Deliverable #2 Report

1. Project Title

E-commerce App

Github link - https://github.com/Rafid-A/3354-Team1

2. Delegation of Tasks

- **Chris Abraham:** Responsible for estimating cost of personnel.
- **Rafid A Ahmed:** Responsible for creating a testing plan and testing the code.
- **Jaya Vardhini Akurathi:** Responsible for project scheduling.
- **Varun Venkat Bhupathi Raju:** Responsible for comparison of our work with similar designs.
- **Quincy J Kemany:** Responsible for estimating cost of hardware products.
- **Varun Mange:** Responsible for cost, effort and pricing estimation.
- **Marc Manoj:** Responsible for estimating cost of software products.
- **Jaime Alejandro Osuna Espinoza:** Responsible for creating a testing plan and testing the code.

3. Deliverable #1 Report

3.1. Project Title

E-commerce App

Github link - https://github.com/Rafid-A/3354-eam1

3.2. Delegation of Tasks

- **Chris Abraham:** Responsible for the **customer**-focused Use Case Diagram and Sequence Diagram.
- **Rafid A Ahmed:** Responsible for the Functional Requirements & Github Set up.
- **Jaya Vardhini Akurathi:** Responsible for the Non-Functional Requirements.
- **Varun Venkat Bhupathi Raju:** Responsible for contributing to the Use Case Diagram.
- **Quincy J Kemany:** Responsible for the Architectural Design.
- **Varun Mange:** Responsible for the **vendor**-focused Use Case Diagram and Sequence Diagrams.
- **Marc Manoj:** Responsible for the software process model.
- **Jaime Alejandro Osuna Espinoza:** Responsible for the Class Diagram and Scope document.

3.3. Assumptions

1. Only authorized vendors are allowed to sell products on our ecommerce platform.
2. The payment processing will be handled by a third-party payment gateway.

3.4. Addressing Proposal Feedback

- **Feedback:** Well Done

3.5. Software Process Model

3.5.1. Model Description

A hybrid **Agile** model, specifically one using core principles from **Scrum**

3.5.2. Rationale

This Agile/Scrum model was chosen over a strict Waterfall model for several key reasons specific to your project:

- **Manages Complexity:** we are building a "complex, real-world system" with multiple facets, including customer-facing features (search, filter) and separate vendor-facing tools (analytics, product management). An iterative approach is ideal for tackling this complexity piece by piece, rather than trying to build everything at once.
- **Promotes Adaptability:** our goal is a "highly adaptable" design. An e-commerce app's requirements often change as features are built. Agile allows your team to adapt and refine features (like the specifics of "sales analytics") as you go, which is very difficult in a rigid Waterfall model.
- **Ideal for Team Coordination:** we have eight members, Agile and Scrum provide a clear framework for managing parallel work. The team can self-organize to tackle different features within a sprint, and short daily (or bi-weekly) meetings ensure everyone stays in sync with what others are doing.

3.6. Software Requirements

3.6.1. Functional Requirements

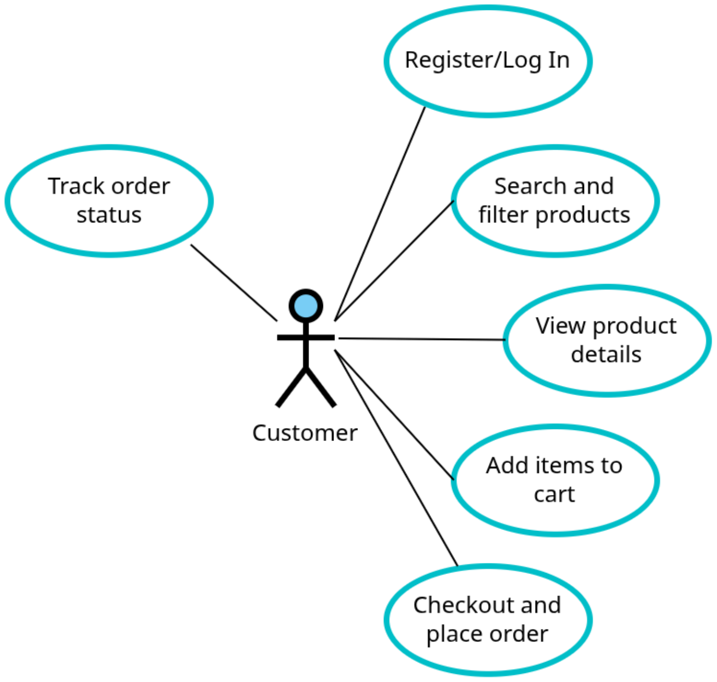
- The system shall allow a new user to register and an existing user to log in with an email and password.
- The system shall allow customers to search, filter, and sort products.
- The system shall display detailed information for each product, including description, images, price, and vendor information.
- The system shall allow customers to add products to a virtual shopping cart and modify its contents.
- The system shall guide the customer through a checkout process to buy the items in their cart.
- The system shall allow a vendor to create and manage their product listings.
- The system shall allow a vendor to view incoming orders and update their shipment.

3.6.2. Non-Functional Requirements

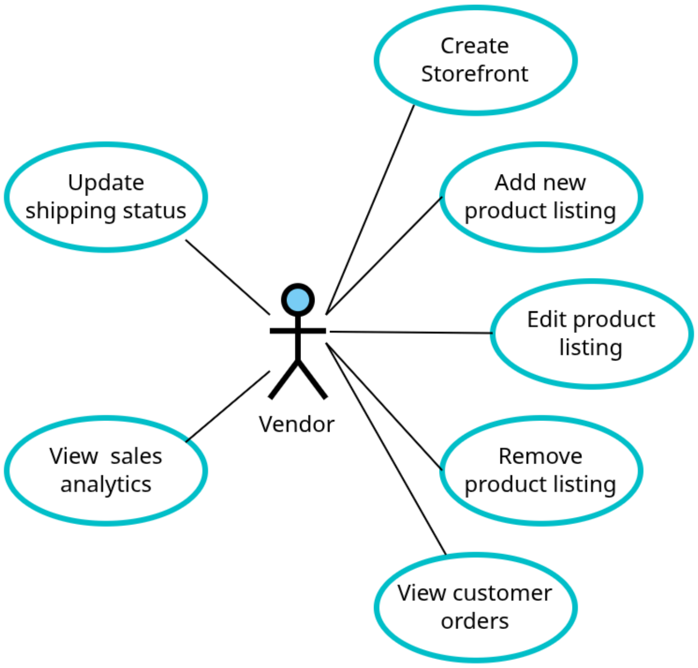
- **Usability Requirement:**
    - The system shall provide a responsive user interface that lets users and vendors explore, search and transact with minimum training or assistance.
- **Performance Requirement:**
    - The system shall load product listings and search results in 3 seconds under normal network conditions and support 500 concurrent users without performance deterioration.
- **Space Requirement:**
    - The system shall efficiently store product data and images using optimized compression and indexing to minimize database storage.
- **Dependability Requirement:**
    - The system shall maintain 99.5% uptime and recover from failures within 5 minutes using automated backups and restores.
- **Security Requirement:**
    - The system shall protect user data with AES-256 encryption and secure vendor accounts through two-factor authentication.
- **Environmental Requirement:**
    - The system shall use energy-efficient cloud infrastructure to reduce power usage and carbon footprint while ensuring high performance.
- **Operational Requirement:**
    - The system shall function 24/7 and be compatible with desktop and mobile browsers on all major operating systems.
- **Development Requirement:**
    - The system shall employ a modular architecture to support easy maintenance, scalability, and integration with external payment and shipping APIs.
- **Regulatory Requirement:**
    - The system shall comply with GDPR and CCPA regulations to protect user data privacy and consent management. It must also follow PCI DSS rules for processing payment information.
- **Ethical Requirement: Accounting Requirement:**
    - The system shall maintain accurate sales and tax logs and prevent fraud to ensure ethical operations and compliance with consumer protection laws.
- **Safety/Security (Legislative) Requirement:**
    - The system shall adhere to safety and security rules, including data protection, fraud prevention, and compliance with any consumer protection legislation.

