Contents

Cloud Architecture Design Document

Executive Summary

This document provides a comprehensive overview of a production-grade, cloud-native e-commerce platform built using microservices architecture with multi-cloud deployment across AWS and GCP. The system demonstrates modern DevOps practices including Infrastructure as Code, GitOps, event-driven architecture, and comprehensive observability.

System Overview

Business Context

The platform is a scalable e-commerce solution that enables: - **Customer Operations**: User registration, authentication, and profile management - **Product Management**: Dynamic product catalog with search and filtering - **Order Processing**: Real-time order creation, payment processing, and order tracking - **Notifications**: Multi-channel customer notifications (Email, SMS, Push) - **Analytics**: Real-time business intelligence and reporting

Technical Goals

1. **High Availability**: 99.9% uptime with multi-AZ deployment
2. **Scalability**: Auto-scaling from 2 to 100+ concurrent users
3. **Event-Driven Architecture**: Asynchronous processing with Kafka
4. **Real-Time Analytics**: Stream processing with Apache Flink
5. **Infrastructure as Code**: Reproducible environments with Terraform
6. **GitOps Deployment**: Automated deployments with ArgoCD
7. **Observability**: Comprehensive monitoring with Prometheus/Grafana
8. **Security**: Zero-trust architecture with IAM and network policies

Technology Stack

- **Container Orchestration**: Kubernetes (EKS on AWS, GKE on GCP)
- **Microservices Framework**: Python Flask, Spring Boot (Analytics)
- **Message Broker**: Apache Kafka (AWS MSK)
- **Databases**: PostgreSQL (RDS), DynamoDB, Redis
- **Stream Processing**: Apache Flink on GCP Dataproc
- **Infrastructure**: Terraform
- **CI/CD**: ArgoCD, GitHub Actions

- **Monitoring**: Prometheus, Grafana, EFK Stack

Cloud Deployment Architecture

High-Level Architecture Diagram

INTERNET / USERS □ (HTTPS)

AWS CLOUD (Primary)

VPC (10.0.0.0/16) - Multi-AZ Architecture

1. **Public Subnets** (us-east-1a, us-east-1b)
   - AWS Application Load Balancer (Layer 7)
   - Ingress Controller (NGINX)
2. **Amazon EKS Cluster** (Kubernetes 1.27)
   - **Microservices Pods:**
     - API Gateway (2-10 replicas with HPA)
     - User Service (2 replicas static)
     - Product Service (2 replicas static)
     - Order Service (2-8 replicas with HPA)
     - Payment Service (2 replicas static)
     - Notification Service (1 replica)
   - **Management:**
     - Horizontal Pod Autoscaler (HPA)
     - Prometheus (Metrics Collection)
     - ArgoCD (GitOps Deployment)
3. **Private Subnets** (us-east-1a, us-east-1b)
   - RDS PostgreSQL (Multi-AZ) - Users, Orders, Payments
   - DynamoDB (On-demand) - Product Catalog
   - Redis (ElastiCache) - Caching Layer
   - AWS MSK (Kafka) - Event Streaming
     - orders topic (3 partitions, replication factor 2)
     - analytics-results topic (3 partitions, replication factor 2)
4. **Additional AWS Services**
   - S3 (Object Storage)
   - Lambda (Serverless Functions)
   - ECR (Container Registry)
   - Route 53 (DNS)
   - CloudWatch (Monitoring)

□ (Cross-Cloud Kafka Connection)

GOOGLE CLOUD (Analytics)

GCP Dataproc Cluster - Apache Flink Analytics Job - Kafka Consumer (orders topic) - Stream Processing (Window Aggregation) - Kafka Producer (analytics-results topic) - Google Cloud Storage (GCS) - Flink checkpoints - Analytics results - Data lake storage

Multi-Cloud Strategy

Primary Cloud: AWS - **EKS (Elastic Kubernetes Service)**: Primary container orchestration platform - **RDS PostgreSQL**: Relational data for users, orders, payments - **DynamoDB**: NoSQL storage for product catalog - **MSK (Managed Streaming for Kafka)**: Event streaming backbone - **S3**: Object storage for static assets and backups - **Lambda**: Serverless functions for event processing - **ALB**: Application Load Balancer for traffic distribution - **ECR**: Container image registry - **Route 53**: DNS management - **CloudWatch**: Basic monitoring and logging

Secondary Cloud: GCP - **Dataproc**: Managed Spark/Flink cluster for stream processing - **Cloud Storage (GCS)**: Data lake and checkpoints - **Cloud Pub/Sub**: Alternative messaging (future) - **BigQuery**: Data warehousing (future)

Deployment Rationale

Why AWS as Primary Cloud?

1. **Mature Kubernetes Platform**: EKS is battle-tested with excellent AWS service integration
2. **Rich Database Options**: RDS and DynamoDB cover all data storage needs
3. **Managed Kafka**: MSK eliminates operational overhead of running Kafka clusters
4. **Cost Effectiveness**: Competitive pricing with reserved instances and savings plans
5. **Enterprise Support**: Extensive documentation and community support
6. **Compliance**: SOC 2, HIPAA, PCI-DSS certifications

Why GCP for Analytics?

1. **Best-in-Class Analytics**: Dataproc with Flink provides superior stream processing
2. **Workload Isolation**: Separates compute-intensive analytics from transactional workloads
3. **Cost Optimization**: Pay only for analytics compute when needed
4. **Data Processing Excellence**: GCP's heritage with Hadoop ecosystem
5. **Future Integration**: Easy path to BigQuery and ML services

Multi-Cloud Benefits

- **Vendor Lock-in Avoidance**: Not dependent on single cloud provider
- **Best-of-Breed Services**: Use optimal service from each provider
- **Risk Mitigation**: Disaster recovery across cloud providers
- **Skills Development**: Team gains multi-cloud expertise
- **Negotiation Power**: Better pricing negotiations with multiple options

