

Cascade

Event-Driven Team Task Board on GKE



Comprehensive Documentation

Kovács Bálint-Hunor

January 8, 2026

Contents

1	Introduction	3
1.1	Overview	3
1.2	Key Features	3
1.3	Technology Stack	3
2	Architecture	5
2.1	Collaborative Model	5
2.1.1	Business Use Cases	5
2.2	Strategic Design	5
2.2.1	Subdomains	5
2.2.2	Bounded Contexts	6
2.2.3	Context Map	6
2.3	Tactical Design	7
2.3.1	Technology Stack	7
2.3.2	Service Architecture	7
2.4	System Architecture	8
2.4.1	Network Architecture	8
2.4.2	Microservices & CQRS	9
2.4.3	Event Flow	10
2.4.4	Data Model	10
2.5	Microservices Patterns	10
2.5.1	Pattern 1: CQRS (Command Query Responsibility Segregation) . .	11
2.5.2	Pattern 2: API Gateway	11
2.5.3	Pattern 3: Event Sourcing (Partial)	11
2.5.4	Pattern 4: Database per Service	11
2.5.5	Pattern 5: Circuit Breaker (via Retry Logic)	11
2.6	Key Architectural Decisions	12
2.7	Kubernetes Infrastructure	12
2.7.1	Workloads	12
2.7.2	Services and Networking	13
2.7.3	Autoscaling (HPA)	13
2.7.4	Event Streaming Infrastructure	13
3	Services	14
3.1	Auth Service	14
3.2	Board Command Service	14
3.3	Board Query Service	14

3.4	Activity Service	15
3.5	Audit Service	15
3.6	API Docs Service	15
4	Deployment	16
4.1	Prerequisites	16
4.2	Local Development	16
4.3	GKE Deployment	16
4.3.1	Setup Script	16
4.3.2	Helm Charts	17
4.3.3	Manual Deployment Steps	17
4.4	Configuration	17
4.4.1	Environment Variables	17
4.4.2	Secrets Management	17
5	API Reference	18
5.1	Overview	18
5.2	Authentication Endpoints	18
5.3	Board Endpoints	18
5.4	Task Endpoints	18
5.5	Interactive Documentation	19

Chapter 1

Introduction

1.1 Overview

Cascade is a modern, event-driven team task board application deployed on Google Kubernetes Engine (GKE). It leverages a microservices architecture to provide a scalable, resilient, and responsive user experience.

The project demonstrates advanced cloud-native patterns including:

- **CQRS (Command Query Responsibility Segregation):** Separation of read and write operations for performance optimization.
- **Event-Driven Architecture:** Asynchronous communication between services using Apache Kafka.
- **Database per Service:** Logical separation of data to ensure service autonomy.
- **API Gateway Pattern:** Centralized routing and handling of external traffic via Nginx.

1.2 Key Features

- **Real-time Updates:** Changes are reflected instantly across all connected clients.
- **Scalable Infrastructure:** Built on GKE with Horizontal Pod Autoscaling (HPA).
- **Secure Authentication:** Supports Email/Password and GitHub OAuth.
- **Audit Logging:** Immutable audit trails for compliance and security.
- **Comprehensive Monitoring:** Activity logging and system health monitoring.

1.3 Technology Stack

The application is built using a modern tech stack:

- **Frontend:** React, Vite, TanStack Router, TailwindCSS
- **Backend:** Node.js, Hono, TypeScript

- **Infrastructure:** Kubernetes (GKE), Helm, Docker
- **Data:** MongoDB (StatefulSet), Redis (Cache), Kafka (Strimzi Operator)

Chapter 2

Architecture

2.1 Collaborative Model

The collaborative model defines the core business use cases and how users interact with the system to achieve their goals.

2.1.1 Business Use Cases

The system supports the following key collaborative scenarios:

- **Board Management:** Users can create, update, and delete project boards.
- **Task Workflow:** Users can create tasks, move them between columns (e.g., To Do → In Progress), and assign them to team members.
- **Real-time Collaboration:** Multiple users can view and edit the same board simultaneously, with changes reflected instantly.
- **Auditability:** All critical actions (e.g., task deletion, permission changes) are logged for compliance and security.