3.7. Use Case Diagram

3.7.1 Customer



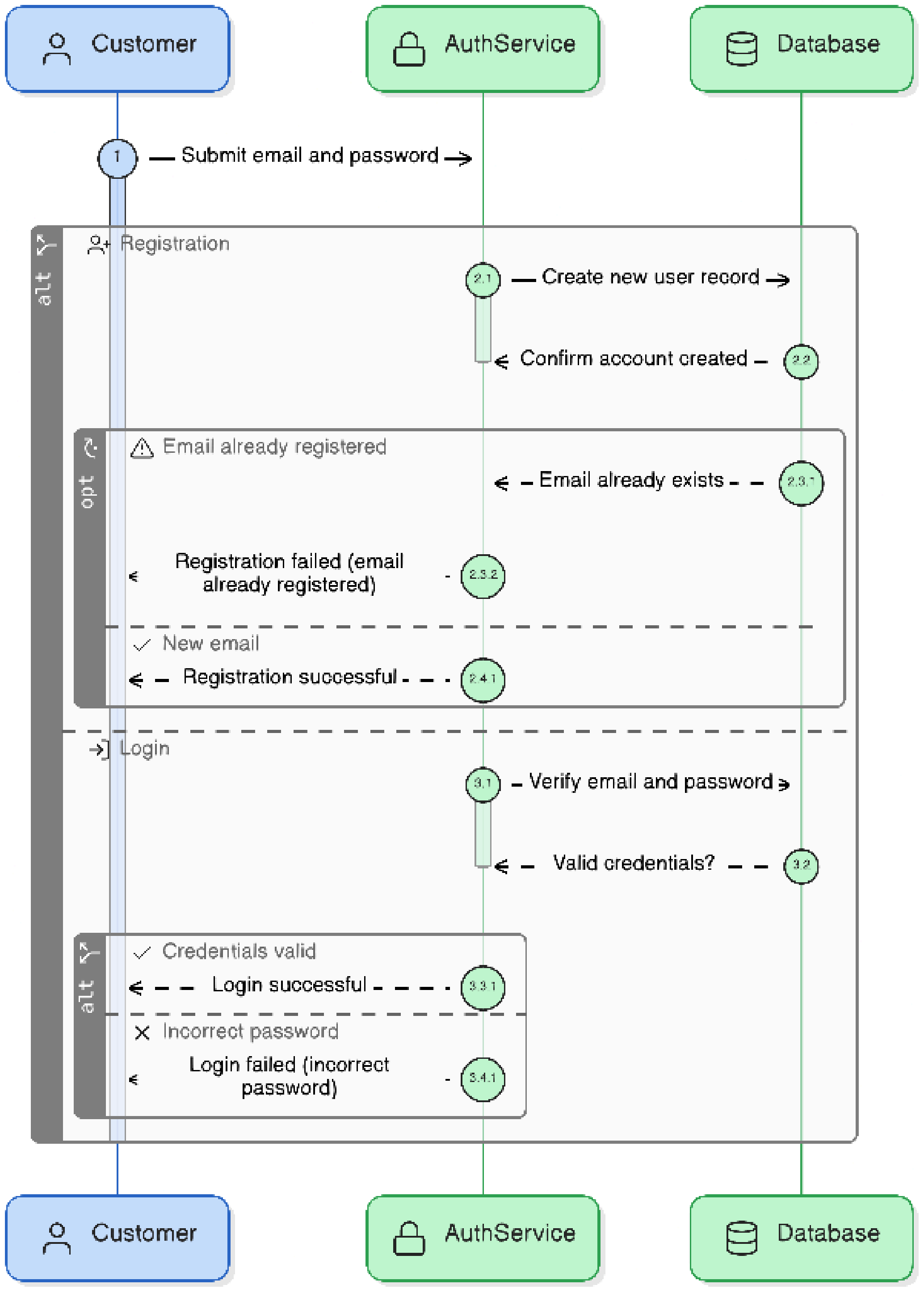
3.7.2 Vendor



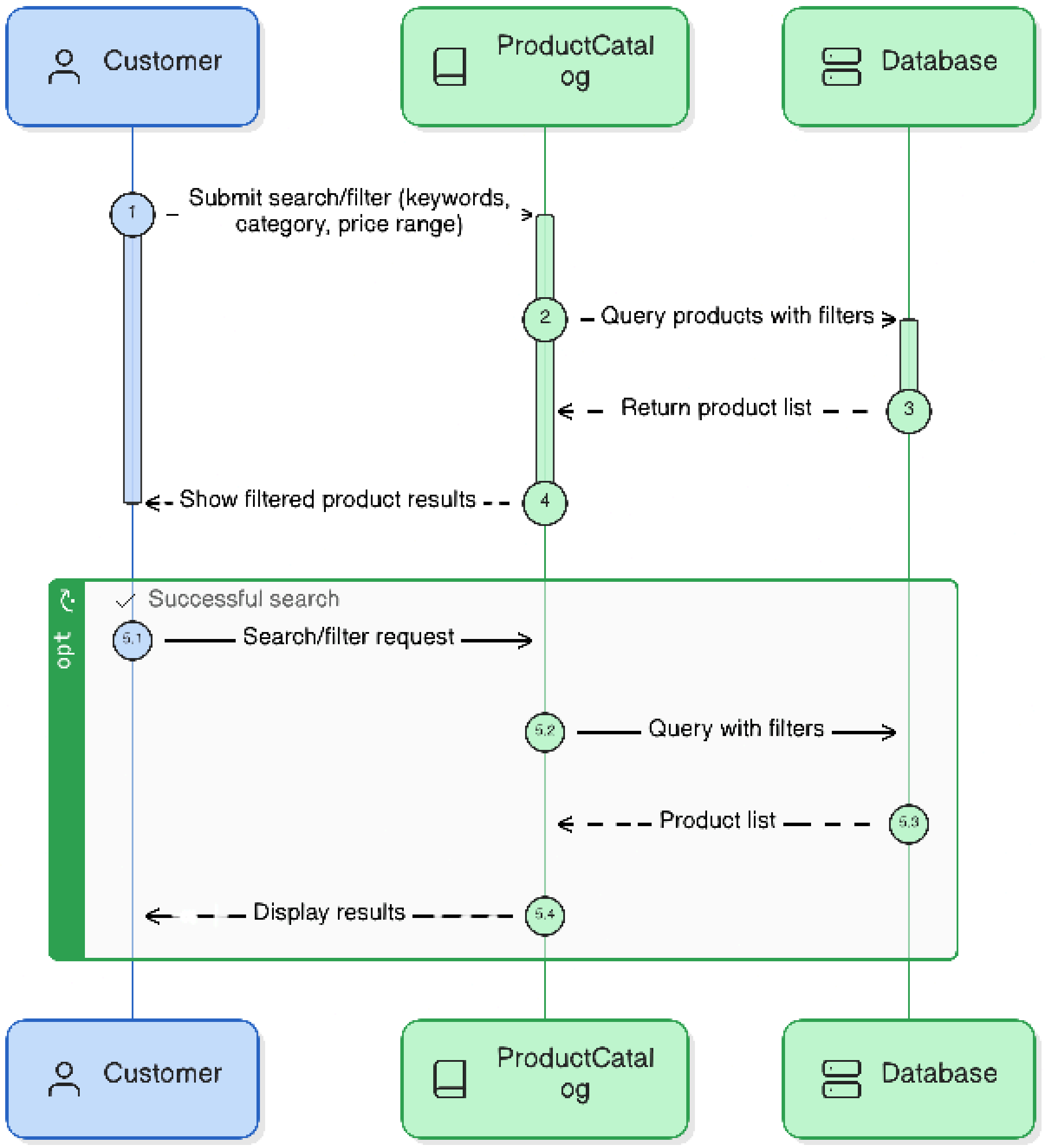
3.8. Sequence Diagrams

3.8.1 Customer

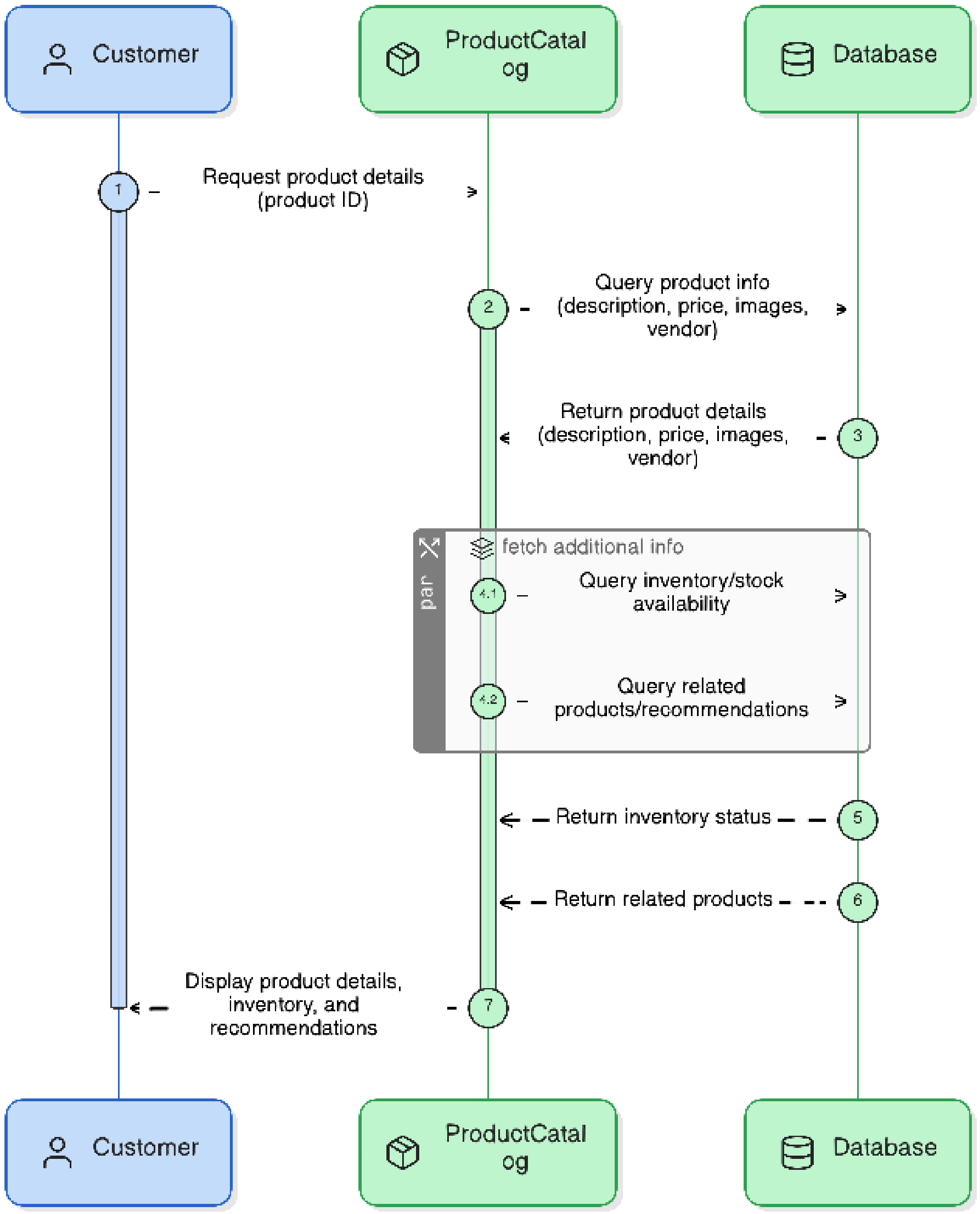
3.8.1.1 Customer - Register/Log In



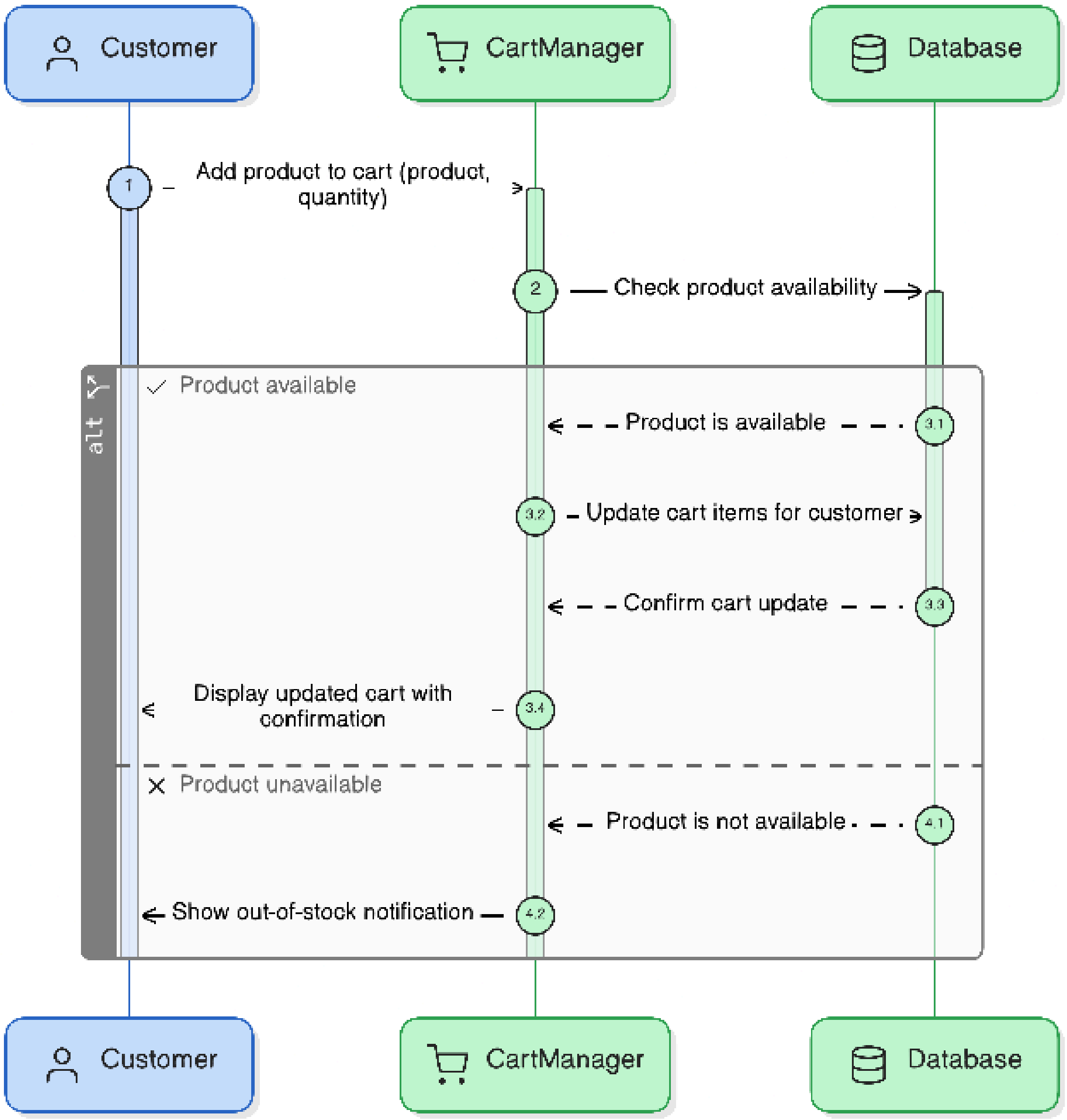
3.8.1.2 Customer - Search and Filter Products



