# Technical Design Document: AI Cost & Insights Copilot

## Architecture Overview

### High-Level Architecture

┌─────────────────────────────────────────────────────────────┐  
│ 🖥️ Presentation Layer │  
├─────────────────────────────────────────────────────────────┤  
│ • Streamlit Dashboard (frontend/app.py) │  
│ • Interactive Chat Interface │  
│ • KPI Dashboards & Visualizations │  
│ • System Monitoring & Admin Panel │  
└─────────────────────────────────────────────────────────────┘  
 ↕ HTTP/REST API  
┌─────────────────────────────────────────────────────────────┐  
│ 🚀 Application Layer │  
├─────────────────────────────────────────────────────────────┤  
│ • FastAPI Application (app/main.py) │  
│ • /api/ask - AI-powered Q&A │  
│ • /api/kpi - Cost metrics & KPIs │  
│ • /api/recommendations - Optimization suggestions │  
│ • /api/metrics - System observability │  
└─────────────────────────────────────────────────────────────┘  
 ↕  
┌─────────────────────────────────────────────────────────────┐  
│ 🧠 AI & Processing Layer │  
├─────────────────────────────────────────────────────────────┤  
│ • Enterprise RAG Service (enhanced\_rag\_service.py) │  
│ • Google Gemini AI Integration │  
│ • FAISS Vector Store (semantic search) │  
│ • FinOps Knowledge Base (25+ best practices) │  
│ • Query Classification & Processing │  
└─────────────────────────────────────────────────────────────┘  
 ↕  
┌─────────────────────────────────────────────────────────────┐  
│ 💾 Data & Storage Layer │  
├─────────────────────────────────────────────────────────────┤  
│ • SQLite Database (data/app.db) │  
│ • Billing Table (cost, usage, metadata) │  
│ • Resources Table (tags, ownership) │  
│ • Data Quality Validation │  
└─────────────────────────────────────────────────────────────┘

## Data Model

### Core Database Schema

#### Billing Table

CREATE TABLE billing (  
 id INTEGER PRIMARY KEY AUTOINCREMENT,  
 invoice\_month TEXT NOT NULL, -- YYYY-MM format  
 account\_id TEXT NOT NULL, -- Cloud account identifier  
 subscription TEXT NOT NULL, -- Subscription/project ID  
 service TEXT NOT NULL, -- Cloud service name  
 resource\_group TEXT NOT NULL, -- Resource group/project  
 resource\_id TEXT NOT NULL, -- Unique resource identifier  
 region TEXT NOT NULL, -- Geographic region  
 usage\_qty REAL NOT NULL, -- Usage quantity  
 unit\_cost REAL NOT NULL, -- Cost per unit  
 cost REAL NOT NULL, -- Total cost  
 created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP  
);

#### Resources Table

CREATE TABLE resources (  
 id INTEGER PRIMARY KEY AUTOINCREMENT,  
 resource\_id TEXT UNIQUE NOT NULL, -- Links to billing.resource\_id  
 owner TEXT, -- Resource owner email  
 env TEXT, -- Environment (prod/dev/staging)  
 tags\_json TEXT, -- JSON blob of additional tags  
 created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP  
);

### Data Quality Checks

1. **Null Value Validation**: No null costs, resource IDs, or critical fields
2. **Cost Range Validation**: No negative costs, reasonable upper bounds
3. **Duplicate Detection**: No duplicate resource\_id + invoice\_month combinations
4. **Foreign Key Integrity**: All billing.resource\_id must exist in resources table
5. **Date Format Validation**: invoice\_month must follow YYYY-MM pattern

## AI/RAG Architecture

### Knowledge Retrieval System

User Question → Query Processing → Context Retrieval → Response Generation  
 │ │ │ │  
 ▼ ▼ ▼ ▼  
Input Validation Classification Vector Search AI Generation  
Security Check Intent Detection Database Query Knowledge Fusion  
Normalization Context Building Knowledge Base Quality Assurance

### RAG Components

#### 1. Knowledge Base

* **FinOps Best Practices**: 25+ curated guidelines
* **Cost Optimization Strategies**: Service-specific recommendations
* **Governance Policies**: Tagging standards and compliance
* **Trend Analysis Patterns**: Common cost variation explanations

#### 2. Vector Store (FAISS)

* **Embedding Model**: sentence-transformers/all-MiniLM-L6-v2
* **Index Type**: Flat L2 distance for accuracy
* **Fallback**: Enhanced keyword matching for reliability
* **Retrieval**: Top-k similarity search (k=5)

#### 3. Query Processing Pipeline

async def ask\_question(question: str) -> Dict[str, Any]:  
 # 1. Security validation and sanitization  
 validated\_question = validate\_input(question)  
   
 # 2. Context retrieval from knowledge base  
 sources = retrieve\_context(validated\_question)  
   
 # 3. Data analysis from database  
 data\_context = analyze\_database(validated\_question)  
   
 # 4. AI response generation  
 if gemini\_available:  
 response = generate\_ai\_response(question, sources, data\_context)  
 else:  
 response = generate\_fallback\_response(question, data\_context)  
   
 # 5. Post-processing and enhancement  
 return enhance\_response(response, data\_context)

## System Components

### 1. FastAPI Application (main.py)

**Responsibilities**: - HTTP request handling and routing - Input validation and security - Response formatting and error handling - Metrics collection and observability - Database connection management

**Key Endpoints**: - POST /api/ask - Natural language query processing - GET /api/kpi - KPI metrics with filtering - GET /api/recommendations - Cost optimization suggestions - GET /api/metrics - System performance metrics - GET /health - Health check and status

