Cloud project 23-24

Table of Contents

[Phase 1: Project Planning and Design 3](#_Toc155193162)

[Define Requirements and Use Cases (4 hours) 3](#_Toc155193163)

[Requirements 4](#_Toc155193164)

[Use Cases 4](#_Toc155193165)

[Choose Technologies (8 hours) 3](#_Toc155193166)

[Architecture Design (12 hours) 3](#_Toc155193167)

[Architecture Overview 4](#_Toc155193168)

[AWS Services Used: 4](#_Toc155193169)

[Communication Flow: 4](#_Toc155193170)

[Setup AWS Account and Resources (4 hours) 3](#_Toc155193171)

[Phase 2: Application Development 3](#_Toc155193172)

[Implement Core Functionality (40 hours) 3](#_Toc155193173)

[User Authentication and Registration: 4](#_Toc155193174)

[Product Catalog Management: 4](#_Toc155193175)

[Shopping Cart Management: 4](#_Toc155193176)

[Order Processing: 4](#_Toc155193177)

[Payment Processing: 4](#_Toc155193178)

[Notification Management: 4](#_Toc155193179)

[Frontend Development: 4](#_Toc155193180)

[Database Design and Integration (16 hours) 3](#_Toc155193181)

[Authentication and Authorization (12 hours) 3](#_Toc155193182)

[API Development (20 hours) 3](#_Toc155193183)

[Phase 3: Deployment and Scaling 3](#_Toc155193184)

[Containerization (16 hours) 3](#_Toc155193185)

[Container Orchestration (20 hours) 3](#_Toc155193186)

[AWS Deployment (16 hours) 3](#_Toc155193187)

[Scaling Strategies (8 hours) 3](#_Toc155193188)

[Phase 4: Testing and Optimization 3](#_Toc155193189)

[Unit and Integration Testing (20 hours) 3](#_Toc155193190)

[Performance Testing and Optimization (16 hours) 3](#_Toc155193191)

[Phase 5: Documentation and Review 4](#_Toc155193192)

[Documentation (20 hours) 4](#_Toc155193193)

[Code Review and Refinement (20 hours) 4](#_Toc155193194)

[Phase 6: Project Completion 4](#_Toc155193195)

[Final Deployment (8 hours) 4](#_Toc155193196)

[Post-Deployment Support and Monitoring (16 hours) 4](#_Toc155193197)

# Phase 1: Project Planning and Design

## Define Requirements and Use Cases (4 hours)

* Meet with your team to outline the application's functionality and purpose.
* Identify key use cases and features.

Certainly! Let's define the requirements and use cases for an online shop deployed in a microservices architecture using AWS and Python.

### Requirements

#### User Authentication and Authorization

* \*\*Use Case:\*\* Users should be able to register, log in, and log out securely.
* \*\*Use Case:\*\* Differentiate between customer and admin roles with specific access permissions.

#### Data Storage

* \*\*Use Case:\*\* Store product information, user profiles, and order history.
* \*\*Use Case:\*\* Utilize a database (e.g., Amazon DynamoDB) to efficiently manage product catalog and customer data.

Scalability

* \*\*Use Case:\*\* Ensure the ability to scale individual microservices independently based on demand.
* \*\*Use Case:\*\* Implement auto-scaling for microservices handling high traffic, especially during promotions or sales events.

#### APIs

* \*\*Use Case:\*\* Develop microservices for core functionalities like product catalog, user management, and order processing.
* \*\*Use Case:\*\* Document APIs for external access and integration with third-party services.

1. \*\*User Service:\*\*

- \*\*Authentication APIs:\*\*

- `/api/register` - Register a new user.

- `/api/login` - Authenticate a user and generate an access token.

- `/api/logout` - Log out a user.

- \*\*Profile APIs:\*\*

- `/api/user/profile` - Retrieve and update user profile information.

2. \*\*Product Catalog Service:\*\*

- \*\*Product APIs:\*\*

- `/api/products` - Retrieve a list of products.

- `/api/products/{product\_id}` - Retrieve details of a specific product.

- `/api/products/create` - Create a new product (Admin access).

3. \*\*Shopping Cart Service:\*\*

- \*\*Shopping Cart APIs:\*\*

- `/api/cart` - Retrieve the contents of the user's shopping cart.

- `/api/cart/add` - Add a product to the shopping cart.

- `/api/cart/remove` - Remove a product from the shopping cart.

- `/api/cart/clear` - Clear the entire shopping cart.

4. \*\*Order Service:\*\*

- \*\*Order APIs:\*\*

- `/api/orders` - Retrieve a list of user orders.

- `/api/orders/{order\_id}` - Retrieve details of a specific order.

- `/api/orders/place` - Place a new order.