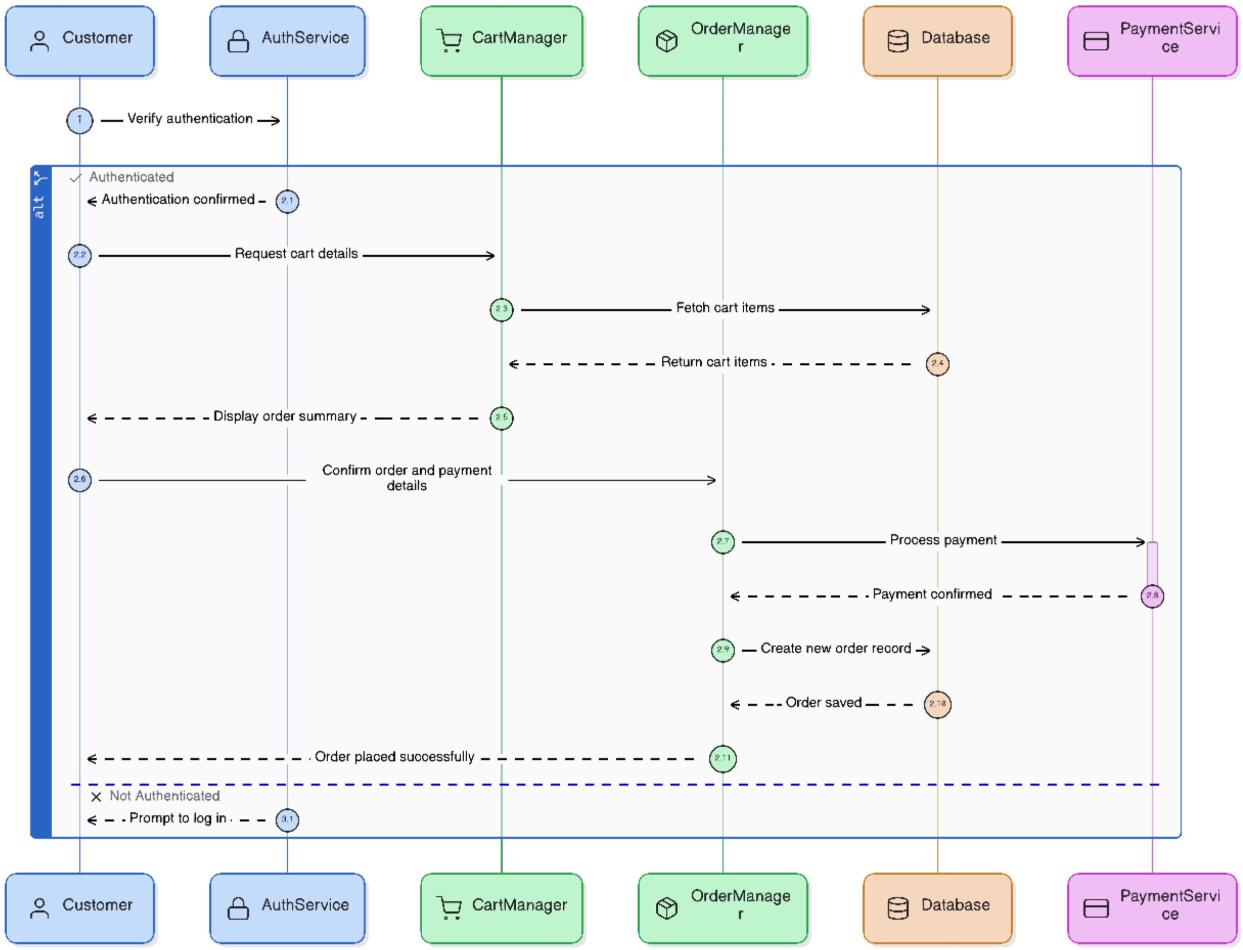
3.8.1.3 Customer - View product details


Customer
ProductCatal og
Database
1
Request product details (product ID)
2
Query product info (description, price, images, vendor)
3
Return product details (description, price, images, vendor)
par
fetch additional info
4.1
Query inventory/stock availability
4.2
Query related products/recommendations
5
Return inventory status
6
Return related products
7
Display product details, inventory, and recommendations
Customer
ProductCatal og
Database

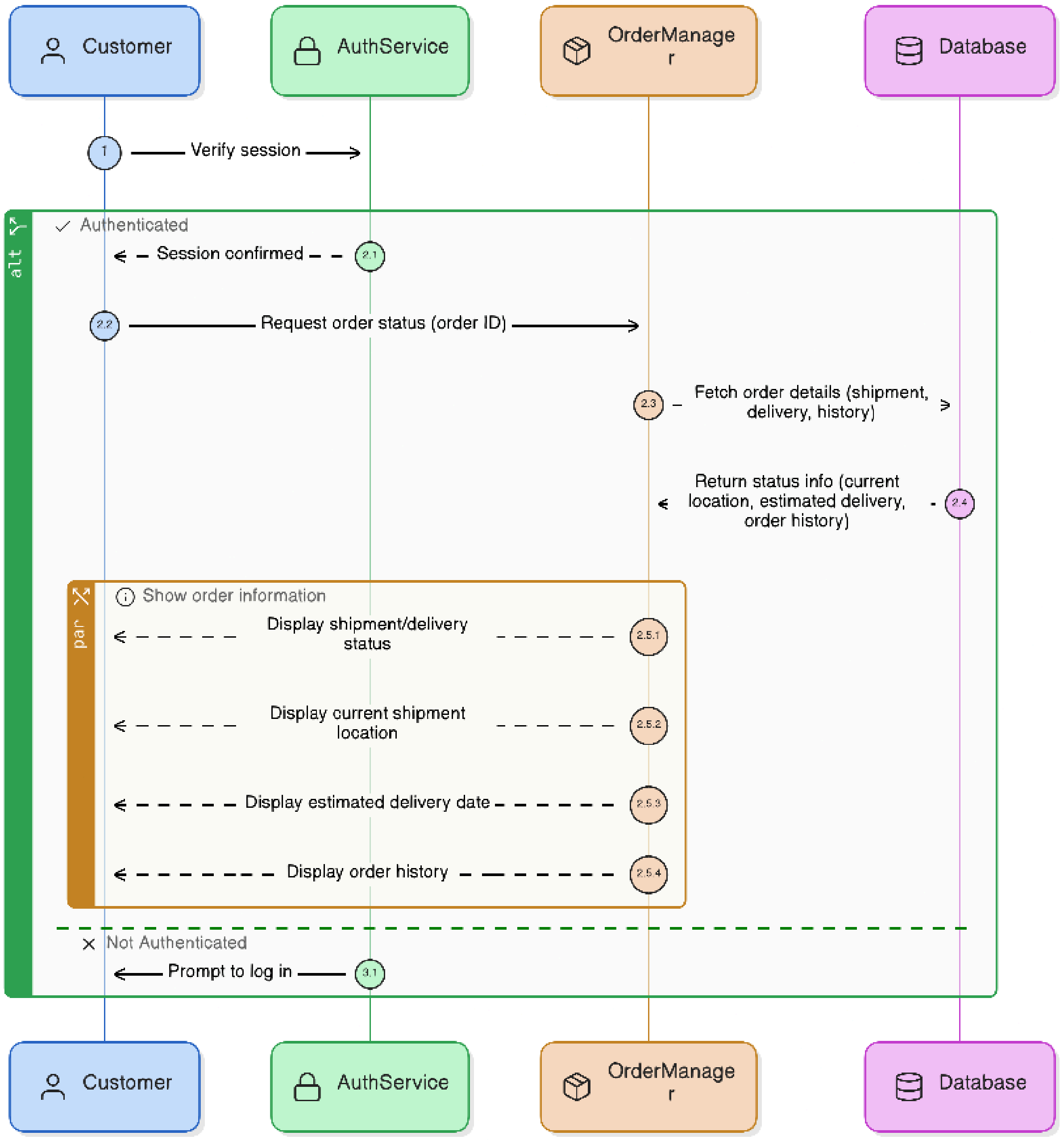
3.8.1.4 Customer - Add items to cart



3.8.1.5 Customer - Checkout and place order

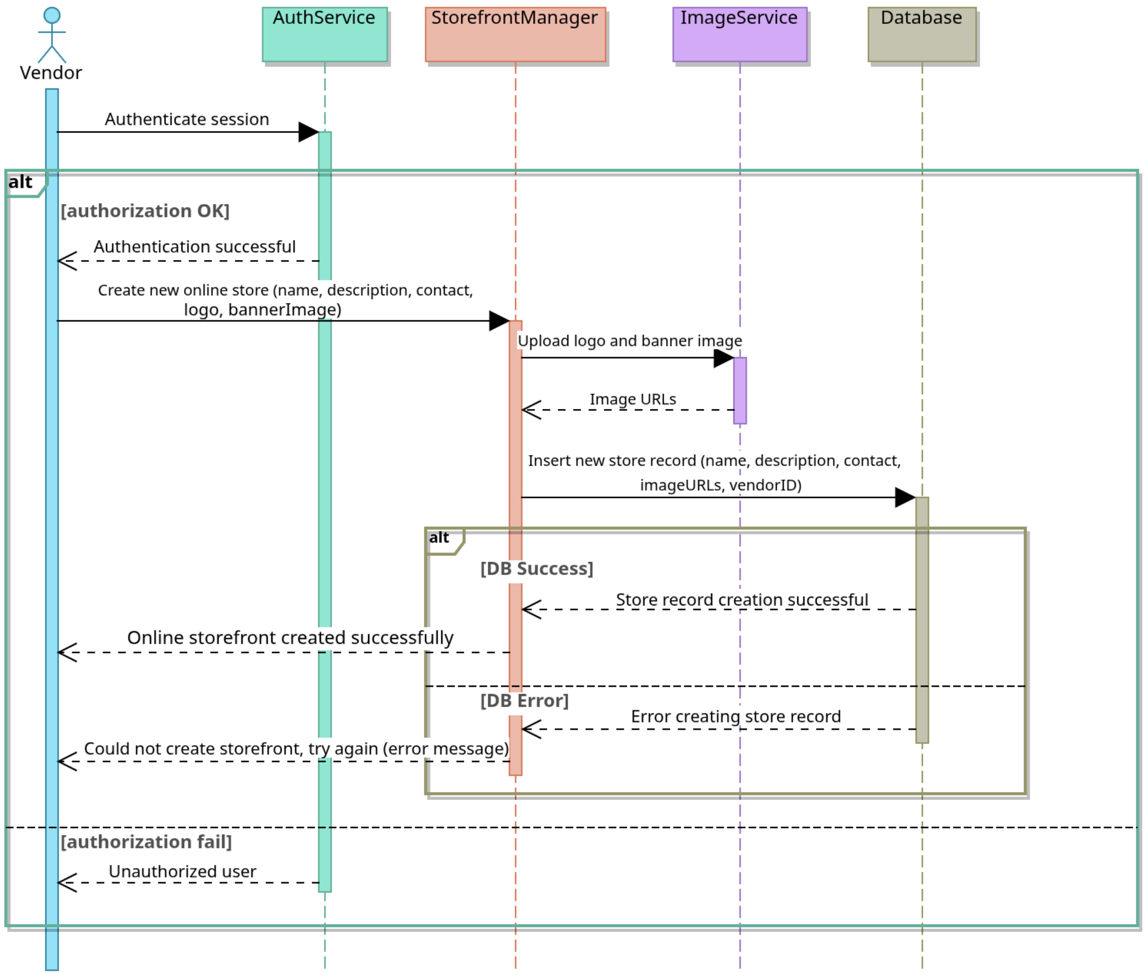


3.8.1.6 Customer - View order status


Customer
AuthService
OrderManager
Database
1
Verify session
alt
Authenticated
2.1
Session confirmed
2.2
Request order status (order ID)
2.3
Fetch order details (shipment, delivery, history)
2.4
Return status info (current location, estimated delivery, order history)
par
Show order information
2.5.1
Display shipment/delivery status
2.5.2
Display current shipment location
2.5.3
Display estimated delivery date
2.5.4
Display order history
Not Authenticated
3.1
Prompt to log in
Customer
AuthService
OrderManager
Database

