MICROSERVICE BASED USER AND ORDER MANAGEMENT

Samantha Menezes
Department of Electrical and Computer Engineering
Concordia University
Montreal, Canada

***Abstract*—Microservices have become increasingly popular due to their ability to enable independent service development, reduce code interdependencies, and improve modularity and maintainability within software systems. Building microservices-based architectures typically involves integrating multiple technologies that must communicate reliably while adhering to the CAP theorem and ensuring consistent data synchronization.**

This project presents the design and implementation of a microservices architecture that supports synchronized user and order data using Kong API Gateway, FastAPI, RabbitMQ, and MongoDB.The system includes two primary microservices: a User Service that manages user information, and an Order Service that maintains order-related data. Shared fields—such as email and delivery address—are synchronized using an event-driven mechanism powered by RabbitMQ.The architecture utilizes Kong as the API Gateway to provide routing, version control, and traffic management, enabling seamless gradual deployment of new service versions using the strangler pattern. Both microservices are containerized with Docker and deployed in a unified environment to simplify development and testing. The work further explores RESTful API design, database schema modeling, event-handling workflows, and a multithreaded RabbitMQ consumer within the FastAPI application.Overall, the solution ensures scalability, modularity, and maintainability, while adhering to modernn software engineering practices.

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

I. Introduction

An application could be developed as a monolith or as a microservice. With monolithic architectures, all processes are tightly coupled and run as a single service. This means that if one process of the application experiences a spike in demand, the entire architecture must be scaled.Adding or improving a monolithic application's features becomes more complex as the code base grows. This complexity limits experimentation and makes it difficult to implement new ideas.Monolithic architectures add risk for application availability because many dependent and tightly coupled processes increase the impact of a single process failure.

With a microservices architecture, an application is built as independent components that run each application process as a service. These services communicate via a well-defined interface using lightweight APIs.Services are built for business capabilities and each service performs a single function. Because they are independently run, each service can be updated, deployed, and scaled to meet demand for specific functions of an application.

This is submitted as part of Assignment 2 - Programming on the Cloud

II. System Architecture

Given below is the System Architecture diagram. The system architecture consists of two versions of the User Microservice and a single Order Microservice, all developed using FastAPI. MongoDB is used as the primary datastore, with two separate databases: `user_db`, which contains the `users` collection, and `order_db`, which contains the `orders` collection. Service-to-service communication is facilitated through RabbitMQ, which acts as an event-driven messaging layer to maintain data consistency across the microservices.

An API Gateway, also built with FastAPI, applies the Strangler Pattern to gradually transition traffic between User Service v1 and v2. The gateway not only determines which version handles each request but also allows the user to configure the proportion of requests routed to each service at runtime.Each microservice is containerized using Docker, and a unified docker-compose configuration orchestrates the entire system. The full setup is deployed on an Amazon EC2 virtual machine, enabling external API access. To enhance reliability and maintainability, CI/CD pipelines were integrated to automate build, deployment, and update processes.

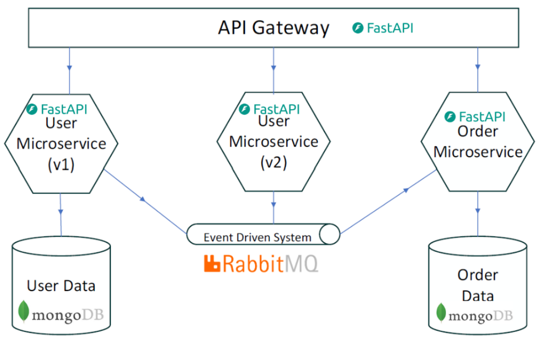


Fig. 1. System Architecture

III. Data Models Design

The system uses two separate databases—one for User data and one for Order data—each responsible for managing critical information.Both databases follow predefined JSON Schemas to enforce structure, while additional runtime validation is performed using Pydantic models within the FastAPI services to ensure correctness and data integrity before any record is stored or updated.

A. *User Database*

The User Database contains all user-related information required by the microservices.It stores identity details and contact information. The primary fields stored in the User Database include:

- **userId**: A unique identifier generated by the backend for each user.
- **firstName** and **lastName**: The user's first and last names.
- **emails**: A list of email addresses associated with the user. At least one email must be provided.
- **deliveryAddress**: The user's delivery location, containing the fields `street`, `city`, `province`, `postalCode`, and `country`.
- **phoneNumber**: An optional contact number provided by the user.

The user submits a JSON payload in the format shown below. Field validation is performed using Pydantic to ensure that mandatory fields are provided and conform to the expected structure.

```
{
  "firstName": "string",
  "lastName": "string",
  "phoneNumber": "string",
  "emails": [
    "string"
  ],
  "deliveryAddress": {
    "street": "string",
    "city": "string",
    "province": "string",
    "postalCode": "string",
    "country": "string"
  }
}
```

The backend service automatically generates a unique `userId` for each new user before storing the final record in the MongoDB collection.

B. *Order Database*

The Order Database stores information related to the orders placed by users.