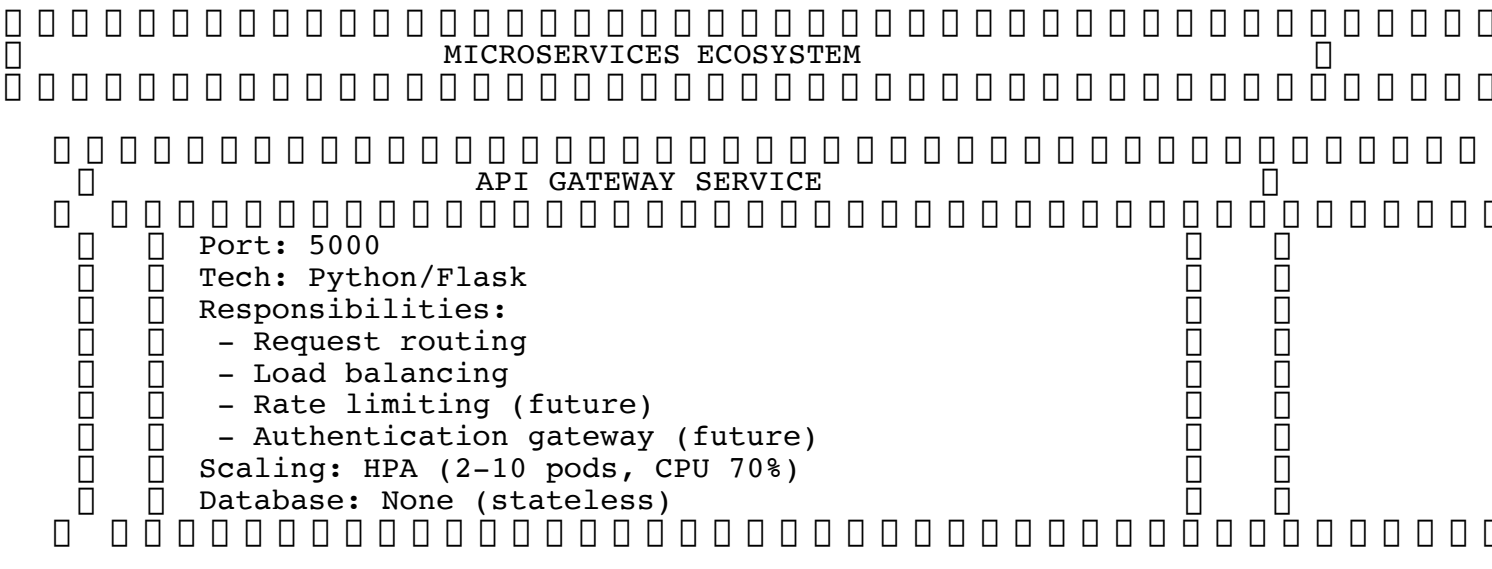
---

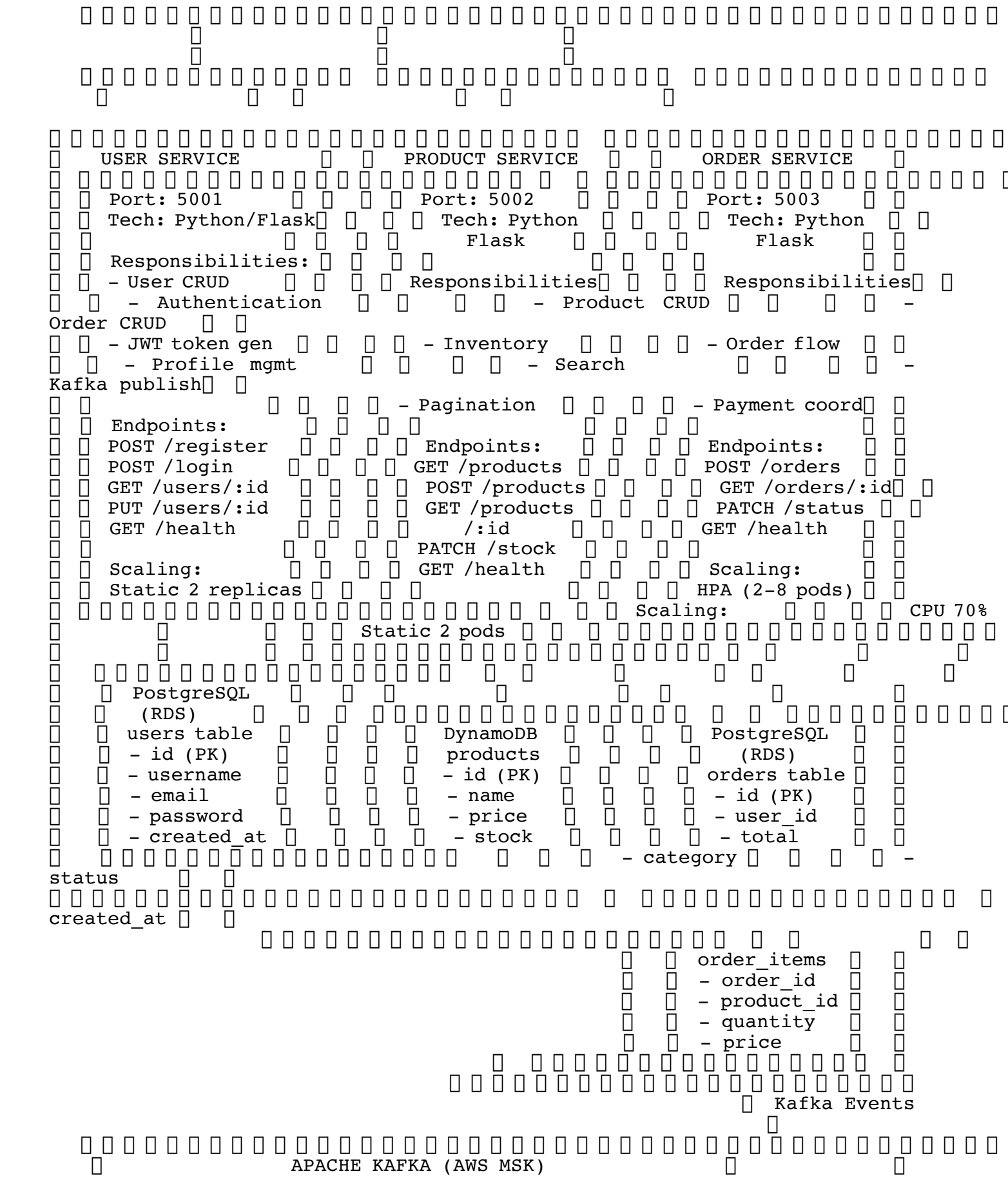
Microservices Architecture

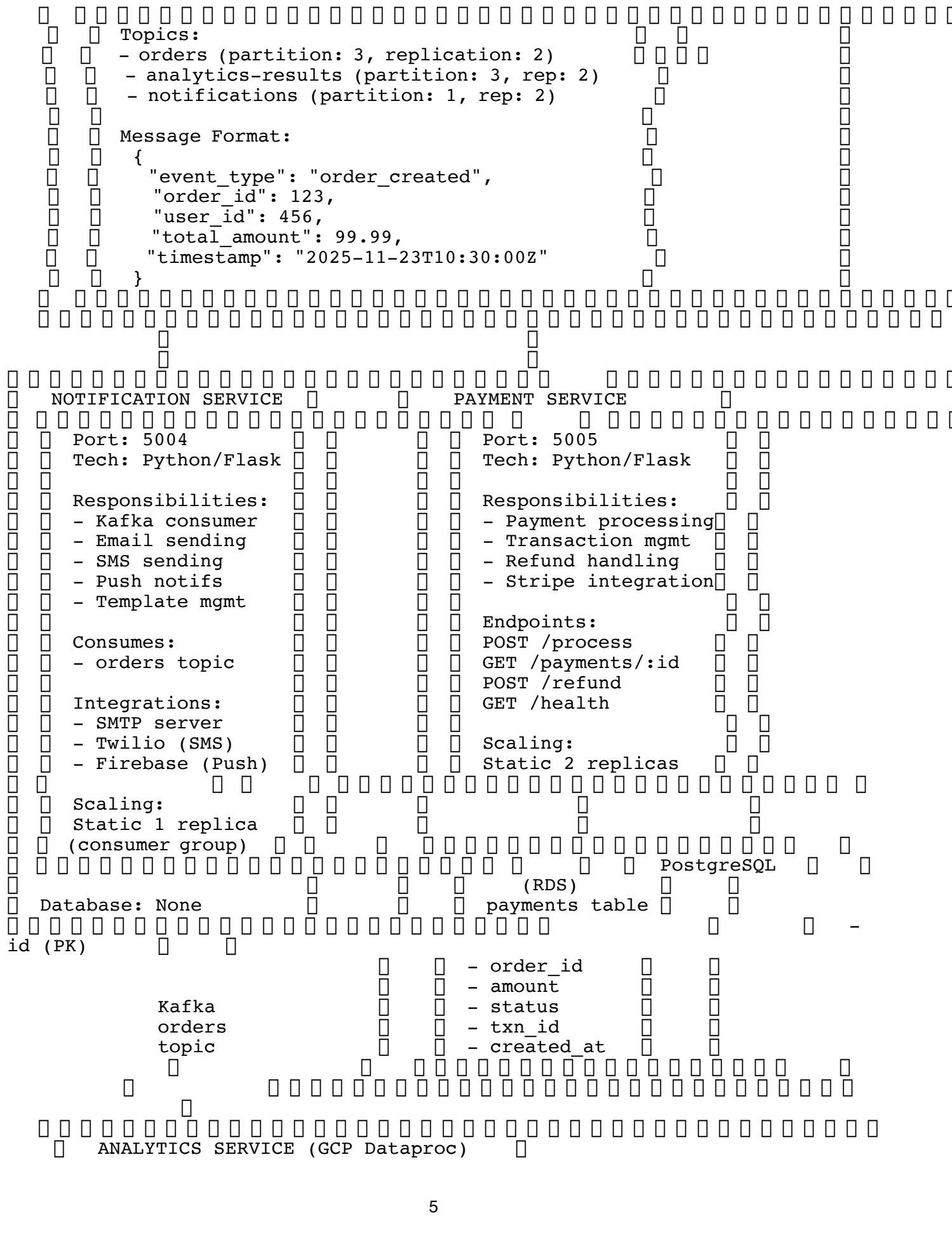
Architecture Principles

1. **Single Responsibility**: Each service owns a specific business capability
2. **Loose Coupling**: Services communicate via well-defined APIs
3. **High Cohesion**: Related functionality grouped within services
4. **Independent Deployment**: Services can be deployed independently
5. **Technology Diversity**: Different tech stacks per service needs
6. **Data Ownership**: Each service owns its database

Detailed Microservices Diagram




USER SERVICE
Port: 5001
Tech: Python/Flask
Flask
Responsibilities:
- User CRUD
- Authentication
- JWT token gen
- Profile mgmt
Endpoints:
POST /register
POST /login
GET /users/:id
PUT /users/:id
GET /health
Scaling:
Static 2 replicas
PRODUCT SERVICE
Port: 5002
Tech: Python
Flask
Responsibilities
- Product CRUD
- Inventory
- Search
- Pagination
Endpoints:
GET /products
POST /products
GET /products
/:id
PATCH /stock
GET /health
Scaling:
Static 2 pods
ORDER SERVICE
Port: 5003
Tech: Python
Flask
Responsibilities
- Order CRUD
- Order flow
- Kafka publish
- Payment coord
Endpoints:
POST /orders
GET /orders/:id
PATCH /status
GET /health
Scaling:
HPA (2-8 pods)
CPU 70%
PostgreSQL
(RDS)
users table
- id (PK)
- username
- email
- password
- created_at
DynamoDB
products
- id (PK)
- name
- price
- stock
- category
PostgreSQL
(RDS)
orders table
- id (PK)
- user_id
- total
- status
created_at
order_items
- order_id
- product_id
- quantity
- price
Kafka Events
APACHE KAFKA (AWS MSK)


Topics:
- orders (partition: 3, replication: 2)
- analytics-results (partition: 3, rep: 2)