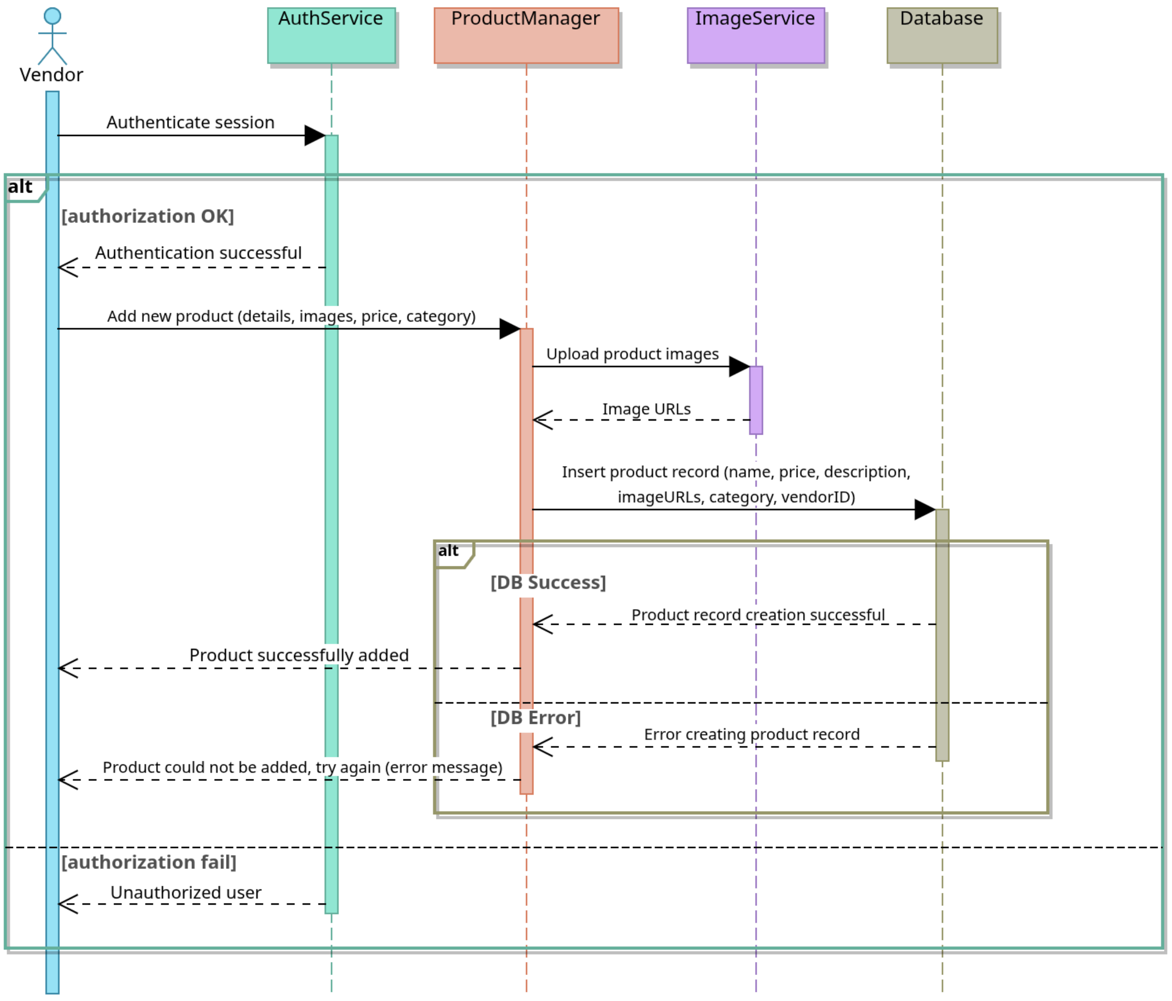
3.8.2 Vendor

3.8.2.1 Vendor - Create Storefront

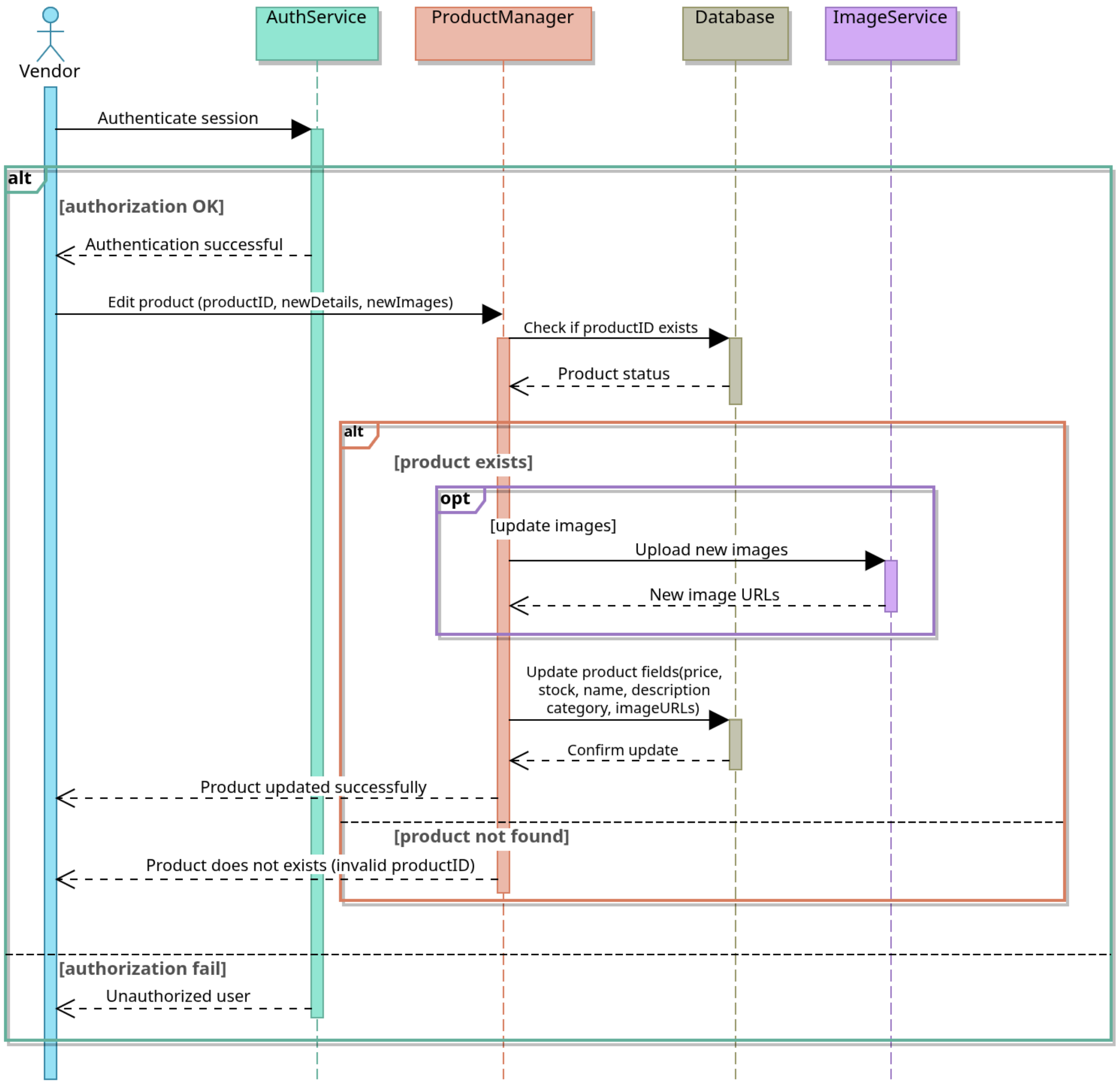


Vendor
AuthService
StorefrontManager
ImageService
Database
Authenticate session
alt
[authorization OK]
Authentication successful
Create new online store (name, description, contact, logo, bannerImage)
Upload logo and banner image
Image URLs
Insert new store record (name, description, contact, imageURLs, vendorID)
alt
[DB Success]
Store record creation successful
Online storefront created successfully
[DB Error]
Error creating store record
Could not create storefront, try again (error message)
[authorization fail]
Unauthorized user

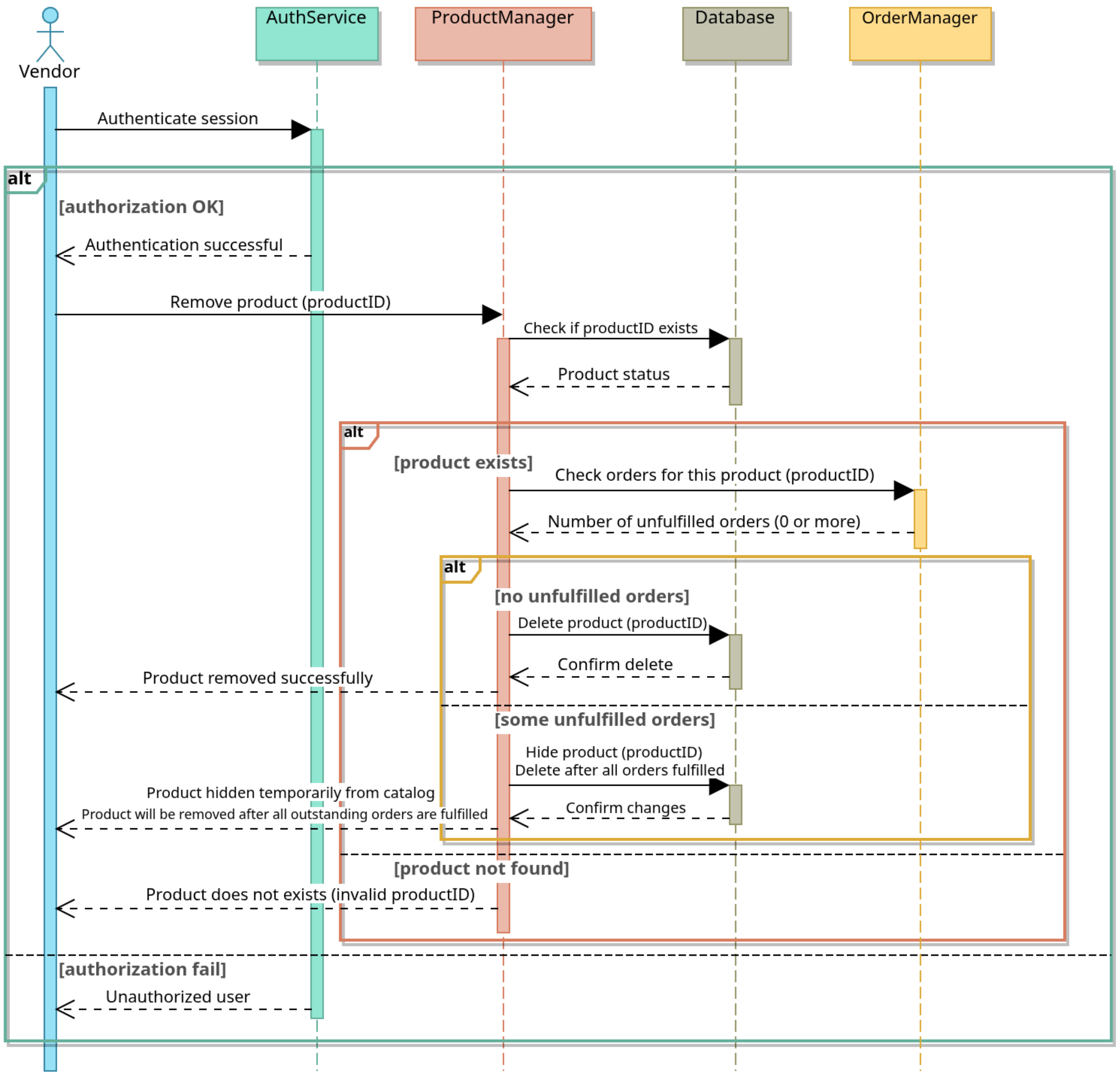
3.8.2.2 Vendor - Add new product listing


Vendor
AuthService
ProductManager
ImageService
Database
Authenticate session
alt
[authorization OK]
Authentication successful
Add new product (details, images, price, category)
Upload product images
Image URLs
Insert product record (name, price, description, imageURLs, category, vendorID)
alt
[DB Success]
Product record creation successful
Product successfully added
[DB Error]
Error creating product record
Product could not be added, try again (error message)
[authorization fail]
Unauthorized user

