Architecture Decision Record: Microservices Migration

TechFlow Solutions, Inc.
ADR Number: ADR-2024-003
Decision Date: September 25, 2024
Status: APPROVED - Implementation Phase
Supersedes: ADR-2023-008 (Monolithic Architecture Review)
Next Review: March 25, 2025

Architecture Review Board: - **Chief Technology Officer:** Jennifer Liu (jennifer.liu@techflow.com) - Decision Authority - **VP of Engineering:** Sarah Kim (sarah.kim@techflow.com) - Technical Lead - **Principal Architect:** Marcus Rodriguez (marcus.rodriguez@techflow.com) - Architecture Owner - **Senior Staff Engineer:** Alex Chen (alex.chen@techflow.com) - Implementation Lead - **DevOps Lead:** David Martinez (david.martinez@techflow.com) - Infrastructure Lead - **Security Architect:** Emma Rodriguez (emma.rodriguez@techflow.com) - Security Review

---

Decision Summary

Context and Problem Statement

TechFlow Solutions' CustomerInsight Pro platform has grown from a startup MVP to an enterprise-grade analytics platform serving 150+ enterprise customers with 50,000+ active users. Our current monolithic architecture, while successful in early stages, has become a bottleneck for scalability, team productivity, and system reliability as we scale toward $50M ARR.

Current Architecture Challenges: - **Deployment Bottlenecks:** Single deployment pipeline affects entire platform - **Team Dependencies:** 45+ engineers blocked by shared codebase conflicts - **Scaling Limitations:** Cannot scale individual components based on demand - **Technology Constraints:** Locked into single technology stack across all features - **Reliability Issues:** Single point of failure affecting entire platform

Decision Outcome

APPROVED: Migrate from monolithic architecture to microservices architecture over 18-month timeline with phased decomposition approach.

Key Decision Points: - **Architecture Pattern:** Domain-driven microservices with API gateway - **Technology Stack:** Polyglot approach with standardized infrastructure - **Deployment Strategy:** Kubernetes-based container orchestration - **Data Strategy:** Database-per-service with event-driven consis-

tency - **Migration Approach:** Strangler Fig pattern with incremental extraction

Investment Required: - **Total Budget:** $2.8M over 18 months - **Engineering Resources:** 12 FTE engineers dedicated to migration - **Infrastructure Costs:** $150K additional monthly cloud spend - **Timeline:** Q4 2024 - Q1 2026 completion

---

Technical Context and Current State

Current Monolithic Architecture

System Overview: - **Application:** Single Ruby on Rails application (CustomerInsight Pro) - **Database:** PostgreSQL primary with Redis caching - **Infrastructure:** AWS EC2 with load balancer and auto-scaling - **Deployment:** Single Git repository with unified CI/CD pipeline - **Monitoring:** Centralized logging and monitoring through DataDog

Application Components:

```
                CustomerInsight Pro Monolith

  Web UI Layer (React SPA + Rails API)

  Authentication & Authorization (Devise + CanCan)

  Dashboard Engine (Chart.js + D3.js rendering)

  Data Processing Engine (Sidekiq + Redis queues)

  Connector Framework (API integrations + ETL)

  Analytics Engine (Real-time + Batch processing)

  Notification System (Email + Slack + Webhooks)

  User Management (Organizations + Teams + Permissions)



            PostgreSQL Database + Redis Cache
```

Current Performance Metrics: - **Application Response Time:** 95th percentile 1.2 seconds - **Database Connections:** 85% of connection pool utilized -

Memory Usage: 16GB per application instance - **CPU Utilization:** 70% average during peak hours - **Deployment Frequency:** 2-3 deployments per week - **Deployment Time:** 45 minutes including testing and rollback capability

Technical Debt and Constraints

Code Quality Issues: - **Lines of Code:** 485,000 lines of Ruby code in single repository - **Test Coverage:** 78% overall (target: 85%+) - **Cyclomatic Complexity:** 45% of classes exceed complexity thresholds - **Technical Debt:** Estimated 180 engineer-days of accumulated debt

Scalability Bottlenecks: - **Database Scaling:** Read replicas at 80% capacity during peak usage - **Background Processing:** Sidekiq queues backing up during data ingestion spikes - **File Storage:** S3 storage costs growing 35% quarterly - **CDN Performance:** CloudFront cache hit ratio 73% (target: 90%+)

Development Team Challenges: - **Merge Conflicts:** 23% of pull requests require conflict resolution - **Build Times:** 35-minute CI/CD pipeline for full test suite - **Feature Development:** Average 3.2 weeks from concept to production - **Bug Fix Deployment:** 48-hour average time from fix to production

Business Drivers for Change

Scale and Performance Requirements: - **User Growth:** 300% growth in active users over next 24 months - **Data Volume:** 10x increase in data processing requirements - **Geographic Expansion:** European and Asia-Pacific market entry - **Enterprise Features:** SOC 2, HIPAA, and multi-tenancy requirements

Development Velocity Needs: - **Team Scaling:** Growing from 45 to 80+ engineers by 2025 - **Release Frequency:** Target daily deployments for rapid iteration - **Feature Independence:** Parallel development of major features - **Technology Innovation:** Adoption of AI/ML and real-time technologies

Competitive Pressure: - **Time to Market:** Competitors shipping features 40% faster - **Reliability Requirements:** 99.99% uptime SLA for enterprise customers - **Performance Expectations:** Sub-second dashboard loading requirements - **Integration Capabilities:** 200+ third-party integrations roadmap

---

Architectural Decision Analysis

Options Considered

Option 1: Continue with Monolithic Architecture + Optimization

Approach: - Optimize existing monolithic codebase and infrastructure - Implement vertical scaling and performance improvements - Modularize code within

monolith using Ruby engines - Enhance CI/CD pipeline and development processes

Pros: - **Lower Risk:** No major architectural changes required - **Faster Short-term:** Immediate performance improvements possible - **Simpler Operations:** Single application to deploy and monitor - **Lower Cost:** $800K investment vs $2.8M for microservices - **Team Familiarity:** Existing team expertise in current stack

Cons: - **Scalability Ceiling:** Cannot solve fundamental scalability limitations - **Technology Lock-in:** Continued constraint to Ruby/Rails ecosystem - **Team Scaling Issues:** Developer productivity decreases with team size - **Deployment Risk:** Single deployment pipeline remains bottleneck - **Competitive Disadvantage:** Slower feature development vs competitors

Option 2: Complete Rewrite with Microservices

Approach: - Build entirely new microservices platform from ground up - Modern technology stack (Node.js, Python, Go) - Cloud-native architecture with Kubernetes - Parallel development while maintaining existing system