The primary fields stored in the User Database include:

- **orderId**: A unique identifier generated by the backend for each order.
- **userId**: The identifier of the user placing the order.This value corresponds to the `userId` returned when the user was originally created.
- **items**: A list of items included in the order. Each item contains:
  - **itemId**: The unique identifier for the item.
  - **quantity**: The number of units of the item being ordered.
  - **price**: The price of a single unit of the item.
- **emails**: A list of email addresses associated with the order.
- **deliveryAddress**: The address where the order will be delivered, consisting of the fields `street`, `city`, `province`, `postalCode`, and `country`.
- **orderStatus**: The current status of the order, which can be one of three values: `under process`, `shipping`, or `delivered`.

The user submits a JSON payload in the format shown below. Field validation is performed using Pydantic to ensure that mandatory fields are provided and conform to the expected structure.

```
{
  "userId": "string",
  "items": [
    {
      "itemId": "string",
      "quantity": 1,
      "price": 0
    }
  ],
  "emails": [
    "string"
  ],
  "deliveryAddress": {
    "street": "string",
    "city": "string",
    "province": "string",
    "postalCode": "string",
    "country": "string"
  },
  "orderStatus": "string"
}
```

The backend service automatically generates a unique `orderId` for each new user before storing the final record in the MongoDB collection.

IV. API Gateway and Microservices APIs

The architecture employs the FastAPI Gateway to manage routing between client requests and the microservices. The User and Order microservices expose REST APIs to handle various operations, such as creating, updating, and retrieving data.

A. *API Gateway Configuration*

The API Gateway serves as the single unified entry point for all external requests to the system. The API Gateway is responsible for inspecting each request, determining the correct backend service, and routing the request accordingly. This allows the legacy User Service v1 and the new User Service v2 to coexist while traffic is gradually shifted to v2 without breaking any clients. This design significantly simplifies client interaction, improves maintainability, and enhances overall system security.

The gateway is published as port `8000` externally.The `docker-compose.yml` of the gateway service is given below.

```
  api_gateway:
  build:
    context: ./src
    dockerfile: api_gateway/Dockerfile
  container_name: api_gateway
  env_file:
  - .env
  ports:
    - "8000:8080"
  volumes:
    - ./src/api_gateway/config.json:
    /app/api_gateway/config.json
  depends_on:
    - user_service_v1
    - user_service_v2
    - order_service
```

The API Gateway is configured to depend on the User v1, User v2, and Order services. This ensures that all essential microservices are fully initialized and running before the system begins accepting any external API requests. A JWT-based authentication layer can also be integrated at the gateway level to prevent unauthorized access to protected endpoints. Additionally, the mounted volume for the `config.json` file allows routing rules—such as traffic distribution percentages—to be updated dynamically without requiring a container rebuild, enabling real-time adjustments to the gateway's behavior.

B. User Microservice REST APIs

The User Microservice exposes two main endpoints for managing user information. These endpoints are implemented using FastAPI and enforce input validation through Pydantic before any data is stored or updated.

- **POST /users/**: Creates a new user. The request payload is validated to ensure that required fields such as `emails` and `deliveryAddress` are provided.
- **PUT /users/{id}**: Updates the email addresses or delivery address of an existing user. Upon a successful update, the microservice publishes an event to synchronize these changes with the Order Microservice.

The corresponding FastAPI route definitions are shown below:

```
@router.post("/")
async def create_user(request: Request, user:
↪   UserModel):
    """Create a new user"""

@router.put("/{id}")
async def update_user(id: str, request:
↪   Request, data: dict):
    """Allows only emails and deliveryAddress
     ↪   updates"""
```

C. Order Microservice REST APIs

The Order Microservice provides a set of endpoints for creating and managing orders. These endpoints are implemented using FastAPI and perform validation to ensure that only permitted fields and valid updates are processed.

- **POST /orders/**: Creates a new order containing the user identifier, list of items, associated email addresses, delivery address, and order status.
- **GET /orders/**: Retrieves all orders that match a specified status. The status value must be one of `under process`, `shipping`, or `delivered`.
- **PUT /orders/{id}/status**: Updates the status of an existing order. The request is validated to ensure that the new status is permitted.
- **PUT /orders/{id}/details**: Updates only the email addresses or delivery address associated with an order. Any field outside these two is rejected.

The corresponding FastAPI route definitions are shown below:

```
@router.post("/")
async def create_order(request: Request,
↪   order: OrderModel):
    """Create a new order"""

@router.get("/")
async def get_orders(request: Request, status:
↪   str):
    """Retrieve orders filtered by their
     ↪   status"""

@router.put("/{id}/status")
async def update_order_status(id: str,
↪   request: Request, data: dict):
    """Update the status of an existing
     ↪   order"""

@router.put("/{id}/details")
async def update_order_details(id: str,
↪   request: Request, data: dict):
    """Update the emails or delivery address
     ↪   of an order"""
```

V. API Testing

API testing was performed using Postman to verify the correctness of the User and Order microservices. The tests validate core functionalities such as creating resources, updating existing records, and retrieving filtered results. Representative test cases are summarised below.

