Aware Micro-Services: A Data-Synchronized Micro-service Architecture Based on Kong, Flask, RabbitMQ, MongoDB, and Amazon EC2

Ismael Talib Ridha Barzani
Student ID: 40188139
Department of Electrical and Computer Engineering
Concordia University - Gina Cody School
Montreal, Quebec, Canada
ismaelmergasori@gmail.com

***Abstract*—Microservices has recently gained significant recognition as it enables teams to develop and deploy their services independently, reduce code interdependency and increase readability and modularity within a codebase. Design and implementation of microservices usually require multiple technologies that have to communicate with each other to adhere to the CAP theorem and ensure data synchronization. This work presents the design and implementation of a microservices-based architecture for a data synchronization solution using Kong API Gateway, Flask, RabbitMQ, and MongoDB. The project integrates two core microservices: a User Service for managing user information and an Order Service for handling order data. Synchronization of shared fields, such as email and delivery addresses, is achieved through an event-driven system powered by RabbitMQ. The architecture incorporates an API Gateway implemented with Kong, which facilitates routing, version management, and traffic distribution, enabling the seamless deployment of new service versions through the strangler pattern. Both microservices are containerized using Docker and deployed in a single environment to streamline development and testing. The work also explores the implementation of RESTful APIs, database schemas, event handling mechanisms, and a multithreaded approach to integrate the RabbitMQ consumer with the Flask application. Finally, the solution ensures scalability, modularity, and maintainability, while aligning with modern software engineering practices.**

***Index Terms*—Cloud Computing, Micro-services, Kong API Gateway, RabbitMQ, MongoDB, Amazon EC2, REST**

I. Introduction

A microservice architecture is an architectural pattern that arranges an application as a collection of loosely coupled, fine-grained services, communicating through lightweight protocols. This approach ensures scalability, modularity, and ease of maintenance compared to traditional monolithic systems. In this work, I developed a solution for a microservice-based system aimed at integrating User and Order Services while ensuring data synchronization and supporting evolutionary changes through versioning.

The architecture consists of two versions of the User Microservice and one Order Microservice, implement using the Flask framework [1], all interacting with MongoDB databases [2] for storage. Communication between the services is enabled via an event-driven system powered by RabbitMQ [3], which ensures data consistency across the microservices. An API Gateway, implemented with Kong API Gateway engine [4], configured with the strangler pattern, facilitates request routing, ensuring backward compatibility and a seamless transition between versions of the user service. Additionally, Docker [5] was used to containerize the entire system, offering flexibility for both monolithic and microservice deployment approaches. The implementation is designed to run on cloud infrastructure with endpoints for API interaction. The final docker setup is then deployed to an Amazon EC2 VM instance [6]. Fig. 1 shows the architectural design for this system.

The system adheres to key requirements, including user and order management, event-based synchronization, version control of services, and cloud-based deployment. Advanced features such as configuration-driven traffic splitting between microservice versions and optional continuous integration/deployment (CI/CD) pipelines were incorporated to enhance reliability and maintainability. Nevertheless, documentation for the APIs and the data model were generated by exposing the APIs to SwaggerHub [7].

This report outlines the system's design, implementa-

This is submitted as part of requirement 3 of the Programming on the Cloud (COEN-6313) Course Assignment #2.

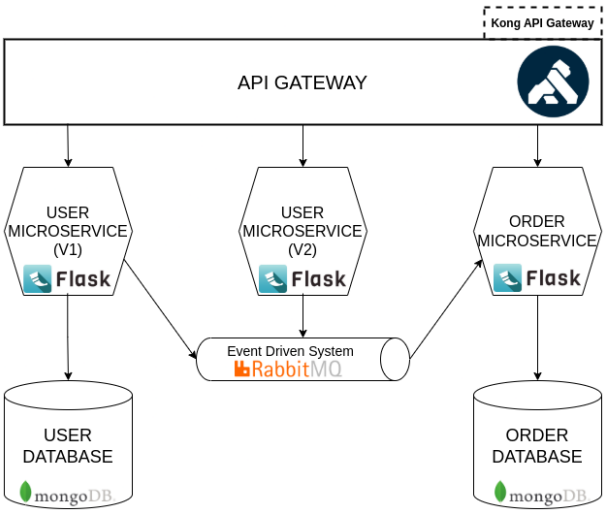

Fig. 1. Aware Micro-Services Architecture

tion details, and the methods used to meet the specified requirements. The goal is to provide a comprehensive overview of the architecture and its operational capabilities, supported by design diagrams and API definitions.

II. Data Models Design

The system relies on two distinct databases: the **User Database** and the **Order Database**, each designed to manage critical information related to users and their orders. Both databases use JSON Schema to define the structure of their respective data models, ensuring consistency and data integrity.

A. User Database

The **User Database** stores essential details about users, including their identification, contact information, and delivery address. The primary attributes include:

- **userId**: A unique identifier for each user.
- **firstName** and **lastName**: The user's first and last names.
- **emails**: A list of email addresses associated with the user, ensuring communication channels are available.
- **deliveryAddress**: The physical address where the user's orders will be delivered. This includes fields like street, city, state, postal code, and country.
- **phoneNumber**: An optional field for the user's phone number, which follows a specific numeric pattern (10-15 digits).
- **createdAt** and **updatedAt**: Timestamps representing when the user account was created and last updated.