Pros: - **Clean Architecture:** No legacy technical debt - **Modern Technology:** Latest frameworks and best practices - **Optimal Design:** Purpose-built for current requirements - **Fast Development:** No migration complexity for new features - **Team Autonomy:** Complete freedom in technology choices

Cons: - **Extreme Risk:** Complete business disruption if migration fails - **Massive Investment:** $8M+ total cost with 3-year timeline - **Feature Freeze:** Limited new feature development during rewrite - **Data Migration Risk:** Complex customer data migration requirements - **Market Risk:** Competitors gain significant advantage during rewrite

Option 3: Incremental Microservices Migration (Strangler Fig Pattern)

Approach: - Gradually extract services from monolith using Strangler Fig pattern - API gateway to route requests between monolith and new services - Domain-driven service boundaries based on business capabilities - Dual-write pattern for data migration and consistency

Pros: - **Manageable Risk:** Incremental migration reduces deployment risk - **Continuous Value:** New features delivered throughout migration - **Learning Opportunity:** Team learns microservices while building - **Rollback Capability:** Can revert individual services if needed - **Customer Continuity:** Zero customer-facing disruption

Cons: - **Complexity:** Managing both monolith and microservices simultaneously - **Longer Timeline:** 18-month migration vs 6-month optimization - **Technical Debt:** Some legacy code remains during transition - **Operational Overhead:** Multiple deployment and monitoring systems - **Data Consistency:** Complex eventual consistency patterns required

Decision Matrix and Analysis

Evaluation Criteria:

Criteria	Weight	Monolith + Optimization	Complete Rewrite	Strangler Fig Migration
Business Risk	25%	9/10	3/10	7/10
Technical Scalability	20%	4/10	10/10	8/10
Development Velocity	20%	5/10	9/10	7/10
Cost Efficiency	15%	9/10	2/10	6/10
Timeline to Value	10%	8/10	3/10	6/10
Team Capability	10%	8/10	5/10	7/10
TOTAL SCORE	100%	**6.85/10**	**5.65/10**	**7.05/10**

Rationale for Strangler Fig Migration: - **Balanced Risk/Reward:** Achieves scalability goals while managing risk - **Continuous Delivery:** Maintains feature development velocity - **Learning Organization:** Builds microservices expertise incrementally - **Customer Impact:** Zero disruption to existing customer base - **Competitive Position:** Improves development speed within 6 months

---

Target Microservices Architecture

Service Decomposition Strategy

Domain-Driven Service Boundaries:

1. User Management Service - **Responsibility:** Authentication, authorization, user profiles, organizations - **Technology:** Node.js + Express + JWT - **Database:** PostgreSQL (user data) - **API:** REST + GraphQL - **Team:** Identity and Access Management (3 engineers)

2. Dashboard Service - **Responsibility:** Dashboard creation, layout management, sharing, permissions - **Technology:** React + TypeScript frontend, Node.js backend - **Database:** MongoDB (dashboard metadata) - **API:** GraphQL + WebSocket - **Team:** Dashboard Experience (4 engineers)

3. Data Connector Service - **Responsibility:** Third-party integrations, data source management, ETL - **Technology:** Python + FastAPI + Celery - **Database:** PostgreSQL (connector config) + Redis (job queue) - **API:** REST + Async messaging - **Team:** Data Integration (4 engineers)

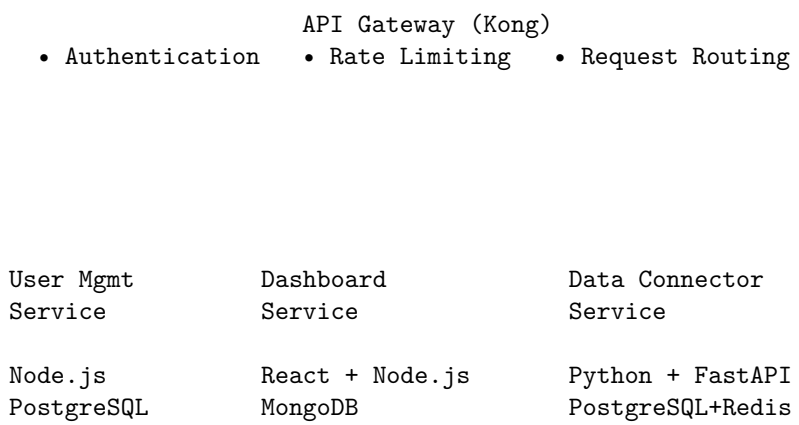
4. Analytics Engine Service - **Responsibility:** Data processing, aggregations, real-time analytics - **Technology:** Go + Apache Kafka + ClickHouse - **Database:** ClickHouse (analytics data) + Redis (caching) - **API:** gRPC + REST - **Team:** Analytics Platform (5 engineers)

5. Visualization Service - **Responsibility:** Chart rendering, visualization types, export functionality - **Technology:** Node.js + D3.js + Puppeteer - **Database:** Redis (cache) + S3 (exports) - **API:** REST + WebSocket - **Team:** Visualization Engine (3 engineers)

6. Notification Service - **Responsibility:** Alerts, email notifications, webhooks, Slack integration - **Technology:** Python + FastAPI + Celery - **Database:** PostgreSQL (notification config) + Redis (queue) - **API:** REST + Event-driven - **Team:** Platform Services (2 engineers)

7. Customer Success Service - **Responsibility:** Usage analytics, health scoring, onboarding tracking - **Technology:** Python + Django + Pandas - **Database:** PostgreSQL (customer data) + ClickHouse (usage analytics) - **API:** REST + GraphQL - **Team:** Customer Success Platform (3 engineers)

Target Architecture Diagram



Analytics Engine	Visualization Service	Notification Service
Go + Kafka ClickHouse	Node.js + D3.js Redis + S3	Python + Celery PostgreSQL+Redis

Customer Success Service

Python + Django + Pandas
PostgreSQL + ClickHouse Analytics

Infrastructure and Platform Services

Container Orchestration: - **Platform:** Amazon EKS (Elastic Kubernetes Service) - **Container Registry:** Amazon ECR - **Ingress Controller:** AWS Load Balancer Controller - **Service Mesh:** Istio for service-to-service communication - **Autoscaling:** Horizontal Pod Autoscaler + Cluster Autoscaler

Data Layer: - **Relational Database:** Amazon RDS PostgreSQL with read replicas - **Document Database:** MongoDB Atlas for flexible schema requirements - **Analytics Database:** ClickHouse on AWS for time-series analytics - **Cache Layer:** Amazon ElastiCache Redis for session and data caching - **Message Queue:** Amazon MSK (Managed Kafka) for event streaming

Observability and Monitoring: - **Metrics:** Prometheus + Grafana for system and business metrics - **Logging:** ELK Stack (Elasticsearch + Logstash + Kibana) - **Tracing:** Jaeger for distributed request tracing - **APM:** DataDog for application performance monitoring - **Alerting:** PagerDuty integration for incident management

Security and Governance: - **Secret Management:** AWS Secrets Manager + Kubernetes secrets - **Network Security:** VPC with private subnets + security groups - **Identity Management:** AWS IAM + RBAC for Kubernetes - **Compliance:** SOC 2 + HIPAA compliant infrastructure - **Vulnerability Scanning:** Snyk for container and dependency scanning

Migration Strategy and Implementation Plan

Strangler Fig Migration Approach

Migration Philosophy: Implement the Strangler Fig pattern where new microservices gradually "strangle" the monolith by intercepting and handling requests that would previously go to the monolithic application. This allows for incremental migration with minimal risk and continuous value delivery.