- notifications (partition: 1, rep: 2)
Message Format:
{
"event_type": "order_created",
"order_id": 123,
"user_id": 456,
"total_amount": 99.99,
"timestamp": "2025-11-23T10:30:00Z"
}
NOTIFICATION SERVICE
Port: 5004
Tech: Python/Flask
Responsibilities:
- Kafka consumer
- Email sending
- SMS sending
- Push notifs
- Template mgmt
Consumes:
- orders topic
Integrations:
- SMTP server
- Twilio (SMS)
- Firebase (Push)
Scaling:
Static 1 replica
(consumer group)
PAYMENT SERVICE
Port: 5005
Tech: Python/Flask
Responsibilities:
- Payment processing
- Transaction mgmt
- Refund handling
- Stripe integration
Endpoints:
POST /process
GET /payments/:id
POST /refund
GET /health
Scaling:
Static 2 replicas
Database: None
PostgreSQL
(RDS)
payments table
id (PK)
- order_id
- amount
- status
- txn_id
- created_at
Kafka
orders
topic
ANALYTICS SERVICE (GCP Dataproc)

Service Inventory & Detailed Responsibilities

1. API Gateway Service

- **Port**: 5000
- **Technology**: Python 3.11, Flask 3.0, Gunicorn
- **Storage**: None (stateless)
- **Primary Functions**:
  - Route HTTP requests to appropriate microservices
  - Load distribution across service instances
  - Request/response transformation
  - Error handling and standardization
  - Health check aggregation
- **Communication Pattern**:
  - Inbound: HTTP REST from Internet via AWS ALB
  - Outbound: HTTP REST to all internal services
- **Scaling Strategy**:
  - HPA based on CPU (70%) and memory (80%)
  - Min: 2 replicas, Max: 10 replicas
  - Scale up: 100% increase every 15s or 2 pods every 15s (whichever is higher)
  - Scale down: 50% decrease every 60s with 300s stabilization window

- **Resource Allocation**:
  - Requests: 128Mi memory, 100m CPU
  - Limits: 256Mi memory, 200m CPU

