Course: **Guided Research I**

Title: **Scalability experiment of microservice architecture on an online bookstore application**

Midterm Report

Student: **Tural Mehtiyev**

Instructors: Dr. Stephen Kaisler, Dr. Jamal Hasanov

Table of Contents

Project Objectives

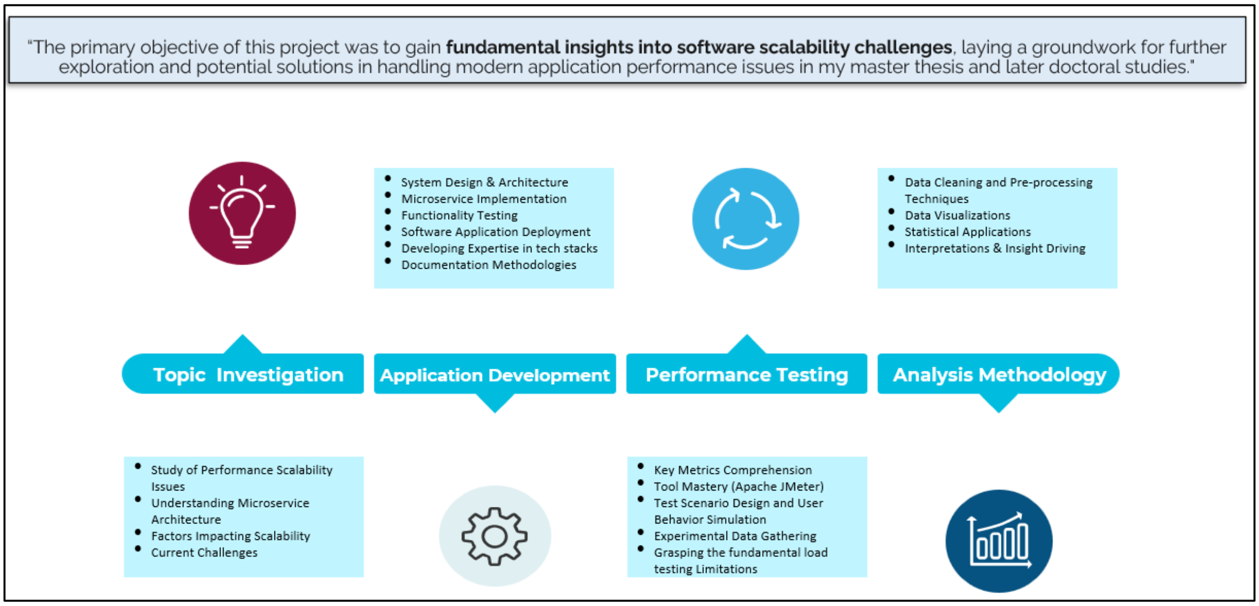


Figure 1

Project Status

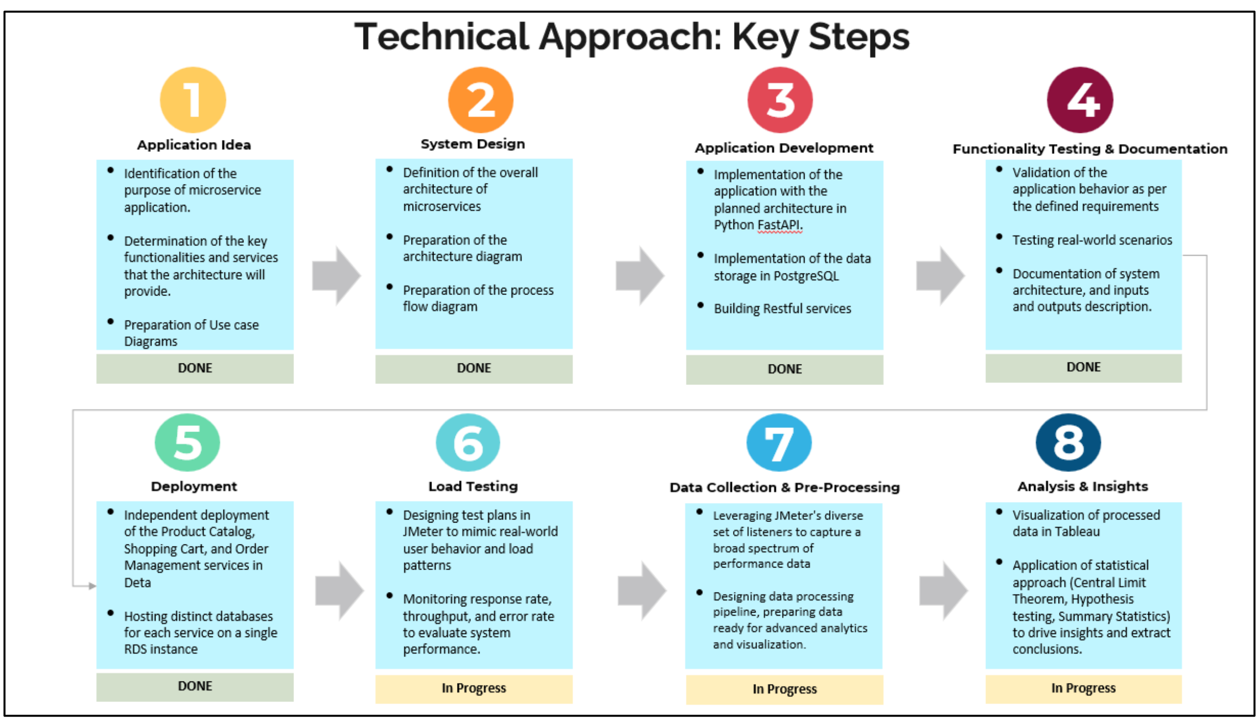


Figure 2

Midterm Deliverables

Application Idea & System Design

Architecture Description Diagram

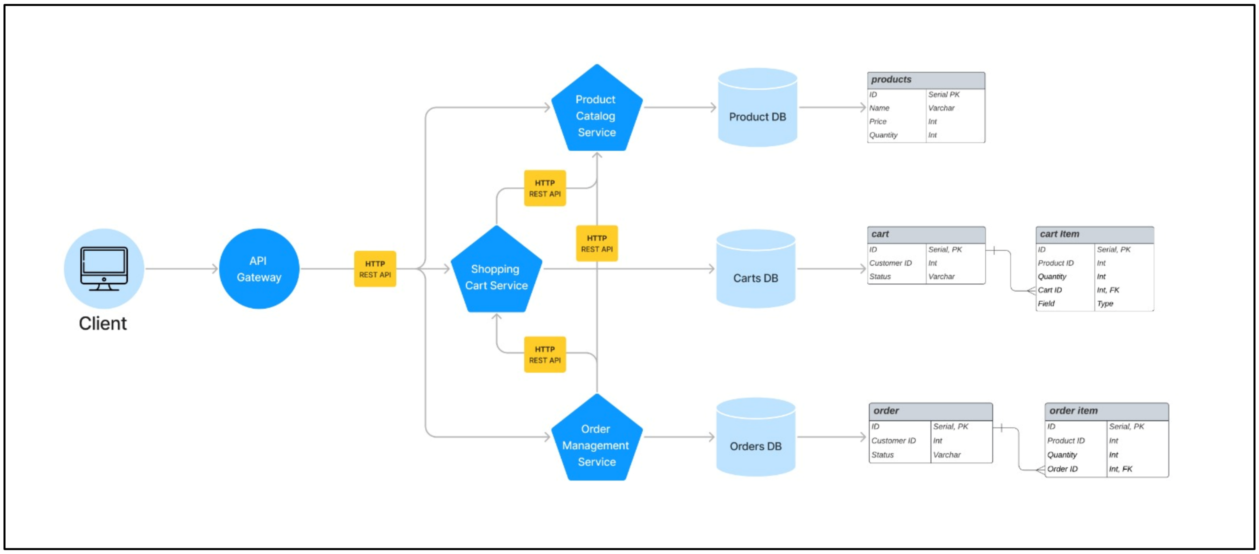


