COINBASE CRYPTO EXCHANGE SOLUTION REPORT

1. Introduction

This report outlines the design and implementation of a container-based service solution for the Coinbase crypto exchange to manage and monitor their services. The solution incorporates cloud-native architecture using Kubernetes, automated deployment via a CI/CD pipeline, and follows a Blue-Green deployment strategy to ensure high availability and 100% uptime of services.

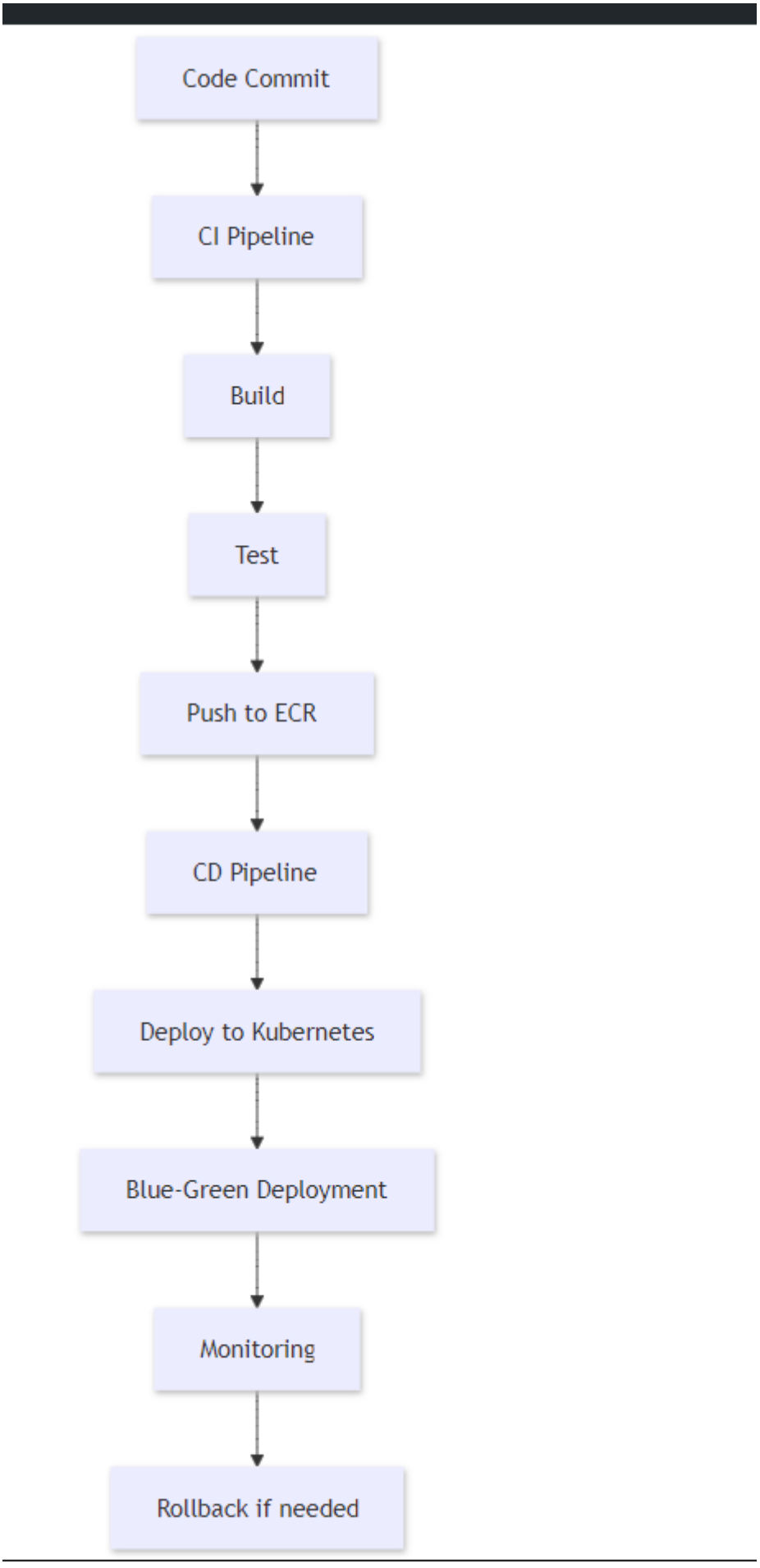
2. Solution Design

2.1 Architecture Diagram

The architecture consists of the following core components:

- **Backend Service**: A mock service handling business logic
- **CI/CD Pipeline**: A pipeline that automates the deployment process, testing, and promotion of services across environments.
- **Kubernetes Cluster**: Manages deployments and scaling of services using various Kubernetes objects such as Deployments, Services,

Below is the architecture diagram:


Code Commit
CI Pipeline
Build
Test
Push to ECR
CD Pipeline
Deploy to Kubernetes
Blue-Green Deployment
Monitoring
Rollback if needed

2.1 Architecture Diagram

- **User Requests**: Users interact with the Frontend Service through HTTP requests. The Frontend Service communicates with the Backend Service.
- **Backend Processing**: The Backend Service processes the request and interacts with the Database to fetch or store data.
- **CI/CD Flow**: The CI/CD Pipeline automates the deployment and testing of services from code commit to production release.

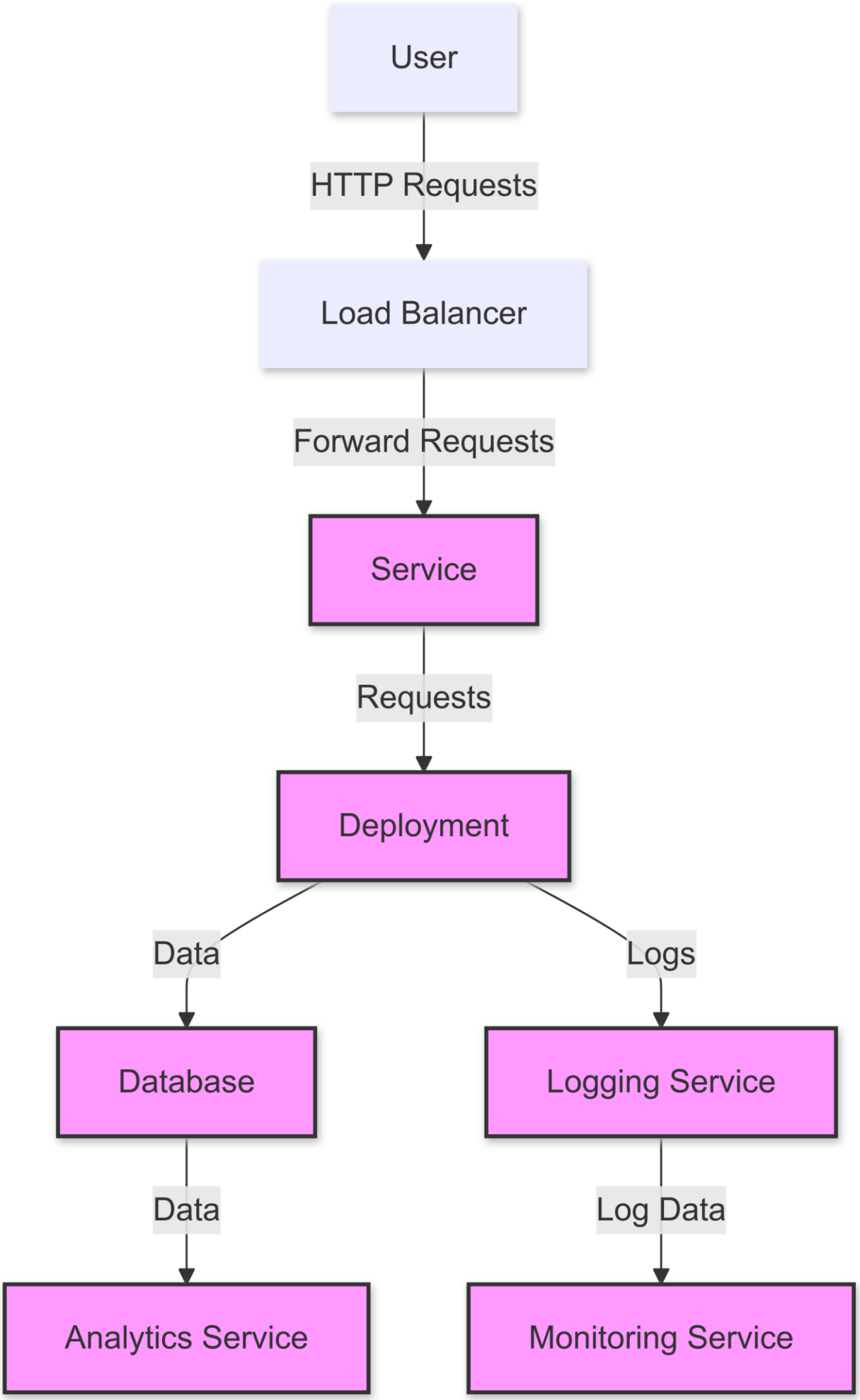
3. Deployment Architecture

3.1 Deployment Architecture Diagram

The solution follows a multi-environment deployment strategy, encompassing:

- **Backend Service**: A mock service handling business logic and interactions with the database.
- **CI/CD Pipeline**: A pipeline that automates the deployment process, testing, and promotion of services across environments.
- **Kubernetes Cluster**: Manages deployments and scaling of services using various Kubernetes objects such as Deployments, Services,

Below is the architecture diagram:


User
HTTP Requests
Load Balancer
Forward Requests
Service
Requests
Deployment
Data
Logs
Database
Logging Service
Data
Log Data
Analytics Service
Monitoring Service

3.2 Blue-Green Deployment Strategy

This method is a deployment strategy designed to ensure minimal downtime during updates to production environments. It involves running two parallel environments: one for the currently active version of the application (blue) and another for the new version (green). The process can be described in the following steps:

1. **Setup**: Initially, the blue environment is serving all user traffic, while the green environment remains idle or in standby.
2. **Deploy to Green**: When a new version of the application is ready to be deployed, it is rolled out to the green environment. This process happens without affecting the users currently using the blue environment.
3. **Testing**: After the deployment, automated smoke tests or integration tests are run in the green environment to ensure that the new version functions correctly and meets all requirements.

4. **Switch Traffic**: Once the green environment has been thoroughly tested and validated, traffic is gradually or immediately switched from the blue environment to the green environment. This switch is typically managed through load balancers or DNS routing adjustments.
5. **Monitor**: After the switch, the green environment is monitored to detect any issues or performance bottlenecks. Monitoring tools and error logs help ensure that the system runs smoothly.
6. **Rollback**: If issues arise in the green environment, the system can quickly revert traffic back to the blue environment, minimizing the impact on users and ensuring continuity of service.
7. **Clean Up**: Once the green environment is confirmed to be stable and fully operational, the blue environment can be updated or decommissioned to free up resources.

The advantage of Blue-Green deployment is that it ensures zero downtime during updates. Users experience seamless transitions between application versions, and rollback mechanisms provide safety in case of deployment failures. This strategy is ideal for services like Coinbase, which require high availability and minimal service disruption.

4. CI/CD Pipeline Design

4.1 Pipeline Structure

The CI/CD pipeline is designed to automate deployment and testing in three stages:

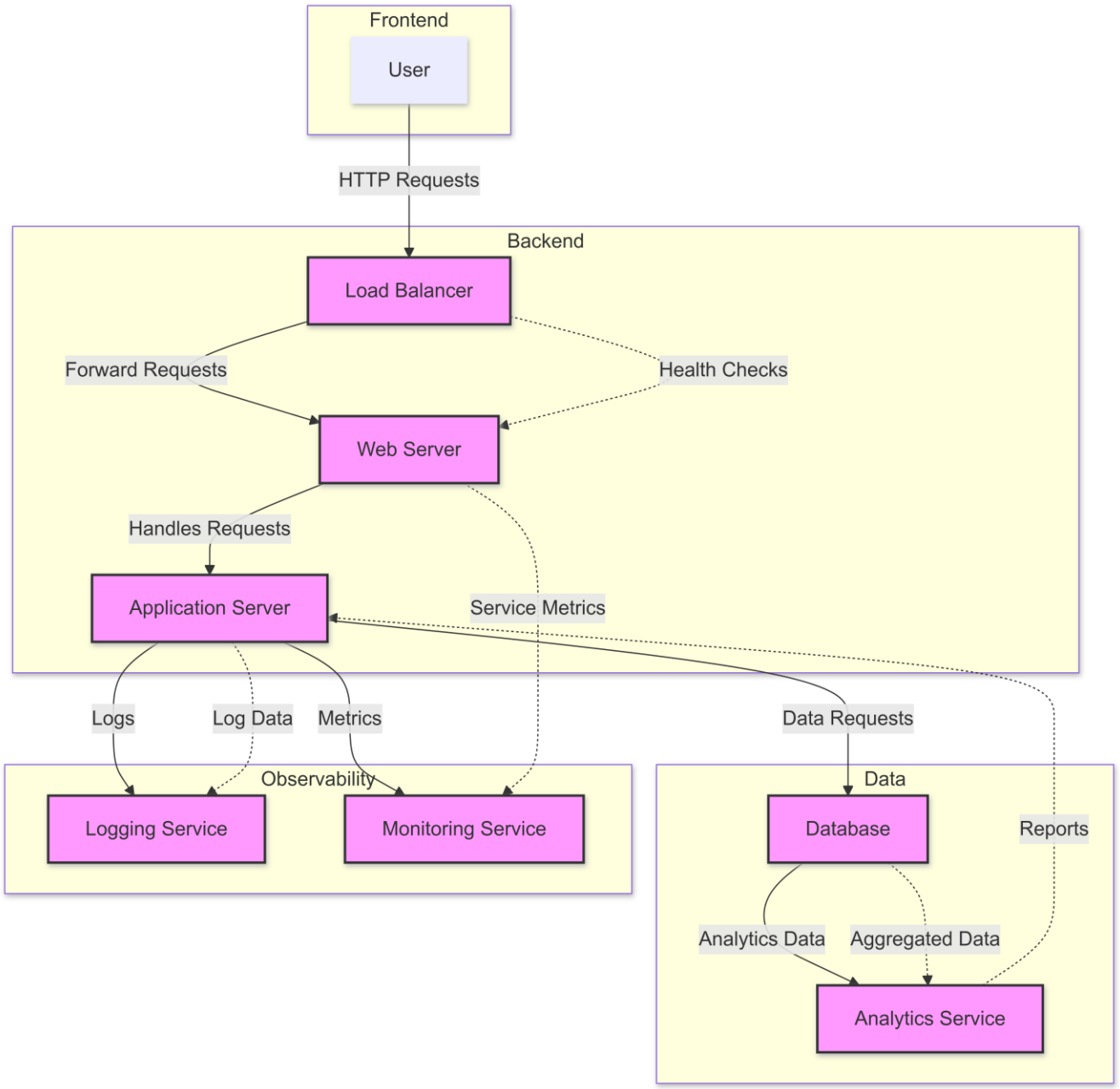
1. **Build Stage**: Upon code commit, the CI pipeline builds the service images using Docker and stores them in a container registry.
2. **Test Stage**: Automated tests are triggered, covering unit tests, integration tests, and security scans.
3. **Deployment Stage**: The pipeline deploys the service artifacts across the development, testing, and production environments. In production, the Blue-Green deployment model ensures a smooth rollout.

4.2 CI/CD Process Description

- **CI**: Continuous Integration ensures that all changes to the codebase are automatically built and tested.

- **CD**: Continuous Deployment automates the release of the tested code to different environments, including production, without manual intervention.

Below is the architecture diagram:



5. Security and Ethics Considerations

5.1 Security

To ensure data integrity and system security, the following measures are taken:

- **TLS/SSL** is enforced for all communications between services.
- **Role-Based Access Control (RBAC)** is used in Kubernetes to ensure limited and secure access to critical resources.
- Sensitive data is stored securely using **Kubernetes Secrets** and is encrypted both in transit and at rest.

6. Implementation Details

6.1 Kubernetes Artifacts

The solution is deployed using various Kubernetes objects, such as:

- **Deployments**: Manages the lifecycle and scaling of the services.
- **Services**: Exposes the frontend and backend services internally and externally.

6.2 Persistence Layer

A cloud-native PostgreSQL database is deployed within the same Kubernetes cluster to handle persistent data. The database is configured to ensure high availability and redundancy.

7. Automated Testing

7.1 Test Suite

A simple automated test suite was created to validate the services after every deployment. The suite includes:

- **Unit Tests**: Validate individual components of the services.

- **Integration Tests**: Test the interaction between the frontend, backend, and database services.
- **Smoke Tests**: Ensure that the core functionality of the services works after each deployment.

7.2 Test Automation

Automated tests are run in the CI pipeline after each build to ensure the integrity of the services before they are promoted to production.Automated tests are run in the CI pipeline after each build to ensure the integrity of the services before they are promoted to production.