Technical Implementation: 1. **API Gateway Layer:** Deploy Kong API Gateway to route requests 2. **Service Extraction:** Extract services one domain at a time 3. **Dual Write Pattern:** Write to both old and new systems during transition 4. **Data Migration:** Migrate data incrementally with validation 5. **Cutover:** Switch traffic to new service once validated 6. **Cleanup:** Remove old code and infrastructure after successful migration

Phase 1: Foundation and Infrastructure (Q4 2024 - Q1 2025)

Duration: 4 months
Team Size: 8 engineers (Infrastructure + Platform)
Budget: $650K

Week 1-4: Infrastructure Setup - **EKS Cluster Setup:** Production-ready Kubernetes cluster with monitoring - **CI/CD Pipeline:** Jenkins-X or GitLab CI for containerized deployments - **Service Mesh:** Istio installation and configuration - **Monitoring Stack:** Prometheus, Grafana, ELK stack deployment - **Security Baseline:** Network policies, RBAC, secret management

Week 5-8: API Gateway Implementation - **Kong Deployment:** API gateway with authentication and rate limiting - **Request Routing:** Route specific endpoints to monolith initially - **Monitoring Integration:** Gateway metrics and logging - **Load Testing:** Performance validation under production load - **Documentation:** API gateway configuration and operational procedures

Week 9-12: Development Environment - **Local Development:** Docker Compose setup for local microservices development - **Staging Environment:** Complete staging infrastructure mirroring production - **Testing Framework:** Integration and end-to-end testing infrastructure - **Developer Tooling:** Service discovery, debugging, and monitoring tools - **Team Training:** Kubernetes and microservices training for engineering teams

Week 13-16: First Service Preparation - **Service Templates:** Standardized service templates and best practices - **Database Strategy:** Database-per-service setup and migration tooling - **Data Consistency:** Event sourcing and CQRS pattern implementation - **Security Framework:** Service-to-service authentication and authorization - **Rollback Procedures:** Automated rollback and disaster recovery procedures

Phase 1 Success Criteria: - EKS cluster handling 100% of staging traffic with 99.9% uptime - API gateway successfully routing 50% of production traffic - Developer productivity metrics show <10% impact on feature delivery - Complete infrastructure monitoring and alerting operational - First service extraction ready for production deployment

Phase 2: Core Services Migration (Q1 2025 - Q3 2025)

Duration: 6 months
Team Size: 12 engineers (6 services × 2 engineers average)
Budget: $1.2M

Service 1: User Management Service (Month 1-2) - **Scope:** Authentication, authorization, user profiles, organizations - **Strategy:** Extract user management while maintaining session compatibility - **Data Migration:** User table migration with zero downtime - **Testing:** Security testing and load testing with production data - **Rollout:** Gradual traffic migration starting with 10% of users

Service 2: Notification Service (Month 2-3) - **Scope:** Email notifications, webhooks, Slack integrations, alerts - **Strategy:** Async service with message queue integration - **Data Migration:** Notification templates and configuration - **Testing:** End-to-end notification delivery testing - **Rollout:** Shadow deployment followed by cutover

Service 3: Data Connector Service (Month 3-4) - **Scope:** Third-party API integrations, data source management - **Strategy:** API-first service with connector plugin architecture - **Data Migration:** Connector configurations and credentials - **Testing:** Integration testing with all 150+ data sources - **Rollout:** Connector-by-connector migration

Service 4: Customer Success Service (Month 4-5) - **Scope:** Usage analytics, health scoring, onboarding tracking - **Strategy:** Analytics-focused service with ClickHouse backend - **Data Migration:** Historical usage data migration - **Testing:** Data accuracy validation and performance testing - **Rollout:** Parallel analytics calculation with validation

Service 5: Visualization Service (Month 5-6) - **Scope:** Chart rendering, visualization types, export functionality - **Strategy:** Stateless service with Redis caching - **Data Migration:** Chart templates and configuration - **Testing:** Visual regression testing and performance benchmarks - **Rollout:** A/B testing with 50/50 traffic split

Phase 2 Success Criteria: - 5 microservices handling 60% of platform functionality - 99.95% uptime maintained throughout migration period - Developer velocity improved by 25% due to service independence - Zero customer-facing incidents during service migrations - Database performance improved by 40% due to service separation

Phase 3: Advanced Services and Optimization (Q3 2025 - Q1 2026)

Duration: 6 months
Team Size: 10 engineers (Optimization focus)
Budget: $950K

Service 6: Dashboard Service (Month 1-2) - **Scope:** Dashboard creation, layout management, sharing - **Strategy:** Frontend-heavy service with GraphQL API - **Data Migration:** Dashboard metadata and permissions - **Testing:** UI testing and performance optimization - **Rollout:** Feature flag-based gradual rollout

Service 7: Analytics Engine Service (Month 2-4) - **Scope:** Data processing, aggregations, real-time analytics - **Strategy:** High-performance service with Kafka and ClickHouse - **Data Migration:** Analytics data and processing pipelines - **Testing:** Performance testing with 10x data volume - **Rollout:** Shadow processing with data validation

Performance Optimization (Month 4-6) - **Service Communication:** gRPC optimization and connection pooling - **Database Optimization:** Query optimization and caching strategies - **Resource Optimization:** Right-sizing containers and autoscaling tuning - **Cost Optimization:** Resource utilization analysis and cost reduction - **Monitoring Enhancement:** Advanced observability and business metrics

Legacy Cleanup (Month 5-6) - **Monolith Decomposition:** Remove extracted functionality from monolith - **Database Cleanup:** Remove unused tables and optimize schema - **Infrastructure Cleanup:** Decommission legacy infrastructure - **Documentation Update:** Complete architectural documentation - **Team Transition:** Final team structure and ownership transfer

