TeamPulse: System Architecture & AI Flow

Technical Architecture Document v1.0

Document Owner: Engineering Leadership
Date: December 16, 2025
Status: Technical Design

Table of Contents

Architecture Overview

High-Level Architecture

```
┌─────────────────────────────────────────────────────────────────
───────┐
│                  CLIENT LAYER                        │
├─────────────────────────────────────────────────────────────────
───────┤
│ Web App (React) │ Mobile Apps (React Native) │ API Clients │
└────────────────────────┬────────────────────────────────────────
───────┘
       │
       ▼
┌─────────────────────────────────────────────────────────────────
───────┐
```

```
│                 API GATEWAY LAYER                          │
├──────────────────────────────────────────────────────────────┤
│  • Authentication/Authorization (Auth0)                     │
│  • Rate Limiting & Throttling                               │
│  • Request Routing & Load Balancing                         │
│  • API Versioning                                           │
└──────────────────────┬───────────────────────────────────────┘
                       │
                       ▼
┌──────────────────────────────────────────────────────────────┐
│                 APPLICATION LAYER                           │
├──────────────┬──────────────┬──────────────┬─────────────────┤
│  Query       │  Report      │  Alert       │  Integration    │
│  Service     │  Service     │  Service     │  Service        │
└──────┬───────┴──────┬───────┴──────┬───────┴────────┬────────┘
       │              │              │                │
       ▼              ▼              ▼                ▼
┌──────────────────────────────────────────────────────────────┐
│                 AI ORCHESTRATION LAYER                       │
├──────────────────────────────────────────────────────────────┤
│  • Query Understanding (Claude API)                         │
│  • Context Management                                       │
│  • Report Generation (Claude API)                           │
│  • Anomaly Detection (ML Models)                            │
│  • Response Synthesis                                       │
└──────────────────────┬───────────────────────────────────────┘
                       │
                       ▼
┌──────────────────────────────────────────────────────────────┐
│                 DATA LAYER                                   │
├──────────────┬──────────────┬──────────────┬─────────────────┤
│  PostgreSQL  │   Redis      │  Vector DB   │  Object Storage │
│  (Primary)   │  (Cache)     │  (Pinecone)  │  (S3)           │
```

```
└──────────────┴──────────────┴──────────────┴──────────────
──────┘
          │
          ▼
┌──────────────────────────────────────────────────────────
───────┐
│             DATA INGESTION LAYER                 │
├──────────────────────────────────────────────────────────
───────┤
│  • ETL Pipeline (Apache Airflow)                 │
│  • Data Validation & Cleaning                    │
│  • Schema Recognition (ML)                       │
│  • External API Connectors                       │
└──────────────────────────────────────────────────────────
───────┘
          │
          ▼
┌──────────────────────────────────────────────────────────
───────┐
│             EXTERNAL DATA SOURCES                    │
├──────────────────────────────────────────────────────────
───────┤
│  Google Sheets │ CSV/Excel │ CRMs │ HR Systems │ Project Mgmt   │
└──────────────────────────────────────────────────────────
───────┘
```

System Components

1. Client Layer

Web Application (React + TypeScript)

Responsibilities:

- User interface for queries, dashboards, and reports
- Real-time updates via WebSocket
- Responsive design for desktop and mobile web
- Offline capability with service workers

Key Features:

- Chat interface for natural language queries

- Dashboard with customizable widgets
- Report viewer and export functionality
- Alert management console
- Data source configuration UI

Technology:

- React 18+ with TypeScript
- TanStack Query for state management
- Tailwind CSS for styling
- Recharts for data visualization
- Socket.io-client for real-time updates

Mobile Applications (React Native)

Responsibilities:

- Native mobile experience for iOS and Android
- Push notifications for alerts
- Quick query access
- Report viewing on-the-go

2. API Gateway Layer

Kong API Gateway

Responsibilities:

- Single entry point for all client requests
- Authentication and authorization
- Rate limiting (100 requests/min per user)
- Request/response transformation
- API analytics and monitoring

Configuration:

```
services:
  - name: teampulse-api
    url: http://app-service:8000
    routes:
      - name: query-route
        paths: [/api/v1/query]
        methods: [POST]
        plugins:
```

```
    - name: rate-limiting
      config:
        minute: 100
    - name: jwt
    - name: request-validator
```

3. Application Layer

Query Service (Python/FastAPI)

Responsibilities:

- Receive and validate user queries
- Manage conversation context
- Coordinate with AI Orchestration Layer
- Format and return responses
- Log query patterns for optimization

API Endpoints:

```
POST   /api/v1/query              # Submit natural language query
GET    /api/v1/query/history      # Retrieve query history
POST   /api/v1/query/feedback     # Submit feedback on response
GET    /api/v1/query/suggestions  # Get suggested queries
```

Key Functions:

```
async def process_query(
    query: str,
    user_id: str,
    workspace_id: str,
    context: Optional[ConversationContext]
) -> QueryResponse:
    # 1. Validate query
    # 2. Retrieve user permissions and data access
    # 3. Build context with relevant data schema
    # 4. Send to AI Orchestration
    # 5. Process AI response
    # 6. Log query and response
    # 7. Return formatted response
```

Report Service (Python/FastAPI)

Responsibilities:

- Schedule and generate automated reports
- Handle custom report requests
- Manage report templates
- Deliver reports via multiple channels
- Track report engagement metrics

API Endpoints:

```
GET    /api/v1/reports              # List available reports
POST   /api/v1/reports/generate     # Generate on-demand report
PUT    /api/v1/reports/schedule     # Schedule recurring report
GET    /api/v1/reports/{id}         # Retrieve specific report
DELETE /api/v1/reports/{id}         # Delete report
```

