Please Visit My Personal Page

https://myprofile.4.220.41.138.nip.io

# eShopMicroServices — High Level Architecture & Service Responsibilities

Summary

- eShopMicroServices is a containerized **.NET** & **Java** microservices suite composed of four primary backend services (Basket, Discount, Order, Catalog).

- RAG + ChatGPT Integration: Leveraging **Python**, **Azure OpenAI**, and **Azure Cognitive Search**, the platform supports natural-language queries over product data and PDF manuals. Users can ask about features, specifications, pricing and user manual answers using semantic vector search and hybrid search.

- Deployed on **AKS** using **Flux** **GitOps** and Azure infrastructure provisioned via **Terraform** (CICD-Templates + terraform-modules).

- CI builds images, pushes to **ACR**, and updates the GitOps repo; Flux reconciles cluster state.

## 🔗 Repository Links

Repository	Purpose
[eShopMicroservices](https://github.com/RijoyP/eShopMicroservices)	Microservices source code
[GitOps](https://github.com/RijoyP/GitOps)	Kubernetes manifests and FluxCD configs
[terraform-modules](https://github.com/RijoyP/terraform-modules)	Infrastructure as Code modules
[CICD-Templates](https://github.com/RijoyP/CICD-Templates)	Reusable pipeline templates
[helm-templates](https://github.com/RijoyP/helm-templates)	Helm chart boilerplate

## 🔗 Infra Monitoring / Logging / Tracing / Messaging Urls Links

Resource	Url
Jaeger	http://jaeger.20.251.183.221.nip.io/
Grafana	http://grafana.20.251.183.221.nip.io/
Kibana	http://kibana.20.251.183.221.nip.io/
Prometheus	http://prometheus.20.251.183.221.nip.io/
Zipkin	http://zipkin.20.251.183.221.nip.io/
Rabbit MQ	http://rabbitmq.20.251.183.221.nip.io/

## 🔗 Backend Urls Links

Backend	Urls
Front End React (React Typescript)	https://eshopreact.4.220.41.138.nip.io/
Catatlog API (ASP.NET Core 8.0)	https://catalog.4.220.41.138.nip.io/swagger/index.html
Discount API (ASP.NET Core 8.0)	https://discount.4.220.41.138.nip.io/swagger/index.html
Basket API (ASP.NET Core 8.0)	https://basket.4.220.41.138.nip.io/swagger/index.html
Order API (ASP.NET Core 8.0)	https://order.4.220.41.138.nip.io/swagger/index.html
Customer API (Java : Maven)	https://customer.4.220.41.138.nip.io/swagger-ui/index.html
Chat API (Python)	https://chatgptrag.4.220.41.138.nip.io/docs

# eShop Microservices - High Level Design

## 🎯 Overview

eShop Microservices is a production-grade distributed e-commerce system demonstrating modern cloud-native patterns and practices.

Technology Stack:

- **.NET 8** - Backend microservices

- **Azure Application Gateway** - API Gateway with Azure AD authentication

- **Docker & AKS** - Containerization and orchestration

- **FluxCD** - GitOps continuous delivery

- **RabbitMQ** - Event-driven messaging

- **Terraform** - Infrastructure as Code

- **Azure DevOps** - CI/CD automation

## 🏗️ Architecture Diagram

### High-Level System Architecture

![Project Diagram](https://github.com/RijoyP/RijoyP/blob/main/assets/eshopchat.png)

## 🔧 Backend Services

### Architecture Layers

Order microservices follow Clean Architecture principles with clear separation of concerns:

```
┌─────────────────────────────────────────────────────────┐
│       API Layer (Carter - Minimal APIs)          │
│  • HTTP Endpoints                          │
│  • Request/Response DTOs                    │
│  • FluentValidation for input validation          │
└────────────────────┬────────────────────────────────────┘
                     ▼
┌────────────────────────────────────────────────────────┐
│      Application Layer (MediatR - CQRS)          │
│  ┌──────────────┐        ┌──────────────┐        │
│  │  Commands    │        │  Queries    │        │
│  │(Write Model) │        │(Read Model)  │        │
│  └──────┬───────┘        └──────┬───────┘        │
│        │                 │              │
│        ▼                 ▼              │
│  Command Handlers       Query Handlers          │
│  + Business Logic      + Caching Logic          │
└────────────────────┬───────────────────────────────────┘
                     ▼
┌────────────────────────────────────────────────────────┐
│          Domain Layer (DDD)                  │
│  • Aggregates & Entities                    │
│  • Value Objects                            │
│  • Domain Events                            │
│  • Business Rules                           │
└─────────────────────┬───────────────────────┘
                      ▼
┌─────────────────────────────────────────────┐
│       Infrastructure Layer                  │
│  • Entity Framework Core (Traditional ORM)  │
│  • Marten (Event Sourcing for Ordering)     │
│  • PostgreSQL / SQL Server / SQLite         │
│  • Redis (Caching)                          │
│  • RabbitMQ (Messaging)                     │
│  • gRPC (Inter-service communication)       │
└─────────────────────────────────────────────┘

      Cross-Cutting Concerns (Building Blocks)
┌─────────────────────────────────────────────┐
│  OpenTelemetry | Serilog | Polly | FluentValidation  │
└─────────────────────────────────────────────┘
```

