System Architecture & Infrastructure

Production infrastructure specifications and scalability patterns

Version: 1.0

Author: Platform Engineering Team

Category: System Architecture | Infrastructure

Last Updated: December 12, 2025

This document provides comprehensive technical specifications for Monzo's production infrastructure, detailing the microservices architecture, data storage systems, networking infrastructure, and scalability patterns that power 9+ million customer accounts processing billions of transactions annually. Understanding this architecture is essential for developers integrating with Monzo APIs and operations teams managing infrastructure.

High-Level Architecture Overview

Monzo operates a cloud-native microservices architecture deployed on AWS and Kubernetes, designed for horizontal scalability, fault tolerance, and rapid feature deployment.

Architecture Principles

Microservices-first: 2,800+ independent Go microservices, each owning specific business domains

Event-driven: Asynchronous communication via NSQ message queue (3+ billion events/day)

API gateway pattern: Single entry point with routing, authentication, rate limiting

Database per service: Each microservice owns its data store (Cassandra or PostgreSQL)

Zero-trust networking: Every service-to-service call authenticated and encrypted

Immutable infrastructure: Container-based deployments, no server SSH access

Scale Metrics

Metric	Value	Notes
Active Customers	9+ million	UK retail and business accounts
Microservices	2,800+	Written in Go, deployed on Kubernetes
API Requests	2 million/sec	Peak read throughput
Transactions/Day	10+ million	Card payments, transfers, direct debits

Events/Day	3+ billion	NSQ message queue throughput
Data Storage	19 petabytes	BigQuery data warehouse
Database Writes	100,000/sec	Cassandra write capacity
Database Reads	2,000,000/sec	Cassandra read capacity
Infrastructure Cost	~£100M/year	AWS + third-party services (estimated)

Compute Infrastructure

Kubernetes Orchestration

All services run on Amazon EKS (Elastic Kubernetes Service) across multiple availability zones:

Cluster Configuration: Multiple production clusters segmented by criticality (core banking, customer-facing, analytics)

Node Types: Mix of on-demand and spot instances for cost optimization (c5.xlarge, c5.2xlarge, r5.xlarge for memory-intensive workloads)

Scaling: Cluster Autoscaler dynamically adjusts node count; Horizontal Pod Autoscaler scales individual services based on CPU/memory/custom metrics

Namespaces: Logical isolation per team/service group with resource quotas and network policies

Microservice Standards

All microservices follow standardized patterns:

```
# Example Kubernetes deployment manifest apiVersion: apps/v1 kind: Deployment metadata:
name: transaction-processor namespace: payments spec: replicas: 10 selector: matchLabels:
app: transaction-processor template: metadata: labels: app: transaction-processor version:
v2.4.1 spec: containers: - name: transaction-processor image:
monzo/transaction-processor:v2.4.1 ports: - containerPort: 8080 resources: requests:
memory: "256Mi" cpu: "250m" limits: memory: "512Mi" cpu: "500m" livenessProbe: httpGet:
path: /health port: 8080 initialDelaySeconds: 30 periodSeconds: 10 readinessProbe: httpGet:
path: /ready port: 8080 initialDelaySeconds: 5 periodSeconds: 5
```

Data Storage Architecture

Apache Cassandra (Primary Operational Database)

Mission-critical transactional data stored in Cassandra for high write throughput and linear scalability:

Use Cases: Account balances, transaction records, customer profiles, audit logs

Deployment: Multi-datacenter clusters across 3 AWS availability zones

Replication: RF=3 (replication factor 3) with LOCAL_QUORUM consistency

Node Count: 100+ nodes in production clusters

Performance: 100K writes/sec, 2M reads/sec, P99 latency <10ms

Data Model: Optimized for query patterns (denormalization, materialized views)

PostgreSQL (Relational Data)

Used for services requiring ACID transactions and complex queries:

Use Cases: User management, product configuration, regulatory reporting data

Deployment: Amazon RDS for PostgreSQL with Multi-AZ failover

Versions: PostgreSQL 14.x with pgvector extension for ML embeddings

Backup: Automated daily snapshots + 5-minute point-in-time recovery

Google BigQuery (Analytics Data Warehouse)

Scale: 19 petabytes of historical data

Ingestion: Real-time streaming from Cassandra via change data capture (CDC)

dbt Models: 4,700+ data transformation models

Query Performance: Petabyte-scale queries in seconds

Use Cases: ML model training, financial reporting, fraud analytics, customer insights

Messaging & Event-Driven Architecture

NSQ Message Queue

Monzo uses NSQ (distributed message queue) for asynchronous service communication:

Throughput: 3+ billion messages per day

Topics: 1,000+ distinct event topics (e.g., transaction.created, account.updated)

Delivery Guarantee: At-least-once delivery (consumers must be idempotent)

Ordering: No global ordering; use partition keys for ordered processing within groups

Event Schema Example


```
{ "event_type": "transaction.created", "event_id": "evt_00008zIcpb1TB4yeIFXMzx",
"timestamp": "2024-12-12T10:30:45.123Z", "version": "v2", "data": { "transaction_id":
"tx_00008zIcpb1TB4yeIFXMzx", "account_id": "acc_00009237aqC8c79WTZ", "amount": -350,
"currency": "GBP", "merchant": { "id": "merch_00008zL2INM3xZ47zCx5j9", "name": "Starbucks"
}, "status": "settled", "created_at": "2024-12-12T10:30:45.000Z" } }
```

Consumer Pattern: Services subscribe to relevant topics and process events asynchronously. Example: fraud detection service subscribes to transaction.created, ML risk scoring service subscribes to account.updated.

Network Architecture & Security

Zero-Trust Network Model

Monzo implements zero-trust networking where every connection requires authentication:

Service Mesh: Istio for service-to-service communication with mTLS

Network Policies: Kubernetes NetworkPolicy objects restrict traffic to allowed paths only

Certificate Management: Automated cert rotation via cert-manager

Identity: SPIFFE/SPIRE for workload identity

API Gateway Layer

All external traffic routes through API gateway:

Function	Implementation	Purpose
Rate Limiting	Token bucket algorithm	Prevent abuse, enforce quotas
Authentication	OAuth 2.0 Bearer tokens	Verify client identity
Request Routing	Path-based routing	Direct to appropriate microservice
TLS Termination	AWS ALB + ACM certs	HTTPS encryption
DDoS Protection	AWS Shield + CloudFront	Mitigate volumetric attacks
Request Validation	JSON Schema validation	Reject malformed requests
Response Caching	Redis cache layer	Reduce backend load