Phase 3 Success Criteria: - 100% of platform functionality migrated to microservices - 50% improvement in development velocity and deployment frequency - 30% reduction in infrastructure costs through optimization - 99.99% platform uptime with sub-second response times - Complete monolith decommissioning with zero technical debt

Risk Mitigation and Contingency Planning

High-Risk Scenarios and Mitigation:

Risk 1: Data Consistency Issues - **Probability:** 40% - Complex data relationships and transactions - **Impact:** High - Customer data integrity and business continuity - **Mitigation:** Comprehensive dual-write validation and rollback procedures - **Contingency:** Immediate rollback to monolith with data reconciliation

Risk 2: Performance Degradation - **Probability:** 30% - Network latency and service communication overhead - **Impact:** Medium - Customer experience and SLA compliance - **Mitigation:** Extensive load testing and performance

monitoring - **Contingency:** Performance optimization sprint and caching implementation

Risk 3: Team Productivity Loss - **Probability:** 25% - Learning curve and operational complexity - **Impact:** Medium - Feature delivery and development velocity - **Mitigation:** Comprehensive training and gradual responsibility transfer - **Contingency:** Extended timeline and additional training resources

Risk 4: Security Vulnerabilities - **Probability:** 20% - Increased attack surface and service communication - **Impact:** High - Data breach and compliance violations - **Mitigation:** Security-first design and continuous vulnerability scanning - **Contingency:** Immediate security audit and remediation

Risk 5: Cost Overrun - **Probability:** 35% - Infrastructure complexity and operational overhead - **Impact:** Medium - Budget constraints and resource allocation - **Mitigation:** Monthly cost monitoring and optimization - **Contingency:** Cost reduction initiatives and timeline adjustment

---

Technology Stack and Implementation Details

Service-Specific Technology Decisions

User Management Service - Node.js + Express

Rationale: - **Performance:** V8 engine provides excellent I/O performance for auth operations - **Ecosystem:** Rich ecosystem for authentication libraries (Passport.js, JWT) - **Team Expertise:** Frontend team familiar with JavaScript/TypeScript - **Integration:** Seamless integration with React frontend

Technology Stack: - **Runtime:** Node.js 18 LTS with TypeScript - **Framework:** Express.js with TypeScript decorators - **Authentication:** JWT with refresh tokens + OAuth 2.0 - **Database:** PostgreSQL with TypeORM - **Caching:** Redis for session storage - **Testing:** Jest + Supertest for unit and integration testing

Dashboard Service - React + Node.js

Rationale: - **Frontend Continuity:** Leverage existing React frontend expertise - **Real-time Features:** WebSocket support for collaborative editing - **Component Reuse:** Reuse existing dashboard component library - **GraphQL Benefits:** Efficient data fetching for complex dashboard queries

Technology Stack: - **Frontend:** React 18 + TypeScript + Apollo Client - **Backend:** Node.js + GraphQL + Apollo Server - **Database:** MongoDB for flexible dashboard schema - **Real-time:** WebSocket with Socket.io - **State Management:** Redux Toolkit for complex state - **Testing:** React Testing Library + Cypress for E2E

Data Connector Service - Python + FastAPI

Rationale: - **Data Integration:** Python's rich ecosystem for data processing - **API Performance:** FastAPI provides automatic OpenAPI documentation - **Async Processing:** Native async/await support for I/O-heavy operations - **Library Support:** Extensive third-party API client libraries

Technology Stack: - **Framework:** FastAPI with Pydantic for data validation - **Database:** PostgreSQL with SQLAlchemy ORM - **Task Queue:** Celery with Redis broker - **HTTP Client:** httpx for async HTTP requests - **Data Processing:** Pandas for data transformation - **Testing:** pytest + httpx for async testing

Analytics Engine Service - Go + Apache Kafka

Rationale: - **Performance:** Go's concurrency model ideal for data processing - **Resource Efficiency:** Lower memory footprint for high-throughput processing - **Kafka Integration:** Excellent Kafka client libraries - **Scaling:** Efficient horizontal scaling capabilities

Technology Stack: - **Language:** Go 1.21 with goroutines for concurrency - **Message Broker:** Apache Kafka with Confluent Go client - **Database:** ClickHouse for analytics with native Go driver - **HTTP Framework:** Gin for REST API endpoints - **gRPC:** Protocol Buffers for high-performance service communication - **Testing:** Go testing package + testcontainers

Visualization Service - Node.js + D3.js

Rationale: - **Frontend Integration:** Seamless integration with React components - **D3.js Expertise:** Leverage existing visualization library investment - **Server-side Rendering:** Puppeteer for PDF/PNG export generation - **WebSocket Support:** Real-time chart updates

Technology Stack: - **Runtime:** Node.js with TypeScript - **Visualization:** D3.js + Canvas for high-performance rendering - **Export:** Puppeteer for server-side rendering - **Caching:** Redis for rendered chart caching - **API:** REST + WebSocket for real-time updates - **Testing:** Jest + Canvas testing utilities

Notification Service - Python + Celery

Rationale: - **Background Processing:** Celery's mature task queue system - **Email Libraries:** Rich ecosystem for email and notification libraries - **Integration APIs:** Excellent libraries for Slack, webhooks, SMS - **Template Engine:** Jinja2 for email template rendering

Technology Stack: - **Framework:** FastAPI for REST API - **Task Queue:** Celery with Redis broker - **Database:** PostgreSQL for notification configuration - **Email:** SendGrid API with Python SDK - **Templates:** Jinja2 for email and notification templates - **Testing:** pytest + mock for external API testing

Cross-Cutting Infrastructure Decisions

API Gateway - Kong

Rationale: - **Performance:** NGINX-based with excellent throughput - **Plugin Ecosystem:** Rich plugin ecosystem for common requirements - **Enterprise Features:** Authentication, rate limiting, analytics - **Kubernetes Integration:** Native Kubernetes ingress controller

Configuration: - **Authentication:** JWT validation with public key verification - **Rate Limiting:** Per-user and per-API endpoint rate limiting - **Logging:** Structured logging with correlation IDs - **Monitoring:** Prometheus metrics and DataDog integration - **Security:** WAF integration and DDoS protection

Service Mesh - Istio

Rationale: - **Security:** mTLS for service-to-service communication - **Observability:** Automatic metrics, logs, and traces - **Traffic Management:** Advanced routing and load balancing - **Policy Enforcement:** Network policies and access control

Configuration: - **mTLS:** Automatic certificate management and rotation - **Traffic Splitting:** Canary deployments and A/B testing - **Circuit Breaking:** Automatic failure detection and isolation - **Observability:** Jaeger tracing and Prometheus metrics