2.2 Strategic Design

Strategic design focuses on defining the bounded contexts and subdomains to ensure a modular and maintainable architecture.

2.2.1 Subdomains

The problem space is divided into the following subdomains:

- **Project Management (Core Domain):** The heart of the application, handling boards, tasks, and columns. This is where the unique business value lies.
- **Identity & Access (Generic Subdomain):** Handles authentication and authorization using industry-standard protocols.
- **Compliance (Supporting Subdomain):** Manages audit logs. It supports the core business but is not the primary product.

- **Engagement (Supporting Subdomain):** Tracks user activity for notifications and analytics.

2.2.2 Bounded Contexts

The solution space maps these subdomains to specific Bounded Contexts, which are implemented as microservices:

- **Auth Context:** Corresponds to the Identity subdomain. Implemented using **Better-Auth**.
- **Board Context:** Corresponds to the Project Management subdomain. Implements CQRS (Command and Query).
- **Audit Context:** Corresponds to the Compliance subdomain.
- **Activity Context:** Corresponds to the Engagement subdomain.

2.2.3 Context Map

The following diagram illustrates the relationships and integration patterns between the bounded contexts.

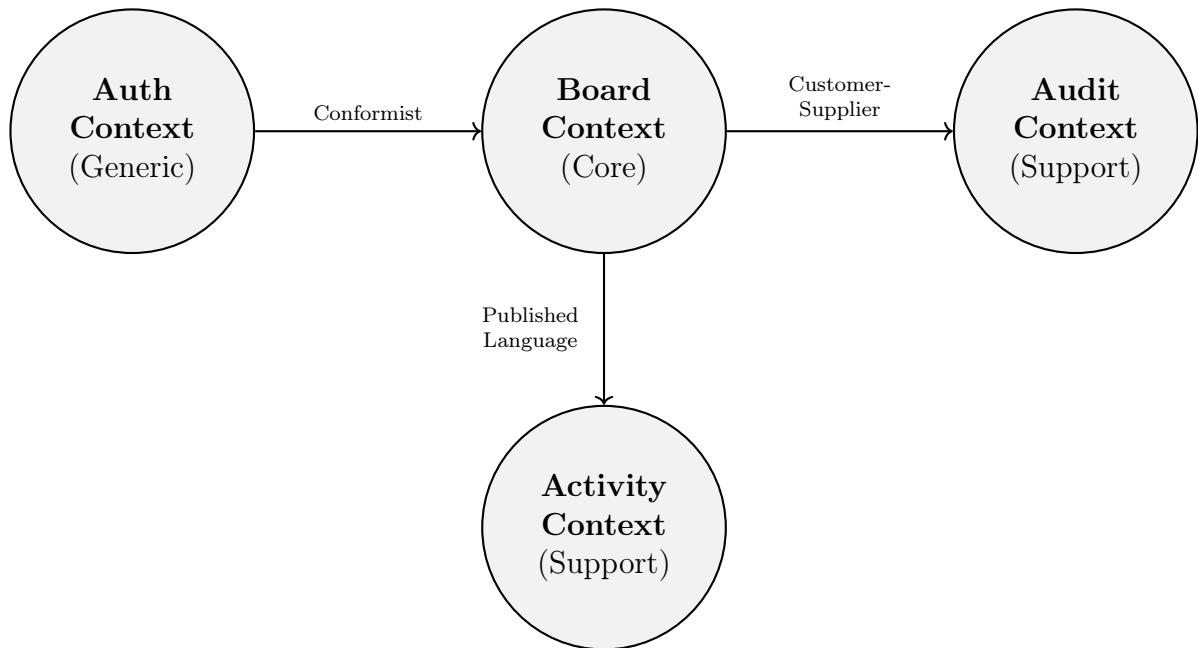


Figure 2.1: Context Map

Relationship Definitions:

- **Conformist:** The Board Context conforms to the Auth Context's model (user IDs, roles) without negotiation.
- **Customer-Supplier:** The Board Context (Supplier) produces events that the Audit Context (Customer) consumes. The Audit Context relies on the Board Context to provide data in a specific format.