### 2. Enterprise RAG Service (enhanced\_rag\_service.py)

**Responsibilities**: - Natural language understanding and processing - Database query generation and execution - Knowledge base retrieval and ranking - Response synthesis and recommendation generation - Confidence scoring and quality assessment

**Core Features**: - Multi-dimensional cost analysis - Trend detection and comparison - Optimization opportunity identification - Executive summary generation

### 3. Streamlit Frontend (frontend/app.py)

**Responsibilities**: - User interface and interaction - Data visualization and charting - Real-time system monitoring - Admin panel and configuration

**Key Sections**: - **Dashboard**: KPI overview and trends - **AI Assistant**: Chat interface for queries - **Recommendations**: Cost optimization insights - **System Monitor**: Health and performance metrics

## Technical Trade-offs & Decisions

### Database Choice: SQLite vs PostgreSQL

**Chosen**: SQLite **Rationale**: - Simple deployment and setup - No external dependencies - Sufficient for demo/prototype scale (< 100K records) - Limited concurrent write performance - No advanced analytical functions

**Alternative**: PostgreSQL for production scaling

### AI Service: Gemini vs OpenAI vs Local Models

**Chosen**: Google Gemini with Local Fallbacks **Rationale**: - Cost-effective for prototype - Good performance for business queries - Comprehensive fallback system - Requires API key and internet connection - Potential latency for complex queries

### Vector Store: FAISS vs Chroma vs Pinecone

**Chosen**: FAISS with Keyword Fallback **Rationale**: - Local deployment, no external dependencies - Fast similarity search - Good performance with small knowledge base - Requires sentence-transformers model - No built-in persistence layer

## Security Architecture

### Input Validation Layer

SECURITY\_PATTERNS = [  
 'ignore', 'system:', 'assistant:', 'prompt:',  
 '<script>', 'javascript:', 'DROP TABLE', 'DELETE FROM',  
 'rm -rf', '../', 'passwd', 'sudo'  
]

### Security Headers

* X-Content-Type-Options: nosniff
* X-Frame-Options: DENY
* X-XSS-Protection: 1; mode=block
* Referrer-Policy: strict-origin-when-cross-origin

### Query Sanitization

* SQL injection prevention through parameterized queries
* Input length limits (2000 characters max)
* Pattern-based malicious content detection
* Request rate monitoring and logging

## Observability & Monitoring

### Metrics Collection

METRICS\_STORE = {  
 'api\_requests\_total': 0,  
 'ai\_queries\_total': 0,  
 'response\_time\_sum': 0.0,  
 'errors\_total': 0,  
 'database\_queries': 0,  
 'gemini\_api\_calls': 0,  
 'security\_blocks': 0  
}

### Structured Logging

logger.info(f"[{request\_id}] {endpoint} - {processing\_time:.2f}s - {status}")

### Health Checks

* Database connectivity and record counts
* AI service availability and response times
* System resource usage and performance
* Error rates and anomaly detection

## Deployment Architecture

### Container Strategy

# Multi-stage build for optimization  
FROM python:3.11-slim as builder  
# Dependencies and model downloads  
  
FROM python:3.11-slim as runtime   
# Application code and runtime

### Docker Compose Services

services:  
 api:  
 build: .  
 ports: ["8000:8000"]  
 environment: [GOOGLE\_API\_KEY]  
   
 frontend:  
 build: ./frontend  
 ports: ["8501:8501"]  
 depends\_on: [api]

## Performance Considerations

### Response Time Targets

* **Simple KPI queries**: < 500ms
* **Complex AI analysis**: < 3s
* **Trend comparisons**: < 1s
* **Health checks**: < 100ms

### Optimization Strategies

* Database indexing on frequently queried columns
* Response caching for repeated questions
* Connection pooling for database access
* Lazy loading of AI models and embeddings

### Scalability Limits (Current Architecture)

* **Concurrent Users**: ~50-100
* **Database Records**: ~100K
* **Monthly Query Volume**: ~10K
* **Response Cache Size**: ~1GB memory

## Risk Assessment & Mitigation

### High-Risk Areas

1. **AI Response Accuracy**
   * **Risk**: Incorrect cost analysis leading to bad decisions
   * **Mitigation**: Confidence scoring, source citations, fallback responses
2. **Data Quality Issues**
   * **Risk**: Missing or incorrect billing data
   * **Mitigation**: Comprehensive validation, quality checks, data profiling
3. **Security Vulnerabilities**
   * **Risk**: Prompt injection, data exposure
   * **Mitigation**: Input validation, security headers, audit logging

### Medium-Risk Areas

1. **Performance Degradation**
   * **Risk**: Slow responses as data grows
   * **Mitigation**: Database optimization, caching, monitoring
2. **Service Dependencies**
   * **Risk**: Gemini API outages
   * **Mitigation**: Fallback responses, local processing

## Future Enhancement Roadmap

### Phase 2 (Next 3 months)

* PostgreSQL migration for production scale
* Advanced caching layer (Redis)
* Multi-cloud provider support
* Enhanced security with authentication

### Phase 3 (3-6 months)

* Real-time cost monitoring and alerting
* Advanced forecasting and budgeting
* Integration with cloud provider APIs
* Mobile-responsive interface

### Phase 4 (6+ months)

* Machine learning for anomaly detection
* Advanced optimization recommendations
* Enterprise SSO integration
* Multi-tenant architecture