Report Generation Flow:

```
async def generate_report(
    report_type: ReportType,
    date_range: DateRange,
    workspace_id: str,
    filters: Optional[Dict]
) -> Report:
    # 1. Fetch relevant data from database
    # 2. Apply filters and transformations
    # 3. Generate insights using AI
    # 4. Create visualizations
    # 5. Format report (PDF, HTML, JSON)
    # 6. Store in object storage
    # 7. Notify recipients
    # 8. Return report metadata
```

Alert Service (Python/FastAPI)

Responsibilities:

- Monitor data for alert conditions
- Evaluate custom alert rules
- Prioritize and deduplicate alerts
- Deliver notifications via multiple channels
- Track alert acknowledgment and resolution

API Endpoints:

```
GET    /api/v1/alerts              # List active alerts
POST   /api/v1/alerts/rules        # Create alert rule
PUT    /api/v1/alerts/{id}/ack     # Acknowledge alert
DELETE /api/v1/alerts/rules/{id}   # Delete alert rule
GET    /api/v1/alerts/history      # Alert history
```

Alert Processing Engine:

```
class AlertProcessor:
    async def evaluate_rules(self, workspace_id: str):
        # 1. Fetch all active rules for workspace
        # 2. Query relevant data
        # 3. Evaluate each rule condition
        # 4. For triggered rules:
        #    - Check if already alerted (deduplication)
        #    - Determine priority using ML model
        #    - Create alert record
        #    - Send notifications
        # 5. Update rule evaluation timestamps
```

Integration Service (Node.js/Express)

Responsibilities:

- Manage OAuth flows for external services
- Sync data from connected sources
- Handle webhooks from external systems
- Transform external data to internal schema
- Maintain connection health

Supported Integrations:

```
const integrations = {
  googleSheets: GoogleSheetsConnector,
  slack: SlackConnector,
  salesforce: SalesforceConnector,
  bambooHR: BambooHRConnector,
  asana: AsanaConnector,
  microsoftTeams: TeamsConnector
};
```

4. AI Orchestration Layer

AI Processing Pipeline

```
┌─────────────────────────────────────────────────────────────────┐
│                    AI ORCHESTRATION FLOW                        │
└─────────────────────────────────────────────────────────────────┘

User Query: "Who didn't come to work yesterday?"
    │
    ▼
┌─────────────────────────────────────────┐
│  1. Query Classification                │
│  ─────────────────────────────────────  │
│  Type: Attendance Query                 │
│  Intent: List Absent Employees          │
│  Time Frame: Yesterday                  │
│  Required Data: Attendance records      │
└───────────────────┬─────────────────────┘
                    │
                    ▼
┌─────────────────────────────────────────┐
│  2. Context Enrichment                  │
│  ─────────────────────────────────────  │
│  • Workspace: Acme Corp                 │
│  • User permissions: Can view all       │
│  • "Yesterday" = Dec 15, 2025           │
│  • Available data sources:              │
│    - attendance_logs table              │
│    - employee_roster table              │
│  • Relevant schema:                     │
│    {employee_id, name, check_in}        │
└───────────────────┬─────────────────────┘
                    │
                    ▼
┌─────────────────────────────────────────┐
│  3. SQL Generation (via Claude)         │
│  ─────────────────────────────────────  │
│  Prompt to Claude:                      │
│  "Given this schema: [schema]           │
│   Generate SQL to find employees        │
│   without attendance on 2025-12-15"     │
│
```

```
│                              │
│  Claude Response:            │
│  SELECT e.name, e.employee_id       │
│  FROM employee_roster e          │
│  LEFT JOIN attendance_logs a       │
│    ON e.employee_id = a.employee_id│
│    AND DATE(a.check_in) =         │
│        '2025-12-15'             │
│  WHERE a.check_in IS NULL         │
│    AND e.status = 'active'        │
└──────────────┬──────────────────┘
               │
               ▼
┌─────────────────────────────────┐
│  4. Query Execution           │
│  ──────────────────────              │
│  Execute SQL against PostgreSQL    │
│  Results: [                   │
│    {name: "John Doe", id: "E123"}, │
│    {name: "Jane Smith", id: "E456"}│
│  ]                            │
└──────────────┬──────────────────┘
               │
               ▼
┌─────────────────────────────────┐
│  5. Response Synthesis (Claude)    │
│  ─────────────────────────────────        │
│  Prompt to Claude:            │
│  "Format these results as natural  │
│   language response: [results]"    │
│                               │
│  Claude Response:             │
│  "Yesterday (Dec 15), 2 employees  │
│  were absent:                 │
│  • John Doe (E123)            │
│  • Jane Smith (E456)          │
│                               │
│  This is 20% higher than your      │
│  typical daily absence rate."      │
└──────────────┬──────────────────┘
               │
               ▼
┌─────────────────────────────────┐
│  6. Response Delivery          │
│  ─────────────────────────              │
│  Return to user with:           │
│  • Natural language answer         │
│  • Structured data (JSON)         │
│  • Confidence score               │
│  • Suggested follow-up queries     │
│  • Execution time: 2.3s           │
└───────────────────────────────────────────────┘
```

AI Components Architecture

1. Query Understanding Module

```
class QueryUnderstandingModule:
    def __init__(self, claude_client):
        self.claude = claude_client

    async def classify_query(self, query: str, context: Context) -> QueryIntent:
        """
        Uses Claude to understand query intent and extract parameters
        """
        prompt = f"""
        Analyze this business query and extract:
        1. Primary intent (attendance, sales, tasks, client, general)
        2. Entities mentioned (people, dates, products, clients)
        3. Time frame (specific date, relative time, date range)
        4. Requested output format (list, summary, comparison, trend)
        5. Filters or conditions

        User Query: "{query}"

        Current Date: {context.current_date}
        Workspace: {context.workspace_name}
        Available Data: {context.data_sources}

        Respond in JSON format.
        """

        response = await self.claude.messages.create(
            model="claude-sonnet-4-20250514",
            max_tokens=1000,
            messages=[{"role": "user", "content": prompt}]
        )
        return parse_query_intent(response.content)
```