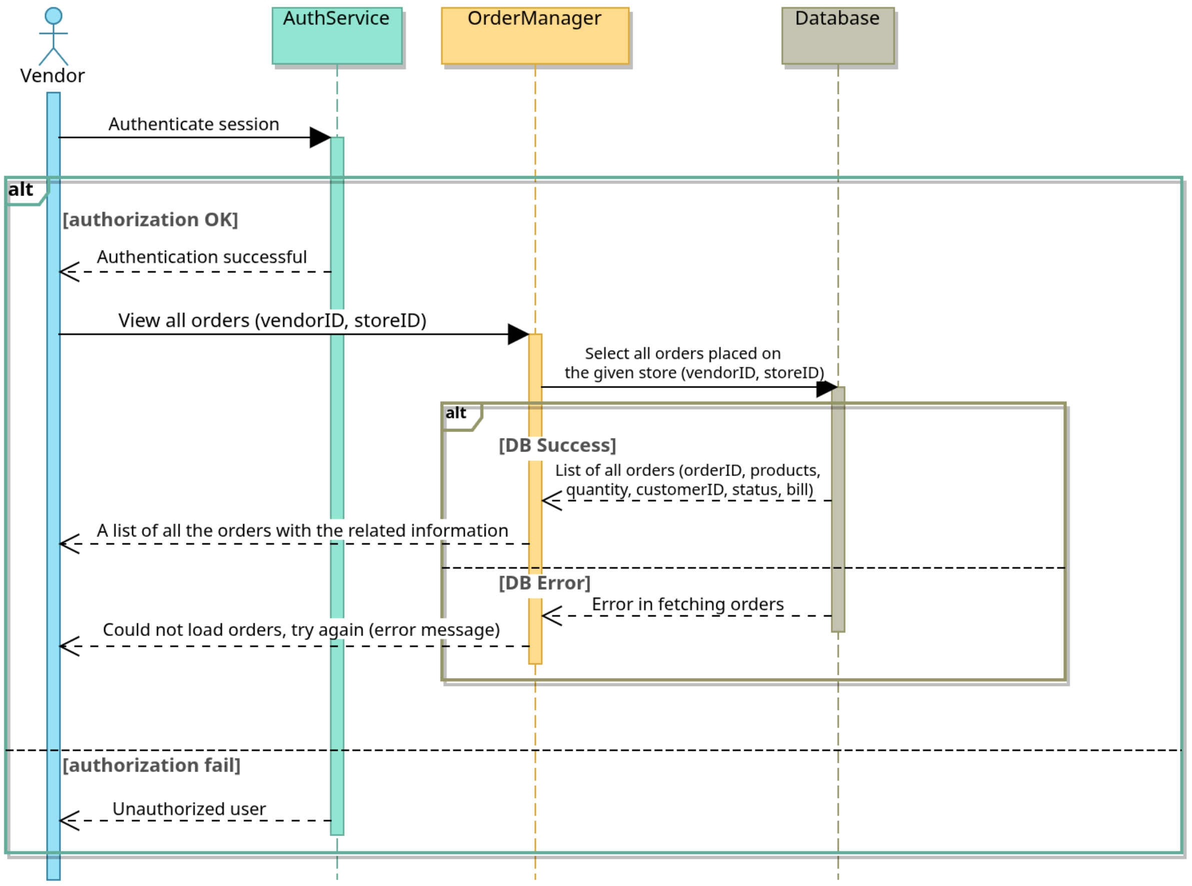
3.8.2.3 Vendor - Edit product listing



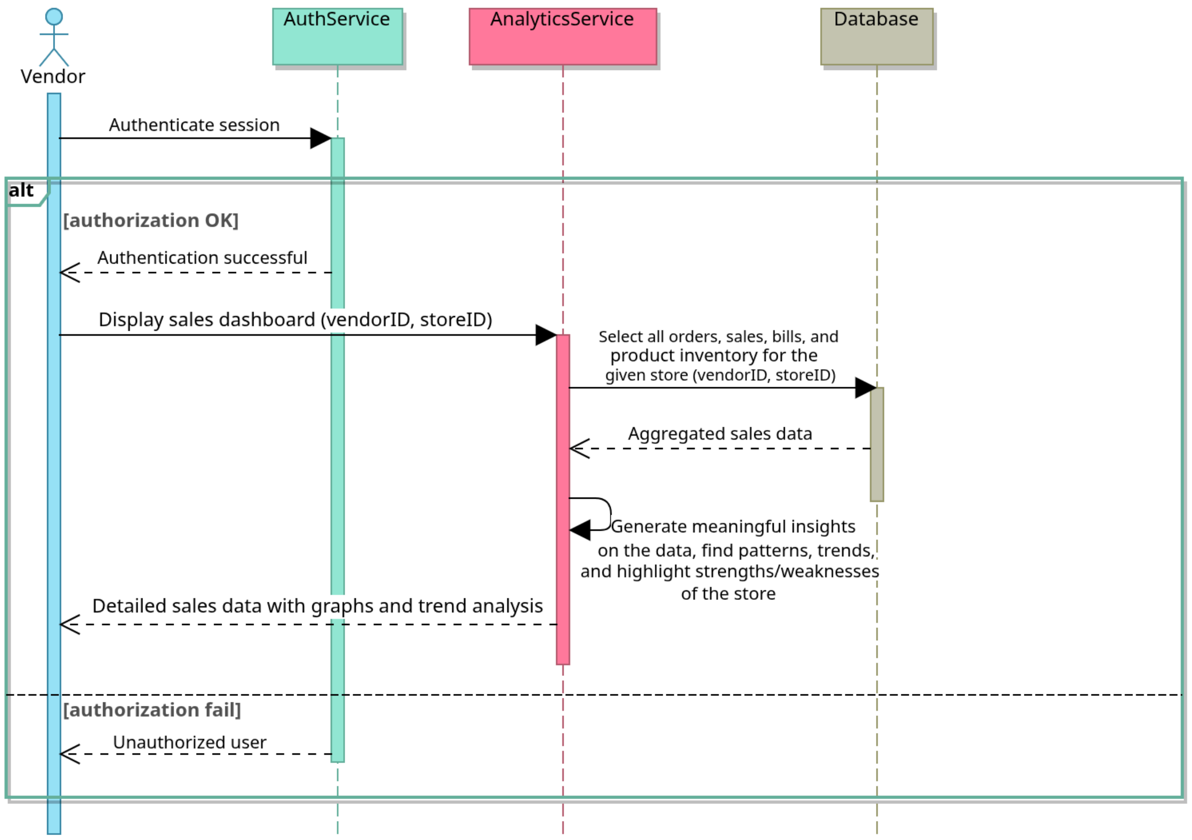
3.8.2.4 Vendor - Remove product listing


Vendor
AuthService
ProductManager
Database
OrderManager
Authenticate session
alt
[authorization OK]
Authentication successful
Remove product (productID)
Check if productID exists
Product status
alt
[product exists]
Check orders for this product (productID)
Number of unfulfilled orders (0 or more)
alt
[no unfulfilled orders]
Delete product (productID)
Confirm delete
Product removed successfully
[some unfulfilled orders]
Hide product (productID)
Delete after all orders fulfilled
Confirm changes
Product hidden temporarily from catalog
Product will be removed after all outstanding orders are fulfilled
[product not found]
Product does not exists (invalid productID)
[authorization fail]
Unauthorized user

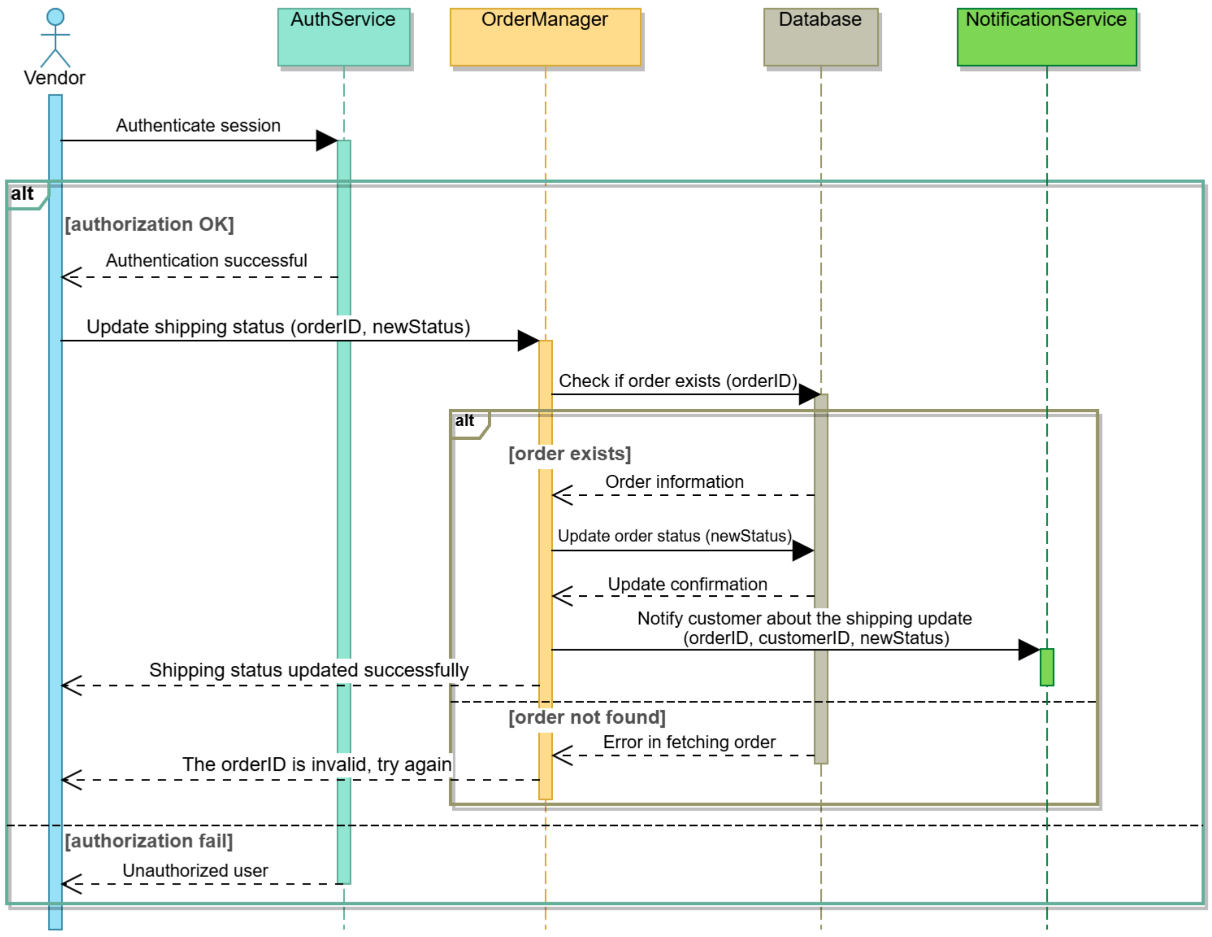
3.8.2.5 Vendor - View customer orders


Vendor
AuthService
OrderManager
Database
Authenticate session
alt
[authorization OK]
Authentication successful
View all orders (vendorID, storeID)
Select all orders placed on the given store (vendorID, storeID)
alt
[DB Success]
List of all orders (orderID, products, quantity, customerID, status, bill)
A list of all the orders with the related information
[DB Error]
Error in fetching orders
Could not load orders, try again (error message)
[authorization fail]
Unauthorized user

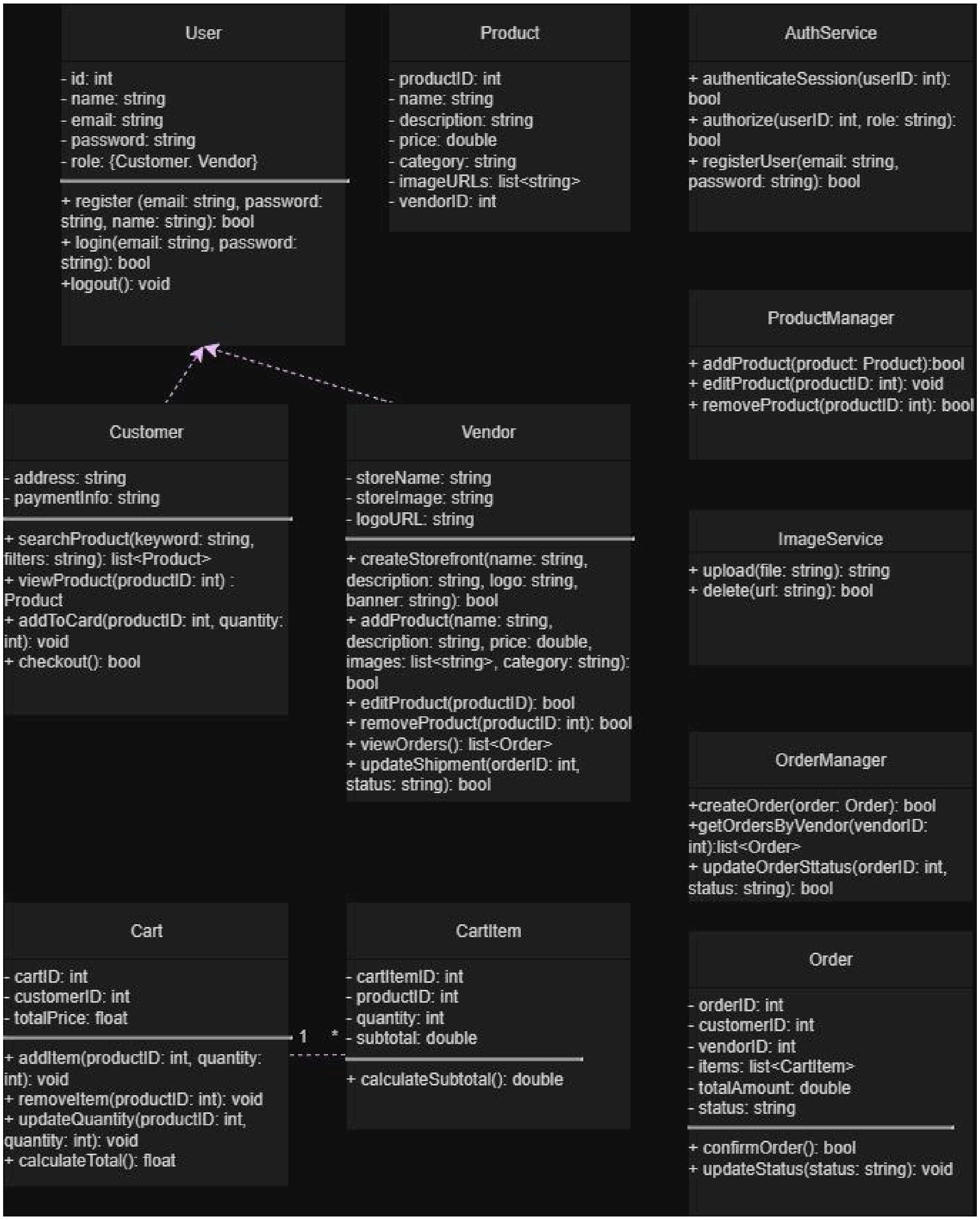
3.8.2.6 Vendor - View sales analytics


Vendor
AuthService
AnalyticsService
Database
Authenticate session
alt
[authorization OK]
Authentication successful
Display sales dashboard (vendorID, storeID)
Select all orders, sales, bills, and product inventory for the given store (vendorID, storeID)
Aggregated sales data
Generate meaningful insights on the data, find patterns, trends, and highlight strengths/weaknesses of the store
Detailed sales data with graphs and trend analysis
[authorization fail]
Unauthorized user

3.8.2.7 Vendor - Update shipping status


Vendor
AuthService
OrderManager
Database
NotificationService
Authenticate session
alt
[authorization OK]
Authentication successful
Update shipping status (orderID, newStatus)
Check if order exists (orderID)
alt
[order exists]
Order information
Update order status (newStatus)
Update confirmation
Notify customer about the shipping update (orderID, customerID, newStatus)
Shipping status updated successfully
[order not found]
Error in fetching order
The orderID is invalid, try again
[authorization fail]
Unauthorized user