- **Published Language:** The Board Context publishes events (e.g., ‘TaskMoved’) using a standard schema that the Activity Context subscribes to.

2.3 Tactical Design

Tactical design details the technical implementation of the bounded contexts, including the technology stack and internal patterns.

2.3.1 Technology Stack

The project utilizes a modern, performance-oriented stack:

- **Runtime:** **Bun** (v1.x) for both backend services and frontend tooling.
- **Backend:**
 - **Framework:** **Hono** for high-performance HTTP APIs.
 - **Validation:** **Zod** for schema validation and type safety.
 - **Auth:** **Better-Auth** for comprehensive authentication.
 - **Logging:** **Pino** for structured logging.
 - **Testing:** **Vitest** for unit and integration tests.
 - **Documentation:** **Scalar** for OpenAPI documentation.
- **Frontend:**
 - **Framework:** **React** with **Vite**.
 - **Full-Stack Features:** **TanStack Start** and **Nitro** server.
 - **Routing:** **TanStack Router**.
 - **State Management:** **TanStack Query**.
 - **Styling:** **TailwindCSS**.
 - **UI Components:** **Lucide-React** (icons), **DnD-Kit** (drag and drop).
- **Data & Messaging:**
 - **Database:** **MongoDB** (Mongoose ORM).
 - **Cache:** **Redis** (IORedis).
 - **Messaging:** **Kafka** (KafkaJS, Strimzi Operator).

2.3.2 Service Architecture

Each service is built independently. The following diagram shows the typical internal structure of a service in Cascade.