Database Strategy

PostgreSQL (User Management, Notifications, Connectors): - **High Availability:** RDS Multi-AZ with automatic failover - **Performance:** Read replicas for read-heavy workloads - **Backup:** Automated backups with point-in-time recovery - **Monitoring:** CloudWatch + DataDog for database metrics

MongoDB (Dashboard Service): - **Flexibility:** Schema flexibility for dashboard metadata - **Performance:** Atlas with automatic scaling - **Replication:** Replica sets for high availability - **Indexing:** Compound indexes for complex queries

ClickHouse (Analytics Engine, Customer Success): - **Performance:** Columnar storage for analytics workloads - **Compression:** Excellent compression ratios for time-series data - **Scalability:** Horizontal scaling with sharding - **Real-time:** Real-time data insertion and querying

Redis (Caching, Sessions, Task Queues): - **High Availability:** ElastiCache with cluster mode - **Performance:** In-memory performance for caching - **Persistence:** Backup and restore capabilities - **Monitoring:** CloudWatch metrics and alerting

---

Implementation Timeline and Milestones

Detailed Project Timeline

Phase 1: Foundation (Q4 2024 - Q1 2025) - 16 weeks

Month 1 (Q4 2024): - Week 1-2: Infrastructure planning and AWS environment setup - Week 3-4: EKS cluster deployment and basic monitoring

Month 2 (Q4 2024): - Week 5-6: API Gateway (Kong) deployment and configuration - Week 7-8: Service mesh (Istio) installation and testing

Month 3 (Q1 2025): - Week 9-10: Development environment and CI/CD pipeline - Week 11-12: Monitoring stack (Prometheus, Grafana, ELK)

Month 4 (Q1 2025): - Week 13-14: Security implementation and compliance validation - Week 15-16: First service preparation and team training

Phase 2: Core Services (Q1 2025 - Q3 2025) - 24 weeks

Month 5-6 (Q1-Q2 2025): User Management Service - Week 17-20: Service development and testing - Week 21-24: Data migration and production deployment

Month 7-8 (Q2 2025): Notification Service - Week 25-28: Service development and integration testing - Week 29-32: Production deployment and validation

Month 9-10 (Q2 2025): Data Connector Service - Week 33-36: Service development and connector migration - Week 37-40: Integration testing and production rollout

Month 11-12 (Q3 2025): Customer Success Service - Week 41-44: Service development and analytics migration - Week 45-48: Data validation and production deployment

Month 13-14 (Q3 2025): Visualization Service - Week 49-52: Service development and rendering optimization - Week 53-56: A/B testing and production cutover

Phase 3: Advanced Services (Q3 2025 - Q1 2026) - 24 weeks

Month 15-16 (Q3-Q4 2025): Dashboard Service - Week 57-60: Service development and GraphQL API - Week 61-64: Frontend integration and user testing

Month 17-20 (Q4 2025): Analytics Engine Service - Week 65-72: High-performance service development - Week 73-80: Data processing migration and optimization

Month 21-22 (Q1 2026): Performance Optimization - Week 81-84: Service communication and database optimization - Week 85-88: Resource optimization and cost reduction

Month 23-24 (Q1 2026): Legacy Cleanup - Week 89-92: Monolith decomposition and infrastructure cleanup - Week 93-96: Documentation and team transition

Critical Milestones and Gates

Milestone 1: Infrastructure Ready (Week 16) - **Success Criteria:** EKS cluster handling 100% staging traffic - **Gate:** Infrastructure security audit passed - **Deliverables:** Production-ready Kubernetes cluster, monitoring, CI/CD - **Go/No-Go Decision:** Proceed to service migration phase

Milestone 2: First Service Live (Week 24) - **Success Criteria:** User Management Service handling 100% auth traffic - **Gate:** Zero customer-facing incidents during migration - **Deliverables:** Operational microservice with monitoring and alerting - **Go/No-Go Decision:** Validate migration approach before scaling

Milestone 3: Core Services Complete (Week 56) - **Success Criteria:** 5 services handling 60% of platform functionality - **Gate:** Performance and reliability targets met - **Deliverables:** Stable multi-service architecture - **Go/No-Go Decision:** Proceed to advanced services migration

Milestone 4: Analytics Migration Complete (Week 80) - **Success Criteria:** High-performance analytics service operational - **Gate:** Data processing performance improved 2x - **Deliverables:** Scalable analytics infrastructure - **Go/No-Go Decision:** Finalize migration and optimization

Milestone 5: Migration Complete (Week 96) - **Success Criteria:** 100% functionality migrated, monolith decommissioned - **Gate:** All performance and reliability targets achieved - **Deliverables:** Complete microservices architecture - **Project Completion:** Architecture migration project closed

Resource Allocation and Team Structure

Phase 1 Team (8 engineers): - **Infrastructure Lead:** 1 Senior DevOps Engineer - **Platform Engineers:** 2 Senior Engineers (Kubernetes, monitoring) - **Security Engineer:** 1 Security Architect - **Backend Engineers:** 2 Senior Engineers (API gateway, service templates) - **Frontend Engineer:** 1 Senior Engineer (development tooling) - **Project Manager:** 1 Technical Program Manager

Phase 2 Team (12 engineers): - **Service Teams:** 6 teams of 2 engineers each (1 senior, 1 mid-level) - **Platform Support:** 2 engineers (infrastructure and DevOps) - **Quality Assurance:** 2 engineers (testing and validation) - **Project Manager:** 1 Technical Program Manager - **Architecture Oversight:** 1 Principal Architect (part-time)

Phase 3 Team (10 engineers): - **Advanced Services:** 4 engineers (Dashboard and Analytics services) - **Performance Team:** 3 engineers (optimization

and monitoring) - **Migration Team:** 2 engineers (cleanup and documentation) - **Project Manager:** 1 Technical Program Manager

Budget Allocation: - **Personnel Costs:** $2.1M (75% of total budget) - **Infrastructure Costs:** $450K (16% of total budget) - **Tools and Software:** $150K (5% of total budget) - **Training and Consulting:** $100K (4% of total budget)

Success Metrics and Monitoring

Technical Performance Metrics

System Performance KPIs:

Response Time and Latency: - **API Response Time:** 95th percentile <500ms (vs current 1.2s) - **Dashboard Load Time:** 95th percentile <2s (vs current 4.5s) - **Service-to-Service Latency:** 99th percentile <100ms - **Database Query Time:** 95th percentile <200ms

Scalability and Throughput: - **Concurrent Users:** Support 50,000+ concurrent users (vs current 15,000) - **API Throughput:** 10,000+ requests/second (vs current 2,500) - **Data Processing:** 1M+ events/minute (vs current 200K) - **Resource Utilization:** <70% CPU/memory during peak load