2. Schema-to-SQL Translator

```
class SQLGenerationModule:
    async def generate_sql(
        self,
        intent: QueryIntent,
        schema: DatabaseSchema
    ) -> SQLQuery:
        """
        Generates SQL query from intent using Claude with schema context
        """
        prompt = f"""
        You are a SQL expert. Generate a SQL query based on:

        Intent: {intent.description}
        Tables: {schema.tables}
        Relationships: {schema.relationships}

        Requirements:
        - Use PostgreSQL syntax
        - Include appropriate JOINs
        - Add WHERE clauses for filters
        - Handle NULL values properly
        - Optimize for performance

        Return only the SQL query, no explanation.
        """

        response = await self.claude.messages.create(
            model="claude-sonnet-4-20250514",
            max_tokens=1500,
            messages=[{"role": "user", "content": prompt}]
        )

        sql = extract_sql(response.content)

        # Validate SQL for safety
        if not self.is_safe_sql(sql):
            raise SecurityException("Query contains unsafe operations")

        return SQLQuery(sql=sql, intent=intent)
```

3. Report Generation Engine

```
class ReportGenerationEngine:
    async def generate_report(
        self,
        report_type: str,
        data: Dict,
        template: Optional[str] = None
    ) -> Report:
        """
        Uses Claude to generate narrative reports from structured data
        """
        prompt = f"""
        Generate a {report_type} report based on this data:

        {json.dumps(data, indent=2)}

        Requirements:
        - Executive summary (2-3 sentences)
        - Key metrics with context
        - Trends and patterns
        - Actionable insights
        - Areas of concern (if any)

        Format as professional business report.
        """

        response = await self.claude.messages.create(
            model="claude-sonnet-4-20250514",
            max_tokens=4000,
            messages=[{"role": "user", "content": prompt}]
        )

        report_content = response.content[0].text

        # Generate visualizations
        charts = self.create_charts(data)

        return Report(
            content=report_content,
            data=data,
            charts=charts,
            generated_at=datetime.now()
        )
```

4. Anomaly Detection System

```
class AnomalyDetectionSystem:
    def __init__(self):
        self.models = {
            'attendance': IsolationForest(),
            'sales': LSTM_Forecaster(),
            'payments': RegressionAnalyzer()
        }

    async def detect_anomalies(
        self,
        data_type: str,
        current_data: pd.DataFrame,
        historical_data: pd.DataFrame
    ) -> List[Anomaly]:
        """
        Detects anomalies using ML models + Claude for explanation
        """
        # Statistical anomaly detection
        model = self.models[data_type]
        anomalies = model.fit_predict(current_data)

        detected = []
        for idx, is_anomaly in enumerate(anomalies):
            if is_anomaly == -1:  # Anomaly detected
                # Use Claude to explain the anomaly
                explanation = await self.explain_anomaly(
                    data_type=data_type,
                    data_point=current_data.iloc[idx],
                    historical_context=historical_data
                )

                detected.append(Anomaly(
                    type=data_type,
                    data=current_data.iloc[idx],
                    severity=self.calculate_severity(explanation),
                    explanation=explanation
                ))

        return detected

    async def explain_anomaly(
        self,
        data_type: str,
        data_point: pd.Series,
        historical_context: pd.DataFrame
    ) -> str:
        """
        Uses Claude to generate human-readable anomaly explanation
        """
        prompt = f"""
        An unusual pattern was detected in {data_type} data:

        Current Data Point: {data_point.to_dict()}

        Historical Context (last 30 days):
        - Average: {historical_context.mean().to_dict()}
        - Std Dev: {historical_context.std().to_dict()}
        - Min: {historical_context.min().to_dict()}
        - Max: {historical_context.max().to_dict()}

        Explain:
        1. What makes this unusual?
        2. Potential business impact
        3. Recommended action

        Be concise and actionable.
        """

        response = await self.claude.messages.create(
            model="claude-sonnet-4-20250514",
            max_tokens=500,
            messages=[{"role": "user", "content": prompt}]
        )

        return response.content[0].text
```

5. Conversation Context Manager

```
class ConversationContextManager:
    def __init__(self, redis_client):
        self.redis = redis_client
        self.context_ttl = 3600  # 1 hour

    async def get_context(
        self,
        user_id: str,
        conversation_id: str
    ) -> ConversationContext:
        """
        Retrieve conversation history and context
        """
        key = f"context:{user_id}:{conversation_id}"
        context_data = await self.redis.get(key)

        if context_data:
            return ConversationContext.from_json(context_data)
        else:
            return ConversationContext.new(user_id, conversation_id)

    async def update_context(
        self,
        context: ConversationContext,
        query: str,
        response: str
    ):
        """
        Add query-response pair to context for follow-up questions
        """
        context.add_turn(query=query, response=response)

        # Keep only last 10 turns to manage context window
        if len(context.turns) > 10:
            context.turns = context.turns[-10:]

        key = f"context:{context.user_id}:{context.conversation_id}"
        await self.redis.setex(
            key,
            self.context_ttl,
            context.to_json()
        )
```