5. \*\*Payment Service:\*\*

- \*\*Payment APIs:\*\*

- `/api/payment/process` - Process a payment for an order.

- `/api/payment/methods` - Retrieve and manage user payment methods.

6. \*\*Notification Service:\*\*

- \*\*Notification APIs:\*\*

- `/api/notifications` - Retrieve a list of user notifications.

- `/api/notifications/{notification\_id}` - Retrieve details of a specific notification.

7. \*\*Frontend Service (Optional):\*\*

- \*\*UI Integration APIs:\*\*

- These APIs are consumed by the frontend to display and manage user interfaces.

#### Containerization

* \*\*Use Case:\*\* Containerize microservices using Docker for consistent deployment across various environments.
* \*\*Use Case:\*\* Utilize AWS ECS (Elastic Container Service) or EKS (Elastic Kubernetes Service) for orchestration.

#### Deployment

* \*\*Use Case:\*\* Deploy microservices independently using AWS Lambda or EC2 instances.
* \*\*Use Case:\*\* Implement blue-green deployment strategies for minimal downtime during updates.

#### Monitoring and Logging

* \*\*Use Case:\*\* Set up monitoring for the health and performance of microservices.
* \*\*Use Case:\*\* Implement centralized logging for debugging and auditing.

#### Security

* \*\*Use Case:\*\* Encrypt sensitive customer data in transit and at rest.
* \*\*Use Case:\*\* Implement OAuth or JWT for secure user authentication between microservices.

#### Testing

* \*\*Use Case:\*\* Conduct unit testing for individual microservices.
* \*\*Use Case:\*\* Implement end-to-end testing for critical user journeys.

#### Documentation

* \*\*Use Case:\*\* Provide comprehensive documentation for microservice APIs, deployment procedures, and troubleshooting.

### Use Cases

#### User Registration and Login

* \*\*Use Case:\*\* Users can register with the online shop, providing necessary details.
* \*\*Use Case:\*\* Registered users can log in securely.

#### Product Management

* \*\*Use Case:\*\* Admins can add, update, or remove products from the catalog.
* \*\*Use Case:\*\* Products should have details like name, description, price, and availability.

#### Shopping Cart

* \*\*Use Case:\*\* Users can add products to their shopping cart.
* \*\*Use Case:\*\* Users can view and modify the contents of their shopping cart.

#### Order Placement and Tracking

* \*\*Use Case:\*\* Users can place orders securely.
* \*\*Use Case:\*\* Users can track the status of their orders.

#### Payment Integration

* \*\*Use Case:\*\* Implement integration with payment gateways for secure transactions.
* \*\*Use Case:\*\* Allow users to save multiple payment methods.

#### Recommendation Engine (Optional)

* \*\*Use Case:\*\* Implement a recommendation engine based on user preferences and purchase history.

#### User Notifications (Additional)

* \*\*Use Case:\*\* Send order confirmation and shipment notifications to users.
* \*\*Use Case:\*\* Notify users of promotions or discounts.

#### Search Functionality

* \*\*Use Case:\*\* Users can search for products using keywords or filters.
* \*\*Use Case:\*\* Implement a recommendation engine based on user preferences and purchase history.

#### Scalability Test

* \*\*Use Case:\*\* Simulate increased traffic during peak hours to test scalability.
* \*\*Use Case:\*\* Automatically scale resources based on demand.

#### Error Handling and Recovery

* \*\*Use Case:\*\* Implement robust error handling to ensure a smooth user experience.
* \*\*Use Case:\*\* Provide mechanisms for recovering from failures and restoring system integrity.

#### Backup and Restore

* \*\*Use Case:\*\* Regularly backup critical data such as user profiles, orders, and product catalog.
* \*\*Use Case:\*\* Implement procedures for quick data restoration in case of data loss.

## Choose Technologies (8 hours)

* Decide on the AWS services to be used (e.g., EC2, S3, RDS, Lambda, API Gateway, etc.).
* Select Python frameworks or libraries for development.