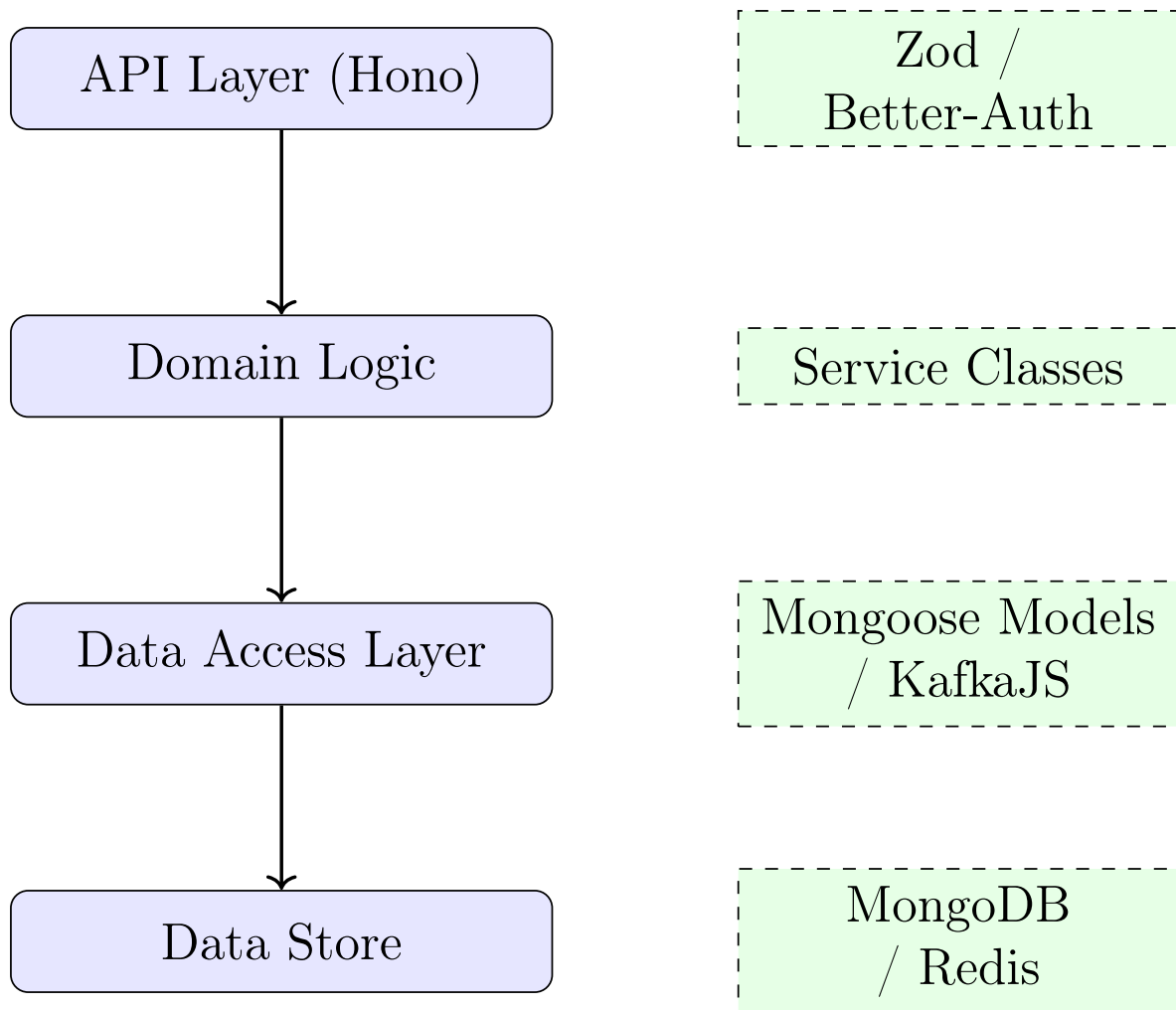


Figure 2.2: Service Internal Architecture

2.4 System Architecture

2.4.1 Network Architecture

The network flow is managed by an Nginx Ingress Controller.

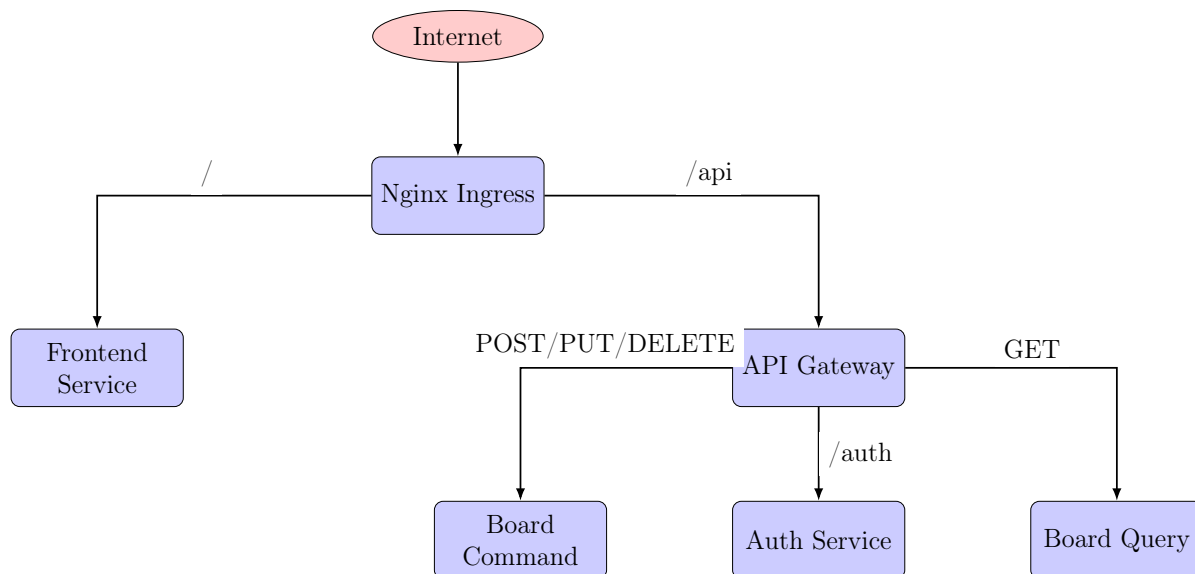


Figure 2.3: Network Flow Diagram

2.4.2 Microservices & CQRS

The system implements CQRS and Event Sourcing patterns.