2. User Service

- **Port**: 5001
- **Technology**: Python 3.11, Flask 3.0, SQLAlchemy 2.0, JWT
- **Storage**: PostgreSQL 15 (RDS Multi-AZ)
- **Primary Functions**:
  - User registration with password hashing (bcrypt)
  - Authentication and JWT token generation
  - User profile CRUD operations
  - Password reset functionality
  - Session management
- **Database Schema**:

```sql
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  username VARCHAR(80) UNIQUE NOT NULL,
  email VARCHAR(120) UNIQUE NOT NULL,
  password_hash VARCHAR(255) NOT NULL,
  full_name VARCHAR(120),
  phone VARCHAR(20),
  address TEXT,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

- **API Endpoints**:
  - `POST /api/users/register` - Create new user
  - `POST /api/users/login` - Authenticate and get JWT
  - `GET /api/users/:id` - Get user profile
  - `PUT /api/users/:id` - Update user profile
  - `DELETE /api/users/:id` - Soft delete user
  - `GET /health` - Health check
- **Scaling Strategy**: Static 2 replicas (low variance in load)
- **Resource Allocation**:
  - Requests: 128Mi memory, 100m CPU
  - Limits: 256Mi memory, 200m CPU

3. Product Service

- **Port**: 5002
- **Technology**: Python 3.11, Flask 3.0, Boto3 (DynamoDB SDK)
- **Storage**: DynamoDB (On-demand billing)
- **Primary Functions**:
  - Product catalog management
  - Inventory tracking
  - Product search and filtering
  - Category management
  - Stock level monitoring

- **Database Schema** (DynamoDB):

```
Table: products
Partition Key: id (Number)
Attributes:
  - name (String)
  - description (String)
  - price (Number)
  - category (String)
  - stock_quantity (Number)
  - image_url (String)
  - sku (String)
  - created_at (String - ISO 8601)
  - updated_at (String - ISO 8601)

Global Secondary Indexes:
  - category-index (category as partition key)
  - sku-index (sku as partition key)
```

- **API Endpoints**:
  - `GET /api/products` - List products (paginated)
  - `GET /api/products/:id` - Get product details
  - `POST /api/products` - Create product (admin)
  - `PUT /api/products/:id` - Update product (admin)
  - `DELETE /api/products/:id` - Delete product (admin)
  - `PATCH /api/products/:id/stock` - Update stock
  - `GET /health` - Health check
- **Scaling Strategy**: Static 2 replicas
- **Resource Allocation**:
  - Requests: 256Mi memory, 250m CPU
  - Limits: 512Mi memory, 500m CPU

4. Order Service

- **Port**: 5003
- **Technology**: Python 3.11, Flask 3.0, SQLAlchemy 2.0, Kafka-Python
- **Storage**: PostgreSQL 15 (RDS Multi-AZ)
- **Primary Functions**:
  - Order creation and validation
  - Order status management workflow
  - Integration with payment service
  - Integration with product service (stock check)
  - Publish order events to Kafka
  - Order history tracking
- **Database Schema**:

```sql
CREATE TABLE orders (
  id SERIAL PRIMARY KEY,
  user_id INTEGER NOT NULL,
  total_amount DECIMAL(10, 2) NOT NULL,
  status VARCHAR(50) DEFAULT 'pending',
  shipping_address TEXT,
  payment_id VARCHAR(100),
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
```

```
  updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE order_items (
  id SERIAL PRIMARY KEY,
  order_id INTEGER REFERENCES orders(id),
  product_id INTEGER NOT NULL,
  product_name VARCHAR(200) NOT NULL,
  quantity INTEGER NOT NULL,
  price DECIMAL(10, 2) NOT NULL
);
```

- **Order Status Flow**:
  1. `pending` → Order created, awaiting payment
  2. `processing` → Payment confirmed
  3. `shipped` → Order dispatched
  4. `delivered` → Order completed
  5. `cancelled` → Order cancelled
- **API Endpoints**:
  - `POST /api/orders` - Create order
  - `GET /api/orders/:id` - Get order details
  - `GET /api/orders/user/:userId` - Get user orders
  - `PATCH /api/orders/:id/status` - Update status
  - `DELETE /api/orders/:id` - Cancel order
  - `GET /health` - Health check
- **Kafka Integration**:
  - Produces to `orders` topic on order creation
  - Event payload includes order details for downstream processing
- **Scaling Strategy**:
  - HPA based on CPU (70%) and memory (80%)
  - Min: 2 replicas, Max: 8 replicas
  - Scale up: 100% increase every 15s or 2 pods every 15s
  - Scale down: 50% decrease every 60s
- **Resource Allocation**:
  - Requests: 128Mi memory, 100m CPU
  - Limits: 256Mi memory, 200m CPU

5. Payment Service

- **Port**: 5005
- **Technology**: Python 3.11, Flask 3.0, SQLAlchemy 2.0, Stripe SDK
- **Storage**: PostgreSQL 15 (RDS Multi-AZ)
- **Primary Functions**:
  - Payment processing integration
  - Transaction management
  - Refund handling
  - Payment method management
  - PCI compliance (via Stripe)
- **Database Schema**:

```sql
CREATE TABLE payments (
  id SERIAL PRIMARY KEY,
  order_id INTEGER UNIQUE NOT NULL,
  amount DECIMAL(10, 2) NOT NULL,
  payment_method VARCHAR(50),
  transaction_id VARCHAR(100) UNIQUE,
  status VARCHAR(50) DEFAULT 'pending',
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

- **API Endpoints**:
  - `POST /api/payments/process` - Process payment
  - `GET /api/payments/:id` - Get payment details
  - `POST /api/payments/refund` - Issue refund
  - `GET /api/payments/order/:orderId` - Get payment by order
  - `GET /health` - Health check
- **Scaling Strategy**: Static 2 replicas
- **Resource Allocation**:
  - Requests: 128Mi memory, 100m CPU
  - Limits: 256Mi memory, 200m CPU

6. Notification Service

- **Port**: 5004
- **Technology**: Python 3.11, Flask 3.0, Kafka-Python, SMTP, Twilio
- **Storage**: None (stateless)
- **Primary Functions**:
  - Consume order events from Kafka
  - Send email notifications (SMTP/SendGrid)
  - Send SMS notifications (Twilio)
  - Send push notifications (Firebase)
  - Template management
  - Notification retry logic
- **Notification Types**:
  - Order confirmation
  - Order status updates
  - Payment confirmation
  - Shipping notifications
  - Promotional emails (future)
- **API Endpoints**:
  - `POST /api/notifications/send` - Manual notification
  - `GET /health` - Health check
- **Kafka Integration**:
  - Consumes from `orders` topic
  - Consumer group: `notification-service-group`
  - Auto-commit: false (manual offset management)
- **Scaling Strategy**: Static 1 replica (single consumer per partition)
- **Resource Allocation**:
  - Requests: 256Mi memory, 250m CPU

