Architecture Diagram Description

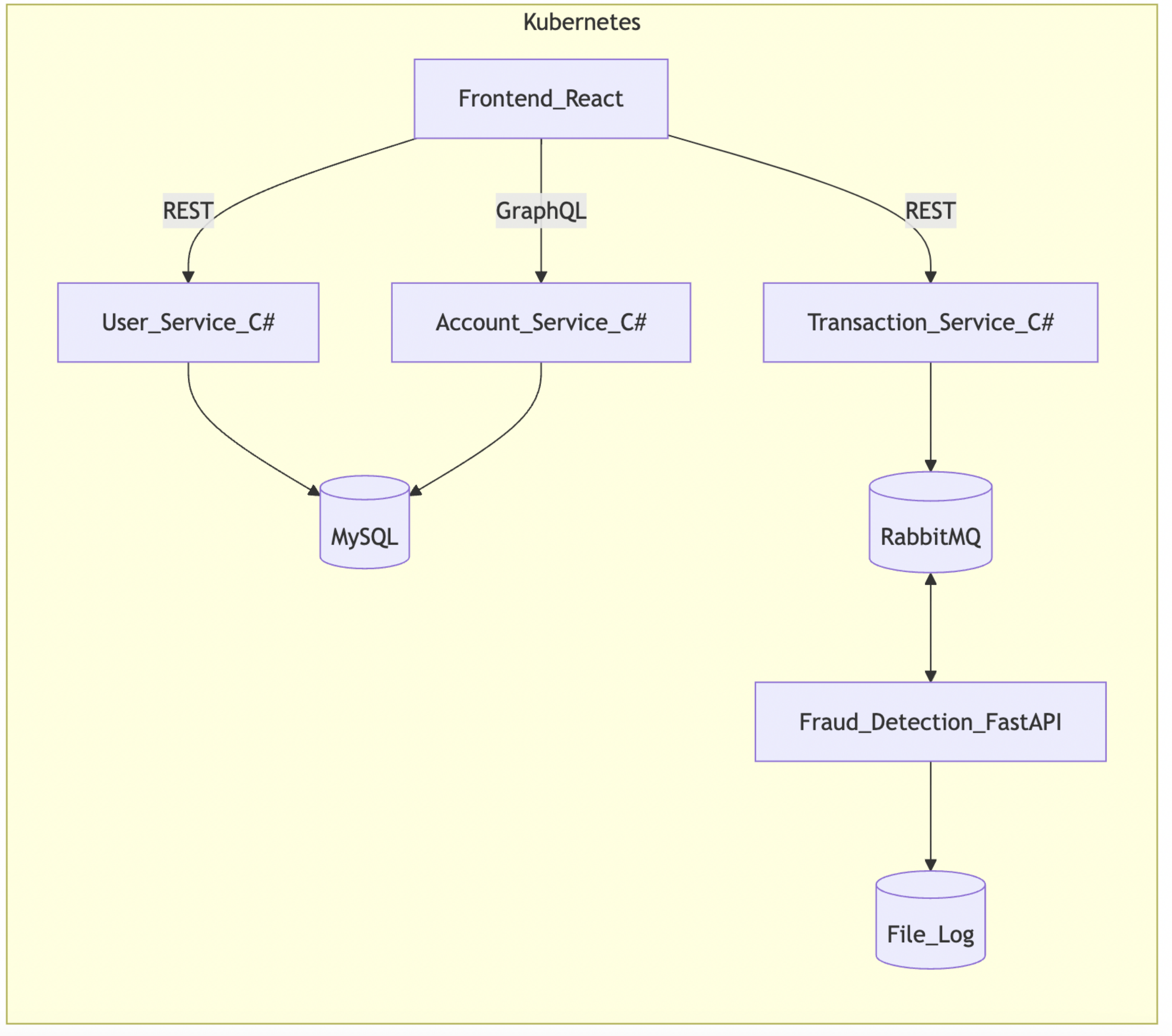
Components

1. **Frontend (React)**:
   - Communicates with:
     - User Service via REST API.
     - Account Service via GraphQL.
     - Transaction Service via REST API (polling for status).
2. **Microservices**:
   - **User Service (C#)**: Handles registration/login, uses MySQL.
   - **Account Service (C#)**: Manages balances, uses MySQL.
   - **Transaction Service (C#)**: Processes transfers, uses MySQL.
   - **Fraud Detection Service (Python with FastAPI)**: Analyzes transfers, logs to a file.
3. **Message Queue (RabbitMQ)**:
   - Facilitates async communication between Transaction Service, Fraud Detection Service, and Account Service.
4. **Database**:
   - MySQL instance shared by User, Account, and Transaction Services, with separate tables per service.
5. **Logging**:
   - File-based logging for Fraud Detection Service.
6. **Kubernetes**:
   - Orchestrates all containers (frontend, microservices, RabbitMQ, MySQL).

Flow

- Arrows represent communication:
  - Solid lines: HTTP (REST/GraphQL) between frontend and microservices.
  - Dashed lines: RabbitMQ messages between microservices.
  - Dotted lines: Database connections from C# services to MySQL.