Reliability and Availability: - **Platform Uptime:** 99.99% uptime SLA (vs current 99.9%) - **Service Availability:** Individual service 99.95% uptime - **Mean Time to Recovery:** <15 minutes (vs current 45 minutes) - **Error Rate:** <0.1% API error rate (vs current 0.3%)

Infrastructure Efficiency: - **Container Startup Time:** <30 seconds for service deployment - **Resource Costs:** 30% reduction in per-user infrastructure cost - **Auto-scaling Response:** <2 minutes scale-up response time - **Storage Efficiency:** 50% improvement in data storage compression

Development Velocity Metrics

Team Productivity KPIs:

Deployment and Release: - **Deployment Frequency:** Daily deployments per service (vs weekly monolith) - **Lead Time:** <24 hours from commit to production (vs 72 hours) - **Change Failure Rate:** <5% deployments requiring rollback (vs 12%) - **Recovery Time:** <15 minutes rollback time (vs 2 hours)

Development Efficiency: - **Build Time:** <10 minutes per service CI/CD (vs 35 minutes monolith) - **Test Execution:** <15 minutes full test suite per service - **Feature Development:** 40% faster feature development cycle - **Code Review Time:** <4 hours average review time (vs 12 hours)

Team Autonomy: - **Service Ownership:** 100% services have dedicated team ownership - **Independent Releases:** 90% releases with zero cross-team dependencies - **Technology Choice:** Teams free to choose optimal tech stack - **Decision Speed:** 50% faster technical decision making

Business Impact Metrics

Customer Experience KPIs:

Performance and Usability: - **Customer Satisfaction:** NPS >70 (vs current 68) - **Platform Performance:** Customer-reported performance issues <5/month - **Feature Adoption:** 40% faster adoption of new features - **Support Tickets:** 30% reduction in performance-related tickets

Business Value: - **Time to Market:** 50% faster new feature delivery - **Customer Retention:** >95% annual retention (vs current 91%) - **Revenue Growth:** Support 300% user growth without proportional cost increase - **Market Expansion:** Enable European data residency and compliance

Monitoring and Observability Strategy

Real-time Monitoring:

Application Performance Monitoring (APM): - **Tool:** DataDog APM for end-to-end performance visibility - **Metrics:** Request traces, database queries, external API calls - **Alerting:** Real-time alerts for performance degradation - **Dashboards:** Service-specific and business-level dashboards

Infrastructure Monitoring: - **Tool:** Prometheus + Grafana for infrastructure metrics - **Metrics:** CPU, memory, network, disk utilization - **Alerting:** PagerDuty integration for infrastructure incidents - **Capacity Planning:** Trend analysis for resource planning

Business Metrics: - **Tool:** Custom dashboards with business KPIs - **Metrics:** User engagement, feature usage, conversion rates - **Alerting:** Business metric anomaly detection - **Reporting:** Executive dashboards and automated reports

Logging and Tracing:

Centralized Logging: - **Tool:** ELK Stack (Elasticsearch, Logstash, Kibana) - **Structure:** Structured logging with correlation IDs - **Retention:** 90-day log retention with archival - **Search:** Full-text search and log analysis capabilities

Distributed Tracing: - **Tool:** Jaeger for request tracing across services - **Coverage:** 100% service-to-service request tracing - **Performance:** <1ms tracing overhead per request - **Analysis:** Root cause analysis and performance bottleneck identification

Security Monitoring: - **Tool:** Security Information and Event Management (SIEM) - **Metrics:** Authentication events, API access patterns, anomalies -

Alerting: Real-time security incident detection - **Compliance:** SOC 2 and audit trail requirements

Success Criteria and Gates

Phase Completion Criteria:

Phase 1 Success (Infrastructure): - EKS cluster operational with 99.9% uptime - API Gateway handling 100% staging traffic - Monitoring and alerting fully operational - Security audit passed with zero critical findings - Development team trained and productive

Phase 2 Success (Core Services): - 5 microservices operational in production - 60% of platform functionality migrated - Zero customer-facing incidents during migration - Performance improvements visible to customers - Development velocity improved by 25%

Phase 3 Success (Complete Migration): - 100% functionality migrated to microservices - Monolith completely decommissioned - All performance and reliability targets achieved - Cost optimization goals met - Team structure and ownership established

Project Success Metrics: - 99.99% platform uptime SLA achieved - 50% improvement in development velocity - 30% reduction in infrastructure costs - Zero security incidents during migration - Customer satisfaction maintained or improved

---

Risk Assessment and Mitigation

Technical Risk Analysis

High-Priority Technical Risks:

Risk 1: Data Consistency and Integrity - **Risk Level:** HIGH (Probability: 40%, Impact: Critical) - **Description:** Eventual consistency and dual-write patterns may cause data inconsistencies - **Potential Impact:** Customer data corruption, financial discrepancies, compliance violations - **Root Causes:** Complex data relationships, concurrent writes, network partitions

Mitigation Strategies: - **Saga Pattern:** Implement distributed transaction management - **Event Sourcing:** Maintain complete audit trail of all data changes - **Dual Write Validation:** Continuous validation between old and new systems - **Rollback Procedures:** Automated rollback with data reconciliation - **Testing:** Chaos engineering and data consistency testing

Monitoring and Detection: - **Data Validation Jobs:** Continuous background validation of data consistency - **Metrics:** Real-time metrics on data

discrepancies and validation failures - **Alerting:** Immediate alerts for data consistency violations - **Dashboards:** Business intelligence dashboards for data quality monitoring

Risk 2: Service Communication and Network Reliability - **Risk Level:** MEDIUM (Probability: 30%, Impact: High) - **Description:** Network latency and service communication failures - **Potential Impact:** Degraded performance, cascading failures, user experience issues - **Root Causes:** Service mesh complexity, network congestion, service dependencies

Mitigation Strategies: - **Circuit Breaker Pattern:** Automatic failure detection and isolation - **Retry Logic:** Exponential backoff and jitter for transient failures - **Timeout Configuration:** Appropriate timeout settings for all service calls - **Load Balancing:** Intelligent load balancing with health checks - **Caching:** Strategic caching to reduce service dependencies

Monitoring and Detection: - **Service Mesh Metrics:** Istio metrics for service-to-service communication - **Latency Monitoring:** Real-time latency monitoring and alerting - **Error Rate Tracking:** Service error rates and failure patterns - **Dependency Mapping:** Service dependency visualization and analysis