3.9. Class Diagram

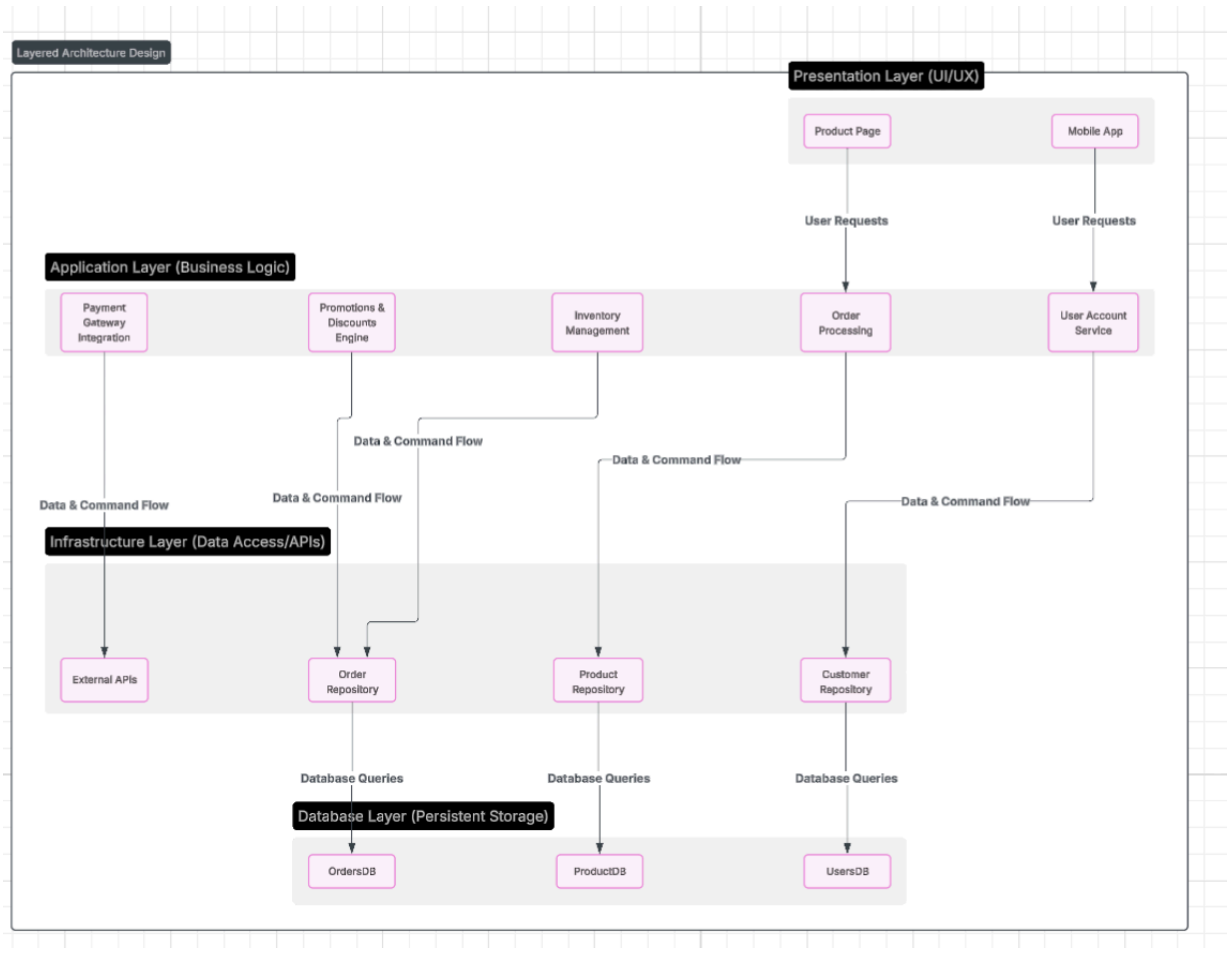

User
- id: int
- name: string
- email: string
- password: string
- role: {Customer. Vendor}
+ register (email: string, password: string, name: string): bool
+ login(email: string, password: string): bool
+logout(): void
Product
- productID: int
- name: string
- description: string
- price: double
- category: string
- imageURLs: list<string>
- vendorID: int
AuthService
+ authenticateSession(userID: int): bool
+ authorize(userID: int, role: string): bool
+ registerUser(email: string, password: string): bool
ProductManager
+ addProduct(product: Product):bool
+ editProduct(productID: int): void
+ removeProduct(productID: int): bool
Customer
- address: string
- paymentInfo: string
+ searchProduct(keyword: string, filters: string): list<Product>
+ viewProduct(productID: int) : Product
+ addToCard(productID: int, quantity: int): void
+ checkout(): bool
Vendor
- storeName: string
- storeImage: string
- logoURL: string
+ createStorefront(name: string, description: string, logo: string, banner: string): bool
+ addProduct(name: string, description: string, price: double, images: list<string>, category: string): bool
+ editProduct(productID): bool
+ removeProduct(productID: int): bool
+ viewOrders(): list<Order>
+ updateShipment(orderID: int, status: string): bool
ImageService
+ upload(file: string): string
+ delete(url: string): bool
OrderManager
+createOrder(order: Order): bool
+getOrdersByVendor(vendorID: int):list<Order>
+ updateOrderSttatus(orderID: int, status: string): bool
Cart
- cartID: int
- customerID: int
- totalPrice: float
+ addItem(productID: int, quantity: int): void
+ removeItem(productID: int): void
+ updateQuantity(productID: int, quantity: int): void
+ calculateTotal(): float
1
*
CartItem
- cartItemID: int
- productID: int
- quantity: int
- subtotal: double
+ calculateSubtotal(): double
Order
- orderID: int
- customerID: int
- vendorID: int
- items: list<CartItem>
- totalAmount: double
- status: string
+ confirmOrder(): bool
+ updateStatus(status: string): void

3.10. Architectural Design

3.10.1. Chosen Architecture:

The chosen Architecture Design for the E-Commerce Project is a **Layered Architecture**.

3.10.2. Design Diagram



3.10.3. Rationale

The reason why a layered architecture is the most appropriate choice for our e-commerce app can be explained in 5 points:

1. There's a clear distinction of concerns in manageable parts, making it to where the team can focus on their designated areas without fear of interrupting other member's area of work.
2. It's enhanced maintainability & evolution which allows for changes in one layer to not affect other layers, making it way easier to adopt new features and technologies without disrupting other layers.
3. The layered design allows for each layer to be scaled independently meaning an improved scalability.
4. Layered models allow for better security due to clear boundaries which prevents sensitive data being accessed from external interfaces.

Lastly a layered design allows for testing to be done in isolated environments, meaning a faster paced developmental process.

4. Project Scheduling and Pricing Estimation

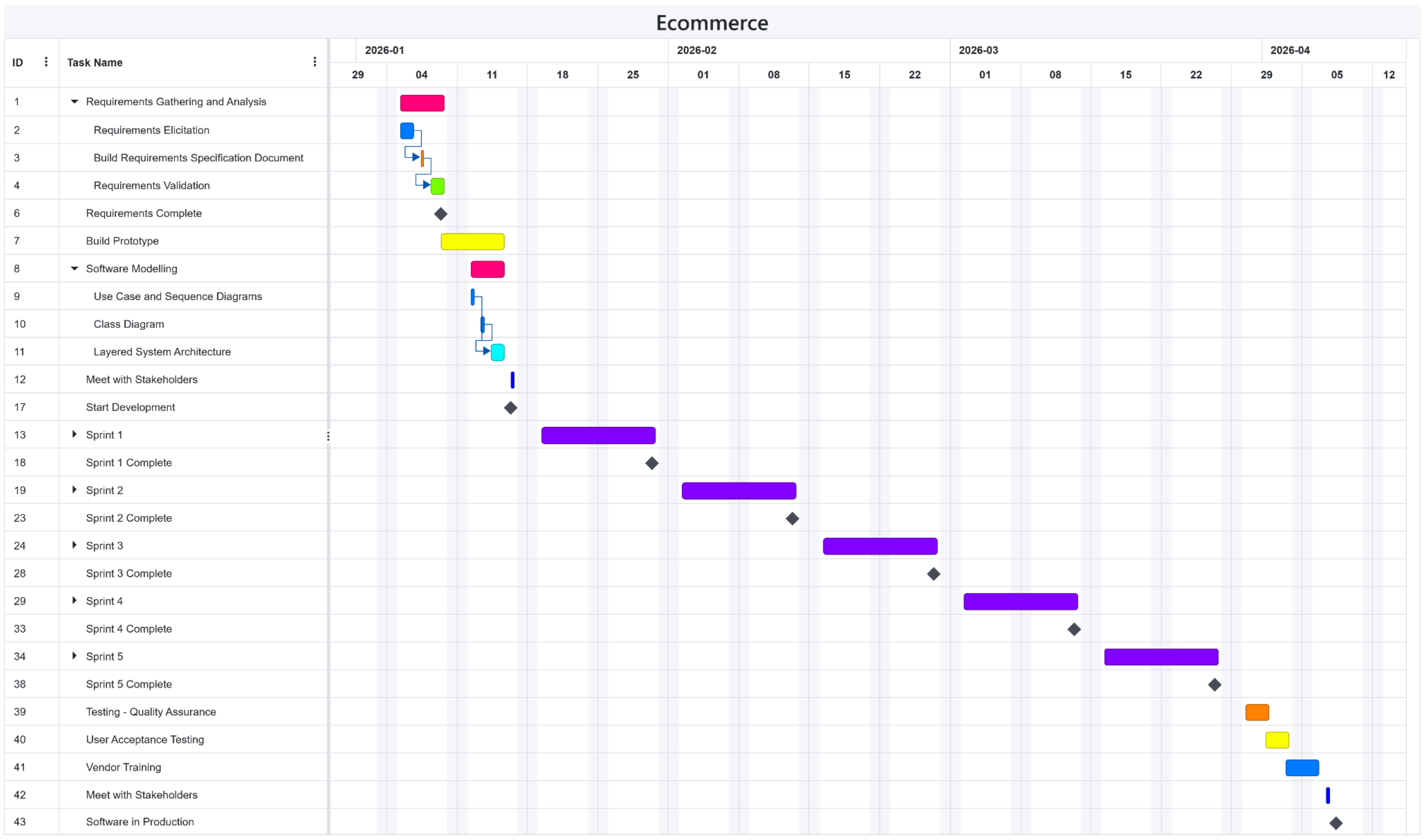
4.1. Project Scheduling

Start date: 5 January 2026

End date: 8 April 2026

Total time: 14 weeks with 40hrs/week with no weekend work included

Sprint: 5 sprints with each sprint being 2 weeks.



ID	Task Name
1	Requirements Gathering and Analysis
2	Requirements Elicitation
3	Build Requirements Specification Document
4	Requirements Validation
6	Requirements Complete
7	Build Prototype
8	Software Modelling
9	Use Case and Sequence Diagrams
10	Class Diagram
11	Layered System Architecture
12	Meet with Stakeholders
17	Start Development
13	Sprint 1
18	Sprint 1 Complete
19	Sprint 2
23	Sprint 2 Complete
24	Sprint 3
28	Sprint 3 Complete
29	Sprint 4
33	Sprint 4 Complete
34	Sprint 5
38	Sprint 5 Complete
39	Testing - Quality Assurance
40	User Acceptance Testing
41	Vendor Training
42	Meet with Stakeholders
43	Software in Production

Requirements Gathering & Analysis (1 week): One week is reasonable because multiple stakeholders must be interviewed, business rules clarified, and requirement conflicts resolved, so it may take several iterative discussions.

Software Modeling (1 week): One week is reasonable because once requirements are confirmed, creating use case diagrams, class diagrams, and the system architecture is a focused task that can be completed efficiently with minimal stakeholder interaction.

Development (10 weeks): Ten weeks is reasonable because the system is divided into multiple sprints, and major e-commerce modules, such as authentication, product catalog, cart, checkout, payment integration, and vendor side features, each require time to build, refine, and integrate without compromising quality or stability.

Testing (1 week): One week is reasonable because although most issues are resolved during sprints, this final phase includes dedicated Quality Assurance (QA) to verify the full system end-to-end and User Acceptance Testing (UAT) to ensure stakeholders approve before deployment.

Final Activities (1.5 weeks): One and a half weeks is reasonable because final activities include deployment preparation, environment setup, vendor training sessions, and ensuring the system operates smoothly and reliably before going live.

4.2. Cost, Effort and Pricing Estimation

We are using the Function Point (FP) method to calculate the estimated cost and price for the project.