The `user_schema.json` defines the structure for these attributes, ensuring that the data stored in the User Database is consistent and follows a defined schema. The following JSON schema describes the user data model:

```
{
    "$schema":
    ↪  "http://json-schema.org/draft-07/schema#",
    "title": "User",
    "type": "object",
    "properties": {
        "userId": { "type": "string",
        ↪  "description": "The unique identifier
        ↪  for a user account" },
        "firstName": { "type": "string",
        ↪  "description": "First name of the user"
        ↪  },
        "lastName": { "type": "string",
        ↪  "description": "Last name of the user"
        ↪  },
        "emails": {
            "type": "array",
            "items": { "type": "string", "format":
            ↪  "email" },
            "description": "A list of email
            ↪  addresses associated with the user"
        },
        "deliveryAddress": {
            "type": "object",
            "properties": {
                "street": { "type": "string",
                ↪  "description": "Street address"
                ↪  },
                "city": { "type": "string",
                ↪  "description": "City" },
                "state": { "type": "string",
                ↪  "description": "State" },
                "postalCode": { "type": "string",
                ↪  "description": "Postal code" },
                "country": { "type": "string",
                ↪  "description": "Country" }
            },
            "required": ["street", "city", "state",
            ↪  "postalCode", "country"],
            "description": "The delivery address of
            ↪  the user"
        },
        "phoneNumber": { "type": "string",
        ↪  "pattern": "^[0-9]{10,15}$",
        ↪  "description": "Optional phone number
        ↪  for the user, 10-15 digits." },
        "createdAt": { "type": "string", "format":
        ↪  "date-time", "description": "Timestamp
        ↪  of when the user was created." },
        "updatedAt": { "type": "string", "format":
        ↪  "date-time", "description": "Timestamp
        ↪  of when the user was last updated." }
    },
    "required": ["userId", "emails",
    ↪  "deliveryAddress"]
}
```

B. Order Database

The **Order Database** stores information related to the orders placed by users. This includes:

- **orderId**: A unique identifier for each order.
- **userId**: The ID of the user who placed the order, linking the order to a specific user.
- **items**: A list of items included in the order, where each item has an **itemId**, **quantity**, and **price**.
- **userEmails**: A list of email addresses associated with the order, synchronized with the user's registered email.

- **deliveryAddress**: The address where the order is to be shipped, similar to the one stored in the User Database.
- **orderStatus**: The status of the order, which can be one of three values: **under process**, **shipping**, or **delivered**.
- **createdAt** and **updatedAt**: Timestamps indicating when the order was created and last updated.

The `order_schema.json` defines the structure for these attributes, ensuring that the order data is organized and conforms to the expected format. The following JSON schema describes the order data model:

```
{
    "$schema":
    ↪  "http://json-schema.org/draft-07/schema#",
    "title": "Order",
    "type": "object",
    "properties": {
        "orderId": { "type": "string",
        ↪  "description": "The unique identifier
        ↪  for an order" },
        "userId": { "type": "string",
        ↪  "description": "The unique identifier
        ↪  for a user" },
        "items": {
            "type": "array",
            "items": {
                "type": "object",
                "properties": {
                    "itemId": { "type": "string",
                    ↪  "description": "The unique
                    ↪  identifier for an item" },
                    "quantity": { "type":
                    ↪  "integer", "minimum": 1,
                    ↪  "description": "Quantity of
                    ↪  the item ordered" },
                    "price": { "type": "number",
                    ↪  "minimum": 0,
                    ↪  "description": "Price of the
                    ↪  item" }
                },
                "required": ["itemId", "quantity",
                ↪  "price"]
            },
            "description": "List of items in the
            ↪  order"
        },
        "userEmails": {
            "type": "array",
            "items": { "type": "string", "format":
            ↪  "email" },
            "description": "A list of email
            ↪  addresses associated with the order"
        },
        "deliveryAddress": {
            "type": "object",
            "properties": {
                "street": { "type": "string",
                ↪  "description": "Street address"
                ↪  },
                "city": { "type": "string",
                ↪  "description": "City" },
                "state": { "type": "string",
                ↪  "description": "State" },
                "postalCode": { "type": "string",
                ↪  "description": "Postal code" },
                "country": { "type": "string",
                ↪  "description": "Country" }
            },
            "required": ["street", "city", "state",
            ↪  "postalCode", "country"],
            "description": "The delivery address of
            ↪  the user"
        },
        "orderStatus": {
            "type": "string",
            "enum": ["under process", "shipping",
            ↪  "delivered"],
            "description": "Current status of the
            ↪  order"
        },
        "createdAt": { "type": "string", "format":
        ↪  "date-time", "description": "Timestamp
        ↪  of when the order was created." },
        "updatedAt": { "type": "string", "format":
        ↪  "date-time", "description": "Timestamp
        ↪  of when the order was last updated." }
    },
    "required": ["orderId", "items", "userEmails",
    ↪  "deliveryAddress", "orderStatus"]
}
```

In summary, the **User Database** stores crucial user-related information, while the **Order Database** handles details related to the user's orders. Both schemas ensure the integrity and consistency of the data, providing a solid foundation for handling user and order management within the system.

III. API Gateway and Microservices APIs

The architecture employs the Kong API Gateway to manage routing between client requests and the microservices. The User and Order microservices expose REST APIs [8] to handle various operations, such as creating, updating, and retrieving data. This section describes the configuration of the API Gateway and the endpoints of the microservices.

A. API Gateway Configuration

The Kong API Gateway is used to route HTTP requests to the appropriate upstream microservices. The configuration includes separate upstreams for the User and Order microservices, with support for load balancing between different service versions. Key features of the configuration are as follows:

- **Load Balancing**: Requests are distributed based on weights assigned to targets in the upstream configurations. This is later used for migrating to a new version of the User Microservice.
- **Routing Rules**: Specific routes are defined for User and Order services using regular expressions and HTTP methods.
- **Service Definitions**: Each microservice is registered as a service in Kong, pointing to its respective upstream.