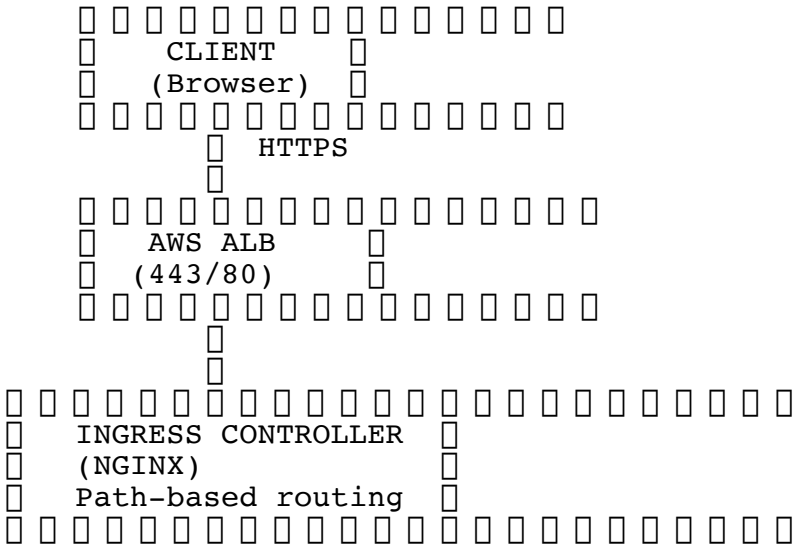
  - Limits: 512Mi memory, 500m CPU

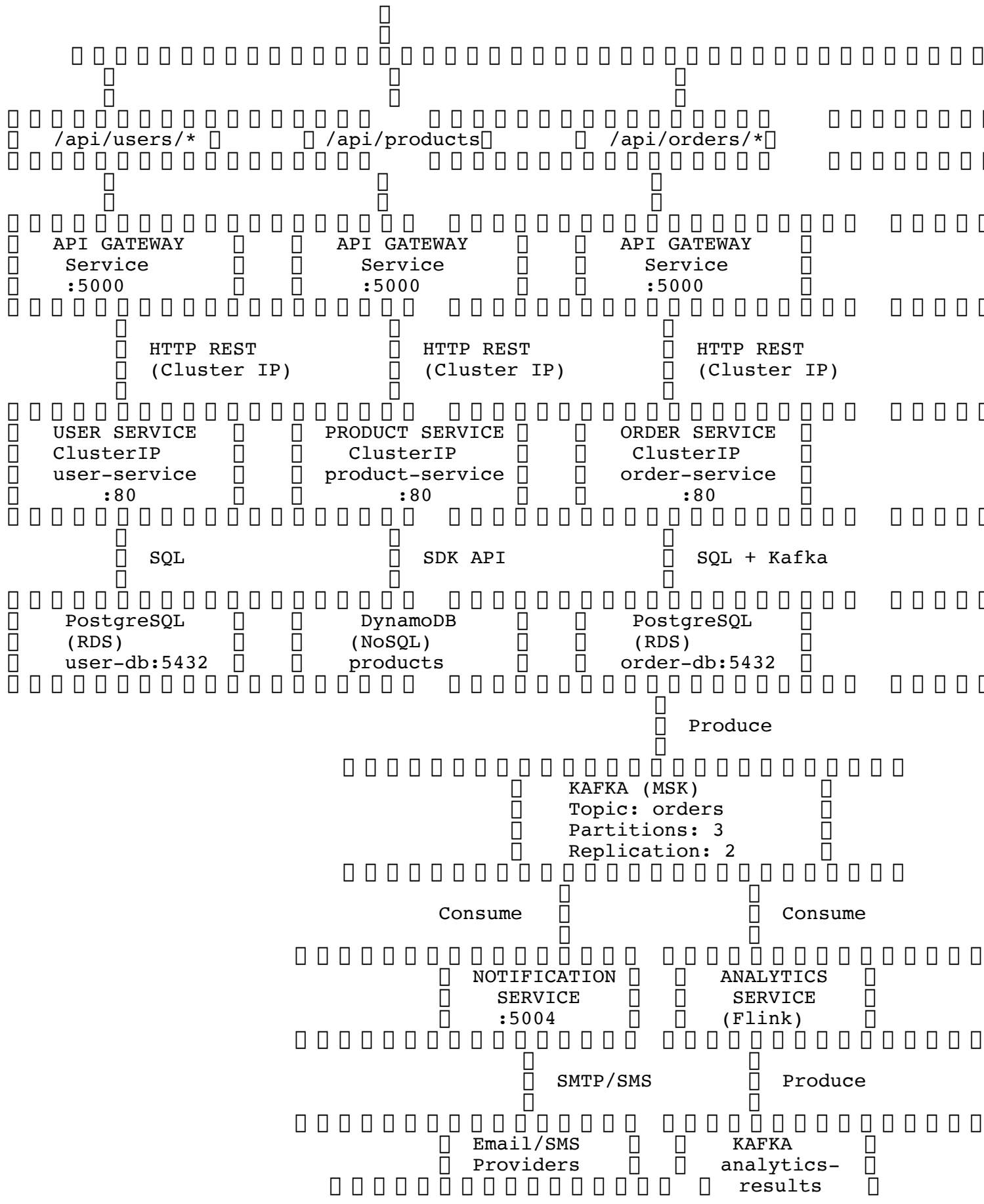
7. Analytics Service (Flink)

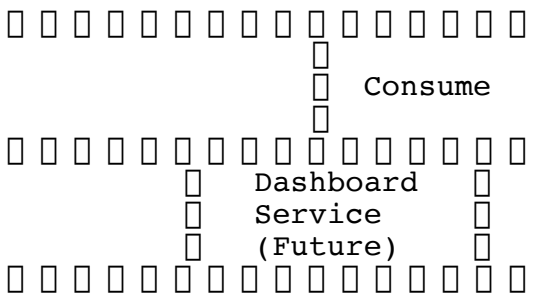
- **Technology**: Apache Flink 1.17, Java 11, Kafka Connector
- **Infrastructure**: GCP Dataproc cluster
- **Primary Functions**:
  - Real-time stream processing
  - Windowed aggregations (tumbling, sliding)
  - Event time processing
  - Stateful computations
  - Anomaly detection
- **Processing Pipeline**:
  1. **Source**: Kafka consumer (orders topic)
  2. **Transform**: Parse JSON, extract fields
  3. **Window**: 5-minute tumbling windows
  4. **Aggregate**: Count, sum, average operations
  5. **Sink**: Kafka producer (analytics-results topic) + GCS
- **Metrics Computed**:
  - Total orders per window
  - Revenue per window
  - Average order value
  - Top-selling products
  - Orders by region
  - Peak hour analysis
- **Cluster Configuration**:
  - Master: n1-standard-2 (2 vCPU, 7.5GB RAM)
  - Workers: 2x n1-standard-2
  - Parallelism: 4
  - Checkpointing: Every 60 seconds to GCS
- **Resource Allocation**:
  - Job Manager: 2GB heap
  - Task Manager: 4GB heap
  - Network buffers: 512MB

Service Interconnections & Communication Mechanisms

Detailed Communication Flow Diagram




/api/users/*
/api/products
/api/orders/*
API GATEWAY Service :5000
API GATEWAY Service :5000
API GATEWAY Service :5000
HTTP REST (Cluster IP)
HTTP REST (Cluster IP)
HTTP REST (Cluster IP)
USER SERVICE ClusterIP user-service :80
PRODUCT SERVICE ClusterIP product-service :80
ORDER SERVICE ClusterIP order-service :80
SQL
SDK API
SQL + Kafka
PostgreSQL (RDS) user-db:5432
DynamoDB (NoSQL) products
PostgreSQL (RDS) order-db:5432
Produce
KAFKA (MSK)
Topic: orders
Partitions: 3
Replication: 2
Consume
Consume
NOTIFICATION SERVICE :5004
ANALYTICS SERVICE (Flink)
SMTP/SMS
Produce
Email/SMS Providers
KAFKA analytics-results