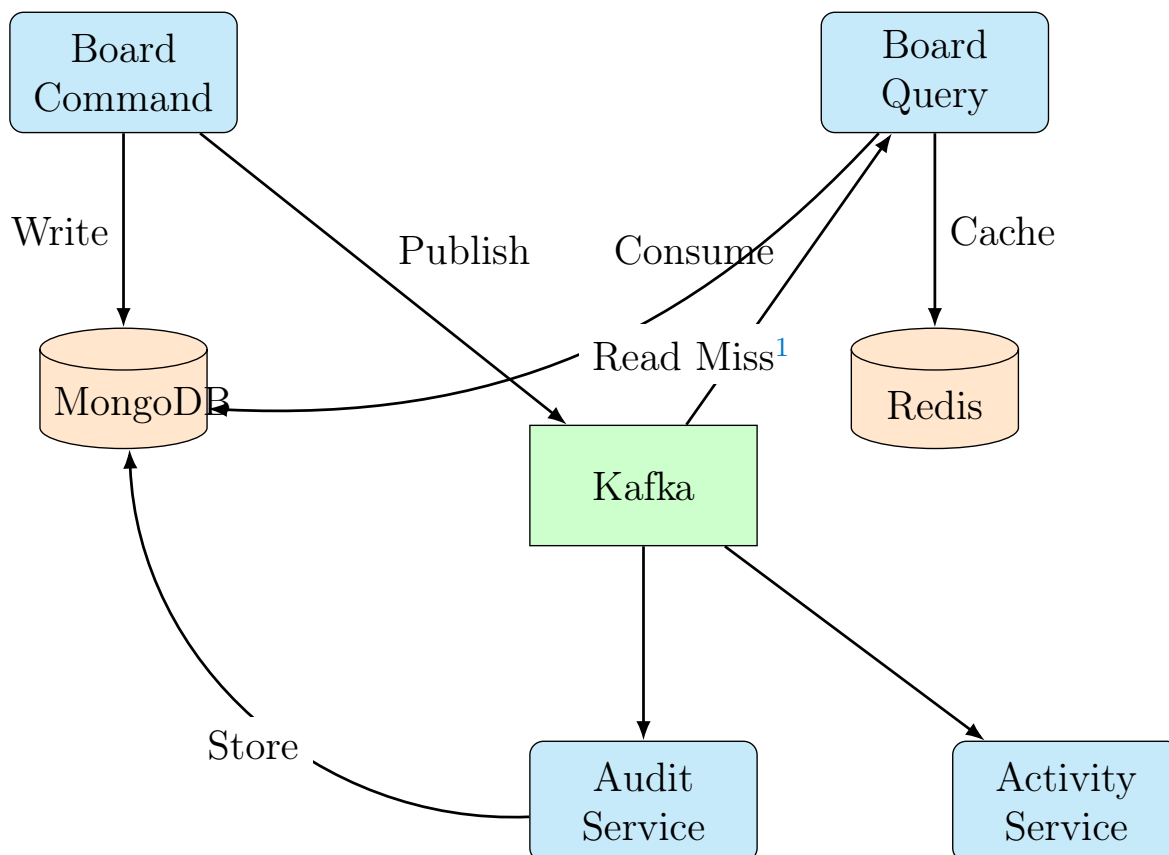


Figure 2.4: CQRS and Event-Driven Architecture

2.4.3 Event Flow

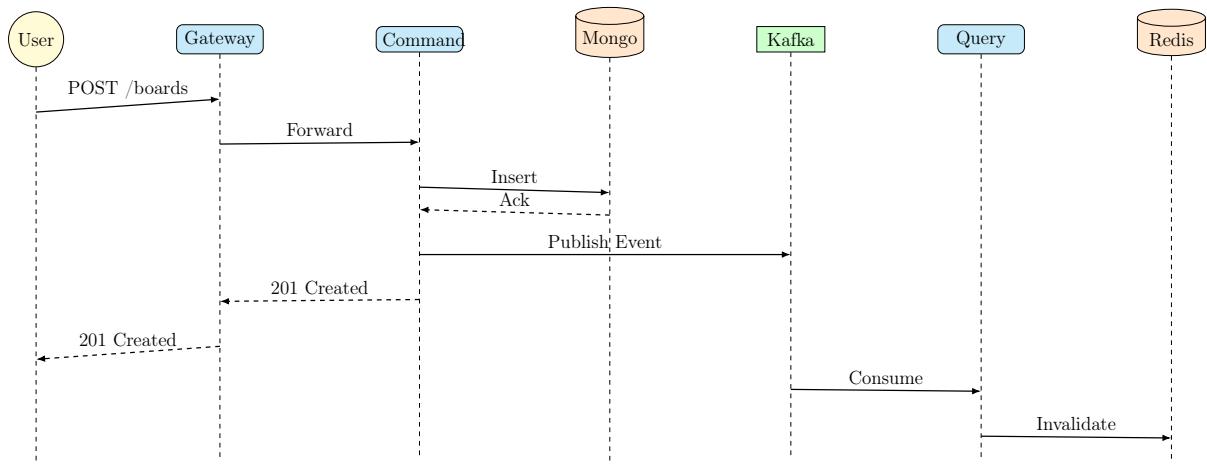


Figure 2.5: Board Creation Sequence

2.4.4 Data Model

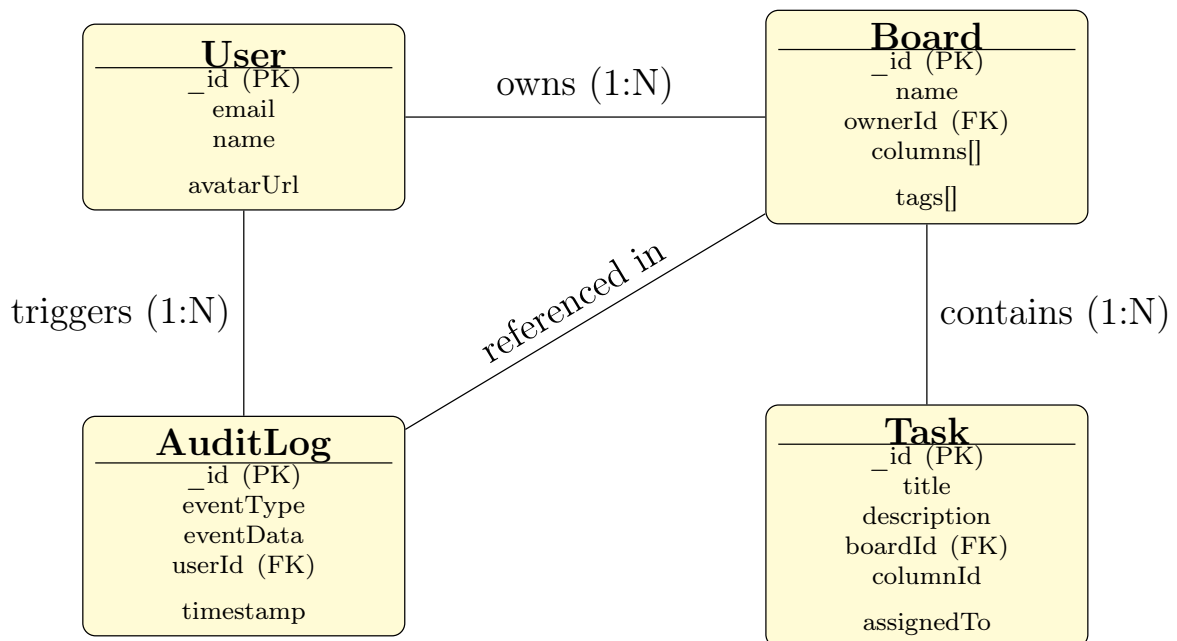


Figure 2.6: Entity Relationship Diagram

2.5 Microservices Patterns

The architecture implements several key microservices patterns to ensure scalability, resilience, and maintainability.

¹Read Miss: If data is not found in Redis, the service queries MongoDB directly.

2.5.1 Pattern 1: CQRS (Command Query Responsibility Segregation)

- **Implementation:** Separate services for write operations (`board-command`) and read operations (`board-query`).
- **Benefits:** Allows independent scaling of read and write workloads; optimizes read performance using Redis caching.

2.5.2 Pattern 2: API Gateway

- **Implementation:** Nginx acts as the single entry point, routing requests based on path and method.
- **Routing:**
 - GET `/api/boards` → `board-query`
 - POST `/api/boards` → `board-command`
- **Benefits:** Simplifies client interaction; provides a centralized place for cross-cutting concerns (though Auth is handled by a separate service).