The `kong.yml` configuration file is shown below:

```
_format_version: "3.0"
_transform: true

upstreams:
  - name: user_service_upstream
    targets:
      - target: user-service-v1:5000
        weight: ${USER_SERVICE_V1_WEIGHT}
      - target: user-service-v2:5000
        weight: ${USER_SERVICE_V2_WEIGHT}
```

```
  - name: order_service_upstream
    targets:
      - target: order-service:5000
        weight: 100

services:
  - name: kong_order_service
    url: http://order_service_upstream
    protocol: http
    routes:
      - name: order_service_route
        paths:
          - "/orders"
          - "/orders/(?<order_id>[\\w-]+)/(?<action
          ↪  >(status|details))"
        strip_path: false
        preserve_host: true
        methods:
          - GET
          - POST
          - PUT

  - name: kong_user_service
    url: http://user_service_upstream
    protocol: http
    routes:
      - name: user_service_route
        paths:
          - "/users/"
          - "/users/(?<user_id>[\\w-]+)"
        strip_path: false
        methods:
          - POST
          - PUT
```

B. User Microservice REST APIs

The User microservice provides endpoints for creating new users and updating existing user information. It is implemented using Flask-RESTx [9] and validates all inputs to ensure data integrity. The following endpoints are available:

- **POST /users/**: Creates a new user. Validates the request payload for required fields, such as `emails` and `deliveryAddress`, and ensures no duplicate emails exist in the database.
- **PUT /users/<id>**: Updates an existing user's email addresses or delivery address. Publishes an event to synchronize the updated data across microservices.

The API route configuration for the User microservice is shown below:

```
@api.route('/')
class UserList(Resource):
    @api.expect(user_model)
    def post(self):
        """Create a new user"""
        ...

@api.route('/<string:id>')
class User(Resource):
    @api.expect(user_model)
    def put(self, id):
        """Update user information"""
        ...
```

The implementation includes:

- Input validation for required fields and correct data types.
- Integration with RabbitMQ for event-driven synchronization when user details are updated.
- Handling of duplicate emails with appropriate error responses.

C. Order Microservice REST APIs

The Order microservice provides endpoints for managing orders. It supports operations such as creating new orders, retrieving orders by status, and updating order details. The endpoints are as follows:

- **POST /orders/**: Creates a new order with a list of items, user emails, and delivery address.
- **GET /orders/**: Retrieves orders filtered by their status (`under process`, `shipping`, or `delivered`).
- **PUT /orders/<id>/status**: Updates the status of an existing order.
- **PUT /orders/<id>/details**: Updates the email addresses or delivery address associated with an order.

The Order microservice is implemented using Flask-RESTx, with features such as:

- Comprehensive input validation for order fields, including `items`, `userEmails`, and `deliveryAddress`.
- Support for filtering orders by status using query parameters.
- Robust error handling for invalid input and missing or non-existent orders.

The API routes for the Order microservice are illustrated in the following example configuration:

```
@api.route('/')
class OrderList(Resource):
    @api.expect(order_model)
    def post(self):
        """Create a new order"""
        ...

    @api.param('status', 'The status of the orders
    ↪  to retrieve')
    def get(self):
        """Retrieve orders by status"""
        ...

@api.route('/<string:id>/status')
class OrderStatus(Resource):
    @api.expect(order_status_model)
    def put(self, id):
        """Update the status of an order"""
        ...

@api.route('/<string:id>/details')
class OrderDetails(Resource):
    @api.expect(order_details_model)
    def put(self, id):
        """Update the email or delivery address of
        ↪  an order"""
        ...
```

D. API Testing

This subsection provides an overview of the API tests conducted using Insomnia for both the User and Order microservices. The tests validate key functionalities such

as creating, updating, and retrieving data through HTTP requests.

1) User Microservice Testing:

a) Create User:

- **Method:** POST
- **URL:** http://0.0.0.0:8000/users/ (Local) http://ec2-18-117-174-177.us-east-2.compute.amazonaws.com:8000/users/ (AWS)

```
{
  "firstName": "Jane",
  "lastName": "Smith",
  "emails": [
    "jane.smith@personal.com"
  ],
  "deliveryAddress": {
    "street": "456 Technology Boulevard, Apt 789",
    "city": "San Francisco",
    "state": "CA",
    "postalCode": "94107",
    "country": "USA"
  },
  "phoneNumber": "14155551234"
}
```

b) Update User by ID:

- **Method:** PUT
- **URL:** http://0.0.0.0:8000/users/u1 (Local) http://ec2-18-117-174-177.us-east-2.compute.amazonaws.com:8000/users/u1 (AWS)

```
{
    "emails": [
        "newemail@example.com",
        "anotheremail@example.com"
    ],
    "deliveryAddress": {
        "street": "123 New Street",
        "city": "New City",
        "state": "New State",
        "postalCode": "12345",
        "country": "New Country"
    }
}
```

2) Order Microservice Testing:

a) Create New Order:

- **Method:** POST
- **URL:** http://0.0.0.0:8000/orders/ (Local) http://ec2-18-117-174-177.us-east-2.compute.amazonaws.com:8000/orders/ (AWS)

```
{
  "userId": "08884b79-e16c-4976-a52e-ba1b2b62cfaa",
  "items": [
    {
      "itemId": "item001",
      "quantity": 2,
      "price": 29.99
    },
    {
      "itemId": "item002",
      "quantity": 1,
      "price": 49.99
    }
  ],
  "userEmails": [
    "jane.smith@personal.com",
    "jsmith@work.com"
  ],
  "deliveryAddress": {
    "street": "456 Technology Boulevard, Apt 789",
    "city": "San Francisco",
    "state": "CA",
    "postalCode": "94107",
    "country": "USA"
  },
  "orderStatus": "under process"
}
```

b) Update Order Status:

- **Method:** PUT
- **URL:** http://0.0.0.0:8000/orders/o1/status (Local) http://ec2-18-117-174-177.us-east-2.compute.amazonaws.com:8000/orders/o1/status (AWS)

```
{
  "orderStatus": "shipping"
}
```

c) Retrieve Orders by Status:

- **Method:** GET
- **URL:** http://0.0.0.0:8000/orders?status=under%20process (Local) http://ec2-18-117-174-177.us-east-2.compute.amazonaws.com:8000/orders?status=shipping (AWS)

E. Summary

The Kong API Gateway effectively routes requests to the microservices, while the User and Order microservices provide robust APIs for managing user and order data. Input validation, event-driven synchronization, and error handling ensure the consistency of the system.