Risk 3: Operational Complexity and Monitoring - **Risk Level:** MEDIUM (Probability: 35%, Impact: Medium) - **Description:** Increased operational complexity with multiple services - **Potential Impact:** Longer incident response times, difficult troubleshooting - **Root Causes:** Service proliferation, monitoring tool complexity, team knowledge gaps

Mitigation Strategies: - **Standardized Monitoring:** Consistent monitoring and alerting across all services - **Centralized Logging:** ELK stack for centralized log analysis - **Distributed Tracing:** Jaeger for end-to-end request tracing - **Runbook Documentation:** Comprehensive operational runbooks - **Team Training:** Extensive training on microservices operations

Monitoring and Detection: - **Unified Dashboards:** Single pane of glass for platform monitoring - **Correlation IDs:** Request correlation across all services - **Automated Diagnostics:** AI-powered incident detection and diagnosis - **Escalation Procedures:** Clear escalation paths for different incident types

Business and Organizational Risks

Medium-Priority Business Risks:

Risk 4: Team Productivity During Transition - **Risk Level:** MEDIUM (Probability: 45%, Impact: Medium) - **Description:** Temporary productivity loss during learning curve - **Potential Impact:** Delayed feature delivery, customer complaints, competitive disadvantage - **Root Causes:** New technology learning, process changes, dual system maintenance

Mitigation Strategies: - **Gradual Transition:** Phased approach minimizes disruption - **Comprehensive Training:** Extensive microservices and DevOps training - **Mentoring Program:** Pair experienced engineers with teams - **Tool Investment:** Best-in-class development and debugging tools - **Process Documentation:** Clear processes and best practices

Monitoring and Detection: - **Velocity Metrics:** Track development velocity and deployment frequency - **Team Surveys:** Regular surveys on team confidence and productivity - **Code Quality Metrics:** Technical debt and code quality tracking - **Feature Delivery Tracking:** Time from concept to production measurement

Risk 5: Customer Impact During Migration - **Risk Level:** LOW (Probability: 15%, Impact: High) - **Description:** Customer-facing issues during service migration - **Potential Impact:** Customer churn, revenue loss, reputation damage - **Root Causes:** Migration bugs, performance degradation, service outages

Mitigation Strategies: - **Strangler Fig Pattern:** Zero customer impact migration approach - **Feature Flags:** Ability to instantly rollback problematic changes - **Blue-Green Deployment:** Zero-downtime deployment strategy - **Customer Communication:** Proactive communication about maintenance - **Support Team Preparation:** Enhanced support during migration periods

Monitoring and Detection: - **Customer Health Metrics:** Real-time customer usage and satisfaction metrics - **Support Ticket Volume:** Monitoring support ticket trends and categories - **Performance Monitoring:** Customer-facing performance metrics - **Feedback Channels:** Direct customer feedback collection and analysis

Financial and Resource Risks

Low-Priority Financial Risks:

Risk 6: Budget Overrun and Cost Escalation - **Risk Level:** MEDIUM (Probability: 35%, Impact: Medium) - **Description:** Project costs exceeding $2.8M budget - **Potential Impact:** Reduced profitability, project scope reduction, resource constraints - **Root Causes:** Infrastructure costs, extended timeline, additional tooling needs

Mitigation Strategies: - **Monthly Budget Reviews:** Regular cost tracking and forecasting - **Cost Optimization:** Continuous infrastructure cost optimization - **Scope Management:** Clear scope definition and change control - **Resource Planning:** Detailed resource allocation and utilization tracking - **Vendor Negotiations:** Strategic vendor relationships and contract negotiations

Monitoring and Detection: - **Cost Dashboards:** Real-time infrastructure and project cost tracking - **Budget Variance Reports:** Monthly budget vs

actual analysis - **Resource Utilization:** Cloud resource utilization and optimization opportunities - **ROI Tracking:** Regular ROI analysis and business case validation

Contingency Planning

Scenario-Based Response Plans:

Scenario 1: Critical Data Consistency Issue - **Response Time:** <30 minutes detection to response - **Actions:** Immediate traffic routing to monolith, data reconciliation, root cause analysis - **Escalation:** CTO and CEO notification within 1 hour - **Recovery:** Service restoration within 4 hours, complete analysis within 24 hours

Scenario 2: Major Performance Degradation - **Response Time:** <15 minutes detection to response - **Actions:** Auto-scaling activation, traffic load balancing, performance profiling - **Escalation:** Engineering leadership notification within 30 minutes - **Recovery:** Performance restoration within 2 hours, optimization within 48 hours

Scenario 3: Service Communication Failure - **Response Time:** <5 minutes detection to response - **Actions:** Circuit breaker activation, fallback service routing, network diagnostics - **Escalation:** DevOps team notification immediately - **Recovery:** Service restoration within 1 hour, post-mortem within 24 hours

Emergency Response Team: - **Incident Commander:** Principal Architect or VP Engineering - **Technical Lead:** Service owner engineer - **DevOps Engineer:** Infrastructure and deployment specialist - **Customer Success:** Customer communication and impact assessment - **Communications:** Internal and external communication coordination

---

Appendix A: Service Interface Definitions

User Management Service API

Authentication Endpoints:

```
POST /auth/login
POST /auth/logout
POST /auth/refresh
POST /auth/forgot-password
POST /auth/reset-password
```

User Management Endpoints:

```
GET /users/{id}
PUT /users/{id}
```

```
DELETE /users/{id}
GET /users/{id}/organizations
POST /users/{id}/organizations
```

Organization Management:

```
GET /organizations/{id}
PUT /organizations/{id}
GET /organizations/{id}/users
POST /organizations/{id}/users
DELETE /organizations/{id}/users/{userId}
```

Dashboard Service GraphQL Schema

```graphql
type Dashboard {
  id: ID!
  name: String!
  description: String
  layout: [Widget!]!
  permissions: DashboardPermissions!
  createdAt: DateTime!
  updatedAt: DateTime!
}

type Widget {
  id: ID!
  type: WidgetType!
  position: Position!
  size: Size!
  configuration: WidgetConfig!
  dataSource: DataSource!
}

type Query {
  dashboard(id: ID!): Dashboard
  dashboards(organizationId: ID!): [Dashboard!]!
  widgets(dashboardId: ID!): [Widget!]!
}

type Mutation {
  createDashboard(input: CreateDashboardInput!): Dashboard!
  updateDashboard(id: ID!, input: UpdateDashboardInput!): Dashboard!
  deleteDashboard(id: ID!): Boolean!
}

type Subscription {
  dashboardUpdated(id: ID!): Dashboard!
```

```
  widgetDataUpdated(widgetId: ID!): WidgetData!
}
```