Inter-Service Authentication Flow:

1. User Login:
   Client → API Gateway → User Service
   POST /api/users/login {username, password}
   ← JWT Token (expires in 1 hour)

2. Authenticated Request:
   Client → API Gateway → Order Service
   Headers: {Authorization: Bearer <JWT>}
   Order Service validates JWT
   ← Response with order data

3. Service-to-Service:
   Order Service → Product Service
   Headers: {Authorization: Bearer <SERVICE_TOKEN>}
   (Service tokens from Kubernetes secrets)

Communication Patterns in Detail

1. Synchronous Communication (HTTP REST) **Pattern**: Request-Response over HTTP/HTTPS

Use Cases: - Client → API Gateway: All user-initiated actions - API Gateway → Microservices: Request routing - Order Service → Product Service: Stock validation - Order Service → Payment Service: Payment processing - Order Service → User Service: User validation

Service Discovery: - Uses Kubernetes DNS: `http://product-service.default.svc.cluster.local:80` - Short form: `http://product-service:80` - Environment variables for service URLs

Benefits: - Simple to understand and debug - Immediate response - Strong consistency

Challenges: - Tight coupling - Cascading failures - Increased latency

Mitigation: - Circuit breakers (future: Istio) - Timeouts and retries - Health checks - Fallback responses

2. Asynchronous Communication (Apache Kafka) **Pattern**: Event-driven messaging

Use Cases: - Order Service → Notification Service: Order events - Order Service → Analytics Service: Business intelligence - Payment Service → Order Service: Payment status (future)

Kafka Configuration:

```
Cluster: MSK (3 brokers)
Topic: orders
  Partitions: 3
  Replication Factor: 2
  Retention: 7 days
  Compression: snappy
```

```
Topic: analytics-results
  Partitions: 3
  Replication Factor: 2
  Retention: 30 days
  Compression: gzip
```

Implementation: Order Service publishes events to Kafka with `acks='all'` for durability. Notification Service consumes with manual commits for guaranteed processing.

Benefits: - Loose coupling - Asynchronous processing - Event replay capability - Scalability - Fault tolerance

Challenges: - Eventual consistency - Message ordering - Duplicate handling

Mitigation: - Idempotent consumers - Message deduplication - Offset management - Dead letter queues

3. Database Access Patterns **Pattern**: Direct database connections

User Service → **PostgreSQL**: Uses SQLAlchemy ORM for database operations with session-based transactions

Product Service → **DynamoDB**: Uses Boto3 SDK for NoSQL operations (get_item, put_item, query with GSI)

Connection Pooling: - PostgreSQL: pgbouncer (session pooling) - Max connections per service: 20 - Connection timeout: 30 seconds

4. Caching Strategy (Redis) **Pattern**: Read-through cache with 1-hour TTL for product catalog, user sessions, and rate limiting

Service Resilience Patterns

Circuit Breaker (Planned) Uses pybreaker library with 5-failure threshold in 60-second window for service degradation

Retry with Exponential Backoff Implemented using tenacity library: 3 attempts with exponential wait (1s to 10s)

Timeout Configuration

- API Gateway → Services: 30 seconds
- Service → Service: 5 seconds
- Service → Database: 10 seconds
- Kafka producer: 10 seconds
- Health checks: 3 seconds

Data Consistency Patterns

Strong Consistency (ACID Transactions) User, Order, and Payment services use PostgreSQL transactions with rollback on failure

Eventual Consistency (Event-Driven)

- Order events to notifications
- Order events to analytics
- Payment status updates

Strategy: - Idempotent operations - Event versioning - Compensating transactions for failures

Security in Service Communication

Mutual TLS (Future Enhancement) Service mesh (Istio) planned for strict mTLS enforcement between all services

API Authentication

- **External**: JWT tokens from User Service
- **Internal**: Service accounts with Kubernetes secrets

Network Policies Kubernetes NetworkPolicy configured to restrict traffic: Order Service only accepts ingress from API Gateway, egress to Product Service

---

Design Rationale & Decision Log

Architectural Decisions

Decision 1: Why Microservices Over Monolith? **Context**: Need to build scalable e-commerce platform

Options Considered: 1. Monolithic architecture 2. Microservices architecture 3. Modular monolith

Decision: Microservices Architecture

Rationale: - **Independent Scaling**: Order service experiences 10x traffic during sales, while user service remains steady. Microservices allow scaling only what's needed - **Technology Flexibility**: Analytics needs Java/Flink for stream processing, while REST APIs work well in Python. Microservices enable polyglot architecture - **Fault Isolation**: Payment service failure shouldn't crash product browsing. Microservices provide bulkheads - **Team Autonomy**: Different teams can own different services with independent release cycles - **Deployment Flexibility**: Can deploy order service updates without touching user service

Trade-offs Accepted: - Increased operational complexity (mitigated by Kubernetes) - Network latency between services (acceptable for our SLAs) - Eventual consistency challenges (managed with event sourcing) - Higher infrastructure costs (offset by independent scaling efficiency)

Decision 2: Why Apache Kafka for Event Streaming? **Context**: Need asynchronous communication between services

Options Considered: 1. Apache Kafka 2. RabbitMQ 3. AWS SQS 4. Direct HTTP callbacks

Decision: Apache Kafka (AWS MSK)

Rationale: - **Durability**: Messages persist for 7 days, enabling event replay and debugging - **Scalability**: Handles millions of events per second with partitioning - **Decoupling**: Order service doesn't know about consumers (notification, analytics) - **Stream Processing**: Native integration with Apache Flink for analytics - **Managed Service**: MSK eliminates operational overhead of running Kafka - **Ordering Guarantees**: Per-partition ordering crucial for order processing - **Multiple Consumers**: Both notification and analytics consume same events

Trade-offs Accepted: - Higher cost than SQS ($300/month for MSK vs $50 for SQS) - Learning curve for team - Eventual consistency model - More complex than simple queues

Why Not Alternatives: - RabbitMQ: Less suited for high-throughput streaming - SQS: No event replay, limited retention, not ideal for stream processing - HTTP callbacks: Tight coupling, retry complexity, no persistence

---

Decision 3: Why Multi-Cloud (AWS + GCP)? **Context**: Need to deploy application infrastructure

Options Considered: 1. Single cloud (AWS only) 2. Single cloud (GCP only) 3. Multi-cloud (AWS + GCP) 4. On-premises + cloud hybrid

Decision: Multi-Cloud (AWS Primary + GCP for Analytics)

Rationale: - **Best-of-Breed**: AWS excels at application hosting, GCP excels at data processing - **Risk Mitigation**: Not locked into single vendor; disaster recovery across clouds - **Cost Optimization**: GCP Dataproc more cost-effective for Flink than AWS EMR - **Learning Opportunity**: Team gains multi-cloud expertise - **Workload Isolation**: Analytics workload separated from transactional load