### Service Overview

Service	Database	Purpose	Key Features

Catalog API	PostgreSQL	Product management	Product CRUD, categories, inventory, search
Basket API	PostgreSQL + Redis	Shopping cart	Cart operations, session management, checkout
Discount API	SQLite	Promotions	Coupon validation, discount calculation
Ordering API	SQL Server	Order processing	DDD implementation, order lifecycle, payments

Source Code: [eShopMicroservices/Services](https://github.com/RijoyP/eShopMicroservices/tree/main/Services)

Admin Azure AD Authentication

```
┌─────────────────────────────────────────────────────────────┐
│                    Azure AD B2C Tenant                     │
│                                                             │
│  ┌──────────────┐  ┌──────────────┐  ┌──────────────┐      │
│  │ Catalog API  │  │ Discount API │  │    React     │      │
│  │  (Backend)   │  │  (Backend)   │  │  (Frontend)  │      │
│  │ Roles:       │  │ Roles:       │  │              │      │
│  │ CatalogWrite │  │ DiscountWrite│  │ Permissions: │      │
│  │              │  │              │  │ - Catalog    │      │
│  │ Scopes:      │  │ Scopes:      │  │ - Discount   │      │
│  │ Catalog.Write│  │ Discount.Write│  │            │          │
│  └────────────────┘  └────────────────┘  └────────────────┘          │
│  ┌────────────────┐              ┌────────────────┐          │
│  │ Catalog User │              │ Discount User│          │
│  │            │              │            │          │
│  │ Assigned:    │              │ Assigned:    │          │
│  │ CatalogWrite │              │ DiscountWrite│          │
│  └────────────────┘              └────────────────┘          │
└──────────────────────────────────────────────────────────────────────┘
```

Azure Cognitive Search + Azure OpenAI Integration

This project integrates Azure Cognitive Search with Azure OpenAI embeddings to enable vector search and hybrid search for product data and PDF manuals.

** 📌 Features**

Create/update Cognitive Search index with vector search (HNSW)

Generate embeddings using Azure OpenAI

Upload product and PDF chunk documents into Azure Search

Perform pure vector search and hybrid search (keyword + vector)

Extract, chunk, embed, and index PDF manuals

** ⚙️ Prerequisites**

Azure Subscription

Azure Cognitive Search (Basic tier or higher)

Azure OpenAI resource with Embedding deployment

Python 3.9+

🟦 Example usage:

You can ask about the products in application, for example:

“What is the price of the IPhone 17 Pro Max?”

“Compare iPhone 16e and iPhone 17”

“Show me Samsung phones under 5000.”

“Tell me the battery details of iPhone models.”

### 1. Catalog API

Purpose: Product catalog and inventory management

Tech Stack: ASP.NET Core (.NET 8) + PostgreSQL + Redis

Key Responsibilities:

- Product and category management

- Real-time inventory tracking

- Search and filtering

Why PostgreSQL? Complex queries, JSON support, full-text search, ACID compliance for inventory

---

### 2. Basket API

Purpose: Shopping cart and session management

Tech Stack: ASP.NET Core (.NET 8) + PostgreSQL + Redis

Key Responsibilities:

- Add/remove cart items

- Calculate totals

- Apply discounts

- Session persistence

- Checkout coordination

Dual Database Strategy:

- **PostgreSQL**: Persistent cart history and audit trails

- **Redis**: High-speed active cart operations and sessions

Event Publishing: Publishes `BasketCheckedOut` event to trigger order creation

---

### 3. Discount API

Purpose: Coupon and promotion management

Tech Stack: ASP.NET Core (.NET 8) + SQLite + Redis

Key Responsibilities:

- Coupon code validation

- Discount calculation engine

- Promotion rules management

- Time-bound offers

Why SQLite? Lightweight, low data volume, simplified deployment for rules-based data

---

### 4. Ordering API - Domain-Driven Design

Purpose: Complete order lifecycle management

Tech Stack: ASP.NET Core (.NET 8) + SQL Server + RabbitMQ

Key Responsibilities:

- Order creation and validation
- Payment coordination
- Order status workflow
- Fulfillment tracking

Why SQL Server? Enterprise transactions, audit capabilities, reporting features, financial data consistency

#### Domain-Driven Design Implementation

Bounded Context: Ordering

Core domain concepts:

- **Order** (Aggregate Root) - Enforces business rules, controls transactions
- **OrderItem** (Entity) - Line items within orders

- **Address** (Value Object) - Immutable shipping/billing address

- **Payment Info** (Value Object) - Payment details

- **OrderStatus** (Enumeration) - Order state machine

Domain Events Published:

- `OrderCreated` → Triggers inventory reservation

#### Context Mapping

Integration Patterns:

Catalog → Ordering

- Ordering translates Catalog models to its own domain objects

- Prevents breaking changes from cascading

Basket → Ordering (Event-Driven)

- Basket publishes events, Ordering subscribes

- Loose coupling through RabbitMQ

Ordering → Payment (Synchronous)

- Direct API calls with shared payment concepts

- Transactional consistency maintained

Discount → Ordering

- Ordering calls Discount API for price calculations

- Read-only relationship with graceful degradation

Technology Stack Explained

Carter - Minimal API Endpoints

Lightweight alternative to traditional Controllers

Functional endpoint definitions

Clean, organized API routes

Supports route grouping and modules

MediatR - CQRS Implementation

Commands: Handle write operations (Create, Update, Delete)

Queries: Handle read operations (Get, List, Search)

Separates read and write concerns

Pipeline behaviors for validation, logging, and performance tracking

FluentValidation - Input Validation

Strongly-typed validation rules

Automatic validation in MediatR pipeline

Clear error messages

Reusable validation logic

Marten - Event Sourcing (Ordering Service)

PostgreSQL as document database

Event store for order history

Stores aggregates as JSON documents

Optimistic concurrency control

Event projections to read models

Entity Framework Core - Traditional ORM

Used in Catalog and Discount services

Relational data access

LINQ query support

Database migrations

Change tracking

Building Blocks - Shared Libraries

Location: eShopMicroservices/BuildingBlocks/

Reusable components shared across all microservices:

BuildingBlocks.CQRS: ICommand, IQuery, ICommandHandler, IQueryHandler interfaces

BuildingBlocks.Messaging: Domain events, integration events, MassTransit configuration

BuildingBlocks.Behaviors: Validation, logging, and performance pipeline behaviors

BuildingBlocks.Exceptions: Custom domain exceptions

Polly - Resilience & Retry Policies

Handles transient failures

Retry policies with exponential backoff

Circuit breaker pattern

Timeout policies

Applied to HTTP clients and database operations

OpenTelemetry - Distributed Tracing

End-to-end request tracing across all services

Automatic instrumentation for HTTP, SQL, RabbitMQ

Exports traces to Jaeger

Correlates logs, traces, and metrics

Performance monitoring

Serilog - Structured Logging

JSON-formatted logs

Enriched with context (trace IDs, user IDs, etc.)

Centralized logging to Elasticsearch

Queryable log data

Different log levels per environment

gRPC - Inter-Service Communication

High-performance RPC framework

Type-safe service contracts

Used for synchronous service-to-service calls

Example: Basket API calls Discount API via gRPC for real-time discount calculations

CQRS Pattern in Action

Write Path (Commands):

User submits request → API endpoint (Carter)

Command created and validated (FluentValidation)

Command sent through MediatR pipeline

Command handler processes business logic

Changes persisted to database

Domain events published to RabbitMQ

Read Path (Queries):

User requests data → API endpoint (Carter)

Query created and sent through MediatR

Query handler retrieves data

Data cached in Redis (if applicable)

Response returned to user

Benefits:

Optimized read and write models

Independent scaling of reads vs writes

Improved performance with caching

Clear separation of concerns

## ☁️ Infrastructure & Deployment

### Terraform Infrastructure

Repository: [terraform-modules](https://github.com/RijoyP/terraform-modules)