Certainly! Here's a proposed list of technologies for the key microservices in your simplified online shop project:

1. \*\*User Service:\*\*

- \*\*Technology Stack:\*\*

- Language: Python (Flask or Django for web frameworks)

- Database: PostgreSQL or Amazon DynamoDB

- Authentication: OAuth or JWT

- Framework: Flask-Security or Django Rest Framework

2. \*\*Product Catalog Service:\*\*

- \*\*Technology Stack:\*\*

- Language: Python (Flask or Django)

- Database: PostgreSQL or Amazon DynamoDB

- ORM: SQLAlchemy (for Flask) or Django ORM (for Django)

3. \*\*Shopping Cart Service:\*\*

- \*\*Technology Stack:\*\*

- Language: Python (Flask or Django)

- Database: Redis (for caching shopping cart data) or PostgreSQL

- Integration: WebSocket for real-time updates or RESTful API

4. \*\*Order Service:\*\*

- \*\*Technology Stack:\*\*

- Language: Python (Flask or Django)

- Database: PostgreSQL or Amazon DynamoDB

- Message Queue: Amazon SQS or RabbitMQ for order processing

5. \*\*Payment Service:\*\*

- \*\*Technology Stack:\*\*

- Language: Python (Flask or Django)

- Payment Gateway Integration: Stripe, PayPal, or others

- Encryption: HTTPS for secure communication

6. \*\*Notification Service:\*\*

- \*\*Technology Stack:\*\*

- Language: Python (Flask or Django)

- Notification Delivery: Amazon SNS, Twilio, or custom email service

- Background Tasks: Celery (optional)

7. \*\*Frontend Service (Optional):\*\*

- \*\*Technology Stack:\*\*

- Framework: React, Angular, or Vue.js

- State Management: Redux (for React), Vuex (for Vue.js), or Context API

- Communication: Axios or Fetch API for API calls

This proposed technology stack leverages Python as the primary programming language for the backend services, which is well-suited for web development and has a variety of frameworks and libraries available. PostgreSQL is recommended as the database choice for its reliability and features, but you can also consider Amazon DynamoDB for a serverless and fully managed option. The choice of the frontend framework depends on your team's expertise and preferences.

Additionally, serverless options like AWS Lambda can be considered for microservices to minimize infrastructure management overhead. AWS API Gateway can be used for creating RESTful APIs, and Amazon DynamoDB can serve as a fully managed and scalable database solution.

Remember to adapt the technology choices based on your team's expertise, project requirements, and any specific constraints or preferences you may have.

## Architecture Design (12 hours)

* Design the overall architecture of your application.
* Decide on data storage, API endpoints, and other key components.

### Architecture Overview

1. \*\*User Service:\*\*

- \*\*AWS Lambda Function:\*\*

- Handles user authentication and registration.

- Invokes AWS Cognito for user pool management.

- Utilizes Amazon DynamoDB for user profile storage.

2. \*\*Product Catalog Service:\*\*

- \*\*AWS Lambda Function:\*\*

- Manages the product catalog and handles CRUD operations.

- Utilizes Amazon DynamoDB for storing product details.

- Exposes RESTful APIs through Amazon API Gateway.

3. \*\*Shopping Cart Service:\*\*

- \*\*AWS Lambda Function:\*\*

- Manages user shopping carts and handles cart operations.

- Uses Amazon DynamoDB for cart data storage.

- Integrates with the Product Catalog Service for product information.

- Exposes RESTful APIs through Amazon API Gateway.

4. \*\*Order Service:\*\*

- \*\*AWS Lambda Function:\*\*

- Manages the order lifecycle, from placement to fulfillment.

- Utilizes Amazon DynamoDB for storing order details.

- Integrates with the Shopping Cart Service and Payment Service.

- Exposes RESTful APIs through Amazon API Gateway.

- \*\*AWS Step Functions:\*\*

- Orchestrates the order processing workflow, coordinating interactions between microservices.

- Handles complex business logic and state transitions.

5. \*\*Payment Service:\*\*

- \*\*AWS Lambda Function:\*\*

- Handles payment processing and payment method management.

- Integrates with payment gateways (e.g., Stripe) for transactions.

- Utilizes Amazon DynamoDB for storing payment-related data.

- Exposes RESTful APIs through Amazon API Gateway.

- \*\*AWS Simple Queue Service (SQS):\*\*

- Sends payment-related events to an SQS queue for asynchronous processing.

6. \*\*Notification Service:\*\*

- \*\*AWS Lambda Function:\*\*

- Manages user notifications and sends notifications.

- Uses Amazon DynamoDB for storing notification data.

- Integrates with Amazon Simple Notification Service (SNS) for messaging.

- Exposes RESTful APIs through Amazon API Gateway.

7. \*\*Frontend Service (Optional):\*\*

- \*\*React Application:\*\*

- Manages the user interface and interacts with backend microservices.

- Communicates with microservices APIs through Amazon API Gateway.

- \*\*AWS S3:\*\*

- Stores static assets for the frontend application.

### AWS Services Used:

- \*\*Amazon DynamoDB:\*\*

- Used for storing user data, product catalog, shopping cart information, order details, payment-related data, and notifications.

- \*\*Amazon Lambda:\*\*

- Used for implementing serverless functions for each microservice.

- \*\*Amazon API Gateway:\*\*

- Exposes RESTful APIs for each microservice.