Data Connector Service API

Connector Management:

```
GET /connectors
POST /connectors
GET /connectors/{id}
PUT /connectors/{id}
DELETE /connectors/{id}
```

Data Source Operations:

```
POST /connectors/{id}/test-connection
POST /connectors/{id}/sync
GET /connectors/{id}/schema
GET /connectors/{id}/data
```

Integration Status:

```
GET /connectors/{id}/status
GET /connectors/{id}/logs
GET /connectors/{id}/metrics
```

Appendix B: Database Schema Evolution

Data Migration Strategy

User Management Service - PostgreSQL

Migration Script Example:

```sql
-- Create new user management database
CREATE DATABASE user_management;

-- Migrate users table
CREATE TABLE users (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    email VARCHAR(255) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL,
    first_name VARCHAR(100),
    last_name VARCHAR(100),
    status VARCHAR(20) DEFAULT 'active',
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW()
);
```

```sql
-- Migrate organizations table
CREATE TABLE organizations (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    name VARCHAR(255) NOT NULL,
    subdomain VARCHAR(100) UNIQUE,
    plan VARCHAR(50) DEFAULT 'professional',
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW()
);

-- Migrate user_organizations relationship
CREATE TABLE user_organizations (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID REFERENCES users(id),
    organization_id UUID REFERENCES organizations(id),
    role VARCHAR(50) DEFAULT 'member',
    created_at TIMESTAMP DEFAULT NOW()
);

-- Data migration from monolith
INSERT INTO users (id, email, password_hash, first_name, last_name, status, created_at, upda
SELECT id, email, encrypted_password, first_name, last_name,
       CASE WHEN active THEN 'active' ELSE 'inactive' END,
       created_at, updated_at
FROM monolith.users;
```

Dashboard Service - MongoDB

Document Schema:

```javascript
// Dashboard document schema
{
  _id: ObjectId,
  name: String,
  description: String,
  organizationId: String,
  createdBy: String,
  layout: {
    grid: {
      rows: Number,
      columns: Number
    },
    widgets: [
      {
        id: String,
        type: String, // 'chart', 'metric', 'table', 'map'
```

```
        position: {
          x: Number,
          y: Number,
          width: Number,
          height: Number
        },
        configuration: {
          title: String,
          dataSource: {
            connectorId: String,
            query: Object
          },
          visualization: {
            type: String,
            options: Object
          }
        }
      }
    ]
  },
  permissions: {
    owner: String,
    editors: [String],
    viewers: [String],
    public: Boolean
  },
  createdAt: Date,
  updatedAt: Date
}
```

Analytics Engine - ClickHouse

Table Schema:

```
-- Events table for real-time analytics
CREATE TABLE events (
    timestamp DateTime,
    organization_id String,
    user_id String,
    event_type String,
    event_data String,
    session_id String,
    ip_address String,
    user_agent String
) ENGINE = MergeTree()
PARTITION BY toYYYYMM(timestamp)
ORDER BY (organization_id, timestamp);
```

```sql
-- Aggregated metrics table
CREATE TABLE metrics_hourly (
    hour DateTime,
    organization_id String,
    metric_type String,
    metric_value Float64,
    dimensions Map(String, String)
) ENGINE = SummingMergeTree()
PARTITION BY toYYYYMM(hour)
ORDER BY (organization_id, metric_type, hour);

-- User activity summary
CREATE TABLE user_activity (
    date Date,
    organization_id String,
    user_id String,
    page_views UInt32,
    dashboard_views UInt32,
    queries_run UInt32,
    session_duration UInt32
) ENGINE = SummingMergeTree()
PARTITION BY toYYYYMM(date)
ORDER BY (organization_id, date, user_id);
```

Appendix C: Monitoring and Alerting Configuration

Prometheus Metrics Configuration

Service-Level Metrics:

```yaml
# HTTP request metrics
http_requests_total:
  type: counter
  labels: [method, endpoint, status]
  help: "Total HTTP requests"

http_request_duration_seconds:
  type: histogram
  labels: [method, endpoint]
  help: "HTTP request duration"

# Database metrics
db_connections_active:
  type: gauge
```

```
  labels: [database, service]
  help: "Active database connections"

db_query_duration_seconds:
  type: histogram
  labels: [database, query_type]
  help: "Database query duration"

# Business metrics
dashboards_created_total:
  type: counter
  labels: [organization_id]
  help: "Total dashboards created"

user_sessions_active:
  type: gauge
  labels: [organization_id]
  help: "Active user sessions"
```

Alerting Rules:

```
groups:
  - name: microservices-alerts
    rules:
      - alert: HighErrorRate
        expr: |
          (
            sum(rate(http_requests_total{status=~"5.."}[5m]))
            /
            sum(rate(http_requests_total[5m]))
          ) > 0.05
        for: 5m
        labels:
          severity: critical
        annotations:
          summary: "High error rate detected"
          description: "Error rate is {{ $value | humanizePercentage }}"

      - alert: HighLatency
        expr: |
          histogram_quantile(0.95, sum(rate(http_request_duration_seconds_bucket[5m])) by (l
        for: 10m
        labels:
          severity: warning
        annotations:
          summary: "High latency detected"
          description: "95th percentile latency is {{ $value }}s"
```

```yaml
      - alert: ServiceDown
        expr: up == 0
        for: 2m
        labels:
          severity: critical
        annotations:
          summary: "Service is down"
          description: "{{ $labels.instance }} is down"
```

Grafana Dashboard Configuration

Service Overview Dashboard:

```
{
  "dashboard": {
    "title": "Microservices Overview",
    "panels": [
      {
        "title": "Request Rate",
        "type": "graph",
        "targets": [
          {
            "expr": "sum(rate(http_requests_total[5m])) by (service)",
            "legendFormat": "{{ service }}"
          }
        ]
      },
      {
        "title": "Error Rate",
        "type": "graph",
        "targets": [
          {
            "expr": "sum(rate(http_requests_total{status=~\"5..\"}[5m])) by (service)",
            "legendFormat": "{{ service }}"
          }
        ]
      },
      {
        "title": "Response Time",
        "type": "graph",
        "targets": [
          {
            "expr": "histogram_quantile(0.95, sum(rate(http_request_duration_seconds_bucket
            "legendFormat": "{{ service }} p95"
          }
```

```
        ]
      }
    ]
  }
}
```

Architecture Decision Record compiled by: Marcus Rodriguez, Principal Architect
Technical Review by: Sarah Kim, VP of Engineering
Business Review by: Jennifer Liu, Chief Technology Officer
Security Review by: Emma Rodriguez, Security Architect
Executive Approval by: David Park, Chief Executive Officer
Classification: Internal - Engineering and Executive Use Only