IV. Event-Driven Sychronization System

The event-driven system is implemented to ensure synchronization between the User and Order databases when user data, such as email addresses or delivery addresses, is updated. This system leverages RabbitMQ as the message broker and uses the Python `pika` [10] library for event publishing and consumption. The following subsections detail the system's workflow, configuration, and implementation.

A. Architecture and Workflow

The event-driven architecture operates as follows:

1) When a PUT request updates the user's email or delivery address via the User microservice, an event is published to the RabbitMQ message broker.

2) The Order microservice subscribes to these events and updates the corresponding fields in the Order database.
3) This ensures that user-related information is synchronized across both microservices.

The system uses a durable queue bound to a direct exchange to guarantee reliable message delivery. Messages are acknowledged upon successful processing, ensuring no data loss.

B. RabbitMQ Configuration

The RabbitMQ service is deployed using Docker and configured with the following `docker-compose.yml`:

```
services:
  rabbitmq:
    image: rabbitmq:3-management
    container_name: rabbitmq-container
    hostname: rabbitmq
    ports:
      - "5673:5673" # RabbitMQ messaging port
      - "15672:15672" # RabbitMQ management UI
    expose:
      - 5673
      - 15672
    environment:
      RABBITMQ_DEFAULT_USER: ${RABBITMQ_USER}
      RABBITMQ_DEFAULT_PASS: ${RABBITMQ_PASSWORD}
      RABBITMQ_NODE_PORT: 5673 # Changed default
      ↪  port due to conflict
    healthcheck:
      test: rabbitmq-diagnostics
      ↪  check_port_connectivity
      interval: 30s
      timeout: 30s
      retries: 3
```

The RabbitMQ instance exposes the management UI on port 15672 and the messaging service on port 5673. Key environment variables, such as the username and password, are set using a `.env` file for secure access.

C. User Microservice: Event Publishing

The User microservice publishes events to RabbitMQ whenever a user's email or delivery address is updated. This is implemented in a dedicated module (`events.py`) with the following functionality:

- Establishes a connection to RabbitMQ using credentials and host information loaded from environment variables.
- Declares a direct exchange named `user_order` and a durable queue for user update events.
- Publishes a message containing the updated user data (`userId`, `userEmails`, and `deliveryAddress`) to the queue.
- Ensures the connection is gracefully closed after the message is published.

This ensures that updates to user data trigger a real-time event, making the information available to other microservices.

D. Order Microservice: Event Consumption

The Order microservice listens for user update events published by the User microservice. The event consumer, implemented in `events.py`, performs the following:

- Connects to RabbitMQ and binds to the durable queue for user update events.
- Listens for messages containing updated user information and processes them sequentially.
- Updates all matching orders in the Order database with the new email addresses and delivery address extracted from the message payload.
- Acknowledges the message upon successful processing to ensure reliable delivery.

The consumer handles errors gracefully, logging any failures and ensuring unacknowledged messages remain in the queue for reprocessing.

E. Shared Configuration for RabbitMQ

Both the User and Order microservices share a configuration module (`rabbitmq_config.py`) for establishing RabbitMQ connections and managing queues. The module provides:

- A reusable function to establish a connection with RabbitMQ using credentials and host details from environment variables.
- A function to create and configure queues and exchanges, ensuring consistent setup across microservices.

This modular approach enhances reusability and ensures consistency in RabbitMQ configurations across the system.

F. Summary

The event-driven system ensures data consistency between the User and Order databases by leveraging RabbitMQ for message-based communication. The User microservice acts as a publisher, broadcasting update events, while the Order microservice acts as a subscriber, consuming these events to update its database. The use of durable queues, direct exchanges, and Python's `pika` library ensures reliability and efficiency in handling user updates.

V. User Microservice New Version (v2)

The User microservice has been updated to version 2 (v2), introducing some simple improvements and refactorings. One significant change is the automatic handling of the `createdAt` and `updatedAt` fields, which were previously set manually. This change improves consistency and reduces the likelihood of errors when creating or updating user data.

The major enhancement in version 2 of the User microservice is the automatic assignment of the `createdAt` and `updatedAt` timestamps when a

new user is created or an existing user is updated. The following snippet highlights the key sections of the updated `routes.py` file in the User microservice:

```
# Automatically set createdAt and updatedAt fields
↪  during user creation
current_time = datetime.utcnow()
data['createdAt'] = current_time
data['updatedAt'] = current_time

# Automatically set updatedAt field during user
↪  update
current_time = datetime.utcnow()
data['updatedAt'] = current_time
```

In this version, both user creation and update operations ensure that the timestamps are handled consistently, reducing manual errors and ensuring data integrity.

A. User Creation (POST /users/)

In version 2 of the User microservice, when a new user is created, the following changes are made to the process:

- The `createdAt` and `updatedAt` fields are automatically set to the current UTC time when the user is created.
- The user ID (`userId`) is automatically generated using UUID.

The code for creating a new user automatically assigns the `createdAt` and `updatedAt` fields as shown below:

- `createdAt` and `updatedAt` are set to the current time using the `datetime.utcnow()` method.
- The user data is then inserted into the database, ensuring that these fields are stored consistently.

B. User Update (PUT /users/<id>)

In version 2, when updating user information, the following improvements are introduced:

- The `updatedAt` field is automatically updated to the current UTC time whenever any change is made to the user data.
- If there are changes to the user data, the event-driven system publishes a synchronization event to keep the Order microservice in sync with the updated user data.

The code for updating a user ensures that the `updatedAt` field is automatically updated:

- The `updatedAt` field is set to the current UTC time using the `datetime.utcnow()` method.
- The updated data is then persisted in the database, with the updated timestamp reflecting the latest change.