Data Flow Diagrams

Daily Report Generation Flow

```
┌──────────────────────────────────────────────────────────────────
──┐
│ CRON Trigger (Daily 6:00 AM local time)                  │
```

```
└──────────────────────┬──────────────────────
──┘
                  │
                  ▼
┌─────────────────────────────────────────────
──┐
│ 1. Report Scheduler                         │
│   • Fetch all workspaces with daily reports enabled   │
│   • Determine report types for each workspace        │
│   • Queue report generation jobs                │
└──────────────────────┬──────────────────────
──┘
                  │
                  ▼
┌─────────────────────────────────────────────
──┐
│ 2. Data Aggregation                         │
│   For each workspace, fetch:                  │
│   • Yesterday's sales data                    │
│   • Yesterday's attendance records               │
│   • Open/completed tasks                       │
│   • Client payment status                     │
│   • Any triggered alerts                      │
└──────────────────────┬──────────────────────
──┘
                  │
                  ▼
┌─────────────────────────────────────────────
──┐
│ 3. Statistical Analysis                     │
│   Calculate:                              │
│   • Day-over-day changes                       │
│   • Week-over-week comparisons                   │
│   • Running averages                          │
│   • Variance from targets                     │
└──────────────────────┬──────────────────────
──┘
                  │
                  ▼
┌─────────────────────────────────────────────
──┐
│ 4. AI Report Generation (Claude)                 │
│   Prompt: "Generate daily report with:           │
│   - Executive summary                          │
│     - Key metrics                              │
│     - Notable changes                          │
│     - Recommendations"                         │
│                                                │
│     Include: [aggregated data + analysis]      │
└───────────────────────┬────────────────────────
──┘
          │
          ▼
┌────────────────────────────────────────────────
──┐
│  5. Visualization Generation                   │
│     Create charts:                             │
│     • Sales trend graph                        │
│     • Attendance heatmap                       │
│     • Task completion progress                 │
└───────────────────────┬────────────────────────
──┘
          │
          ▼
┌────────────────────────────────────────────────
──┐
│  6. Report Formatting                          │
│     Generate:                                  │
│     • HTML version (email)                     │
│     • PDF version (download)                   │
│     • JSON version (API access)                │
└───────────────────────┬────────────────────────
──┘
          │
          ▼
┌────────────────────────────────────────────────
──┐
│  7. Multi-channel Delivery                     │
│     • Email to configured recipients           │
│     • Slack/Teams notification with summary    │
│     • In-app notification                      │
│     • Store in S3 for retrieval                │
└───────────────────────┬────────────────────────
──┘
          │
          ▼
┌────────────────────────────────────────────────
──┐
```

```
│ 8. Engagement Tracking                    │
│   Monitor:                        │
│   • Email opens                   │
│   • Report downloads              │
│   • User feedback                 │
└──────────────────────────────────────────────────────
──┘
```

Real-time Alert Processing Flow

```
┌──────────────────────────────────────────────────────
──┐
│ Data Change Event (New record or Update)            │
└──────────────────────┬───────────────────────────────
──┘
              │
              ▼
┌──────────────────────────────────────────────────────
──┐
│ 1. Event Stream (Kafka)                   │
│   Topic: data-changes                     │
│   Event: {                        │
│     type: "attendance_record",            │
│     action: "insert",             │
│     workspace_id: "ws_123",               │
│     data: {...}                   │
│   }                             │
└──────────────────────┬───────────────────────────────
──┘
              │
              ▼
┌──────────────────────────────────────────────────────
──┐
│ 2. Alert Rules Matcher                    │
│   Query active rules for workspace           │
│   Match event against rule conditions:        │
│   • Rule: "Alert if > 5 employees absent"     │
│   • Current state: 6 employees absent         │
│   • Match: TRUE                       │
└──────────────────────┬───────────────────────────────
──┘
              │
              ▼
```

```
┌─────────────────────────────────────────────────┐
│ 3. Deduplication Check                          │
│   • Check Redis: Was this alert sent in last 2 hours? │
│   • If yes: Suppress (avoid alert fatigue)      │
│   • If no: Proceed                              │
└────────────────────┬────────────────────────────┘
                     │
                     ▼
┌─────────────────────────────────────────────────┐
│ 4. Priority Classification (ML Model)           │
│   Analyze:                                      │
│   • Historical impact of similar events         │
│   • Current business context (day of week, season) │
│   • User's past engagement with similar alerts  │
│                                                 │
│   Output: Priority = HIGH                       │
└────────────────────┬────────────────────────────┘
                     │
                     ▼
┌─────────────────────────────────────────────────┐
│ 5. Context Enrichment (Claude)                  │
│   Prompt: "Explain this alert:                  │
│   6 employees are absent today (usual is 2-3).  │
│   Is this significant? What should admin do?"   │
│                                                 │
│   Response: "This is 100% above normal. Check if │
│   there's a team event or illness outbreak..."  │
└────────────────────┬────────────────────────────┘
                     │
                     ▼
┌─────────────────────────────────────────────────┐
│ 6. Alert Creation                               │
│   Create alert record:                          │
│   {                                             │
│     id: "alert_789",                            │
│     workspace_id: "ws_123",                     │
│     type: "attendance_threshold",               │
│     priority: "high",                          │
│     message: "6 employees absent (100% above normal)",     │
│     context: "...",                            │
│     suggested_actions: [...],                  │
│     created_at: "2025-12-16T09:15:00Z"         │
│   }                                            │
└──────────────────────┬──────────────────────────────────────
───┘
                 │
                 ▼
┌─────────────────────────────────────────────────────────────
───┐
│ 7. Multi-channel Notification                  │
│   Based on priority and user preferences:      │
│   • HIGH: Push + SMS + Email                   │
│   • MEDIUM: Push + Email                       │
│   • LOW: In-app only                           │
│                                                │
│   Deliver through:                             │
│   • WebSocket (real-time in-app)               │
│   • FCM (mobile push)                          │
│   • SendGrid (email)                           │
│   • Twilio (SMS for critical)                  │
│   • Slack/Teams webhook                        │
└──────────────────────┬──────────────────────────────────────
───┘
                 │
                 ▼
┌─────────────────────────────────────────────────────────────
───┐
│ 8. Acknowledgment Tracking                     │
│   Monitor:                                     │
│   • User views alert                           │
│   • User acknowledges alert                    │
│   • User takes suggested action                │
│   • Alert auto-resolves or escalates           │
└─────────────────────────────────────────────────────────────
───┘
```