A. User Microservice Testing

1) Create User:

Method: POST
URL: `http://18.224.212.122:8000/users/`
Request:

```
{
  "firstName": "Robert",
  "lastName": "Williams",
  "phoneNumber": "+1-438-222-7890",
  "emails": [
    "robert.williams@hotmail.com",
    "rob.w@gouv.qc.ca"
  ],
  "deliveryAddress": {
    "street": "456 Elm Avenue",
    "city": "Ottawa",
    "province": "ON",
    "postalCode": "K2P 1L4",
    "country": "Canada"
  }
}
```

Response:

```
{
   "status": "success",
   "user": {
      "_id": "692c9dff56f43543f996c217",
      "firstName": "Robert",
      "lastName": "Williams",
      "phoneNumber": "+1-438-222-7890",
      "emails": [
         "robert.williams@hotmail.com",
         "rob.w@gouv.qc.ca"
      ],
      "deliveryAddress": {
         "street": "456 Elm Avenue",
         "city": "Ottawa",
         "province": "ON",
         "postalCode": "K2P 1L4",
         "country": "Canada"
      },
      "userId": "8eab397f-6890-4c61-b3f8-4
          f6c099e4c90",
   }
}
```

2) Update User by ID:

Method: PUT
URL: http://18.224.212.122:8000/users/8eab397f-6890-4c61-b3f8-4f6c099e4c90
Request:

```
{
    "emails" :
    ↪  ["robert.williams@outlook.com"],
    "deliveryAddress": {
        "street": "99 Park Ave",
        "city": "Montreal",
        "province": "QC",
        "postalCode": "H3A 1B1",
        "country": "Canada"
    }
}
```

Response:

```
{
  "_id": "692c9dff56f43543f996c217",
  "firstName": "Robert",
  "lastName": "Williams",
  "phoneNumber": "+1-438-222-7890",
  "emails": [
    "robert.williams@outlook.com"
  ],
  "deliveryAddress": {
    "street": "99 Park Ave",
    "city": "Montreal",
    "province": "QC",
    "postalCode": "H3A 1B1",
    "country": "Canada"
  },
  "userId": "8eab397f-6890-4c61-b3f8-4
      f6c099e4c90"
```

```
}
```

B. Order Microservice Testing

1) Create New Order:

Method: POST
URL: `http://18.224.212.122:8000/orders/`
Request:

```
{
  "userId":
  ↪  "8eab397f-6890-4c61-b3f8-4f6c099e4c90",
  "items": [
    {"itemId": "Study table", "quantity":
    ↪  1, "price": 29.99},
    {"itemId": "Chair", "quantity" : 1,
    ↪  "price": 15.99}
  ],
  "emails":
  ↪  ["robert.williams@hotmail.com"],
  "deliveryAddress": {
    "street": "456 Elm Avenue",
    "city": "Ottawa",
    "province": "ON",
    "postalCode": "K2P 1L4'",
    "country": "Canada"
  },
  "orderStatus": "under process"
}
```

Response:

```
{
   "status": "success",
   "order": {
      "_id": "692ca1d4bb0d73e43e9714a7",
      "userId": "8eab397f-6890-4c61-b3f8-4
          f6c099e4c90",
      "items": [
         {
            "itemId": "Study table",
            "quantity": 1,
            "price": 29.99
         },
         {
            "itemId": "Chair",
            "quantity": 1,
            "price": 15.99
         }
      ],
      "emails": [
         "robert.williams@hotmail.com"
      ],
      "deliveryAddress": {
         "street": "456 Elm Avenue",
         "city": "Ottawa",
         "province": "ON",
         "postalCode": "K2P 1L4'",
         "country": "Canada"
      },
      "orderStatus": "under process",
      "orderId": "9ae31dee-8e10-491e-bea9-2
          ada279e49cf",
      "createdAt": "2025-11-30T19
          :58:12.053000",
      "updatedAt": "2025-11-30T21
          :01:23.450000"
   }
}
```

2) Retrieve Orders by Status:

Method: GET
URL: http://18.224.212.122:8000/orders?status=under%20process
Response:

```
{
   "status": "success",
   "orders": [
      {
         "_id": "692ca1d4bb0d73e43e9714a7",
         "userId": "8eab397f-6890-4c61-b3f8-4
             f6c099e4c90",
         "items": [
            {
               "itemId": "Study table",
               "quantity": 1,
               "price": 29.99
            },
            {
               "itemId": "Chair",
               "quantity": 1,
               "price": 15.99
            }
         ],
         "emails": [
            "robert.williams@hotmail.com"
         ],
         "deliveryAddress": {
            "street": "456 Elm Avenue",
            "city": "Ottawa",
            "province": "ON",
            "postalCode": "K2P 1L4'",
            "country": "Canada"
         },
         "orderStatus": "under process",
         "orderId": "9ae31dee-8e10-491e-bea9-2
             ada279e49cf",
         "createdAt": "2025-11-30T19
             :58:12.053000",
         "updatedAt": "2025-11-30T19
             :58:12.053000"
      }
   ]
}
```

3) Update Order Status:

Method: PUT
URL: http://18.224.212.122:8000/orders/9ae31dee-8e10-491e-bea9-2ada279e49cf/status
Request:

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

Response:

```
{
   "status": "success",
   "after": {
      "_id": "692ca1d4bb0d73e43e9714a7",
      "userId": "8eab397f-6890-4c61-b3f8-4
          f6c099e4c90",
      "items": [
         {
            "itemId": "Study table",
            "quantity": 1,
            "price": 29.99
         },
         {
            "itemId": "Chair",
            "quantity": 1,
            "price": 15.99
         }
      ],
      "emails": [
         "robert.williams@hotmail.com"
      ],
      "deliveryAddress": {
         "street": "456 Elm Avenue",
         "city": "Ottawa",
         "province": "ON",
         "postalCode": "K2P 1L4'",
         "country": "Canada"
      },
      "orderStatus": "shipping",
      "orderId": "9ae31dee-8e10-491e-bea9-2
          ada279e49cf",
      "createdAt": "2025-11-30T19
          :58:12.053000",
      "updatedAt": "2025-11-30T19
          :58:12.053000"
   }
}
```

4) Update Emails or Delivery Address:

Method: PUT
URL: http://18.224.212.122:8000/orders/9ae31dee-8e10-491e-bea9-2ada279e49cf/details
Request:

```
{
  "deliveryAddress": {
    "street": "12 Main St",
    "city": "Toronto",
    "province": "ON",
    "postalCode": "M5V 3L9",
    "country": "Canada"
  }
}
```

Response:

```
{
   "status": "success",
   "_id": "692ca1d4bb0d73e43e9714a7",
   "userId": "8eab397f-6890-4c61-b3f8-4
       f6c099e4c90",
   "items": [
      {
         "itemId": "Study table",
            "quantity": 1,
            "price": 29.99
        },
        {
            "itemId": "Chair",
            "quantity": 1,
            "price": 15.99
        }
    ],
    "emails": [
        "robert.williams@outlook.com"
    ],
    "deliveryAddress": {
        "street": "12 Main St",
        "city": "Toronto",
        "province": "ON",
        "postalCode": "M5V 3L9",
        "country": "Canada"
    },
    "orderStatus": "shipping",
    "orderId": "9ae31dee-8e10-491e-bea9-2
        ada279e49cf",
    "createdAt": "2025-11-30T19:58:12.053000",
    "updatedAt": "2025-11-30T21:01:23.450000"
}
```

VI. Event-Driven Synchronization System

The event-driven system ensures that any updates to a user's email or delivery address—performed through the PUT endpoint of the User Microservice—are automatically propagated to the Order Database. When such an update occurs, the User Microservice publishes an event that triggers the synchronization process. This mechanism is implemented using RabbitMQ as the message broker, with the Python `pika` library handling event publishing and consumption.

A. Architecture and Workflow

The event-driven architecture operates as follows:

1) When a user's email or delivery address is updated via a `PUT` request in the User microservice, an update event is published to the RabbitMQ message broker.
2) The Order microservice subscribes to these events and updates the corresponding user fields in its own database.
3) This mechanism ensures that user information remains consistent and synchronized across both microservices.

A durable queue bound to a direct exchange ensures reliable message delivery. Messages are acknowledged only after successful processing, preventing any data loss.

B. RabbitMQ Configuration

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

```
rabbitmq:
    image: rabbitmq:3-management
    container_name: rabbitmq
    ports:
      - "5672:5672"    # RabbitMQ
       ↪  connection
      - "15672:15672" # Web dashboard
    healthcheck:
      test: ["CMD",
       ↪  "rabbitmq-diagnostics", "ping"]
      interval: 5s
      timeout: 5s
      retries: 10
    environment:
      RABBITMQ_DEFAULT_USER:
       ↪  ${RABBITMQ_USERNAME}
      RABBITMQ_DEFAULT_PASS:
       ↪  ${RABBITMQ_PASSWORD}
```

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 an update event to RabbitMQ whenever a user's email or delivery address is modified. This behavior is implemented in the `publish_user_update_event()` function, which performs the following steps:

- Establishes a connection to RabbitMQ using the `RABBITMQ_URI` loaded from the environment-based configuration.
- Opens a channel and declares a durable `fanout` exchange using the name stored in `RABBITMQ_QUEUE_NAME`.
- Constructs an event message containing the event type (`user.updated`), the user's ID, the updated list of email addresses, and the delivery address.
- Serializes the event into JSON and publishes it to RabbitMQ using the queue name as the routing key.
- Closes the RabbitMQ connection cleanly once the message is published.

This mechanism ensures that any update to user data results in a real-time event that can be consumed by other microservices, enabling consistent and synchronized data across the system.

D. Order Microservice: Event Consumption

The Order microservice consumes user update events published by the User microservice. The event consumer, implemented in `consume_user_update_events()` function, performs the following operations:

- Establishes a connection to RabbitMQ using the configured hostname, port, and credentials, and declares a durable queue for user update events.
- Continuously listens on the queue and processes incoming messages sequentially through a callback function.
- Extracts the updated user information from each event payload (`userId`, `emails`, and `deliveryAddress`) and updates all matching orders in the Order database using `update_many()`.

- Acknowledges each message only after a successful database update, ensuring reliable message delivery and preventing data loss.

This consumer ensures that any modifications to user data are propagated to the Order microservice in real time, maintaining consistency across services.