- \*\*Amazon Cognito:\*\*

- Manages user authentication and user pool.

- \*\*AWS Step Functions:\*\*

- Orchestrates the order processing workflow, coordinating interactions between microservices.

- \*\*AWS Simple Queue Service (SQS):\*\*

- Used for asynchronous processing of payment-related events.

- \*\*Amazon SNS:\*\*

- Used for sending notifications to users.

- \*\*Amazon S3:\*\*

- Stores static assets for the frontend application.

### Communication Flow:

1. User interacts with the React application (optional) or any other frontend application.

2. Frontend communicates with backend microservices through Amazon API Gateway.

3. Each microservice function is triggered by API Gateway and performs its specific functionality.

4. Microservices communicate with each other, and AWS Step Functions orchestrates complex order processing workflows.

5. Payment-related events are sent to an SQS queue for asynchronous processing.

6. Data is stored in Amazon DynamoDB for each microservice.

7. Notifications may be sent asynchronously using Amazon SNS.

This architecture introduces AWS Cognito for user pool management, AWS Step Functions for orchestrating workflows, and AWS SQS for asynchronous event processing. This mix provides a more comprehensive and scalable solution while taking advantage of specialized services for specific tasks. Adjustments can still be made based on your specific requirements and preferences.

## Setup AWS Account and Resources (4 hours)

* Create an AWS account if you don't have one.
* Set up necessary AWS resources based on your architecture.

# Phase 2: Application Development

## Implement Core Functionality (40 hours)

* Develop the basic features and functionality of your application.
* Set up a version control system (e.g., Git) for collaboration.

### User Authentication and Registration:

Implementation:

* Utilize AWS Cognito for managing user authentication and registration.
* Integrate Cognito with your User Service, which will handle user-related operations.
* Use the Cognito SDK or API Gateway to provide secure authentication endpoints.

### Product Catalog Management:

Implementation:

* Develop a Product Catalog Service using AWS Lambda and API Gateway.
* Store product details in Amazon DynamoDB.
* Expose RESTful APIs to manage products (CRUD operations).

### Shopping Cart Management:

Implementation:

* Create a Shopping Cart Service using AWS Lambda and API Gateway.
* Store cart data in Amazon DynamoDB.
* Implement APIs for adding, removing, and retrieving items from the cart.

### Order Processing:

Implementation:

* Develop an Order Service using AWS Lambda and API Gateway.
* Store order details in Amazon DynamoDB.
* Integrate with the Shopping Cart and Payment Services.
* Use AWS Step Functions to orchestrate the order processing workflow.

### Payment Processing:

Implementation:

* Implement a Payment Service using AWS Lambda and API Gateway.
* Integrate with payment gateways like Stripe or others.
* Store payment-related data in Amazon DynamoDB.
* Use AWS SQS for asynchronous processing of payment-related events.

### Notification Management:

Implementation:

* Create a Notification Service using AWS Lambda and API Gateway.
* Store notification data in Amazon DynamoDB.
* Utilize Amazon SNS for sending notifications to users.
* Implement APIs for retrieving and managing user notifications.

### Frontend Development:

Implementation:

* Develop a frontend application using a framework like React.
* Utilize AWS S3 to store and serve static assets.
* Integrate with backend microservices using APIs exposed by API Gateway.

## Database Design and Integration (16 hours)

* Design and create the database schema.
* Integrate the database with your application.

## Authentication and Authorization (12 hours)

* Implement user authentication and authorization mechanisms.

## API Development (20 hours)

* Create APIs for communication between different components of your application.
* Ensure proper error handling and validation.

# Phase 3: Deployment and Scaling

## Containerization (16 hours)

* Dockerize your application components for easy deployment.

## Container Orchestration (20 hours)

* Explore Kubernetes for container orchestration if needed.

## AWS Deployment (16 hours)

* Deploy your application on AWS using services like EC2, Elastic Beanstalk, or Lambda.

## Scaling Strategies (8 hours)

* Implement auto-scaling and load balancing to handle varying workloads.

# Phase 4: Testing and Optimization

## Unit and Integration Testing (20 hours)

* Write and execute tests to ensure the correctness of your code.

## Performance Testing and Optimization (16 hours)

* Identify and optimize performance bottlenecks.

# Phase 5: Documentation and Review

## Documentation (20 hours)

* Document the architecture, setup, and usage of your application.

## Code Review and Refinement (20 hours)

* Conduct a thorough code review and make necessary improvements.

# Phase 6: Project Completion

## Final Deployment (8 hours)

* Deploy the final version of your application.

## Post-Deployment Support and Monitoring (16 hours)

* Set up monitoring tools and provide support for any issues that arise.

### Total Estimated Work Hours: ~370 hours