VI. Strangler Pattern In The API Gateway

The strangler pattern is used to gradually replace an old system with a new one. In this case, the old version of the User microservice (v1) is replaced by the new version (v2) over time. The API Gateway, Kong, facilitates this by routing a specified percentage of the requests to v1 and the remaining requests to v2, allowing the transition to occur incrementally. The percentage (P) can be dynamically adjusted through a configuration file, ensuring flexibility without requiring hardcoding within the API Gateway.

A. Overview of the Strangler Pattern

In the context of the User microservice, the strangler pattern allows the following:

- A certain percentage (P) of user requests are routed to the legacy version of the service (v1).
- The remaining percentage (1-P) of requests are routed to the new version (v2).
- The proportion (P) can be dynamically specified without requiring a restart or code changes in the API Gateway.

This approach ensures that the new version (v2) of the microservice is gradually rolled out, allowing for testing and verification in production without fully cutting over from v1 to v2 immediately.

B. Implementation Details

The strangler pattern is implemented in the Kong API Gateway configuration. The routing logic is determined based on the weight assigned to each version of the User microservice. The weights are dynamically calculated using an environment variable, `P_VALUE`, which specifies the percentage of requests to route to v1.

The Kong configuration file (`kong.yml`) is updated with the appropriate weights for both versions of the User service. The weight for v1 is set to the value of `P_VALUE`, while the weight for v2 is calculated as `100 - P_VALUE`. This allows Kong to route requests proportionally based on the specified weight.

C. Kong Configuration for Strangler Pattern

The Kong configuration file (`kong.yml`) is defined as follows, where the weights for `user-service-v1` and `user-service-v2` are dynamically set based on the environment variable `P_VALUE`:

```
_format_version: "3.0"
_transform: true

upstreams:
  - name: user_service_upstream  # Single upstream
  ↪  for user service
    targets:
      - target: user-service-v1:5000
        weight: ${USER_SERVICE_V1_WEIGHT}
      - target: user-service-v2:5000
        weight: ${USER_SERVICE_V2_WEIGHT}
```

```
  - name: order_service_upstream
    targets:
      - target: order-service:5000
        weight: 100

services:

  - name: kong_user_service
    url: http://user_service_upstream
    protocol: http
    routes:
      - name: user_service_route
        paths:
          - "/users/"
          - "/users/(?<user_id>[\\w-]+)"
        strip_path: false
        methods:
          - POST
          - PUT
```

D. Dynamic Weight Calculation via Docker Entry Point

The weights for v1 and v2 are calculated dynamically using the `P_VALUE` environment variable, which specifies the percentage of requests to be routed to v1. This environment variable is set in the `docker-entrypoint.sh` script, which runs during the deployment of the API Gateway.

The `docker-entrypoint.sh` script performs the following tasks:

- Verifies that the `P_VALUE` environment variable is set.
- Calculates the weights for v1 and v2 based on `P_VALUE`.
- Substitutes the calculated weights into the Kong configuration template (`kong.yml.template`) using the `envsubst` command.
- Writes the final configuration file (`kong.yml`) and starts the Kong API Gateway with the updated configuration.

Below is the script that calculates the weights and updates the Kong configuration:

```
#!/bin/bash
set -e
...
# Set weights
export USER_SERVICE_V1_WEIGHT=$P_VALUE
export USER_SERVICE_V2_WEIGHT=$((100 - P_VALUE))
...
```

VII. Database Design

The database design for this system utilizes MongoDB, a NoSQL database deployed using Docker. MongoDB provides flexibility with schema-less data models, making it an ideal choice for managing dynamic data such as user and order information. The database is set up with collections for users and orders, each with its own validation schema to ensure data integrity.

A. MongoDB Deployment in Docker

MongoDB is deployed as a containerized service using Docker, as defined in the `docker-compose.yml` file. The MongoDB service is configured with the necessary environment variables, including the database name, username, and password, all loaded from the `.env` file. The service is exposed on port 27017, and a volume is used to persist the data.

In addition to the MongoDB service, a `mongodb-setup` service is defined, which is responsible for setting up the database and seeding it with test data. This service runs Python scripts to initialize the database schema and populate it with sample data.

B. Database Initialization and Schema Validation

The database consists of two primary collections: `users` and `orders`. Each collection has a JSON schema that validates the structure of the documents stored in it, ensuring that only valid data is inserted.

1) Users Collection: The `users` collection is initialized with schema validation to enforce the following structure:

- `userId`: A unique identifier for each user (string).
- `emails`: An array of email addresses, each validated against an email pattern.
- `deliveryAddress`: An object containing the user's address details, including `street`, `city`, `state`, `postalCode`, and `country`.
- `phoneNumber`: A string that must match a specific pattern (optional).
- `createdAt` and `updatedAt`: Date fields that are automatically set when a user is created or updated.

The schema ensures that documents are validated before they are inserted into the collection, and any invalid data is rejected. If the collection already exists, an exception is caught and an error message is displayed.

2) Orders Collection: The `orders` collection also uses a schema validator that enforces the following structure:

- `orderId`: A unique identifier for each order (string).
- `userEmails`: An array of email addresses associated with the order.
- `deliveryAddress`: An object with the same structure as in the `users` collection.
- `items`: An array of items in the order, each with properties `itemId`, `quantity`, and `price`.
- `orderStatus`: A string that must be one of `["under process", "shipping", "delivered"]`.
- `createdAt` and `updatedAt`: Date fields for tracking order creation and updates.

This collection is also validated using a schema to ensure the integrity of the data.

C. Database Setup and Seeding

The `mongodb-setup` service uses two Python scripts to initialize and seed the MongoDB database with test data. These scripts are run when the Docker container starts.

1) setup_mongo.py: The `setup_mongo.py` script sets up the database collections for `users` and `orders`. It connects to MongoDB using credentials from the `.env` file and initializes the collections with schema validation. If the collections already exist, the script handles the error gracefully.

2) seed_database.py: The `seed_database.py` script seeds the `users` and `orders` collections with sample data. It generates random user details, including emails and delivery addresses, and inserts them into the `users` collection. Similarly, it generates sample orders, associates them with users, and inserts them into the `orders` collection.