Distributed system architecture



Explanation

- **Frontend**: Connects to microservices via HTTP.
- **User Service**: REST API to frontend, connects to MySQL.
- **Account Service**: GraphQL to frontend, connects to MySQL, listens to RabbitMQ for balance updates.
- **Transaction Service**: REST API to frontend, connects to MySQL, sends/receives RabbitMQ messages for fraud checks and account updates.
- **Fraud Detection Service**: Built with FastAPI (Python), consumes/produces RabbitMQ messages, logs to a file.
- **RabbitMQ**: Central hub for async messaging between Transaction, Fraud Detection, and Account Services.
- **MySQL**: Single shared database instance for C# services, with dotted connections.
- **Kubernetes**: Encompasses all components.

Detailed Flow (Transfer Example)

1. **Frontend** → **Transaction Service**: REST POST `/transfer {fromAccount, toAccount, amount}`.
2. **Transaction Service** → **RabbitMQ**: Publishes `"CheckFraud": {transferId, amount}`.

3. **RabbitMQ → Fraud Detection Service**: Consumes `"CheckFraud"`.
4. **Fraud Detection Service**: Checks if `amount > 1000`, logs result to file, publishes `"FraudResult": {transferId, isFraud}`.
5. **RabbitMQ → Transaction Service**: Consumes `"FraudResult"`.
6. **Transaction Service → RabbitMQ**: If not fraud, publishes `"UpdateAccounts": {fromAccount, toAccount, amount}`.
7. **RabbitMQ → Account Service**: Consumes `"UpdateAccounts"`, updates balances in MySQL.
8. **Account Service → RabbitMQ**: Publishes `"TransferComplete": {transferId}`.
9. **RabbitMQ → Transaction Service**: Consumes `"TransferComplete"`, updates status in MySQL.
10. **Frontend → Transaction Service**: Polls REST GET `/transfer/{transferId}` for status.

Frontend

- **Technology**: React
- **Functionality**:
  - Register/login via User Service (REST).
  - View account balance via Account Service (GraphQL).
  - Initiate a transfer via Transaction Service (REST).
  - Display transfer status (success or flagged as fraud).
- **Scope**: Minimal UI—just a few pages with basic forms and tables, no styling or real-time updates.

Infrastructure

- **Message Queue**: RabbitMQ
  - Used for basic async communication (e.g., Transaction → Fraud Detection → Account).
  - Simple queues with no advanced features like retries or dead-letter queues.
- **Containerization**: Docker
  - Basic Dockerfiles for each microservice and frontend.
- **Orchestration**: Kubernetes
  - Run locally with Minikube for development; DigitalOcean Kubernetes chosen for its free tier and ease of setup for the final demo.
- **Logging**:
  - File-based logging per service (e.g., text files for Fraud Detection), chosen for simplicity in a demo context.
- **Monitoring**: None, though Prometheus is an option if time permits.
- **CI/CD**:
  - GitHub Actions pipeline: build Docker images, run basic tests, deploy to Kubernetes.

Specifications

Functional Requirements

1. **User Management**:
   - Register with username/password.
   - Log in to access the app.

2. **Account Management**:
   - Create a single account per user with an initial balance.
   - View account balance.
3. **Transaction Processing**:
   - Transfer money between accounts.
   - Display transfer status (success or flagged).
4. **Fraud Detection**:
   - Flag transfers exceeding $1000 and log the decision.

Non-Functional Requirements

- **Scalability**: Basic microservices structure (no high-load optimization needed).
- **Security**: Plain text passwords in MySQL for demo simplicity (not production-ready).
- **Reliability**: Basic error handling; no complex retry logic.
- **Performance**: Adequate for a demo with a few users.
- **Maintainability**: Simple code with comments.
- **Deployability**: Deployable to Kubernetes via CI/CD.

System Architecture Design

- **Architectural Pattern**: Message-Driven Microservices Architecture using RabbitMQ for asynchronous communication.
- **Team Responsibilities**:
  - Daniel: User Service + Frontend.
  - Jakob: Account Service + RabbitMQ.
  - Albert: Transaction Service.
  - Frederik: Fraud Detection Service + DevOps (Kubernetes, CI/CD).

Development and Deployment Plan

1. **Project Management**:
   - Use GitHub with a monorepo (one repo, folders for each service).
   - Simple task board (GitHub Issues) for tracking.
2. **CI/CD Setup**:
   - GitHub Actions:
     - Build Docker images.
     - Run minimal unit tests (e.g., one test per service for core functionality).
     - Deploy to Minikube or DigitalOcean Kubernetes.
3. **Documentation Strategy**:
   - GitHub README for setup instructions.
   - Inline code comments for implementation details.
   - Final report for architecture and design decisions.
4. **Versioning Strategy**:
   - **Code**: Git tags (e.g., v1.0.0).
   - **APIs**: Semantic versioning in endpoints (e.g., `/v1/transfer`).
   - **Database**: Manual schema updates for MySQL tables.

5. **Implementation Steps** (~60-75 hours per person):
   - **Week 1-2**: Define specs, set up repo, Docker/Kubernetes basics (10-15 hours).
   - **Week 3-5**: Build User Service + Frontend login/register (15-20 hours).
   - **Week 6-8**: Build Account Service + balance view (15-20 hours).
   - **Week 9-11**: Build Transaction Service + transfer logic (15-20 hours).
   - **Week 12-13**: Build Fraud Detection Service + RabbitMQ integration (10-15 hours).
   - **Week 14**: Polish UI, test, deploy (5-10 hours).
6. **Deployment**:
   - Minikube locally for development.
   - DigitalOcean Kubernetes for the final demo.