Technology Stack

Backend Services

Component	Technology	Justification
API Services	**Python 3.11 + FastAPI**	Fast development, excellent async support, type hints, OpenAPI auto-generation
Integration Service	**Node.js 20 + Express**	Better ecosystem for OAuth and external API integrations
Task Queue	**Celery + Redis**	Reliable distributed task execution for reports and alerts
Background Jobs	**Apache Airflow**	Orchestrate complex ETL pipelines with monitoring
Message Queue	**Apache Kafka**	High-throughput event streaming for real-time data changes
API Gateway	**Kong**	Production-ready, plugin ecosystem, API management features

AI & Machine Learning

Component	Technology	Purpose
LLM	**Claude 4 Sonnet (Anthropic)**	Query understanding, SQL generation, report writing, anomaly explanation
Vector Database	**Pinecone**	Store embeddings for semantic search and similar query matching
ML Framework	**scikit-learn + PyTorch**	Anomaly detection models, classification
Time Series	**Prophet (Facebook)**	Sales forecasting, trend prediction
Feature Store	**Feast**	Manage ML features for models

Data Layer

Component	Technology	Purpose
Primary Database	**PostgreSQL 15**	Relational data (users, workspaces, configurations, historical data)
Cache	**Redis 7**	Session management, conversation context, rate limiting
Object Storage	**AWS S3**	Reports, uploaded files, backups

Search	**Elasticsearch**	Full-text search across reports and data
Analytics	**ClickHouse**	Time-series analytics, query performance metrics

Frontend

Component	Technology	Purpose
Web Framework	**React 18 + TypeScript**	Type-safe UI development
State Management	**TanStack Query + Zustand**	Server state + local state management
UI Components	**Shadcn/ui + Tailwind CSS**	Consistent, accessible design system
Charts	**Recharts + D3.js**	Interactive data visualizations
Real-time	**Socket.io**	WebSocket connection for live updates
Mobile	**React Native + Expo**	Cross-platform mobile apps

Infrastructure

Component	Technology	Purpose
Cloud Provider	**AWS**	Primary hosting (EC2, RDS, S3, Lambda)
Container Orchestration	**Kubernetes (EKS)**	Microservices deployment and scaling
CI/CD	**GitHub Actions**	Automated testing and deployment
Monitoring	**Datadog**	Application performance monitoring
Logging	**ELK Stack**	Centralized logging (Elasticsearch, Logstash, Kibana)
Error Tracking	**Sentry**	Real-time error monitoring

Security & Auth

Component	Technology	Purpose
Authentication	**Auth0**	User authentication, SSO, MFA

Secrets Management	**AWS Secrets Manager**	API keys, database credentials
Encryption	**AWS KMS**	Key management for data encryption
WAF	**Cloudflare**	DDoS protection, CDN

Security Architecture

Data Security

1. Encryption

```
Data at Rest:
├── Database: AES-256 encryption (PostgreSQL Transparent Data Encryption)
├── Object Storage: S3 server-side encryption (SSE-S3)
├── Backups: Encrypted before storage
└── Secrets: AWS Secrets Manager with automatic rotation

Data in Transit:
├── TLS 1.3 for all client-server communication
├── mTLS between microservices
└── VPN for database access
```

2. Access Control

```
# Role-Based Access Control (RBAC) Model
class Permission:
    READ_SALES = "sales:read"
    READ_ATTENDANCE = "attendance:read"
    READ_TASKS = "tasks:read"
    WRITE_ALERTS = "alerts:write"
    ADMIN_WORKSPACE = "workspace:admin"

class Role:
    ADMIN = [
        Permission.READ_SALES,
        Permission.READ_ATTENDANCE,
        Permission.READ_TASKS,
        Permission.WRITE_ALERTS,
        Permission.ADMIN_WORKSPACE
    ]

    MANAGER = [
        Permission.READ_SALES,
        Permission.READ_ATTENDANCE,
        Permission.READ_TASKS,
        Permission.WRITE_ALERTS
    ]

    VIEWER = [
        Permission.READ_SALES,
        Permission.READ_ATTENDANCE
    ]

# Row-Level Security (RLS) in PostgreSQL
CREATE POLICY workspace_isolation ON sales_data
    FOR ALL
    USING (workspace_id = current_setting('app.workspace_id')::uuid);
```

3. API Security

```
# Rate Limiting Configuration
rate_limits:
  query_api:
    per_user: 100/minute
    per_workspace: 500/minute

  report_generation:
    per_user: 10/hour
    per_workspace: 50/hour

  data_ingestion:
    per_workspace: 1000/hour

# Input Validation
validation_rules:
  sql_generation:
    - No DROP/TRUNCATE/DELETE statements
    - Read-only operations (SELECT) only
    - Query timeout: 30 seconds
    - Result limit: 10,000 rows
```

```
 file_uploads:
   - Max file size: 50 MB
   - Allowed formats: CSV, XLSX, JSON
   - Virus scanning required
```

4. Audit Logging

```
class AuditLogger:
    async def log_event(self, event: AuditEvent):
        """
        Log all security-relevant events
        """
        await self.audit_log.insert({
            'timestamp': datetime.utcnow(),
            'user_id': event.user_id,
            'workspace_id': event.workspace_id,
            'action': event.action,  # READ, WRITE, DELETE, LOGIN, etc.
            'resource': event.resource,  # Table, report, alert
            'ip_address': event.ip_address,
            'user_agent': event.user_agent,
            'result': event.result,  # SUCCESS, FAILURE
            'metadata': event.metadata
        })

# Retention: 2 years for compliance
# Monitored for: Unusual access patterns, privilege escalation, data exfiltration
```