Gross Function Point:

	Function Category	Count	Complexity			Count X Complexity
			Simple	**Average**	Complex	
1	Number of user input	12	3	**4**	6	48
2	Number of user output	10	4	**5**	7	50
3	Number of user queries	13	3	**4**	6	52
4	Number of data files and relational tables	8	7	**10**	15	80
5	Number of external interfaces	3	5	**7**	10	21
					GFP	**251**

Processing Complexity:
0 (no influence), 1 (incidental), = 2 (moderate), = 3 (average), = 4 (significant), = 5 (essential)

	Category	Complexity
1	Does the system require reliable backup and recovery?	5
2	Are data communications required?	3
3	Are there distributed processing functions?	3
4	Is performance critical?	4
5	Will the system run in an existing, heavily utilized operational environment?	3
6	Does the system require online data entry?	4
7	Does the online data entry require the input transaction to be built over multiple screens or operations?	4
8	Are the master files updated online?	3
9	Are the inputs, outputs, files, or inquiries complex?	3
10	Is the internal processing complex?	3
11	Is the code designed to be reusable?	4
12	Are conversion and installation included in the design?	3
13	Is the system designed for multiple installations in different organizations?	4
14	Is the application designed to facilitate change and ease of use by the user?	5
	Total PC	**51**

Gross Functional Point (GFP) = **251**

Processing Complexity Adjustment (PCA) = 0.65 + (0.01 x 51) = **1.16**

Functional Point (FP) = GFP x PCA = 251 x 1.16 = **291.16**

Productivity = 5 FP per person-week
Effort = FP / Productivity = 291.16 / 5 = 58.23 ≈ **58 person-weeks**

Team size = 6
Duration = Effort / team size = 58 / 6 ≈ **10 weeks**
≈ **2.5 months**

Billing rate = $70 /hr = $2800 /week (for 40 hr/week)
Cost = 2800 x 10 x 6 = **$168,000.00**

Assumptions:

1. All the categories in the gross function point table have an average complexity.
2. The empirical constant for calculating Processing Complexity Adjustment (PCA) is 0.65 because 0.65 ≤ PCA ≤ 1.35.
3. The developer productivity is 5 function points per person-week because we are using the agile methodology and it will take around 6-8 hours for a developer to complete a story point with full development and testing.
4. The development team size is 6 because small teams are suitable for the agile method.
5. The billing rate for our developers is $70 /hr because our development team will most consist of medium skilled developers and $70 /hr is a reasonable market average.

4.3. Cost of Hardware Products

1. **Application Compute (Web/API Servers)**
    a. Product: Google Compute Engine/AWS EC2
    b. Purpose: The platform supplies the foundational virtual server instances (vCPUs and RAM) that execute the application code. These instances are essential for automatic scaling to ensure reliable service for 500 concurrent users.
    c. Cost: $300 – $500 / month

2. **Managed Database**
    a. Product: Google Cloud SQL and AWS RDS (such as PostgreSQL)
    b. Purpose: To safely store all product inventory, user accounts, and order transaction data in a fully managed, high-availability database service. It also takes care of the need for dependable backup and recovery.
    c. Cost: $80 – $150 / month

3. **Content Delivery Network (CDN)**
    a. Product: AWS CloudFront / Cloudflare CDN
    b. Purpose: Caches static assets at global edge locations, including product images, JavaScript, and CSS. To meet the 3-second load time NFR, these files must be delivered promptly.
    c. Cost: $10 - $50 / month

4. **Load Balancing & High-Availability**
    a. Product: Cloud Load Balancer (e.g., AWS ELB)
    b. Purpose: The goal is to automatically distribute incoming customer traffic among the application servers in order to minimize downtime, guarantee system uptime, and manage traffic spikes for high concurrency.
    c. Cost: $15 – $30 / month

Category	Estimated Monthly Cost	Rationale
Compute/Server (Managed Cloud)	$400	Provides scalable virtual machines (vCPUs/RAM) to handle up to 500 concurrent users.
Database (Managed Service)	$100	Cost for a managed database (Neon Postgresql) for product, order, and user data.
Content Delivery Network (CDN)	$30	Caches product images and assets globally to ensure the 3-second load time NFR.
Load Balancer & Network	$20	Distributes traffic across servers to manage the 500 concurrent users and ensures 24/7 uptime.
TOTAL (Monthly)	**$550**	

4.4. Cost of Software Products

Internal Admin & Operations
Internal Admin Tools (Back-Office Software) need a GUI for our support team to view orders, issue refunds, and ban users without touching the raw database.

- Product: Retool or Forest Admin.
- Purpose: Provides a drag-and-drop internal dashboard for our customer support and operations teams.
- Cost: $50 - $65 / user / month (Business/Pro Plans).
    - Estimate for 3 Admin Staff: ~$150 - $200 / month.

Tax & Legal Compliance
Automated Tax Compliance Calculating sales tax for 1,000s of users across different states/countries is impossible to do manually.

- Product: TaxJar Professional or Avalara AvaTax.
- Purpose: Real-time API that calculates the exact tax rate at checkout and automates state filing.
- Cost: $99 - $350 / month (Tiered by transaction volume, e.g., ~1k-5k orders).

Legal & Privacy Consent Required for GDPR/CCPA compliance when handling user data.

- Product: Termly or Iubenda.
- Purpose: Manages cookie consent banners, privacy policies, and data deletion requests automatically.
- Cost: $20 - $100 / month (Pro/Ultimate Plans).

Application Health (Software Monitoring)
Application Error Tracking Software that reports "Code Errors" (not server crashes).

- Product: Sentry (Team/Business Tier).
- Purpose: Alerts developers immediately when a user sees a "Something went wrong" error, including the exact line of code responsible.
- Cost: $29 - $89 / month.

Specialized Security Software
Web Application Firewall (WAF) Software layer that sits in front of our hardware to block hackers and Distributed Denial of Service (DDoS) attacks.

- Product: Cloudflare Pro.
- Purpose: Provides a “Software Firewall” (WAF) to block SQL injection attacks and automated bot scraping before they hit our servers.
- Cost: $20 - $25 / month.

Transactional Email Software
Transactional Email Service Specialized software to ensure receipts land in Inboxes, not Spam.

- Product: Postmark or SendGrid.
- Purpose: High-deliverability email API strictly for "system" messages (password resets, receipts).
- Cost: $15 - $50 / month (for ~10k - 50k emails).

Total Software Product Budget
Monthly “SaaS” Budget Summary Recurring licenses required to run the business legally and efficiently.

Category	Software Product	Estimated Monthly Cost
Internal Admin	Retool (3 seats)	$150
Tax Automation	TaxJar Professional	$200
Privacy/Legal	Termly Pro	$20
Error Tracking	Sentry Team	$29
Security WAF	Cloudflare Pro	$20
Email API	Postmark	$15
TOTAL:		**~$434 / Month**

4.5. Cost of Personnel

Required Personnel & Salaries (Full-Time Employees)

The following reflects typical U.S. market compensation for a mid-size software company:

Role	# Employees	Avg. Salary	Rationale
Software Developers	6	$110,000	Core engineering team for backend, frontend, and integrations
QA Engineers	2	$85,000	Testing, automation, regression, and load testing
Project Manager / Scrum Master	1	$100,000	Scheduling, risk management, coordination
DevOps / Cloud Engineer	1	$120,000	AWS deployment, CI/CD, scaling
Security Engineer	1	$125,000	Payment security, PCI-DSS, API hardening
Vendor Onboarding & Support Specialist	1	$60,000	Vendor training, onboarding, issue resolution

Total Annual Salaries (Before Overhead):
(6×110,000) + (2×85,000) +100,000 +120,000 + 125,000 + 60,000= $1,255,000

Employment Overhead (Benefits, Taxes & Equipment)
Industry-standard overhead for full-time tech employees is **25–35%**.
We apply **30%** for benefits, payroll tax, insurance, hardware, and HR expenses.

1,255,000 × 0.30=$ 376,500

Total Annual Personnel Cost:
1,255,000 + 376,500 =$1,631,500

Cost Allocated to This Project (10-Week Duration)
Based on Deliverable 2, the project is estimated to last **10 weeks**.
10 weeks / 52 weeks =19.23% of annual workload

Project-Specific Personnel Cost:
1,631,500 × 0.1923 ≈ **$ 313,700**

Additional Training & Certification Costs (For Full Employees)

Training Area	Description	Cost
AWS Certification (8 engineers)	Developers + DevOps	$1,200 × 8 = **$9,600**
Payment Security / PCI-DSS Training	Required for handling transactions	$800 × 7 = **$5,600**
Secure Architecture & Scalability Workshops	Distributed systems training	**$3,000**
Vendor Platform Training Materials	Documentation + tutorials	**$1,500**

Total Training Cost: $19,700

Final Personnel Cost Estimate:

Category	Cost
Prorated Personnel Cost (10 weeks)	$313, 700
Training & Certification	$19,700
Total Estimated Cost	**$333,400**

Total Personnel Cost for the E-Commerce Project: $333,400

The total personnel cost allocated to this 10-week development effort is **approximately $333,400**. This includes salaries, benefits, cloud/security training, and vendor onboarding support.

5. Test Plan

5.1. Test Plan Description

The test plan uses **Jest** as the main testing framework. It tests the backend API by making live HTTP requests using the **Supertest** library.

The plan involves:

- **Database Management**: Using drizzle-orm, test data (like users, products, categories) is added to the database before tests run (beforeAll) and cleaned up after all tests are finished (afterAll). This makes sure the tests run in a clean, known state.
- **API Endpoint Testing**: Tests are grouped by API routes (e.g., Auth Routes, Product Routes).
- **Request Simulation**: supertest sends requests (like POST and GET) to the API endpoints (e.g., /api/auth/signup, /api/products/:id).
- **Assertions**: After getting a response, Jest's expect function checks if the HTTP status code is correct (like 200 for success or 400 for a bad request) and if the response body contains the expected data (like a user ID, a product name, or an error message).

5.2. Test Code

Auth Routes Test

```
describe("POST /api/auth/signup", () => {
  // Test case: Successful user signup with valid information
  // Expected result: Status 201, user data and JWT token returned
  it("should successfully create a new user account", async () => {
    const response = await request(app)
      .post("/api/auth/signup")
      .send(testUser)
      .expect(201);

    expect(response.body).toHaveProperty("userId");
    expect(response.body).toHaveProperty("name", testUser.name);
    expect(response.body).toHaveProperty("jwt");
    expect(typeof response.body.jwt).toBe("string");
  });

  // Test case: Signup with duplicate email
  // Expected result: Status 400, error message "User already exists"
  it("should reject signup with duplicate email", async () => {
    const response = await request(app)
      .post("/api/auth/signup")
      .send(testUser)
      .expect(400);
    expect(response.body).toHaveProperty("message");
    expect(response.body.message).toBe("User already exists");
  });
});

describe("POST /api/auth/login", () => {
  // Test case: Login with correct credentials
  // Expected result: Status 200, user data and JWT token returned
  it("should successfully login with correct credentials", async () => {
    await request(app).post("/api/auth/signup").send(anotherTestUser);

    const response = await request(app)
      .post("/api/auth/login")
      .send({
        email: anotherTestUser.email,
        password: anotherTestUser.password,
      })
      .expect(200);

    expect(response.body).toHaveProperty("userId");
    expect(response.body).toHaveProperty("name", anotherTestUser.name);
    expect(response.body).toHaveProperty("jwt");
    expect(typeof response.body.jwt).toBe("string");
  });

  // Test case: Login with non-existent email
  // Expected result: Status 400, error message "Invalid Credentials"
  it("should reject login with non-existent email", async () => {
    const response = await request(app)
      .post("/api/auth/login")
      .send({
        email: "nonexistent@example.com",
        password: "password123",
      })
      .expect(400);

    expect(response.body).toHaveProperty("message");
    expect(response.body.message).toBe("Invalid Credentials");
  });
});

describe("GET /api/auth/profile", () => {
  let authToken;

  // Test case: Get user profile with valid authentication token
  // Expected result: Status 200, user profile data returned
  it("should successfully get user profile with valid token", async () => {
    const response = await request(app)
      .get("/api/auth/profile")
      .set("Authorization", `Bearer ${authToken}`)
      .expect(200);

    expect(response.body).toHaveProperty("userId");
    expect(response.body).toHaveProperty("name");
    expect(response.body).toHaveProperty("email");
    expect(response.body).toHaveProperty("role");
  });

  // Test case: Get profile without authentication token
  // Expected result: Status 401, error message "Unauthorized Request"
  it("should reject profile request without token", async () => {
    const response = await request(app).get("/api/auth/profile").expect(401);

    expect(response.body).toHaveProperty("message");
    expect(response.body.message).toBe("Unauthorized Request");
  });
});
```