Trade-offs Accepted: - Increased operational complexity (two cloud consoles) - Cross-cloud networking costs - Skill requirement for two platforms - Inconsistent tooling and APIs

Implementation Strategy: - Keep compute and data gravity in AWS - Use GCP only for analytics workload - Kafka as integration point between clouds - Minimize cross-cloud data transfer

---

Decision 4: Why Kubernetes (EKS) Over ECS or Lambda? **Context**: Need container orchestration platform

Options Considered: 1. Amazon EKS (Kubernetes) 2. Amazon ECS (proprietary) 3. AWS Lambda (serverless) 4. EC2 instances directly

Decision: Amazon EKS (Kubernetes)

Rationale: - **Industry Standard**: Kubernetes is cloud-agnostic, transferable skills - **Rich Ecosystem**: Helm charts, operators, service meshes, monitoring tools - **Portability**: Can migrate to GKE or on-premises if needed - **Advanced Features**: HPA, network policies, custom schedulers, StatefulSets - **Community Support**: Massive community, extensive documentation - **GitOps Ready**: Native integration with ArgoCD for declarative deployments

Trade-offs Accepted: - Steeper learning curve than ECS - Control plane costs ($0.10/hour = $73/month) - More complex than serverless - Requires more operational expertise

Why Not Alternatives: - ECS: Vendor lock-in to AWS, less feature-rich, smaller community - Lambda: Cold starts, 15-minute timeout limit, not suitable for long-running processes - EC2: Manual scaling, no self-healing, higher operational burden

---

Decision 5: Why PostgreSQL (RDS) + DynamoDB? **Context**: Need database for different data patterns

Options Considered: 1. PostgreSQL for everything 2. DynamoDB for everything 3. MongoDB 4. PostgreSQL + DynamoDB (chosen)

Decision: PostgreSQL (RDS) for transactional + DynamoDB for catalog

Rationale:

PostgreSQL for Users/Orders/Payments: - **ACID Transactions**: Critical for financial data integrity - **Complex Queries**: JOIN operations for order items, user orders - **Relational Data**: Natural fit for user-order-payment relationships - **Referential Integrity**: Foreign keys ensure data consistency - **Mature Ecosystem**: ORMs, migration tools, backup solutions

DynamoDB for Products: - **High Throughput**: Product catalog read-heavy (90% reads) - **Flexible Schema**: Product attributes vary by category - **Predictable Performance**: Single-digit millisecond latency at scale - **Auto-Scaling**: Pay-per-request pricing scales with traffic - **No Maintenance**: Fully managed, no patches or upgrades

Trade-offs Accepted: - Two databases to manage instead of one - Different query patterns and SDKs - Cross-database joins not possible (mitigated by API composition)

Decision 6: Why GitOps with ArgoCD? **Context**: Need CI/CD deployment strategy

Options Considered: 1. kubectl apply in CI/CD pipeline 2. Helm direct installs 3. ArgoCD (GitOps) 4. Flux CD (GitOps alternative)

Decision: ArgoCD for GitOps

Rationale: - **Declarative**: Git is single source of truth for cluster state - **Auditability**: Every change tracked in Git history - **Easy Rollback**: `git revert` to previous working state - **Security**: No kubectl credentials in CI/CD; ArgoCD pulls changes - **Drift Detection**: Auto-sync corrects manual kubectl changes - **Multi-Cluster**: Can manage multiple Kubernetes clusters from one place - **Visualization**: UI shows deployment status and sync health

Trade-offs Accepted: - Additional component to run in cluster - Git becomes critical path (mitigated by multiple repos) - Learning curve for GitOps concepts

Why Not Alternatives: - kubectl in CI/CD: Requires credentials in pipeline, no drift detection - Helm direct: No automated sync, manual updates - Flux CD: ArgoCD has better UI and multi-tenancy support

---

Decision 7: Why Python Flask for Microservices? **Context**: Choose technology stack for REST APIs

Options Considered: 1. Python with Flask 2. Node.js with Express 3. Java with Spring Boot 4. Go with Gin

Decision: Python with Flask (except Analytics)

Rationale: - **Team Expertise**: Team already proficient in Python - **Fast Development**: Flask is lightweight and productive - **Rich Ecosystem**: SQLAlchemy, Kafka clients, AWS SDKs - **Readability**: Python code is maintainable and self-documenting - **Data Science Integration**: Easy to add ML features later - **Docker Support**: Official slim Python images are lightweight

Trade-offs Accepted: - Lower raw performance than Go or Java - GIL limitations for CPU-bound tasks (not an issue for I/O-bound APIs) - Async support less mature than Node.js

Why Analytics in Java?: - Apache Flink is Java-native - Better performance for stream processing - Mature Flink SDK in Java

---

Decision 8: Why Horizontal Pod Autoscaling (HPA)? **Context**: Need to scale applications based on load

Options Considered: 1. Static replica count 2. Horizontal Pod Autoscaler (HPA) 3. Vertical Pod Autoscaler (VPA) 4. Manual scaling

Decision: HPA for API Gateway and Order Service, Static for others

Rationale:

HPA for API Gateway & Order Service: - **Variable Load**: Traffic varies 10x between off-peak and sales events - **Cost Optimization**: Scale down during low traffic to save money - **Automatic**: No manual intervention during traffic spikes - **Reliable**: Kubernetes manages scaling decisions based on metrics

Static Replicas for User/Product/Payment/Notification: - **Predictable Load**: User CRUD operations are steady - **Simplicity**: No need for scaling complexity - **Faster Startup**: Always have capacity ready - **Cost**: Minimal cost difference (2-3 pods)

Configuration: - API Gateway: 2-10 replicas at 70% CPU - Order Service: 2-8 replicas at 70% CPU - Scale up fast (100% increase every 30s) - Scale down slow (50% decrease every 5min)

Trade-offs Accepted: - Metric collection overhead (Prometheus) - Potential overprovisioning during scale-up - Complexity in tuning thresholds

---

Decision 9: Why Prometheus + Grafana for Observability? **Context**: Need monitoring and alerting

Options Considered: 1. CloudWatch only 2. Prometheus + Grafana 3. Datadog (commercial) 4. New Relic (commercial)

Decision: Prometheus + Grafana + CloudWatch

Rationale: - **Open Source**: No per-host licensing costs - **Kubernetes Native**: Service discovery, pod metrics out-of-box - **Flexible Querying**: PromQL for custom queries and dashboards - **Rich Ecosystem**: Exporters for databases,

Kafka, etc. - **Alertmanager**: Flexible alerting with Slack/PagerDuty integration - **Grafana**: Beautiful dashboards, template variables, annotations

CloudWatch for: - AWS-specific metrics (EKS, RDS, MSK) - Log aggregation - Billing alerts

Trade-offs Accepted: - Need to run Prometheus in cluster - Data retention limits (30 days default) - Learning curve for PromQL

