import QueuePool

SQL_COPT_SS_ACCESS_TOKEN = 1256  # ODBC access token attribute


def _get_credential():
    env = os.getenv("ENV", "local").lower()

    # Local dev: interactive browser auth
    if env == "local":
        # Optional: set a tenant_id if your org has multiple tenants
        return InteractiveBrowserCredential()

    # Non-local: use the standard chain (Managed Identity, workload identity, etc.)
    return DefaultAzureCredential()


def _get_access_token() -> str:
    credential = _get_credential()
    token = credential.get_token("https://database.windows.net/.default")
    return token.token


def _odbc_access_token(token: str) -> bytes:
    token_bytes = token.encode("utf-16-le")
    return struct.pack(f"<I{len(token_bytes)}s", len(token_bytes), token_bytes)


def build_engine(server: str, database: str, pool_size: int, max_overflow: int, pool_recycle: int) -> Engine:
    if not server or not database:
        raise ValueError("AZURE_SQL_SERVER and AZURE_SQL_DATABASE must be set")

    driver = "ODBC Driver 18 for SQL Server"
    odbc_connect = (
        f"Driver={{{driver}}};"
```

Figure 5: Database Connection (Azure AD Auth)

API endpoint returning data:

```
(.venv) m4llen@Michaels-MacBook-Air src % head -40 ~/Documents/CMSC495-CloudCost/src/backend/routes/v1/clients.py
from flask import jsonify, request
from sqlalchemy import text
from backend.db.session import get_db_session
from backend.routes.v1 import api_v1_bp

@api_v1_bp.get("/clients")
def get_clients():
    limit_param = request.args.get("limit", "10")

    try:
        limit = int(limit_param)
    except ValueError:
        return jsonify({"error": "limit must be an integer"}), 400

    if limit < 1:
        return jsonify({"error": "limit must be >= 1"}), 400

    if limit > 100:
        # Don't allow limits greter than 100
        limit = 100


    db = get_db_session()

    rows = db.execute(
        text(
            """
            SELECT TOP (:limit) ClientID, ClientName, CreatedDate
            FROM Clients
            ORDER BY CreatedDate DESC
            """
        ),
        {"limit": limit},
    ).fetchall()

    clients = []
    for client_id, client_name, created_date in rows:
        clients.append(
            {
                "client_id": client_id,
```

Figure 6: API Endpoint Implementation

Frontend API configuration:

```
(.venv) m4llen@Michaels-MacBook-Air src % head -40 ~/Documents/CMSC495-CloudCost/src/frontend/js/config.js
// API Configuration (Global namespace)
const API_CONFIG = {
    BASE_URL: 'http://localhost:5000',
    BASE_PATH: '/api/v1',
    TIMEOUT: 10000, // 10 seconds
    USE_MOCK_DATA: true, // Set to false when backend is ready
};

// API Endpoints
const ENDPOINTS = {
    HEALTH: '/health',
    HEALTH_DB: '/health/db',
    CLIENTS: '/clients',
    PROVIDERS: '/providers',
    SERVICES: '/services',
    USAGES: '/usages',
    BUDGETS: '/budgets',
    INVOICES: '/invoices',
};

// Build full URL for an endpoint
function getApiUrl(endpoint, params = {}) {
    const url = new URL(API_CONFIG.BASE_URL + API_CONFIG.BASE_PATH + endpoint);

    // Add query parameters
    Object.keys(params).forEach(key => {
        if (params[key] !== null && params[key] !== undefined && params[key] !== '') {
            url.searchParams.append(key, params[key]);
        }
    });

    return url.toString();
}

// Provider IDs (from backend)
const PROVIDER_IDS = {
    AWS: 1,
    AZURE: 2,
};

(.venv) m4llen@Michaels-MacBook-Air src %
```

Figure 7: Frontend API Configuration

4. Documentation

4.1 Code Formatting Standards

Python code follows PEP 8 guidelines. JavaScript uses consistent 4-space indentation. All files use UTF-8 encoding. Function and variable names use snake_case in Python and camelCase in JavaScript.

4.2 Commenting Approach

Backend functions include inline comments explaining key logic. Frontend files include header comments describing file purpose. Complex operations like Azure AD token generation include step-by-step comments.

Sample Commented Code:

```
(.venv) m4llen@Michaels-MacBook-Air CMSC495-CloudCost % head -30 src/frontend/js/dashboard.js
/**
 * File: dashboard.js
 * Project: Cloud Cost Intelligence Platform
 * Author: Ishan (Frontend Lead)
 * Created: January 2026
 * Description: Main dashboard logic. Initializes charts, handles user
 *              interactions, and renders cost data visualizations.
 */

// Main Dashboard Logic & Chart.js Integration (uses global functions from other files)

// Global state
let currentPage = 'dashboard';
let dashboardChart = null;
let currentDateRange = 30;
let unfilteredDashboardData = null; // Cache for filtering

// Global state for waste alerts sorting
let wasteAlertsData = [];
let currentSortColumn = 'potential_savings'; // Default sort by savings (high to low)
let currentSortDirection = 'desc';

// Initialize app when DOM is loaded
document.addEventListener('DOMContentLoaded', initApp);

async function initApp() {
    console.log('Cloud Cost Intelligence Platform - Initializing...');
    console.log('API Mode:', API_CONFIG.USE_MOCK_DATA ? 'MOCK DATA' : 'LIVE API');