Product Routes Test

```
describe("GET /api/products", () => {
  // Test case: Get all products from the API
  // Expected result: Status 200, array of products returned
  it("should successfully get all products", async () => {
    const response = await request(app).get("/api/products").expect(200);

    expect(Array.isArray(response.body)).toBe(true);

    if (response.body.length > 0) {
      const product = response.body[0];
      expect(product).toHaveProperty("productId");
      expect(product).toHaveProperty("productName");
      expect(product).toHaveProperty("price");
      expect(typeof product.price).toBe("number");
    }
  });

  // Test case: Get all products when none exist in database
  // Expected result: Status 400 or 200 (depends on database state), error message
if 400
  it("should return error when no products exist", async () => {
    const response = await request(app).get("/api/products");

    expect([200, 400]).toContain(response.status);

    if (response.status === 400) {
      expect(response.body).toHaveProperty("message");
      expect(response.body.message).toBe("No product found");
    }
  });
});

describe("GET /api/products/:id", () => {
  // Test case: Get product by valid product ID
  // Expected result: Status 200, product details with images returned
  it("should successfully get product by valid ID", async () => {
    if (!testProductId) {
      console.log("Skipping test - test product not created");
      return;
    }

    const response = await request(app)
      .get(`/api/products/${testProductId}`)
      .expect(200);

    expect(response.body).toHaveProperty("productId", testProductId);
    expect(response.body).toHaveProperty("productName");
    expect(response.body).toHaveProperty("description");
    expect(response.body).toHaveProperty("price");
    expect(response.body).toHaveProperty("images");
    expect(Array.isArray(response.body.images)).toBe(true);
  });

  // Test case: Get product with non-existent product ID
  // Expected result: Status 400, error message "Invalid product ID"
  it("should return error for non-existent product ID", async () => {
    const fakeProductId = "00000000-0000-0000-0000-000000000000";

    const response = await request(app)
      .get(`/api/products/${fakeProductId}`)
      .expect(400);

    expect(response.body).toHaveProperty("message");
    expect(response.body.message).toBe("Invalid product ID");
  });
});

describe("POST /api/products", () => {
  // Test case: Create product with valid authentication and vendor role
  // Expected result: Status 201, productId returned
  it("should successfully create a product with valid authentication", async () =>
{
    if (!authToken) {
      console.log("Skipping test - auth token not available");
      return;
    }

    const response = await request(app)
      .post("/api/products")
      .set("Authorization", `Bearer ${authToken}`)
      .send({
        name: "New Test Product",
        brand: "Test Brand",
        description: "Test description for new product",
        price: 50.0,
        stockQuantity: 5,
        category: "electronics",
      })
      .expect(201);

    expect(response.body).toHaveProperty("productId");
  });

  // Test case: Create product without authentication
  // Expected result: Status 401, error message returned
  it("should require authentication to create product", async () => {
    const response = await request(app)
      .post("/api/products")
      .send({
        name: "New Product",
        brand: "Test Brand",
        description: "Test description",
        price: 50.0,
        stockQuantity: 5,
        category: "electronics",
      })
      .expect(401);

    expect(response.body).toHaveProperty("message");
  });
});
```

5.3. Test Cases and their Results

Auth Routes (authRoutes.test.js)

- **POST /api/auth/signup**
  - **Test Case 1**: should successfully create a new user account
  - **Description**: Sends a POST request with a new user's name, email, and password.
  - **Expected Result**: A **201 Created** status. The response body must include the new userId, the user's name, and a jwt (JSON Web Token).

  - **Test Case 2**: should reject signup with duplicate email
  - **Description**: Sends a POST request using the *same email* as the user created in the first test.
  - **Expected Result**: A **400 Bad Request** status. The response body must include a message: "User already exists".
- **POST /api/auth/login**
  - **Test Case 1**: should successfully login with correct credentials
  - **Description**: First, it signs up anotherTestUser. Then, it sends a POST request with that user's correct email and password.
  - **Expected Result**: A **200 OK** status. The response body must include the userId, name, and a jwt token.

  - **Test Case 2**: should reject login with non-existent email
  - **Description**: Sends a POST request with an email that is not in the database.
  - **Expected Result**: A **400 Bad Request** status. The response body must include a message: "Invalid Credentials".
- **GET /api/auth/profile**
  - **Test Case 1**: should successfully get user profile with valid token
  - **Description**: Sends a GET request to the profile endpoint, including a valid jwt token (from a user created in its beforeAll block) in the Authorization header.
  - **Expected Result**: A **200 OK** status. The response body must include the user's userId, name, email, and role.

  - **Test Case 2**: should reject profile request without token
  - **Description**: Sends a GET request to the profile endpoint *without* an Authorization header.
  - **Expected Result**: A **401 Unauthorized** status. The response body must include a message: "Unauthorized Request".

Product Routes (productRoutes.test.js)

- **GET /api/products**
    - **Test Case 1**: should successfully get all products
    - **Description**: Sends a GET request to the main products endpoint.
    - **Expected Result**: A **200 OK** status. The response body must be an array. If the array has products, each product should have a productId, productName, and price.

    - **Test Case 2**: should return error when no products exist
    - **Description**: Sends a GET request to the main products endpoint. This test seems designed to check what happens if the database is empty.
    - **Expected Result**: The status should be either **200 OK** (likely with an empty array) or **400 Bad Request**. If the status is 400, the body must have a message: "No product found".

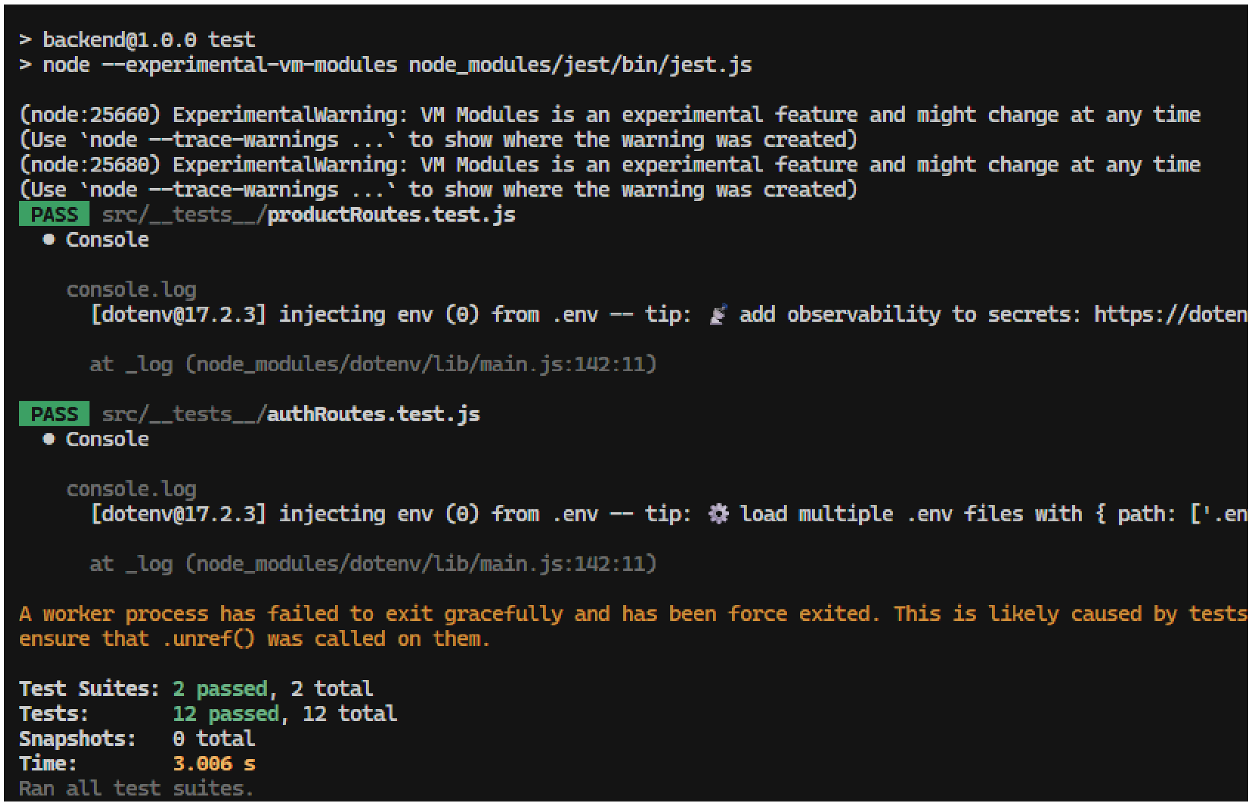
- **GET /api/products/:id**
    - **Test Case 1**: should successfully get product by valid ID
    - **Description**: Sends a GET request to a specific product's URL using a testProductId that was created in the beforeAll setup.
    - **Expected Result**: A **200 OK** status. The response body must have the correct productId and include productName, description, price, and an images array.

    - **Test Case 2**: should return error for non-existent product ID
    - **Description**: Sends a GET request using a "fake" product ID (all zeros).
    - **Expected Result**: A **400 Bad Request** status. The response body must include a message: "Invalid product ID".

- **POST /api/products**
    - **Test Case 1**: should successfully create a product with valid authentication
    - **Description**: Sends a POST request to create a new product. It includes product data (name, brand, price, etc.) and a valid jwt token (for a vendor user) in the Authorization header.
    - **Expected Result**: A **201 Created** status. The response body must include the productId of the newly created product.

    - **Test Case 2**: should require authentication to create product
    - **Description**: Sends a POST request to create a product *without* an Authorization header.
    - **Expected Result**: A **401 Unauthorized** status. The response body must include an error message.

Test Results


```
> backend@1.0.0 test
> node --experimental-vm-modules node_modules/jest/bin/jest.js

(node:25660) ExperimentalWarning: VM Modules is an experimental feature and might change at any time
(Use `node --trace-warnings ...` to show where the warning was created)
(node:25680) ExperimentalWarning: VM Modules is an experimental feature and might change at any time
(Use `node --trace-warnings ...` to show where the warning was created)
 PASS  src/__tests__/productRoutes.test.js
  ● Console

    console.log
      [dotenv@17.2.3] injecting env (0) from .env -- tip: add observability to secrets: https://doten

      at _log (node_modules/dotenv/lib/main.js:142:11)

 PASS  src/__tests__/authRoutes.test.js
  ● Console

    console.log
      [dotenv@17.2.3] injecting env (0) from .env -- tip: load multiple .env files with { path: ['.en

      at _log (node_modules/dotenv/lib/main.js:142:11)

A worker process has failed to exit gracefully and has been force exited. This is likely caused by tests
ensure that .unref() was called on them.

Test Suites: 2 passed, 2 total
Tests:       12 passed, 12 total
Snapshots:   0 total
Time:        3.006 s
Ran all test suites.
```

NOTE: We had both positive and negative test cases. Here the negative test cases also pass because our software effectively catches those test cases and returns the expected status code of 400, 404, or 500.

6. Comparison of our Work with Similar Designs

Shopify - an e-commerce platform that provides tools for businesses to create and manage online stores to sell products.

Similarities:

- Provides services for vendors to manage their products and customer to browse/order products
- Vendors are in complete control of their products, orders, and inventory.

Differences:

- Shopify is a plug-and-play system where the developers have to utilize prebuilt services, limiting the ability to customize according to customer requirements. On the other hand our e-commerce platform is built from the ground-up, as per customer requirements, allowing for greater customization.

Best Buy - a multinational electronics retailer with both offline and online presence.

Similarities:

- Both, Best Buy and our ecommerce platform are built using a Layered Architecture pattern, allowing for separation of concerns, easy maintenance, and high scalability [1].
- The use of CDNs leads to faster loading times, and reduced hosting costs.

Differences:

- Best Buy is a mature system, built over years. It has daily releases, multiple versions, and low cost of change. Our e-commerce platform is still in the early stages, and will require significant time and development to achieve the same velocity of change as Best Buy.

7. Conclusion

Overall, this project has been very successful so far. We were able to develop excellent functional and non-functional requirements, system design and architecture, time and cost estimations, a test plan, and a working prototype. The requirements of our project were quite descriptive, allowing for a single interpretation and no confusion among our team members. Our layered architecture design choice clearly reflects the points mentioned in our non-functional requirements. We chose the agile methodology for our project, and all project scheduling and pricing estimations were based on our use of the agile methodology. This enabled us to make accurate and consistent estimates for the project. Our test plan is highly detailed and tests two different modules with multiple positive and negative test cases. All of these test cases performed as expected, helping us catch several bugs while building the prototype. These test cases are essential for the success of our agile workflow, which includes test-driven development.

One thing we could have done is to specify the types of products our e-commerce platform will sell (such as electronics, clothing, furniture, etc.). This would have helped us narrow down the scope of our project and build a more specific software product. Another thing we noticed is that these e-commerce platforms rely on the trust of their users. During this project, we should have considered how customers can gain confidence in our platform. We should have included a review system for products and vendors, enabling transparency between customers and vendors. We should have also considered the credit card payment processing fee, which is generally 2.9% per transaction. We should have decided who will cover this overhead, whether it is the customer, vendor, or the platform.

References

[1] J. Crabb, “BESTBUY.COM’S CLOUD ARCHITECTURE,” Apr. 29, 2013. https://joelcrabb.com/wp-content/uploads/2024/02/bestbuycloudarch.pdf