VII. User Microservice New Version v2

The User microservice has been upgraded to version 2 (v2), introducing several refinements and internal improvements. In version 1, the `createdAt` and `updatedAt` fields were not stored at all, and email addresses were not validated.

In v2, these concerns are addressed by automatically managing timestamps and enforcing email validation for all user operations. This update improves data quality, consistency, and reliability across the system.

```
    user_dict["createdAt"] =
    ↪  datetime.now(timezone.utc)
    user_dict["updatedAt"] =
    ↪  datetime.now(timezone.utc)

    def is_valid_email(email : str):
    try:
        validate_email(email)
        return True
    except EmailNotValidError:
        return False
```

A) User Creation (POST /users)

In v2, the process of creating a new user has been enhanced with the following changes:

- The `createdAt` and `updatedAt` timestamps are automatically assigned using the current UTC time at the moment of user creation.
- All submitted email addresses are validated using the `email_validator` library. If any email is invalid, the request is rejected with an appropriate error response.

Once validation is complete, the full user record—including the generated `userId` and timestamp fields—is inserted into the database, ensuring that each new user is stored with consistent metadata.

B) User Update (PUT /users/userId)

In version 2, updates to user information also benefit from the new validation and timestamp handling:

- The `updatedAt` field is automatically set to the current UTC time whenever the user's email addresses or delivery address are modified.
- Any updated email addresses are validated using the same `email_validator` logic; invalid emails cause the update request to be rejected.
- After a successful update, the microservice publishes an event to the event-driven messaging system so that the Order microservice can remain synchronized with the latest user data.

These enhancements ensure that user records now include accurate timestamps, that only valid email addresses are stored, and that changes are consistently propagated to dependent microservices.

VIII. Strangler Pattern in the API Gateway

The Strangler pattern is an architectural approach employed during the migration from a monolithic application to a microservices-based architecture. In this case, the old version of the User microservice (v1) is replaced by the new version (v2) over time.

A. Implementation Details

The Strangler Pattern in this system is implemented through a dedicated `api_gateway`. In this design, the gateway applies a probabilistic traffic-splitting strategy to implement the strangler pattern. The configuration parameter P, defined in `config.json`, specifies the percentage of incoming user requests that should be forwarded to User Service v1.

```
{
  "P": 70
}
```

Accordingly, the remaining $1-P$ percent of requests are routed to User Service v2.

The API Gateway service contains the following files:

- `gateway.py` — the core request router,
- `config.json` — stores the routing percentage P used for weighted routing between v1 and v2,
- a Dockerfile used to build and run the gateway independently.

B. Routing Logic Inside Gateway

All requests pass through middleware defined in `gateway.py`. The routing logic operates as follows:

```
# Load routing percentage
def load_config():
    with open("config.json") as f:
        return json.load(f)

@app.middleware("http")
async def gateway_router(request: Request,
↪   call_next):

    if request.url.path.startswith("/orders"):
        backend = "http://order_service:8000"

    else:
        # Weighted routing between v1 and v2
        P = load_config()["P"]
        hit = random.randint(1, 100)

        if hit <= P:
            backend =
            ↪  "http://user_service_v1:8000"
        else:
            backend =
            ↪  "http://user_service_v2:8000"

    async with httpx.AsyncClient() as client:
        response = await client.request(
            request.method, backend +
            ↪  request.url.path,
            content=await request.body()
    )

    return Response(content=response.content,
    ↪   status_code=response.status_code)
```

1) Requests matching the path `/orders` are always routed directly to the Order microservice.

For User-related routes, the Strangler Pattern is implemented through weighted routing:

1) The gateway loads the routing percentage P from `config.json`.
2) A random value between 1 and 100 is generated.
3) If the value is less than or equal to P, the request is forwarded to `user_service_v1`.
4) Otherwise, the request is sent to `user_service_v2`.

By adjusting P in `config.json`, we can perform a gradual migration without redeploying any services:

- $P = 100\%$: all traffic goes to v1 (initial state).
- $P = 70\%$: 70% to v1, 30% to v2 (testing phase).
- $P = 20\%$: 20% to v1, 80% to v2 (stabilization phase).
- $P = 0\%$: all traffic now goes to v2 (v1 fully strangled).

Because all clients always call the same API Gateway URL, no client-side changes are required during the migration.This design allows the new User Service v2 to replace v1 incrementally, ensuring zero downtime and full backward compatibility.

IX. Database Design

The system uses MongoDB as its primary database, deployed as a Docker container. As a schema-flexible NoSQL database, MongoDB is well-suited for storing dynamic and evolving data such as user and order records. The database is organized into two collections for users and orders, with data validation handled by the microservices to ensure consistency and correctness.

A. *MongoDB Deployment in Docker*

MongoDB is deployed as a separate Docker container. The service reads its configuration, such as database name, username, and password, from the `.env` file. The container exposes port `27017` so that the microservices can connect to it using the shared `MONGO_URI`.