2.5.3 Pattern 3: Event Sourcing (Partial)

- **Implementation:** All state changes are published as immutable events to Kafka. The Audit Service maintains a complete log of these events in the `cascade-audit` database.
- **Benefits:** Provides a complete audit trail; enables event replay for debugging or rebuilding state.

2.5.4 Pattern 4: Database per Service

- **Implementation:** Each service owns its data.
 - `cascade-auth` DB → Auth Service
 - `cascade-board` DB → Board Services (Shared for pragmatic reasons)
 - `cascade-audit` DB → Audit Service
- **Benefits:** Ensures loose coupling; prevents one service from inadvertently breaking another's data schema.

2.5.5 Pattern 5: Circuit Breaker (via Retry Logic)

- **Implementation:** Kafka consumers implement retry logic with exponential backoff for handling message processing failures.
- **Benefits:** Prevents cascading failures; allows the system to recover gracefully from transient errors.

2.6 Key Architectural Decisions

1. **Pragmatic CQRS:** Both Command and Query services share the same MongoDB physical cluster but use different logical databases/collections where appropriate. This reduces infrastructure cost while maintaining logical separation.
2. **Eventual Consistency:** The system accepts a small window of inconsistency (typically <500ms) between a write (Command) and a read (Query) to achieve high performance.
3. **Stateless Services:** All application services are stateless, allowing for easy horizontal scaling via Kubernetes Deployments.
4. **Infrastructure as Code:** The entire deployment is managed via Helm charts, ensuring reproducibility.

2.7 Kubernetes Infrastructure

The application is deployed on GKE with a robust configuration of resources.

2.7.1 Workloads

The system consists of 8 deployed microservices:

- **Deployments:**
 - `cascade-frontend` (Port 3000): Serves the React application.
 - `cascade-api-gateway` (Nginx): Reverse proxy and routing.
 - `cascade-auth` (Port 3001): Authentication service.
 - `cascade-board-command` (Port 3002): Write operations.
 - `cascade-board-query` (Port 3003): Read operations.
 - `cascade-activity` (Port 3004): Activity tracking.
 - `cascade-audit` (Port 3005): Audit logging.
 - `cascade-api-docs` (Port 3006): Scalar documentation.
- **StatefulSets:**
 - `cascade-mongodb`: A 3-node replica set. StatefulSets are used here to maintain stable network identities (`cascade-mongodb-0`, etc.) and persistent storage for data safety.
- **Jobs:**
 - `cascade-mongodb-init`: A one-time job that connects to the MongoDB pods and initiates the replica set configuration.

2.7.2 Services and Networking

- **ClusterIP:** The default service type. Used for internal communication (e.g., `cascade-auth`, `cascade-redis-master`). These are not exposed externally.
- **Headless Services:** Used for MongoDB (`cascade-mongodb-headless`) and Kafka to allow direct pod discovery and stable DNS entries for clustering.
- **Ingress:** The single entry point for external traffic, routing to the API Gateway and Frontend based on path rules.

2.7.3 Autoscaling (HPA)

Horizontal Pod Autoscalers are configured for critical services to handle varying loads:

- `cascade-auth`, `cascade-board-command`, `cascade-board-query`, `cascade-frontend`.
- **Policy:** Scale between 1 and 5 replicas based on CPU utilization (target 80%).

2.7.4 Event Streaming Infrastructure

- **Strimzi Operator:** Manages the Kafka cluster.
- **Kafka Cluster:** Running in KRaft mode (no Zookeeper).
- **Kafka UI:** Deployed for easy monitoring and management of topics and messages.

Chapter 3

Services

3.1 Auth Service

The Auth Service handles user authentication and session management.

- **Port:** 3001
- **Database:** `cascade-auth`
- **Features:**
 - Email/Password authentication via Better Auth.
 - GitHub OAuth integration.
 - JWT token generation and validation.

3.2 Board Command Service

Responsible for all write operations to boards and tasks.