The sample data includes:

- `users`: A list of 5 users, each with a unique `userId`, emails, and delivery address.
- `orders`: A list of 15 orders, each associated with a user and containing 1 to 3 items, along with the order status and timestamps.

The script uses the `datetime.utcnow()` method to ensure the `createdAt` and `updatedAt` fields are populated with the current UTC time.

VIII. Docker Deployment Strategy

In this system, the microservice deployment strategy is used, where each component—such as the API Gateway, microservices, and event-driven system (RabbitMQ)—is deployed independently with its own Docker container. This approach provides greater flexibility, scalability, and maintainability compared to a monolithic deployment, as each service can be developed, scaled, and updated independently.

A. Microservice Architecture

The microservice architecture in this deployment is designed to provide modularity and isolate the different components of the system. Each service is contained in its own Docker container, and Docker Compose is used to manage the entire multi-container setup. The services include:

- **API Gateway (Kong):** The API Gateway is responsible for routing requests to the appropriate microservices (User services and Order service). It also handles load balancing and can dynamically route a specified percentage of requests to different versions of the User service (v1 and v2), based on the `P_VALUE` configuration.
- **User Microservice v1 and v2:** Two versions of the User service (v1 and v2) are deployed to allow for incremental migration from the older version to the new version. Requests are routed to these services dynamically based on the specified weights.
- **Order Service:** The Order service is responsible for managing orders, interacting with the User services to retrieve user data and ensure that order processing is carried out correctly.
- **RabbitMQ (Event-Driven System):** RabbitMQ facilitates communication between services, enabling the event-driven architecture that synchronizes the User and Order services when user data is updated.
- **MongoDB:** MongoDB is used as the database for both the User and Order services, with a setup container to initialize the database and seed it with test data.

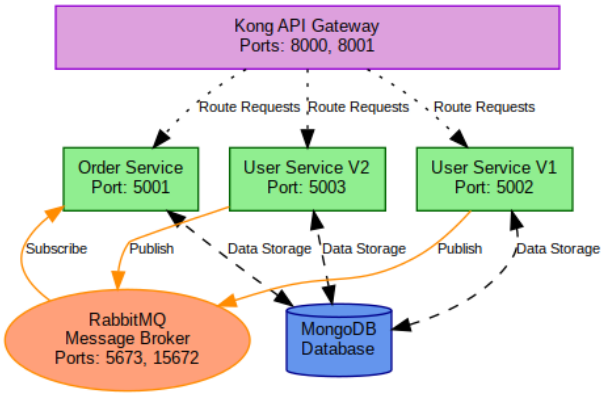

Fig. 2. Microservice Deployment Architecture

Fig. 2 shows the deployed architecture used in my Docker setup.

B. Docker-Compose Configuration

The system is defined in a single `docker-compose.yml` file, where each service is declared with its own configuration. The configuration allows for easy management of service dependencies and ensures that each service runs in its own isolated environment.

Below is the key configuration for the Docker setup:

```
services:
  mongodb:
    image: mongo:latest
    container_name: mongo-container
    environment:
      - MONGO_INITDB_DATABASE=${DATABASE_NAME}
      - MONGO_INITDB_ROOT_USERNAME=${MONGO_USERNAME}
      - MONGO_INITDB_ROOT_PASSWORD=${MONGO_PASSWORD}
    ports:
      - "27017:27017"
    volumes:
      - mongodb_data:/data/db

  mongodb-setup:
    image: python:3.10-slim
    container_name: mongo-setup
```

```
    build:
      context: .
      dockerfile:
      ↪  src/shared/config/mongodb/Dockerfile
    depends_on:
      - mongodb
    env_file:
      - .env
    links:
      - mongodb

  order-service:
    build:
      context: ./src
      dockerfile: order_service/Dockerfile
    container_name: order-service
    environment:
      - FLASK_APP=order_service.wsgi:app
      - MONGO_URI=${MONGO_URI}
    ports:
      - "5001:5000"
    depends_on:
      - rabbitmq

  user-service-v1:
    build:
      context: ./src
      dockerfile: user_service_v1/Dockerfile
    container_name: user-service-v1
    environment:
      - FLASK_APP=user_service_v1.wsgi:app
      - MONGO_URI=${MONGO_URI}
    ports:
      - "5002:5000"
    depends_on:
      - rabbitmq

  user-service-v2:
    build:
      context: ./src
      dockerfile: user_service_v2/Dockerfile
    container_name: user-service-v2
    environment:
      - FLASK_APP=user_service_v2.wsgi:app
      - MONGO_URI=m${MONGO_URI}
    ports:
      - "5003:5000"
    depends_on:
      - rabbitmq

  rabbitmq:
    image: rabbitmq:3-management
    container_name: rabbitmq-container
    ports:
      - "5673:5673"
      - "15672:15672"
    environment:
      RABBITMQ_DEFAULT_USER: ${RABBITMQ_USER}
      RABBITMQ_DEFAULT_PASS: ${RABBITMQ_PASSWORD}

  kong:
    image: kong:latest
    container_name: api-gateway
    build:
      context: ./src/api_gateway
      dockerfile: Dockerfile
    environment:
      - KONG_PROXY_LISTEN=0.0.0.0:8000
      - KONG_ADMIN_LISTEN=0.0.0.0:8001
      - P_VALUE=${P_VALUE}
    ports:
      - "8000:8000"
      - "8001:8001"
    depends_on:
      - order-service
      - user-service-v1

volumes:
  mongodb_data:
```

IX. Cloud Deployment to Amazon EC2

To deploy the application on the cloud, an Amazon EC2 instance running Linux was used. The deployment process involved setting up the Docker containers for the microservices and configuring the necessary network settings to ensure that the APIs are accessible via cloud endpoints.