The relevant `docker-compose` configuration is shown below:

```
mongodb:
    image: mongo:latest
    container_name: mongodb
    ports:
    - "27017:27017"
    command: ["mongod",
    ↪   "--wiredTigerCacheSizeGB=0.25"]
    volumes:
      - mongodb_data:/data/db
    healthcheck:
      test: ["CMD", "mongosh", "--eval",
      ↪   "db.runCommand({ ping: 1 })"]
      interval: 5s
      timeout: 5s
      retries: 10
volumes:
  mongodb_data:
```

The database layer consists of two MongoDB databases, each used by a different microservice. The `user_db` database contains the `users` collection, which is maintained by the User Microservice and stores all user-related data. The `order_db` database contains the `orders` collection, managed by the Order Microservice and kept synchronized with user updates through RabbitMQ events.

X. Docker Deployment Strategy

The system follows a microservice-based deployment strategy, where each component runs inside its own Docker container. This approach provides isolation between services, easier debugging, and the ability to scale or update individual components independently. Docker Compose is used to orchestrate the multi-container environment and define service dependencies.

A. *Microservice Architecture*

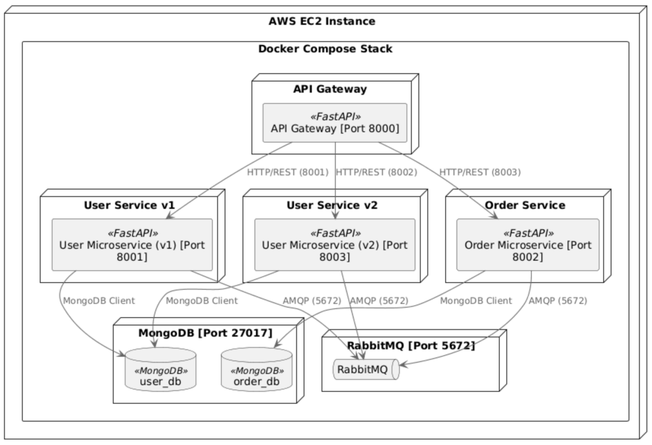


Fig. 2. Microservice Deployment Diagram

Each microservice is packaged with its own Dockerfile, allowing it to run independently:

- **User Service v1** and **User Service v2**: Two versions of the User Microservice are deployed together to enable the Strangler Pattern. Both services run as separate containers and connect to the same MongoDB instance.
- **Order Service**: Runs as an independent FastAPI service and listens for user update events from RabbitMQ to synchronize user-related fields inside the `orders` collection.
- **API Gateway**: Implemented as a dedicated FastAPI service. All external requests pass through this container. The gateway applies weighted routing to direct traffic between User Service v1 and v2 based on the percentage value stored in `config.json`.
- **RabbitMQ**: Deployed as a message broker container, providing asynchronous communication between the User and Order microservices.

- **MongoDB**: Runs as a separate container and stores two databases: `user_db` (containing the `users` collection) and `order_db` (containing the `orders` collection).

B. Docker-Compose Configuration

All services are defined in a single `docker-compose.yml` file. Docker Compose ensures that each container starts in the correct order using the `depends_on` directive.

```
services:
  mongodb:
    image: mongo:latest
    container_name: mongodb
    ports:
    - "27017:27017"
    command: ["mongod",
     ↪  "--wiredTigerCacheSizeGB=0.25"]
    volumes:
      - mongodb_data:/data/db
    healthcheck:
      test: ["CMD", "mongosh", "--eval",
       ↪  "db.runCommand({ ping: 1 })"]
      interval: 5s
      timeout: 5s
      retries: 10

  rabbitmq:
    image: rabbitmq:3-management
    container_name: rabbitmq
    ports:
      - "5672:5672"    # RabbitMQ
       ↪  connection
      - "15672:15672" # Web dashboard
    healthcheck:
      test: ["CMD",
       ↪  "rabbitmq-diagnostics", "ping"]
      interval: 5s
      timeout: 5s
      retries: 10
    environment:
      RABBITMQ_DEFAULT_USER:
       ↪  ${RABBITMQ_USERNAME}
      RABBITMQ_DEFAULT_PASS:
       ↪  ${RABBITMQ_PASSWORD}

  user_service_v1:
    build:
      context: ./src
      dockerfile:
       ↪  user_service_v1/Dockerfile
    container_name: user_service_v1
    ports:
      - "8001:8000"
    depends_on:
      - mongodb
      - rabbitmq
    environment:
      - MONGO_URI=${MONGO_URI}
      - USER_DB=${USER_DB}
      - RABBITMQ_URI=
      ${RABBITMQ_URI}
      - RABBITMQ_USERNAME=
      ${RABBITMQ_USERNAME}
      - RABBITMQ_PASSWORD=
      ${RABBITMQ_PASSWORD}
      - RABBITMQ_QUEUE_NAME=
      ${RABBITMQ_QUEUE_NAME}

  order_service:
    build:
      context: ./src
      dockerfile: order_service/Dockerfile
    container_name: order_service
    env_file:
    - .env
    ports:
      - "8002:8000"
    depends_on:
      - mongodb
      - rabbitmq
    environment:
      - MONGO_URI=${MONGO_URI}
      - ORDER_DB=${ORDER_DB}
      - RABBITMQ_URI=${RABBITMQ_URI}
      - RABBITMQ_QUEUE_NAME=
      ${RABBITMQ_QUEUE_NAME}
      - RABBITMQ_USERNAME=
      ${RABBITMQ_USERNAME}
      - RABBITMQ_PASSWORD=
      ${RABBITMQ_PASSWORD}

  user_service_v2:
    build:
      context: ./src
      dockerfile:
       ↪  user_service_v2/Dockerfile
    container_name: user_service_v2
    env_file:
    - .env
    ports:
      - "8003:8000"
    depends_on:
      - mongodb
      - rabbitmq
    environment:
      - MONGO_URI=${MONGO_URI}
      - USER_DB=${USER_DB}
      - RABBITMQ_URI=${RABBITMQ_URI}
      - RABBITMQ_USERNAME=
      ${RABBITMQ_USERNAME}
      - RABBITMQ_PASSWORD=
      ${RABBITMQ_PASSWORD}
      - RABBITMQ_QUEUE_NAME=
      ${RABBITMQ_QUEUE_NAME}

  api_gateway:
    build:
      context: ./src
      dockerfile: api_gateway/Dockerfile
    container_name: api_gateway
    env_file:
    - .env
    ports:
      - "8000:8080"
    volumes:
      - ./src/api_gateway/config.json:
      /app/api_gateway/config.json
    depends_on:
      - user_service_v1
      - user_service_v2
      - order_service

volumes:
  mongodb_data:
```

