

Revenium React Developer Take-Home Project

Project: Real-Time AI FinOps Dashboard

Overview

Build a real-time dashboard for monitoring AI usage and costs across multiple customers and tenants. This system will display streaming metrics, detect anomalies, and provide interactive cost analysis tools—all while maintaining smooth performance with high-frequency data updates.

Time Expectation

This is designed as a weekend project (8-12 hours). We value quality over completeness - focus on demonstrating deep technical understanding rather than building every feature.

Once the project is complete, please share the link to its GitHub repository.

Business Context

Our platform provides real-time visibility into AI and API costs. Finance teams, engineering leads, and product managers use our dashboards to track token usage, detect cost anomalies, and understand spending patterns across different models, customers, and time periods. The dashboard must handle hundreds of metric updates per second while remaining responsive and easy to use.

Technical Requirements

Core Components

1. Real-Time Metrics Dashboard

Build a **responsive dashboard** that displays:

Primary Metrics Panel:

TypeScript

```
interface MetricUpdate {  
  timestamp: string;  
  tenantId: string;  
  customerId: string;  
  metrics: {  
    totalCalls: number;  
    totalTokens: number;  
    totalCost: number;  
    avgLatencyMs: number;  
  }  
}
```

Required Visualizations:

- **Live cost gauge** - Current spend rate (\$/hour) with animated updates
- **Token usage time series** - Last 5 minutes of token consumption with 5-second resolution
- **Cost by model chart** - Breakdown showing GPT-4, GPT-3.5, Claude, etc.
- **Top customers table** - Real-time sorted list of highest spenders
- **Anomaly alerts panel** - Shows detected cost spikes or unusual patterns

2. Data Connection & Updates

Implement a **polling-based data fetch** system that:

- Polls a REST endpoint at configurable intervals (default: every 2 seconds)
- Handles API failures gracefully with exponential backoff retry logic
- Manages polling state (pause/resume updates)
- Implements client-side buffering to smooth bursty data updates
- Provides connection status indicators (connected, error, retrying)
- Allows users to adjust polling frequency (1s, 2s, 5s, 10s)

Mock Server: You'll create a simple Node/Express or Python server that provides a REST endpoint returning mock metric updates.

Required Endpoints:

TypeScript

```
// GET /api/metrics?since=<timestamp>

// Returns metrics since the last poll

{
  metrics: MetricUpdate[];

  nextPollAfter: number; // milliseconds
}

// GET /api/metrics/history?from=<timestamp>&to=<timestamp>

// Returns historical data for time range selection
```

Bonus/Extra Credit: Real-Time Streaming Implementation

For additional consideration, implement **Server-Sent Events (SSE)** or **WebSocket** as an alternative to polling:

- Automatic fallback to polling if streaming is unavailable
- Connection status indicators specific to streaming
- Proper cleanup on component unmount
- Exponential backoff for reconnection attempts
- Handle browser tab visibility (pause when hidden, resume when visible)

This demonstrates advanced real-time system architecture but is not required for the core assignment.

3. Anomaly Detection Visualization

Implement a **visual anomaly detection** system:

- Show trend lines
- Display contextual information on hover (what caused the spike?)
- Allow users to dismiss or acknowledge anomalies

- Persist acknowledged anomalies (localStorage is fine for this exercise)

4. Time Window Controls

Implement **flexible time window** selection:

- Preset options: Last 5 min, 15 min, 1 hour, 24 hours
 - Custom date range picker
 - Real-time mode vs. historical mode toggle
 - Playback controls for historical data (play/pause/speed)
-

Technology Stack

Required Technologies

- **React 18+** with hooks
- **TypeScript** (strict mode)
- **Vite** as build tool
- **Your choice of:**
 - Charting library (Recharts, Highcharts, Chart.js, D3, or similar)
 - State management (Context + useReducer, Zustand, Jotai, Redux Toolkit, or similar)
 - UI component library (Radix UI, ShadCN UI, MUI, or similar)

Mock Backend Server

- Simple **Node/Express** server (or use Vite server capabilities) or **Python** server
- Generates realistic streaming data
- Includes a few REST endpoints for initial data load

Docker Requirements

- **Dockerfile** for the React application
- **docker-compose.yml** that includes both frontend and mock backend
- Optimized production build
- Multi-stage build to minimize image size

What We're Evaluating

Required Deliverables

1. Working Code

- React application (TypeScript, strict mode)
- Mock backend server generating realistic data
- Dockerfile for the application
- docker-compose.yml for the complete stack
- Include a README.md with setup instructions
- All features functional and demonstrated

2. Component Architecture Document

- Component hierarchy diagram showing the component tree
- State management architecture explanation
- Data flow explanation (from API to UI)
- Justification for component structure and reusability approach

3. Architecture Document

- System design diagram (include frontend, backend, data flow)
- Real-time data handling strategy (polling or streaming)
- Key design decisions and tradeoffs
- Responsive design approach
- **What you would change** with more time

4. Testing Strategy

- Unit tests for critical utility functions and hooks
- Component tests using React Testing Library
- At least one integration test (user interaction flow)
- Performance test demonstrating smooth operation at 100 updates/second
- Document your testing philosophy

5. Code Organization

- Explanation of your folder structure and rationale
- File naming conventions used
- Module organization strategy (features, layers, etc.)
- How you separate concerns (business logic, UI, utilities, types)