A. Amazon EC2 Setup

The EC2 instance was launched with Ubuntu 22.04 LTS, providing a stable environment for running Docker and deploying the application. The instance is equipped with an appropriate security group to manage inbound and outbound network traffic. The following security group rules were configured to allow traffic to the relevant services:

- **Kong Admin API:** Port 8001 is open for managing the Kong API Gateway, allowing administrative access.
- **Kong API Gateway:** Port 8000 is open for external users to access the services via the Kong API Gateway.
- **SSH Access:** Port 22 is open for secure shell access to the instance for management and debugging purposes.

The security group ensures that these services are exposed to the internet, while also restricting access where necessary, such as limiting SSH access to trusted IPs.

Once the EC2 instance was set up and Docker was installed, the application was deployed using the provided `docker-compose.yml` configuration. The following steps were taken to ensure the APIs were accessible:

- **Kong API Gateway** was configured to listen on port 8000, making the microservices accessible to external clients via HTTP requests.
- The **Konga UI** was exposed on port 1337, providing a management interface for monitoring and configuring the Kong API Gateway.
- The **Kong Admin API** was exposed on port 8001 to allow for administrative access and configuration of Kong, facilitating the management of API routing and settings.

With the EC2 instance running and the Docker containers deployed, external clients can now interact with the APIs via the public IP of the EC2 instance and the open ports. HTTP requests are routed through the Kong API Gateway, which handles load balancing between the User microservice versions (v1 and v2) and the Order service.

X. CI/CD Using GitHub Actions *(BONUS)*

To automate the deployment process, a CI/CD pipeline was set up using GitHub Actions. This pipeline ensures

that any changes pushed to the main branch of the repository trigger an automated deployment to the Amazon EC2 instance. The pipeline involves several key steps, including repository checkout, SSH setup, and remote execution of deployment scripts.

A. GitHub Actions Configuration

The CI/CD pipeline is defined in the `ci-cd.yml` file, which is located in the `.github/workflows` directory of the GitHub repository. This configuration file defines the workflow that is triggered on a push to the `main` branch (or any branch specified in the configuration).

The following is the content of the `ci-cd.yml` file:

```
name: CI/CD Pipeline

on:
  push:
    branches:
      - main  # Change to your default branch if
      ↪  different

jobs:
  deploy:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout Repository
        uses: actions/checkout@v3

      - name: Set up SSH
        uses: webfactory/ssh-agent@v0.7.0
        with:
          ssh-private-key: ${{ secrets.EC2_SSH_KEY
          ↪  }}

      - name: Add Known Hosts
        run: |
          mkdir -p ~/.ssh
          ssh-keyscan -H ${{ secrets.EC2_HOST }} >>
          ↪  ~/.ssh/known_hosts

      - name: Deploy to EC2
        run: |
          ssh ${{ secrets.EC2_USER }}@${{
          ↪  secrets.EC2_HOST }} 'bash ~/deploy.sh'
```

B. Explanation of the CI/CD Workflow

The workflow consists of several stages, each designed to automate a specific step in the deployment process:

- **Checkout Repository:** The pipeline starts by checking out the code from the repository using the `actions/checkout` GitHub Action.
- **Set up SSH:** The next step involves setting up SSH authentication by using the `webfactory/ssh-agent` GitHub Action. This action uses an SSH private key stored in the GitHub secrets (`EC2_SSH_KEY`) to authenticate with the EC2 instance.
- **Add Known Hosts:** To prevent SSH from prompting for confirmation when connecting to the EC2 instance, the `ssh-keyscan` command is run to add the EC2 host's SSH fingerprint to the `known_hosts` file. The EC2 host's IP address or domain name is retrieved from the GitHub secrets (`EC2_HOST`).
- **Deploy to EC2:** Finally, the deployment is executed by SSH-ing into the EC2 instance as the user specified in `EC2_USER`. The `deploy.sh` script located on the EC2 instance is triggered to perform the deployment tasks, such as pulling the latest code and restarting Docker containers.

C. GitHub Secrets for Secure Access

To ensure the security of the deployment process, several secrets are used in the GitHub repository. These secrets are stored in the GitHub repository's secrets settings and are used to securely pass sensitive information to the CI/CD pipeline:

- `EC2_SSH_KEY`: The private SSH key used to authenticate with the EC2 instance. This key is securely stored in GitHub secrets and is never exposed in the codebase.
- `EC2_USER`: The username used to log into the EC2 instance.
- `EC2_HOST`: The IP address or hostname of the EC2 instance, which is used to establish an SSH connection.

By utilizing GitHub secrets, sensitive information is securely managed and kept out of the codebase, ensuring that the deployment process is both secure and automated.

XI. Exposing APIs to SwaggerHub *(BONUS)*

To provide a standardized and interactive documentation for the APIs, the APIs of the **Order Service** and **User Service** were exposed using SwaggerHub. This was achieved by generating Swagger API definitions in the form of `swagger.json` files using the `flask-restx` module and importing these definitions to SwaggerHub for easy access and integration.

A. Swagger API Definitions

For each microservice, Swagger API definitions were manually created using the `swagger.yaml` format, which describes the API endpoints, request/response formats, and possible status codes. Below are the Swagger YAML configurations for both the **Order Service** and the **User Service**.