XI. CLOUD DEPLOYMENT TO AMAZON EC2

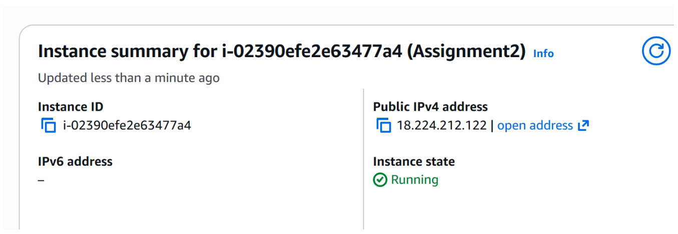
The entire microservices system was deployed on an Amazon EC2 instance running Ubuntu. The EC2 VM serves as the host environment for all Docker containers, allowing the application to run in a cloud environment identical to the local Docker Compose setup.

A. Amazon EC2 Setup

A new EC2 instance was launched using Ubuntu 22.04 LTS. After connecting via SSH, Docker and Docker Compose were installed on the VM. The project files were copied to the instance, along with the `.env` file containing environment variables for MongoDB, RabbitMQ, and the microservices.

- **Port 8000**: Exposed for the API Gateway, allowing external clients to access all microservices through a single entry point.
- **Port 22**: SSH access for administration.

These rules ensure that only the required services are accessible while keeping the instance secure.

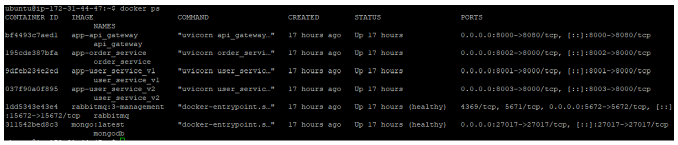


Instance summary for i-02390efe2e63477a4 (Assignment2) Info

Updated less than a minute ago

Instance ID	Public IPv4 address
i-02390efe2e63477a4	18.224.212.122 | open address
IPv6 address	Instance state
–	Running

Fig. 3. AWS EC2 Instance



```
ubuntu@ip-172-31-44-47:~$ docker ps
CONTAINER ID   IMAGE                 COMMAND                  CREATED        STATUS                  PORTS
          NAMES
bf4493c7aed1   app-api_gateway       "uvicorn api_gateway…"   17 hours ago   Up 17 hours             0.0.0.0:8000->8080/tcp, [::]:8000->8080/tcp
          api_gateway
195cde387bfa   app-order_service     "uvicorn order_servi…"   17 hours ago   Up 17 hours             0.0.0.0:8002->8000/tcp, [::]:8002->8000/tcp
          order_service
9dfeb234e2ed   app-user_service_v1   "uvicorn user_servic…"   17 hours ago   Up 17 hours             0.0.0.0:8001->8000/tcp, [::]:8001->8000/tcp
          user_service_v1
037f90a0f895   app-user_service_v2   "uvicorn user_servic…"   17 hours ago   Up 17 hours             0.0.0.0:8003->8000/tcp, [::]:8003->8000/tcp
          user_service_v2
1dd5343e43e4   rabbitmq:3-management "docker-entrypoint.s…"   17 hours ago   Up 17 hours (healthy)   4369/tcp, 5671/tcp, 0.0.0.0:5672->5672/tcp, [::]
:15672->15672/tcp   rabbitmq
311542bed8c3   mongo:latest          "docker-entrypoint.s…"   17 hours ago   Up 17 hours (healthy)   0.0.0.0:27017->27017/tcp, [::]:27017->27017/tcp
          mongodb
```

Fig. 4. Docker containers running on cloud

XII. CI/CD USING GITHUB ACTIONS

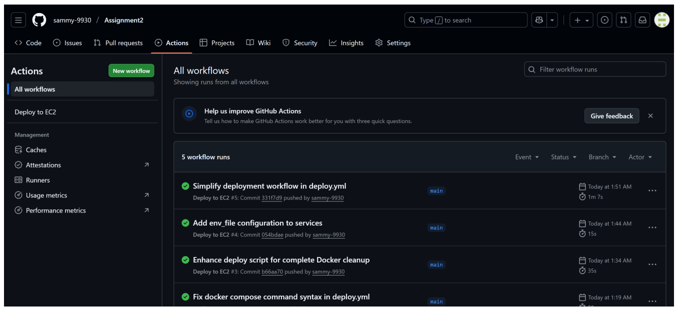
A lightweight CI/CD pipeline was implemented using GitHub Actions to automate the deployment process to the Amazon EC2 instance. The pipeline is triggered whenever changes are pushed to the main branch of the repository, ensuring that the latest code is deployed without requiring manual intervention.