Scalability & Performance

Horizontal Scaling Strategy

```
┌─────────────────────────────────────────────────────────────
──┐
│                 Load Balancer (ALB)                │
│               (Auto-scaling based on traffic)       │
└──────────────────────────┬──────────────────────────────────
──┘
                    │
   ┌──────────────────┼──────────────────┐
   │              │              │
   ▼              ▼              ▼
┌──────────────┐  ┌──────────────┐  ┌──────────────┐
```

```
│  API Server  │    │  API Server  │    │  API Server  │
│   Pod 1      │    │   Pod 2      │    │   Pod 3      │
│              │    │              │    │              │
│  CPU: 40%    │    │  CPU: 45%    │    │  CPU: 38%    │
└──────────────┘    └──────────────┘    └──────────────┘
```

Auto-scaling rules:
- Scale up: When avg CPU > 70% for 2 minutes
- Scale down: When avg CPU < 30% for 5 minutes
- Min replicas: 3
- Max replicas: 20

Caching Strategy

```
class CachingStrategy:
    """
    Multi-layer caching for optimal performance
    """

    # Layer 1: In-memory cache (application level)
    # Duration: 5 minutes
    # Use case: Frequently accessed workspace configs

    # Layer 2: Redis cache
    # Duration: 15-60 minutes
    # Use case: Query results, user sessions, conversation context

    # Layer 3: CDN (Cloudflare)
    # Duration: 24 hours
    # Use case: Static assets, generated reports (PDF)

    async def get_with_cache(self, key: str, fetch_fn: Callable):
        # Try L1 cache
        if key in self.memory_cache:
            return self.memory_cache[key]

        # Try L2 cache (Redis)
        cached = await self.redis.get(key)
        if cached:
            self.memory_cache[key] = cached
            return cached

        # Cache miss: Fetch from database
        data = await fetch_fn()
        # Populate caches
        await self.redis.setex(key, 900, data)  # 15 min
        self.memory_cache[key] = data

        return data
```

Database Optimization

1. Indexing Strategy

```
-- Indexes for common query patterns

-- User queries by workspace and date
CREATE INDEX idx_sales_workspace_date
ON sales_data (workspace_id, sale_date DESC);

-- Attendance lookups
CREATE INDEX idx_attendance_employee_date
ON attendance_logs (employee_id, check_in_date DESC);

-- Task queries
CREATE INDEX idx_tasks_assignee_status
ON tasks (assignee_id, status, due_date);

-- Composite index for complex queries
CREATE INDEX idx_sales_product_date
ON sales_data (workspace_id, product_id, sale_date DESC);

-- Partial index for active records only
CREATE INDEX idx_active_employees
ON employees (workspace_id)
WHERE status = 'active';
```

2. Database Partitioning

```
-- Partition large tables by date for better query performance
CREATE TABLE sales_data (
    id UUID PRIMARY KEY,
    workspace_id UUID NOT NULL,
    sale_date DATE NOT NULL,
    amount DECIMAL(10,2),
    -- other columns
) PARTITION BY RANGE (sale_date);

-- Create partitions for each month
CREATE TABLE sales_data_2025_12
PARTITION OF sales_data
FOR VALUES FROM ('2025-12-01') TO ('2026-01-01');

-- Auto-create partitions using pg_partman extension
-- Retain 24 months, archive older data to cold storage
```

3. Read Replicas

```
Primary Database (Write)
    │
    ├──── Replica 1 (Read) ──► Query Service
    ├──── Replica 2 (Read) ──► Report Service
    └──── Replica 3 (Read) ──► Analytics
```

Read traffic distribution:
- 80% of queries go to replicas
- Only writes and critical reads use primary
- Replica lag monitored (alert if > 1 second)

Performance Targets

Metric	Target	Measurement
API Response Time (p95)	< 300ms	For simple queries
Query Execution (p95)	< 3s	Including AI processing
Report Generation	< 30s	Daily reports
Page Load Time	< 2s	Initial page load
Real-time Alert Latency	< 5s	From event to notification
Database Query Time (p95)	< 100ms	For indexed queries
Uptime	99.9%	~8.7 hours downtime/year

Integration Patterns

Google Sheets Integration

```
// OAuth 2.0 Flow
const googleSheetsConnector = {
  // 1. User initiates connection
  async initiateOAuth(workspaceId, userId) {
    const oauth2Client = new google.auth.OAuth2(
      process.env.GOOGLE_CLIENT_ID,
      process.env.GOOGLE_CLIENT_SECRET,
      `${process.env.APP_URL}/integrations/google/callback`
    );

    const authUrl = oauth2Client.generateAuthUrl({
      access_type: 'offline',
      scope: ['https://www.googleapis.com/auth/spreadsheets.readonly'],
      state: JSON.stringify({ workspaceId, userId })
    });

    return authUrl;
  },

  // 2. Handle OAuth callback
  async handleCallback(code, state) {
    const { workspaceId, userId } = JSON.parse(state);
    const { tokens } = await oauth2Client.getToken(code);

    // Store encrypted tokens
    await db.integrations.create({
      workspace_id: workspaceId,
      user_id: userId,
      provider: 'google_sheets',
      access_token: encrypt(tokens.access_token),
      refresh_token: encrypt(tokens.refresh_token),
      expires_at: new Date(tokens.expiry_date)
    });
  },

  // 3. Sync data from sheet
  async syncSheet(integrationId, sheetUrl) {
    const integration = await db.integrations.findById(integrationId);
    const sheets = google.sheets({
      version: 'v4',
      auth: this.getAuthClient(integration)
    });
    // Extract sheet ID from URL
    const sheetId = this.extractSheetId(sheetUrl);

    // Get sheet metadata
    const metadata = await sheets.spreadsheets.get({
      spreadsheetId: sheetId
    });

    // Read data
    const response = await sheets.spreadsheets.values.get({
      spreadsheetId: sheetId,
      range: 'A1:ZZ',  // Read all data
    });

    const rows = response.data.values;

    // Use AI to detect schema
    const schema = await this.detectSchema(rows[0]);  // headers

    // Transform and load data
    await this.loadData(integration.workspace_id, rows, schema);

    // Set up webhook for real-time updates (if available)
    await this.setupWebhook(sheetId, integration);
  },

  // 4. Real-time sync via polling (Google Sheets doesn't have webhooks)
  async pollForChanges(integrationId) {
    // Run every 15 minutes
    const lastSync = await db.sync_log.getLastSync(integrationId);
    const sheets = this.getAuthClient(integration);

    // Check if sheet modified since last sync
    const metadata = await sheets.spreadsheets.get({
      spreadsheetId: integration.sheet_id,
      fields: 'properties.timeZone,properties.title,sheets.properties'
    });

    // If modified, re-sync
    if (this.isModified(metadata, lastSync)) {
      await this.syncSheet(integrationId, integration.sheet_url);
    }
  }
};
```