1) Order Service API - `order-swagger.yaml`: The Swagger definition for the **Order Service** includes detailed documentation for handling orders, including operations for retrieving, creating, updating, and modifying order details and status. Here is an overview of the `order-swagger.yaml` file:

```yaml
swagger: "2.0"
info:
  version: "1.0"
  title: API
basePath: /
tags:
  - name: orders
    description: Order related operations
paths:
  /orders/:
    get:
      tags:
        - orders
      summary: Retrieves orders by status
      parameters:
        - name: status
          in: query
          description: The status of the orders to
          ↪  retrieve
        - name: X-Fields
          in: header
          description: Optional fields mask
      responses:
        "200":
          description: Success
          schema:
            type: array
            items:
              $ref: '#/definitions/Order'
    post:
      tags:
        - orders
      summary: Creates a new order
      parameters:
        - in: body
          name: payload
          required: true
          schema:
            $ref: '#/definitions/Order'
      responses:
        "201":
          description: Success
          schema:
            $ref: '#/definitions/Order'
  /orders/{id}/status:
    put:
      tags:
        - orders
      summary: Updates the status of an existing
      ↪  order
      parameters:
        - name: id
          in: path
          required: true
          type: string
        - in: body
          name: payload
          required: true
          schema:
            $ref: '#/definitions/OrderStatus'
      responses:
        "200":
          description: Success
          schema:
            $ref: '#/definitions/Order'
        "404":
          description: Order not found
definitions:
  Order:
    type: object
    required:
      - orderId
      - orderStatus
      - userEmails
      - deliveryAddress
    properties:
      orderId:
        type: string
      orderStatus:
        type: string
        enum:
          - under process
          - shipping
          - delivered
      userEmails:
        type: array
        items:
          type: string
      deliveryAddress:
        $ref: '#/definitions/Order_deliveryAddress'
  OrderStatus:
    type: object
    required:
      - orderStatus
    properties:
      orderStatus:
        type: string
        enum:
          - under process
          - shipping
          - delivered
```

2) User Service API - `user-swagger.yaml`: The Swagger definition for the **User Service** covers operations related to user management, such as creating and updating user information. Below is an overview of the `user-swagger.yaml` file:

```yaml
swagger: "2.0"
info:
  version: "1.0"
  title: API
basePath: /
tags:
  - name: users
    description: User related operations
paths:
  /users/:
    post:
      tags:
        - users
      summary: Creates a new user
      parameters:
        - in: body
          name: payload
          required: true
          schema:
            $ref: '#/definitions/User'
      responses:
        "201":
          description: Success
          schema:
            $ref: '#/definitions/User'
  /users/{id}:
    put:
      tags:
        - users
      summary: Updates user information based on
      ↪  user ID
      parameters:
        - name: id
          in: path
          required: true
          type: string
        - in: body
          name: payload
          required: true
          schema:
            $ref: '#/definitions/User'
      responses:
        "200":
          description: Success
          schema:
            $ref: '#/definitions/User'
        "404":
          description: User not found
definitions:
  User:
```

```
    type: object
    required:
      - userId
      - emails
      - deliveryAddress
    properties:
      userId:
        type: string
      firstName:
        type: string
      lastName:
        type: string
      emails:
        type: array
        items:
          type: string
      deliveryAddress:
        $ref: '#/definitions/User_deliveryAddress'
  User_deliveryAddress:
    type: object
    required:
      - street
      - city
      - state
      - postalCode
      - country
    properties:
      street:
        type: string
      city:
        type: string
      state:
        type: string
      postalCode:
        type: string
      country:
        type: string
```

Once the API definitions were generated, they were imported into SwaggerHub for easier documentation and testing.

XII. Conclusion

This project demonstrated the implementation of a robust **microservices architecture** using modern tools and technologies such as **Kong**, **Docker**, **Flask**, **RabbitMQ**, **MongoDB**, and **Amazon EC2**, with a strong emphasis on scalability, flexibility, and security. The system consists of independently deployable **microservices**, including the **User Service**, **Order Service**, and an **Event-Driven System** facilitated by **RabbitMQ**. The **Kong API Gateway** routes requests to the appropriate services, with versioning managed through a **Strangler Pattern** to ensure smooth transitions between service versions.

The system was deployed on an **Amazon EC2 instance**, allowing it to be accessed via public cloud endpoints, while ensuring security through appropriately configured **security groups**. A **continuous integration and continuous deployment (CI/CD)** pipeline was set up using **GitHub Actions**, automating the process of testing and deploying changes to the cloud.

API documentation and testing were made easy by exposing the **User** and **Order** microservices to **SwaggerHub**, where interactive API definitions were created using the `swagger.yaml` format and imported for testing and collaboration.

By employing **Docker** for service containerization, **MongoDB** for data storage, and **RabbitMQ** for event-driven communication, the architecture is highly maintainable and scalable. The integration of modern deployment strategies, including **cloud services** and **CI/CD pipelines**, ensures that the system remains reliable, secure, and easily adaptable to future changes.

In conclusion, the combination of **microservices**, **cloud deployment**, **CI/CD pipelines**, and **API documentation** creates a robust foundation for scalable and maintainable web applications, while ensuring smooth operations and streamlined development processes.

References

[1] (2024) Flask framework. Accessed: Dec. 3, 2024. [Online]. Available: https://flask.palletsprojects.com

[2] (2024) Mongodb. Accessed: Dec. 3, 2024. [Online]. Available: https://www.mongodb.com

[3] (2024) Rabbitmq. Accessed: Dec. 3, 2024. [Online]. Available: https://www.rabbitmq.com

[4] (2024) Kong api gateway. Accessed: Dec. 3, 2024. [Online]. Available: https://konghq.com/products/kong-gateway

[5] (2024) Docker. Accessed: Dec. 3, 2024. [Online]. Available: https://www.docker.com

[6] (2024) Amazon ec2. Accessed: Dec. 3, 2024. [Online]. Available: https://aws.amazon.com/ec2

[7] (2024) Swaggerhub. Accessed: Dec. 3, 2024. [Online]. Available: https://swagger.io/tools/swaggerhub/

[8] R. T. Fielding and R. N. Taylor, *Principled Design of the Modern Web Architecture*. ACM Transactions on Internet Technology (TOIT), 2000, vol. 2, no. 2, accessed: Dec. 3, 2024. [Online]. Available: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

[9] (2024) Flask-restx. Accessed: Dec. 3, 2024. [Online]. Available: https://flask-restx.readthedocs.io/

[10] (2024) Pika: Python rabbitmq client. Accessed: Dec. 3, 2024. [Online]. Available: https://pika.readthedocs.io/en/stable/