- **Port:** 3002
- **Database:** `cascade-board`
- **Responsibilities:**
 - Validating incoming requests (Zod schemas).
 - Persisting changes to MongoDB.
 - Publishing domain events to Kafka (e.g., `board.created`, `task.moved`).

3.3 Board Query Service

Responsible for serving read requests with high performance.

- **Port:** 3003
- **Database:** `cascade-board`

- **Cache:** Redis
- **Responsibilities:**
 - Serving GET requests for boards and tasks.
 - Managing Redis cache (Read-through/Write-through).
 - Consuming Kafka events to invalidate stale cache entries.

3.4 Activity Service

Provides real-time visibility into system activities.

- **Port:** 3004
- **Input:** Kafka topics
- **Function:** Consumes events and logs them for monitoring. It does not maintain its own database but streams activity data.

3.5 Audit Service

Ensures compliance and security through immutable logging.

- **Port:** 3005
- **Database:** `cascade-audit`
- **Function:** Consumes all domain events and stores them in an append-only collection.
- **Schema:** Includes `eventType`, `eventData`, `userId`, and `timestamp`.

3.6 API Docs Service

Hosts the interactive API documentation.

- **Port:** 3006
- **Tool:** Scalar
- **Path:** `/reference`

Chapter 4

Deployment

4.1 Prerequisites

Before deploying Cascade, ensure you have the following tools installed:

- Docker and Docker Compose
- Google Cloud SDK (gcloud)
- kubectl
- Helm

4.2 Local Development

For local development, Docker Compose is used to spin up the entire stack.

```
1 docker-compose up --build
```

Listing 4.1: Start Local Environment

This starts all microservices, MongoDB, Redis, and Kafka locally.

4.3 GKE Deployment

Deploying to Google Kubernetes Engine involves several steps, automated via Helm and shell scripts.

4.3.1 Setup Script

The `setup.sh` script automates the cluster creation and configuration.

```
1 ./setup.sh <YOUR_PROJECT_ID>
```

Listing 4.2: Run Setup Script

4.3.2 Helm Charts

The project uses a unified Helm chart located in `helm/cascade`. Key configuration files:

- `values.yaml`: Default configuration values.
- `templates/`: Kubernetes manifest templates.

4.3.3 Manual Deployment Steps

If not using the setup script:

1. **Create Cluster**: Create a GKE cluster with standard settings.
2. **Install Dependencies**: Install Nginx Ingress Controller and Cert Manager (if needed).
3. **Build Images**: Build and push Docker images to Google Artifact Registry.
4. **Deploy Helm Chart**:

```
1 helm install cascade ./helm/cascade --set project.id=<  
PROJECT_ID>  
2
```

4.4 Configuration

4.4.1 Environment Variables

Services are configured via environment variables injected by Kubernetes.

- `MONGODB_URI`: Connection string for MongoDB.
- `KAFKA_BROKERS`: Address of Kafka bootstrap server.
- `REDIS_HOST`: Address of Redis instance.
- `JWT_SECRET`: Secret key for signing tokens.

4.4.2 Secrets Management

Sensitive data like OAuth secrets and database credentials should be managed using Kubernetes Secrets, not plain text in `values.yaml`.

Chapter 5

API Reference

5.1 Overview

The API is designed around RESTful principles. All API endpoints are prefixed with `/api`.

5.2 Authentication Endpoints

- `POST /api/auth/sign-up`: Register a new user.
- `POST /api/auth/sign-in`: Log in an existing user.
- `POST /api/auth/sign-out`: Log out.
- `GET /api/auth/session`: Get current session.

5.3 Board Endpoints

- `GET /api/boards`: List all boards for the current user.
- `POST /api/boards`: Create a new board.
- `GET /api/boards/:id`: Get details of a specific board.
- `PUT /api/boards/:id`: Update a board.
- `DELETE /api/boards/:id`: Delete a board.

5.4 Task Endpoints

- `POST /api/tasks`: Create a new task.
- `PUT /api/tasks/:id`: Update a task (move, edit).
- `DELETE /api/tasks/:id`: Delete a task.

5.5 Interactive Documentation

For interactive testing and detailed schema information, visit the API Docs service at `/reference` when the application is running.