Slack Integration

```
const slackConnector = {
  // Send daily report to Slack channel
  async sendReport(workspaceId, report) {
    const integration = await db.integrations.findOne({
      workspace_id: workspaceId,
      provider: 'slack'
    });

    const client = new WebClient(decrypt(integration.access_token));

    await client.chat.postMessage({
      channel: integration.channel_id,
      blocks: [
        {
          type: 'header',
          text: {
            type: 'plain_text',
            text: ' Daily Report - December 16, 2025'
          }
        },
        {
          type: 'section',
          text: {
            type: 'mrkdwn',
            text: `*Sales*: ${report.sales.total} (${report.sales.change}%)\n*Attendance*:
${report.attendance.present}/${report.attendance.expected}`
          }
        },
        {
          type: 'section',
          text: {
            type: 'mrkdwn',
            text: report.summary
          }
        },
        {
          type: 'actions',
          elements: [
            {
              type: 'button',
              text: { type: 'plain_text', text: 'View Full Report' },
              url: `${process.env.APP_URL}/reports/${report.id}`
            }
          ]
        }
      ]
    });
  },

  // Interactive slash command: /teampulse ask [question]
  async handleSlashCommand(payload) {
    const { user_id, text, channel_id } = payload;

    // Acknowledge immediately (Slack requires response within 3s)
    await this.acknowledge(payload.response_url);

    // Process query asynchronously
    const response = await queryService.processQuery(
      query: text,
      user_id: user_id,
      channel: 'slack'
    );

    // Send response to channel
    await client.chat.postMessage({
      channel: channel_id,
      text: response.text,
      blocks: this.formatResponse(response)
    });
  }
};
```

Webhook System for External Events

```
class WebhookHandler:
    """
    Handle incoming webhooks from external systems
    """

    async def handle_webhook(self, provider: str, payload: dict):
        """
        Route webhook to appropriate handler
        """
        handlers = {
            'stripe': self.handle_stripe_webhook,
            'salesforce': self.handle_salesforce_webhook,
            'bamboohr': self.handle_bamboohr_webhook
        }

        handler = handlers.get(provider)
        if handler:
            await handler(payload)
        else:
            logger.warning(f"Unknown webhook provider: {provider}")

    async def handle_salesforce_webhook(self, payload: dict):
        """
        Process Salesforce opportunity updates
        """
        # Verify webhook signature
        if not self.verify_signature(payload):
            raise SecurityException("Invalid webhook signature")

        event_type = payload.get('event_type')

        if event_type == 'opportunity.closed':
            # New sale recorded in Salesforce
            sale_data = {
                'workspace_id': self.get_workspace_from_sf_org(payload['org_id']),
                'amount': payload['data']['amount'],
                'client_id': payload['data']['account_id'],
                'sale_date': payload['data']['close_date'],
                'product': payload['data']['product_name']
            }

            # Insert into our database
            await db.sales_data.insert(sale_data)

            # Trigger alert evaluation (might trigger revenue milestone alert)
            await alert_service.evaluate_rules(sale_data['workspace_id'])
```

Deployment Architecture

Kubernetes Cluster Layout

```
# Production Kubernetes Configuration
```

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: teampulse-prod

---
# API Service Deployment
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api-service
  namespace: teampulse-prod
spec:
  replicas: 5
  selector:
    matchLabels:
      app: api-service
  template:
    metadata:
      labels:
        app: api-service
    spec:
      containers:
      - name: api
        image: teampulse/api:v1.2.3
        ports:
        - containerPort: 8000
        env:
        - name: DATABASE_URL
          valueFrom:
            secretKeyRef:
              name: db-credentials
              key: connection-string
        - name: CLAUDE_API_KEY
          valueFrom:
            secretKeyRef:
              name: ai-credentials
              key: claude-key
        resources:
          requests:
            memory: "512Mi"
            cpu: "500m"
          limits:
            memory: "1Gi"
            cpu: "1000m"
        livenessProbe:
          httpGet:
            path: /health
            port: 8000
          initialDelaySeconds: 30
          periodSeconds: 10
        readinessProbe:
          httpGet:
            path: /ready
            port: 8000
          initialDelaySeconds: 10
          periodSeconds: 5

---
# Horizontal Pod Autoscaler
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: api-service-hpa
  namespace: teampulse-prod
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: api-service
  minReplicas: 3
  maxReplicas: 20
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
  - type: Resource
    resource:
      name: memory
      target:
        type: Utilization
        averageUtilization: 80

---
# Service
```

```
apiVersion: v1
kind: Service
metadata:
  name: api-service
  namespace: teampulse-prod
spec:
  selector:
    app: api-service
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8000
  type: LoadBalancer
```

CI/CD Pipeline