Key Modules:

- **AKS**: Kubernetes cluster with node pools, autoscaling, Azure AD integration
- **ACR**: Container registry with geo-replication and vulnerability scanning
- **Networking**: VNet, subnets, NSGs, Application Gateway
- **Databases**: PostgreSQL, SQL Server, Redis managed services
- **Monitoring**: Log Analytics, Application Insights
- **Security**: Key Vault, managed identities, RBAC

Deployment Stages:

```
Terraform Validate → Plan → Apply (Dev) → Apply (Staging) → Apply (Prod)
                         ↓           ↓            ↓
                    Manual Approval Required for Prod
```

### Platform Components Deployment

Repository: [GitOps/infrastructure](https://github.com/RijoyP/GitOps/tree/main/eShopMicroService/infrastructure)

Deployment Order:

```
1. FluxCD System      → GitOps operator
2. Ingress NGINX      → External traffic routing
3. Cert Manager       → TLS certificate automation
4. RabbitMQ           → Message broker
5. Logging Stack      → Elasticsearch + Kibana (or PLG)
6. Monitoring Stack   → Prometheus + Grafana + Alertmanager
7. Tracing            → Jaeger with OpenTelemetry
```

## 🚀 CI/CD Pipeline

### Pipeline Architecture

Repository: [CICD-Templates](https://github.com/RijoyP/CICD-Templates)

### Application Pipeline Flow

![Pipeline Diagram](https://github.com/RijoyP/RijoyP/blob/main/assets/pipelinenew.png)

Pipeline Files: [eShopMicroservices/.pipeline](https://github.com/RijoyP/eShopMicroservices/tree/main/.pipelines)

### Automated Profile Generation

Script: [generate-profiles.ps1](https://github.com/RijoyP/CICD-Templates/blob/main/applications/scripts/generate-profiles.ps1)

Automated Profile Generation

Script: generate-profiles.ps1

Purpose: Automatically generates Azure DevOps pipeline YAML files for different techinical stack which will help the developement team can use easily.

How It Works:

Scans Services directory to discover all microservices

Loads pipeline templates and environment configurations

For each technical stack × task combination, generates complete pipeline YAML

Validates generated files for syntax and schema

Outputs to applications/profiles/ directory

Generated Profiles: [CICD-Templates/applications/profiles](https://github.com/RijoyP/CICD-Templates/tree/main/applications/profiles)

Sample Generated Files:

Profile	Build	Linting	Unit Tests	SCA	Publish	Parameters
dotnet/dotnet-build-linting-none-all-all.yaml	dotnet/build-dotnet.yaml	dotnet/linting-dotnet.yaml	none/unit-tests-none.yaml	sca/sca-all.yaml	publish/publish-all.yaml	vmImage, poolName, applicationFolder, dotnetVersion, sonarQubeProjectKey, sonarQubeProjectName, sonarQubeServiceConnection, trivySeverity, trivyFormat, serviceConnection, dockerfilePath, buildArgs, dockerContext, trivyExitCode, trivyIgnoreUnfixed, trivyTimeout, acrRegistry, imageName, imageTag
dotnet/dotnet-build-linting-none-all-build-image.yaml	publish/publish-build-image.yaml	dotnet/linting-dotnet.yaml	none/unit-tests-none.yaml	sca/sca-all.yaml	-	vmImage, poolName, applicationFolder, dotnetVersion, sonarQubeProjectKey, sonarQubeProjectName, sonarQubeServiceConnection, trivySeverity, trivyFormat, dockerfilePath, imageName, imageTag, buildArgs, dockerContext
dotnet/dotnet-build-linting-none-all-none.yaml	dotnet/build-dotnet.yaml	dotnet/linting-dotnet.yaml	none/unit-tests-none.yaml	sca/sca-all.yaml	publish/publish-none.yaml	vmImage, poolName, applicationFolder, dotnetVersion, sonarQubeProjectKey, sonarQubeProjectName, sonarQubeServiceConnection, trivySeverity, trivyFormat
dotnet/dotnet-build-linting-none-all-push-acr.yaml	dotnet/build-dotnet.yaml	dotnet/linting-dotnet.yaml	none/unit-tests-none.yaml	sca/sca-all.yaml	publish/publish-push-acr.yaml	vmImage, poolName, applicationFolder, dotnetVersion, sonarQubeProjectKey, sonarQubeProjectName, sonarQubeServiceConnection, trivySeverity, trivyFormat, serviceConnection, dockerfilePath, dockerContext, buildArgs, acrRegistry, imageName, imageTag
dotnet/dotnet-build-linting-none-all-scan-image.yaml	dotnet/build-dotnet.yaml	dotnet/linting-dotnet.yaml	none/unit-tests-none.yaml	sca/sca-all.yaml	publish/publish-scan-image.yaml	vmImage, poolName, applicationFolder, dotnetVersion, sonarQubeProjectKey, sonarQubeProjectName, sonarQubeServiceConnection, trivySeverity, trivyFormat, imageName, imageTag, trivyExitCode, trivyIgnoreUnfixed, trivyTimeout
react/react-build-none-unit-tests-all-build-image.yaml	publish/publish-build-image.yaml	none/linting-none.yaml	react/unit-tests-react.yaml	sca/sca-all.yaml	-	applicationFolder, nodeVersion, buildCommand, buildOutputFolder, vmImage, poolName, testCommand, codeCoverageThreshold, sonarQubeProjectKey, sonarQubeProjectName,