    // Setup navigation
```

Figure 8: Sample Commented Code

4.3 README Contents

The repository README includes:

- Project description (Cloud Cost Intelligence Platform)
- Team members and roles
- Technology stack (Python/Flask, Azure SQL, HTML/CSS/JS, Chart.js)
- Setup instructions for local development
- API endpoint documentation

4.4 Dependencies Documentation

Python dependencies documented in `requirements.txt` with pinned versions generated via `pip freeze`. Frontend uses Chart.js loaded via CDN — no package.json required. Database schema documented in `src/database/schema.md` with all tables and column definitions.

5. Unit Testing

5.1 Testing Framework

Backend uses pytest for Python API testing. Tests call API endpoints and verify response codes and data structure. The requests library is used to make HTTPS calls to the Flask server. A warmup fixture handles Azure SQL serverless tier cold start behavior.

5.2 Test Coverage Summary

Module	Tests	Passing	Coverage
Health Endpoints	2	2	100%
Clients API	2	2	100%
Providers API	1	1	100%
Services API	1	1	100%
Usages API	1	1	100%
Budgets API	1	1	100%
Invoices API	1	1	100%
Total	9	9	100%

5.3 Test Cases

ID	Endpoint	Expectation	Actual	Pass
TC-01	GET/api/v1/health	200, status=”ok”	200, status=”ok”	Yes
TC-02	GET/api/v1/health/db	200, db=”connected’	200, db=”connected’	Yes
TC-03	GET/api/v1/clients	200, returns list	200, returns list	Yes
TC-04	GET/api/v1/clients/1001	200, single client	200, single client	Yes
TC-05	GET/api/v1/providers	200, returns list	200, returns list	Yes
TC-06	GET/api/v1/services	200, returns list	200, returns list	Yes
TC-07	GET/api/v1/usages	200, returns list	200, returns list	Yes
TC-08	GET/api/v1/budgets	200, returns list	200, returns list	Yes
TC-09	GET/api/v1/invoices	200, returns list	200, returns list	Yes

5.4 Test Execution Output

```
(.venv) m4llen@Michaels-MacBook-Air CMSC495-CloudCost % TEST_API_URL=http://localhost:5001/api/v1 pytest tests/ -v
============================= test session starts ==============================
platform darwin -- Python 3.12.3, pytest-9.0.2, pluggy-1.6.0 -- /Users/m4llen/Documents/CMSC495-CloudCost/.venv/bin/python
cachedir: .pytest_cache
rootdir: /Users/m4llen/Documents/CMSC495-CloudCost
collected 9 items

tests/test_budgets.py::test_budgets_returns_list PASSED                  [ 11%]
tests/test_clients.py::test_clients_returns_list PASSED                  [ 22%]
tests/test_health.py::test_health_returns_ok PASSED                      [ 33%]
tests/test_health_db.py::test_health_db_connected PASSED                 [ 44%]
tests/test_invoices.py::test_invoices_returns_list PASSED                [ 55%]
tests/test_providers.py::test_providers_returns_list PASSED              [ 66%]
tests/test_services.py::test_services_returns_list PASSED                [ 77%]
tests/test_single_client.py::test_single_client PASSED                   [ 88%]
tests/test_usages.py::test_usages_returns_list PASSED                    [100%]

============================== 9 passed in 4.88s ===============================
(.venv) m4llen@Michaels-MacBook-Air CMSC495-CloudCost %
```

Figure 9: Test Execution Output

6. Collaboration and Version Control

6.1 Team Contributions

Member	Role	Phase I Contributions
Michael	Project Manager	Environment setup, integration testing, documentation, repo organization, meeting coordination
Ishan	Frontend Lead	Dashboard UI, 9 GUI components, Chart.js integration, mock data fallback
Sean	Backend Lead	Flask API, 7 route modules, Azure AD auth, HTTPS configuration
Tony	Database Lead	Azure SQL setup, 6 tables, 40K usage records, seed data scripts
Bryana	Testing/QA Lead	Test plan, unit test development

6.2 Code Review Process

Team coordinates via Microsoft Teams chat before pushing to main. Members announce commits and review each other's code informally. Pull requests used for larger features; direct commits for documentation and minor fixes.

6.3 Communication

- **Saturday syncs:** 3:30 PM EST weekly video call
- **Async check-ins:** Monday/Thursday via Teams chat
- **Daily standups:** Posted in Teams channel
- **GitHub:** Code repository and version history

7. Progress Assessment

7.1 Schedule Status

Milestone	Planned	Status
Database schema deployed	Week 4	Complete
Test data populated	Week 4	Complete
Backend API skeleton	Week 4	Complete
Frontend dashboard	Week 4	Complete
Unit tests passing	Week 4	Complete
Frontend-backend integration	Week 4	Complete

7.2 Risks and Mitigation

Risk	Impact	Mitigation
CORS blocking frontend-backend connection	Medium	Resolved: Docker with nginx reverse proxy serves frontend and proxies API requests from same origin
Azure SQL cold start timeouts	Low	Warmup fixture in test suite handles serverless tier wake-up
Time zone coordination (EST/CET)	Low	Saturday syncs + async chat; working well
Scope creep	Low	Team aligned on rubric-first decisions

7.3 Next Steps

1. Complete Test Plan document covering six test types: unit, integration, functionality, performance, stress, and acceptance testing
2. Define test environment configuration and data preparation procedures for each test category
3. Implement Waste Alerts feature to identify spending anomalies exceeding baseline thresholds
4. Implement Recommendations engine providing cost optimization suggestions based on usage patterns
5. Add budget threshold notifications alerting users when spending approaches defined limits
6. Execute integration tests validating frontend-backend-database connectivity under Docker environment
7. Expand unit test coverage for new Phase II features before source code submission

TEAM APPROVAL

By submitting this document, all team members confirm agreement with this Phase I Report.

Team Member	Role	Date
Ishan	Frontend Lead	2/3/2026
Michael	Project Manager	2/3/2026
Bryana	Test/ QA Lead	2/3/2026
Sean	Backend Lead	2/3/2026
Tony	Database Developer	2/2/2026