```
# .github/workflows/deploy.yml

name: Deploy to Production

on:
  push:
    branches: [main]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Run unit tests
        run: pytest tests/unit

      - name: Run integration tests
        run: pytest tests/integration

      - name: Security scan
        run: |
          pip install bandit
          bandit -r src/

  build:
    needs: test
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Build Docker image
        run: |
          docker build -t teampulse/api:${{ github.sha }} .
          docker tag teampulse/api:${{ github.sha }} teampulse/api:latest

      - name: Push to registry
        run: |
          echo ${{ secrets.DOCKER_PASSWORD }} | docker login -u ${{
secrets.DOCKER_USERNAME }} --password-stdin
          docker push teampulse/api:${{ github.sha }}
          docker push teampulse/api:latest

  deploy:
    needs: build
    runs-on: ubuntu-latest
    steps:
      - name: Deploy to Kubernetes
        run: |
          kubectl set image deployment/api-service \
            api=teampulse/api:${{ github.sha }} \
            -n teampulse-prod

          kubectl rollout status deployment/api-service -n teampulse-prod

      - name: Run smoke tests
        run: |
          curl https://api.teampulse.com/health
          pytest tests/smoke
```

Monitoring & Observability

Metrics Collection

```
from prometheus_client import Counter, Histogram, Gauge

# Query metrics
query_count = Counter(
    'teampulse_queries_total',
    'Total queries processed',
    ['workspace_id', 'query_type']
)

query_duration = Histogram(
    'teampulse_query_duration_seconds',
    'Query processing duration',
    ['query_type']
)

# AI metrics
claude_api_calls = Counter(
    'teampulse_claude_api_calls_total',
    'Total Claude API calls',
    ['operation']
)

claude_api_errors = Counter(
    'teampulse_claude_api_errors_total',
    'Claude API errors',
    ['error_type']
)

# Business metrics
active_workspaces = Gauge(
    'teampulse_active_workspaces',
    'Number of active workspaces'
)

daily_active_users = Gauge(
    'teampulse_daily_active_users',
    'Daily active users'
)
```

Alerting Rules

```
# Prometheus alerting rules

groups:
  - name: teampulse_alerts
    interval: 1m
    rules:
      - alert: HighErrorRate
        expr: rate(http_requests_total{status="500"}[5m]) > 0.05
        for: 5m
        annotations:
          summary: "High error rate detected"
          description: "Error rate is {{ $value }} per second"

      - alert: SlowQueries
        expr: histogram_quantile(0.95, rate(teampulse_query_duration_seconds_bucket[5m])) > 5
        for: 10m
        annotations:
          summary: "95th percentile query time > 5s"

      - alert: ClaudeAPIDown
        expr: rate(teampulse_claude_api_errors_total[5m]) > 0.5
        for: 2m
        annotations:
          summary: "Claude API experiencing high error rate"

      - alert: DatabaseConnectionPoolExhausted
        expr: pg_stat_activity_count > 90
        for: 5m
        annotations:
          summary: "Database connection pool near capacity"
```

Disaster Recovery

Backup Strategy

```
Database Backups:
├── Continuous: WAL archiving to S3 (Point-in-time recovery)
├── Daily: Full database backup at 2 AM UTC
├── Weekly: Full backup with 90-day retention
└── Monthly: Archive backup with 7-year retention

Application Data:
├── Reports: Replicated to S3 with versioning
├── User uploads: S3 with cross-region replication
└── Redis: Daily RDB snapshots

Recovery Time Objective (RTO): 1 hour
Recovery Point Objective (RPO): 5 minutes
```

Disaster Recovery Runbook

```
#!/bin/bash
# disaster_recovery.sh

# 1. Spin up new database instance
aws rds create-db-instance \
  --db-instance-identifier teampulse-recovery \
  --db-instance-class db.r5.2xlarge \
  --engine postgres

# 2. Restore from latest backup
aws rds restore-db-instance-to-point-in-time \
  --source-db-instance-identifier teampulse-prod \
  --target-db-instance-identifier teampulse-recovery \
  --restore-time "2025-12-16T10:00:00Z"

# 3. Update Kubernetes secrets with new database endpoint
kubectl create secret generic db-credentials \
  --from-literal=connection-string="postgresql://..." \
  --namespace teampulse-prod \
  --dry-run=client -o yaml | kubectl apply -f -

# 4. Rolling restart of services
kubectl rollout restart deployment/api-service -n teampulse-prod
kubectl rollout restart deployment/report-service -n teampulse-prod

# 5. Verify health
kubectl get pods -n teampulse-prod
curl https://api.teampulse.com/health
```

Cost Optimization

Estimated Monthly Costs (at 1000 workspaces)

Service	Monthly Cost	Optimization Strategy
AWS EC2 (Kubernetes)	$3,500	Use spot instances for non-critical workloads
RDS PostgreSQL	$800	Right-size instances, use read replicas
S3 Storage	$200	Lifecycle policies, archive old reports to Glacier

Claude API	$2,000	Cache common queries, use Haiku for simple tasks
Redis	$300	Use ElastiCache reserved instances
Data Transfer	$400	Use CloudFront CDN
Total	**$7,200**	**$7.20 per workspace/month**

Cost Reduction Strategies

1. **Caching**: Reduce Claude API calls by 60% through intelligent caching
2. **Query Optimization**: Batch similar queries to reduce database load
3. **Reserved Instances**: 40% savings on predictable compute loads
4. **Data Archival**: Move reports older than 6 months to S3 Glacier
5. **Auto-scaling**: Scale down during off-hours (nights, weekends)

This architecture provides a robust, scalable foundation for TeamPulse that can handle growth from 100 to 10,000+ workspaces while maintaining performance and reliability.