sonarQubeServiceConnection, trivySeverity, trivyFormat, dockerfilePath, imageName, imageTag, buildArgs, dockerContext |

| react/react-build-none-unit-tests-all-none.yaml | react/build-react.yaml | none/linting-none.yaml | react/unit-tests-react.yaml | sca/sca-all.yaml | publish/publish-none.yaml | applicationFolder, nodeVersion, buildCommand, buildOutputFolder, vmImage, poolName, testCommand, codeCoverageThreshold, sonarQubeProjectKey, sonarQubeProjectName, sonarQubeServiceConnection, trivySeverity, trivyFormat |

| react/react-build-none-unit-tests-all-push-acr.yaml | react/build-react.yaml | none/linting-none.yaml | react/unit-tests-react.yaml | sca/sca-all.yaml | publish/publish-push-acr.yaml | applicationFolder, nodeVersion, buildCommand, buildOutputFolder, vmImage, poolName, testCommand, codeCoverageThreshold, sonarQubeProjectKey, sonarQubeProjectName, sonarQubeServiceConnection, trivySeverity, trivyFormat, serviceConnection, dockerfilePath, dockerContext, buildArgs, acrRegistry, imageName, imageTag |

| react/react-build-none-unit-tests-all-scan-image.yaml | react/build-react.yaml | none/linting-none.yaml | react/unit-tests-react.yaml | sca/sca-all.yaml | publish/publish-scan-image.yaml | applicationFolder, nodeVersion, buildCommand, buildOutputFolder, vmImage, poolName, testCommand, codeCoverageThreshold, sonarQubeProjectKey, sonarQubeProjectName, sonarQubeServiceConnection, trivySeverity, trivyFormat, imageName, imageTag, trivyExitCode, trivyIgnoreUnfixed, trivyTimeout |

| react/react-build-none-unit-tests-none-all.yaml | react/build-react.yaml | none/linting-none.yaml | react/unit-tests-react.yaml | sca/sca-none.yaml | publish/publish-all.yaml | applicationFolder, nodeVersion, buildCommand, buildOutputFolder, vmImage, poolName, testCommand, codeCoverageThreshold, serviceConnection, dockerfilePath, buildArgs, dockerContext, trivySeverity, trivyExitCode, trivyIgnoreUnfixed, trivyTimeout, acrRegistry, imageName, imageTag |

## 🔄 GitOps with FluxCD

### Repository Structure

Repository: [GitOps](https://github.com/RijoyP/GitOps)

```
GitOps/
 ├── infrastructure/          # Platform components
|    ├── logging/             # Elasticsearch, Kibana
|    ├── tracing/             # Jaeger, Zipkin
|    ├── monitoring/          # Prometheus, Grafana
|   └── messaging/            # RabbitMQ
|
└── apps/                     # Microservice deployments
   └── base/                  #base overlays
      └── ingress/
      └── frontend-react/
      |     ├── dev
      |     ├── stg
      |    └── prod
       ├── catalog-api/
      |     ├── dev
      |     ├── stg
      |    └── prod
       ├── basket-api/
      |     ├── dev
|    ├── stg
|   └── prod
 ├── discount-api/
|    ├── dev
|    ├── stg
|   └── prod
└── ordering-api/
    ├── dev
    ├── stg
   └── prod
```