Why Not Commercial: - Datadog: $15/host/month = $500+/month for our scale - New Relic: Similar pricing - Budget-conscious for learning project

---

Infrastructure Design Rationale

Multi-AZ Deployment **Decision**: Deploy across 2 Availability Zones

Rationale: - **High Availability**: Survive single AZ failure - **Load Distribution**: Traffic balanced across zones - **Database Resilience**: RDS Multi-AZ automatic failover

Cost Impact: - 2x data transfer costs between AZs - Acceptable for production-grade availability

---

VPC Design (10.0.0.0/16) **Decision**: Public subnets for EKS, private subnets for databases

Rationale: - **Security**: Databases not publicly accessible - **Compliance**: Industry best practice - **NAT Gateway**: Future for egress from private subnets

Subnet Allocation: - 10.0.0.0/24 - Public Subnet AZ1 (256 IPs) - 10.0.1.0/24 - Public Subnet AZ2 (256 IPs) - 10.0.10.0/24 - Private Subnet AZ1 (256 IPs) - 10.0.11.0/24 - Private Subnet AZ2 (256 IPs)

Resource Sizing **EKS Node Group**: - Instance Type: t3.medium (2 vCPU, 4GB RAM) - Min: 2 nodes, Max: 4 nodes - **Rationale**: - t3 burstable for variable workloads - medium size balances cost and capacity - Can run 8-12 pods per node (256Mi pods)

RDS Instance: - Instance Type: db.t3.micro (2 vCPU, 1GB RAM) - Storage: 20GB gp3 - **Rationale**: - Free tier eligible - Sufficient for development/demo - Can scale vertically if needed

---

Security Groups **EKS Node Security Group**: - Ingress: Port 443 from ALB - Egress: All (for pulling images, calling AWS APIs)

RDS Security Group: - Ingress: Port 5432 from EKS nodes only - No public internet access

MSK Security Group: - Ingress: Port 9092 from EKS nodes - Cross-cloud: Port 9092 from GCP IP range

---

Future Enhancements & Trade-offs

What We Would Add in Production

1. **Service Mesh (Istio)**:
   - Mutual TLS between services
   - Advanced traffic management (canary, blue-green)
   - Distributed tracing
   - **Why not now**: Complexity for learning project
2. **API Gateway Layer (Kong/Ambassador)**:
   - Rate limiting per user
   - API key management

- Request transformation
- **Why not now**: Kubernetes Ingress sufficient for demo

3. **Caching Layer (Redis)**:
   - Product catalog caching
   - Session storage
   - Rate limit counters
   - **Why not now**: Database performance adequate currently
4. **CDN (CloudFront)**:
   - Static asset delivery
   - Global edge caching
   - DDoS protection
   - **Why not now**: No static assets currently
5. **Secrets Management (Vault/Secrets Manager)**:
   - Dynamic secrets rotation
   - Centralized secret storage
   - Audit logging
   - **Why not now**: Kubernetes Secrets sufficient for demo
6. **Service Mesh Observability**:
   - Distributed tracing (Jaeger)
   - Service topology visualization
   - Request flow analysis
   - **Why not now**: Prometheus metrics sufficient
7. **Multi-Region Deployment**:
   - Active-active across regions
   - Global load balancing
   - Data replication
   - **Why not now**: Regional deployment sufficient, cost prohibitive

---

Cost-Benefit Analysis

Current Monthly Costs (estimated): - EKS Control Plane: $73 - EC2 Instances (2x t3.medium): $60 - RDS (db.t3.micro): $15 - MSK (kafka.t3.small): $300 - DynamoDB: $10 (on-demand) - Data Transfer: $20 - **Total**: ~$478/month

Cost Optimization Strategies: 1. Reserved Instances: Save 30-40% on EC2/RDS 2. Spot Instances: Use for dev/test environments 3. Auto-scaling: Scale down during off-hours 4. Data lifecycle: Move old data to S3 Glacier

Cost vs. Alternative Architectures: - Serverless (Lambda + API Gateway): ~$200/month but cold starts - Monolith on single EC2: ~$50/month but no scaling - Managed containers (ECS Fargate): ~$350/month but less features

Value Delivered: - Production-grade architecture - Skills development in modern DevOps - Portfolio showcase - Real-world experience with cloud-native patterns

---

Infrastructure as Code

Terraform Modules

1. **VPC Module**: Network isolation, public/private subnets
2. **EKS Module**: Kubernetes cluster with node groups
3. **RDS Module**: MySQL database with security groups
4. **MSK Module**: Kafka cluster configuration
5. **Lambda Module**: S3-triggered function
6. **GCP Module**: Dataproc cluster and Cloud Storage

Benefits

- **Reproducibility**: Spin up identical environments
- **Version control**: Infrastructure changes tracked
- **Collaboration**: Team can review infrastructure changes
- **Automation**: CI/CD can apply infrastructure changes

Scalability & Resilience

Horizontal Pod Autoscaling (HPA)

- **API Gateway**: 2-10 pods based on CPU (70%)
- **Order Service**: 2-8 pods based on CPU (70%)
- **Triggers**: CPU/memory utilization

Database Scaling

- **RDS**: Vertical scaling (instance size)
- **DynamoDB**: Auto-scaling (pay-per-request)

High Availability

- **Multi-AZ**: RDS and EKS nodes across 2 AZs
- **Load Balancing**: AWS ELB for API Gateway
- **Health Checks**: Kubernetes liveness/readiness probes

Observability

Metrics (Prometheus + Grafana)

- Request rate, latency, error rate per service
- Kubernetes cluster health
- Database connections
- Kafka lag

Logging (EFK Stack)

- Centralized log aggregation
- Application logs from all pods
- Searchable via Kibana

Tracing (Future)

- Distributed tracing with OpenTelemetry
- End-to-end request tracking

Security Considerations

1. **Network Isolation**: Private subnets for databases
2. **IAM Roles**: Service accounts with least privilege
3. **Secrets Management**: Kubernetes secrets for credentials
4. **TLS**: HTTPS for public endpoints
5. **Security Groups**: Firewall rules limiting access

Cost Optimization

Free Tier Usage

- **RDS**: db.t3.micro (free tier eligible)
- **EKS**: Control plane free first 30 days
- **Lambda**: 1M requests free
- **S3/DynamoDB**: Generous free tiers

Cost Management

- **Auto-scaling**: Pay only for what you use
- **Spot instances**: (Future) For non-critical workloads
- **Monitoring**: CloudWatch to track spending

Estimated Monthly Cost: $50-100 (mostly MSK Kafka)

Future Enhancements

1. **Service Mesh** (Istio): Traffic management, security
2. **API Gateway** (Kong): Rate limiting, authentication
3. **Caching** (Redis): Reduce database load
4. **CDN** (CloudFront): Static asset delivery
5. **Multi-region**: Global deployment

Conclusion

This architecture demonstrates: - Modern cloud-native patterns - Multi-cloud deployment - Event-driven design - GitOps workflows - Comprehensive observability - Production-ready practices

All while remaining cost-effective and manageable.