A. Workflow Overview

The GitHub Actions workflow performs the following tasks:

- Checks out the latest version of the repository.
- Establishes an SSH connection to the EC2 instance using a private key stored securely in GitHub Secrets.
- Copies the updated project files to the EC2 instance.
- Executes Docker Compose commands on the EC2 host to rebuild and restart the microservices.

This automated process ensures that every push to the main branch results in a consistent and reproducible deployment.



All workflows
Showing runs from all workflows

Help us improve GitHub Actions
Tell us how to make GitHub Actions work better for you with three quick questions.

5 workflow runs

Simplify deployment workflow in deploy.yml
Deploy to EC2 #5: Commit 331f7d9 pushed by sammy-9930 — main — Today at 1:51 AM — 1m 7s

Add env_file configuration to services
Deploy to EC2 #4: Commit 054bdae pushed by sammy-9930 — main — Today at 1:44 AM — 15s

Enhance deploy script for complete Docker cleanup
Deploy to EC2 #3: Commit b66aa70 pushed by sammy-9930 — main — Today at 1:34 AM — 35s

Fix docker compose command syntax in deploy.yml — main — Today at 1:19 AM

Fig. 5. GitHub Workflows

B. GitHub Actions Configuration

The actions configuration file is given below.

```
name: Deploy to EC2

on:
  push:
    branches: ["main"]

jobs:
  deploy:
    runs-on: ubuntu-latest

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

      - name: Copy files to EC2
        uses: appleboy/scp-action@v0.1.7
        with:
          host: ${{ secrets.EC2_HOST }}
          username: ubuntu
          key: ${{ secrets.EC2_KEY }}
          source: "."
          target: "/home/ubuntu/app"

      - name: SSH into EC2 and deploy
        uses: appleboy/ssh-action@v0.1.7
        with:
          host: ${{ secrets.EC2_HOST }}
          username: ubuntu
          key: ${{ secrets.EC2_KEY }}
          script: |
            cd /home/ubuntu/app

            echo "${{ secrets.ENV_FILE }}"
             ↪  > .env
            # Restart services
            docker-compose down -v || true
            docker-compose up -d --build
            docker system prune -f || true
```

C. GitHub Secrets for Secure Access

Sensitive information required for deployment is stored securely using GitHub Secrets, including:

- `EC2_HOST`: Public IP address of the EC2 instance.
- `EC2_KEY`: SSH private key used to authenticate with the VM.
- `ENV_FILE`: The content used to recreate the `.env` file on the EC2 instance.

This prevents credentials from being exposed in the repository and ensures safe, secure automation.

XIII. Conclusion

This project demonstrates the design and deployment of a modular and scalable microservices architecture using FastAPI, Docker, RabbitMQ, and MongoDB. Two versions of the User Microservice and a single Order Microservice were implemented, showcasing the use of versioning and the Strangler Pattern to migrate functionality safely and incrementally. The API Gateway provided a centralized routing layer, enabling dynamic traffic distribution between service versions without requiring changes from the client side.

Data consistency across microservices was achieved through an event-driven synchronization mechanism using RabbitMQ, ensuring that updates to user information were reflected automatically in the Order Microservice. MongoDB served as the primary datastore, with separate databases and collections that kept user and order data isolated yet coordinated through events.

The entire system was containerized using Docker and deployed on an Amazon EC2 instance, enabling a cloud-based runtime environment that mirrors the local development setup. A lightweight CI/CD pipeline built with GitHub Actions automated the deployment process, simplifying updates and ensuring consistent deployments across environments.

Overall, this work demonstrates a complete microservices architecture with version control, an API gateway, event-driven communication, and cloud deployment. The system is maintainable, extensible, and aligned with modern cloud-native application design principles.

XIV. Source Code URL

https://github.com/sammy-9930/Assignment2

References

[1] Amazon Web Services, “Microservices on AWS.” [Online]. Available: https://aws.amazon.com/microservices/

[2] GeeksforGeeks, “Strangler Pattern in Microservices System Design.” [Online]. Available: https://www.geeksforgeeks.org/system-design/strangler-pattern-in-micro-services-system-design/

[3] RabbitMQ Documentation. [Online]. Available: https://www.rabbitmq.com/docs

[4] MongoDB Documentation. [Online]. Available: https://www.mongodb.com/docs/

[5] Docker Compose Documentation. [Online]. Available: https://docs.docker.com/compose/

[6] Amazon EC2 User Guide. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EC2_GetStarted.html