### FluxCD Workflow

How FluxCD Works with Helm Base + Overlay

Base HelmRelease Template (apps/base/helmrelease.yaml):

Generic template with placeholder values

References the boilerplate Helm chart from ACR

Used as foundation for all services

Service-Specific Overlays (e.g., apps/basket/dev/):

kustomization.yaml: Patches the base HelmRelease

Replaces placeholders with service-specific names

Generates ConfigMaps and Secrets from values files

Injects environment-specific configurations

values-dev.yaml: Contains all service configurations

Image repository and tag

Replica count

Resource limits

Database connections

Redis configuration

RabbitMQ settings

Ingress rules

Health check settings

Deployment Flow:

```
1. CI Pipeline completes
   └─> Builds Docker image: basket-api:v1.0.50
   └─> Pushes to ACR
   └─> Updates GitOps repo: apps/basket/dev/values-dev.yaml
```

└─> Commits new image tag

2. FluxCD Source Controller (every 1-5 minutes)

└─> Detects Git commit in GitOps repo

3. FluxCD Kustomize Controller

└─> Reads apps/basket/dev/kustomization.yaml

└─> Loads base HelmRelease template

└─> Generates ConfigMap from values-dev.yaml

└─> Generates Secret from values-dev.yaml

└─> Patches base template:

- Replaces name: app-placeholder → basketapi-dev
- Injects ConfigMap reference
- Injects Secret reference

4. FluxCD Helm Controller

└─> Reads HelmRelease: basketapi-dev

└─> Fetches Helm chart from ACR

└─> Merges values from ConfigMap and Secret

└─> Renders Helm templates

└─> Applies to AKS namespace: flux-apps

5. Kubernetes

└─> Creates/Updates Deployment: basketapi-dev (2 replicas)

└─> Creates/Updates Service: basketapi-dev

└─> Creates/Updates Ingress: basket-api-dev.eshop.com

```
   └─> Health checks pass
   └─> Deployment ready

6. FluxCD Notification
   └─> Posts success message to Teams/Slack
```

Key Benefits:

Single Helm Chart: One boilerplate chart used for all services

Environment-Specific Values: Each environment has unique configuration

No Helm CLI: FluxCD manages everything automatically

Git as Source of Truth: All changes tracked and auditable

Automated Reconciliation: Cluster state always matches Git

Easy Rollback: Git revert automatically reverts deployment

### FluxCD Reconciliation Loop

```
┌─────────────────────────────────────────────────────────┐
│ FluxCD Reconciliation Loop                  │
│                                   │
│ Git Commit (New Image Tag)                  │
│      ↓                              │
│ Source Controller (Detects Change)              │
│      ↓                              │
```

```
| Kustomize Controller (Builds Manifests)         |
|       ↓                              |
| Apply to AKS Cluster                    |
|       ↓                              |
| Health Check & Validation                 |
|       ↓                              |
| Notification (Teams/Slack)                |
|                                    |
| ◄──── Continuous Reconciliation (1-5 min) ────┤
└──────────────────────────────────────────┘
```

Key Features:

Drift Detection: Auto-corrects manual changes back to Git state

Progressive Delivery: Canary deployments with Flagger

Automated Rollback: Reverts on health check failures

Multi-Environment: Separate overlays for dev/staging/prod

Key Features:

- **Drift Detection**: Auto-corrects manual changes back to Git state
- **Progressive Delivery**: Canary deployments with Flagger
- **Automated Rollback**: Reverts on health check failures
- **Multi-Environment**: Separate overlays for dev/staging/prod

### Helm Integration

Repository: [helm-templates](https://github.com/RijoyP/helm-templates)

Boilerplate Helm chart for consistent microservice deployments:

- Deployment, Service, Ingress templates

- ConfigMap and Secret management

- HPA (Horizontal Pod Autoscaling)

- ServiceMonitor for Prometheus

- PodDisruptionBudget

## 📊 Observability Stack

### Logging

Stack Options:

Stack: Serilog → Elasticsearch → Kibana

![Kibana Dashboard](https://github.com/RijoyP/RijoyP/blob/main/assets/elasticsearch.png)

How It Works:

Serilog: Structured logging library integrated into all .NET microservices

Elasticsearch: Stores and indexes log data for fast searching

Kibana: Provides visualization dashboards and log exploration UI

Features:

Centralized structured log aggregation from all services

Full-text search and filtering across all logs

Pre-built Kibana dashboards for error tracking and service health

Trace ID correlation for distributed debugging

Real-time log streaming and alerting

Deployment: [GitOps/infrastructure/logging](https://github.com/RijoyP/GitOps/tree/main/eShopMicroService/infrastructure/logging)

### Tracing

Stack: Zipkin + Jaeger with OpenTelemetry → Grafana

Zipkin

![zipkinlist](https://github.com/RijoyP/RijoyP/blob/main/assets/zipkin_list.png)

![zipkin](https://github.com/RijoyP/RijoyP/blob/main/assets/zipkin.png)

![zipkin_flow](https://github.com/RijoyP/RijoyP/blob/main/assets/zipkin_flow.png)

Jaeger

![jaeger_ist](https://github.com/RijoyP/RijoyP/blob/main/assets/jeager_all.png)

![jaeger](https://github.com/RijoyP/RijoyP/blob/main/assets/jaeger.png)

How It Works:

OpenTelemetry: Instruments .NET services to collect trace data

Zipkin: Collects and processes trace spans from services

Jaeger: Provides advanced trace storage and querying capabilities

Grafana: Visualizes traces with unified dashboard alongside metrics

Features:

End-to-end request tracing across all microservices

Deployment: [GitOps/infrastructure/tracing](https://github.com/RijoyP/GitOps/tree/main/eShopMicroService/infrastructure/tracing)

### Monitoring

Stack: Prometheus + Grafana

![Grafana Dashhboard](https://github.com/RijoyP/RijoyP/blob/main/assets/grafana.png)

How It Works:

Prometheus: Scrapes metrics from all services and infrastructure

Grafana: Provides unified dashboards for metrics, traces, and logs

Metrics Collected:

Application: Request rates, response latencies, error rates, HTTP status codes

Infrastructure: CPU usage, memory consumption, disk I/O, network throughput

Deployment: [GitOps/infrastructure/monitoring](https://github.com/RijoyP/GitOps/tree/main/eShopMicroService/infrastructure/monitoring)

### Messaging

Component: RabbitMQ

Features:

- Event-driven communication between services

- Durable queues for reliable message delivery

- Message tracing and monitoring

Deployment: [GitOps/infrastructure/messaging](https://github.com/RijoyP/GitOps/tree/main/eShopMicroService/infrastructure/messaging/)

## 🔗 Repository Links

Repository	Purpose
[eShopMicroservices](https://github.com/RijoyP/eShopMicroservices)	Microservices source code
[GitOps](https://github.com/RijoyP/GitOps)	Kubernetes manifests and FluxCD configs
[terraform-modules](https://github.com/RijoyP/terraform-modules)	Infrastructure as Code modules
[CICD-Templates](https://github.com/RijoyP/CICD-Templates)	Reusable pipeline templates
[helm-templates](https://github.com/RijoyP/helm-templates)	Helm chart boilerplate

---

## 🎯 Key Architectural Patterns

- **Microservices Architecture**: Independent, loosely coupled services

- **Event-Driven Architecture**: Asynchronous communication via RabbitMQ

- **API Gateway Pattern**: Azure Application Gateway with Azure AD authentication

- **Database per Service**: Each service owns its data

- **CQRS**: Command-Query separation in Ordering service

- **Domain-Driven Design**: Rich domain model in Ordering service

- **GitOps**: Declarative infrastructure and deployments

- **Infrastructure as Code**: Terraform for all Azure resources

---

## 📈 System Characteristics

Scalability: Horizontal scaling through Kubernetes HPA, independent service scaling

Resilience: Circuit breakers, retry policies, health checks

Security: Azure AD authentication, managed identities, Key Vault integration, network policies

Observability: Comprehensive logging, tracing, and monitoring across all layers

Automation: Fully automated CI/CD pipelines, GitOps-driven deployments

---

## 🚀 Deployment Summary

1. **Infrastructure**: Terraform provisions Azure resources (AKS, ACR, databases)

2. **Platform**: FluxCD deploys infrastructure components (logging, monitoring, messaging)

3. **Applications**: CI/CD builds images, updates GitOps repo, FluxCD auto-deploys to AKS

4. **Monitoring**: Observability stack provides full system visibility

All deployments are automated, version-controlled, and auditable through Git.

Built with ❤️ using .NET 8, Azure, Kubernetes, and modern DevOps practices