Figure 3

Use Case Diagram

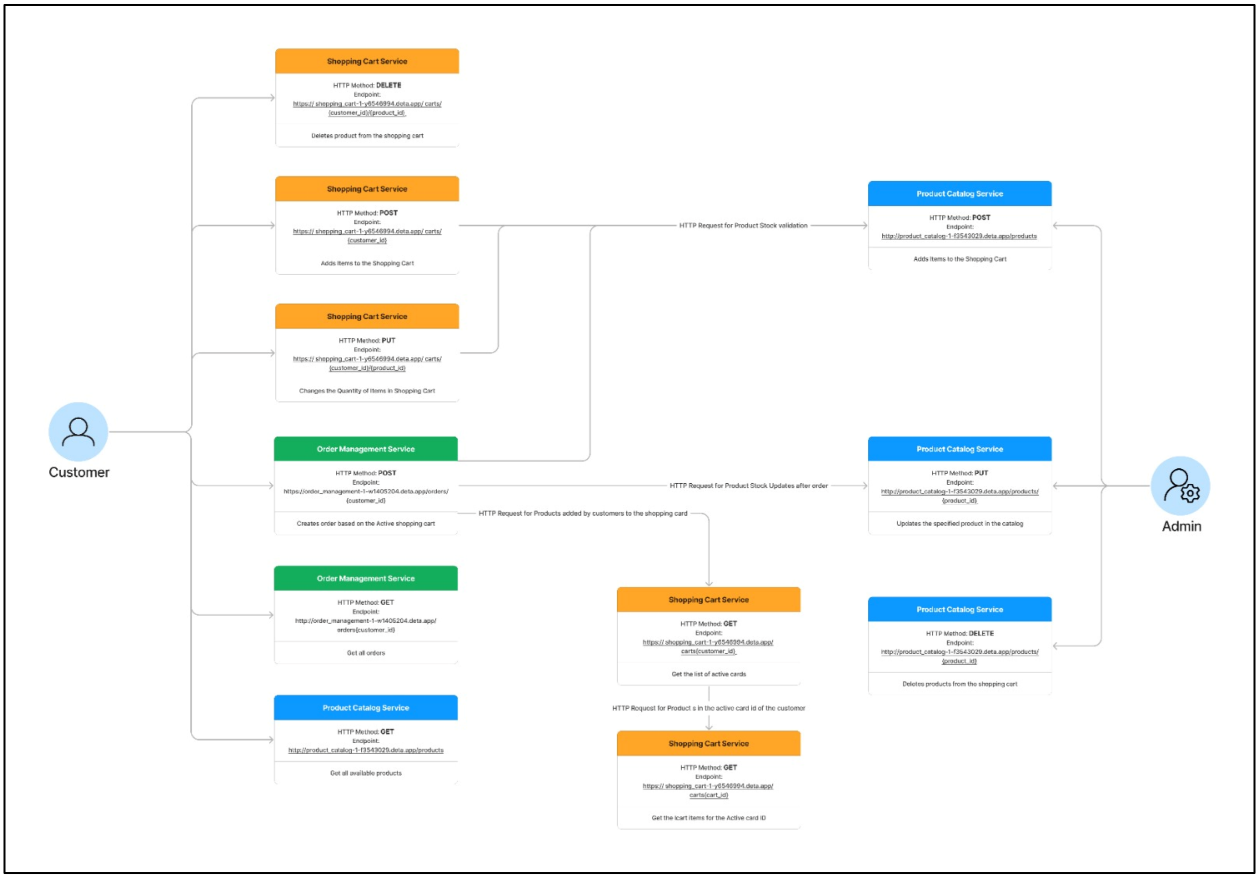


Figure 4

Order Flow Diagram

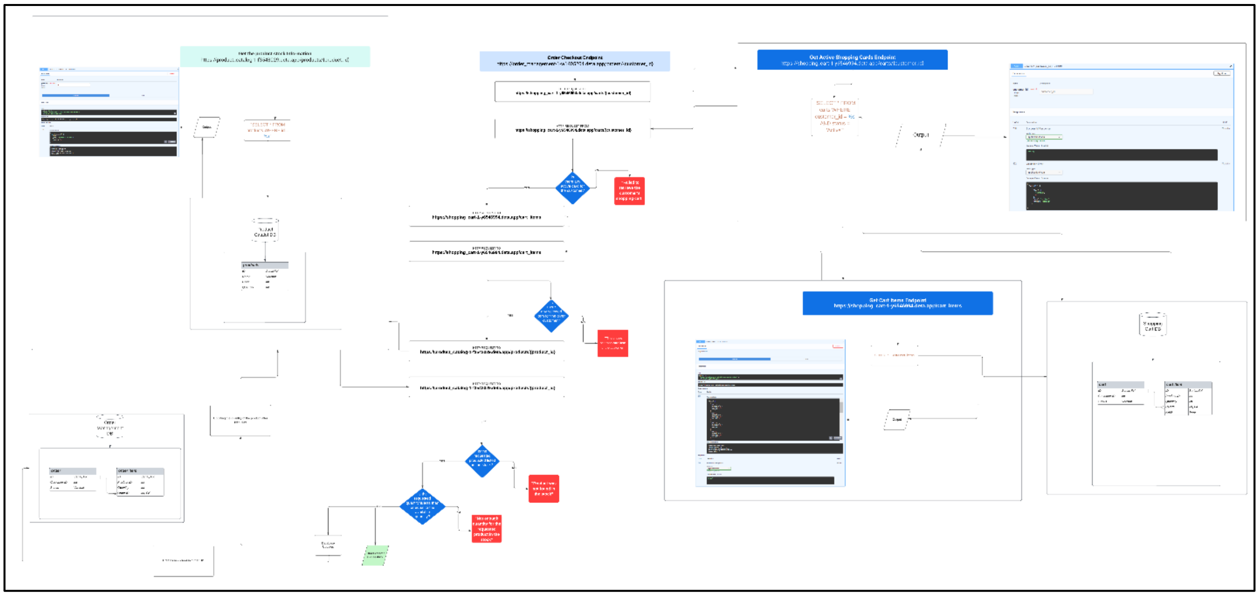


Figure 5

Application Development, Testing & Documentations

Product Catalog Microservice

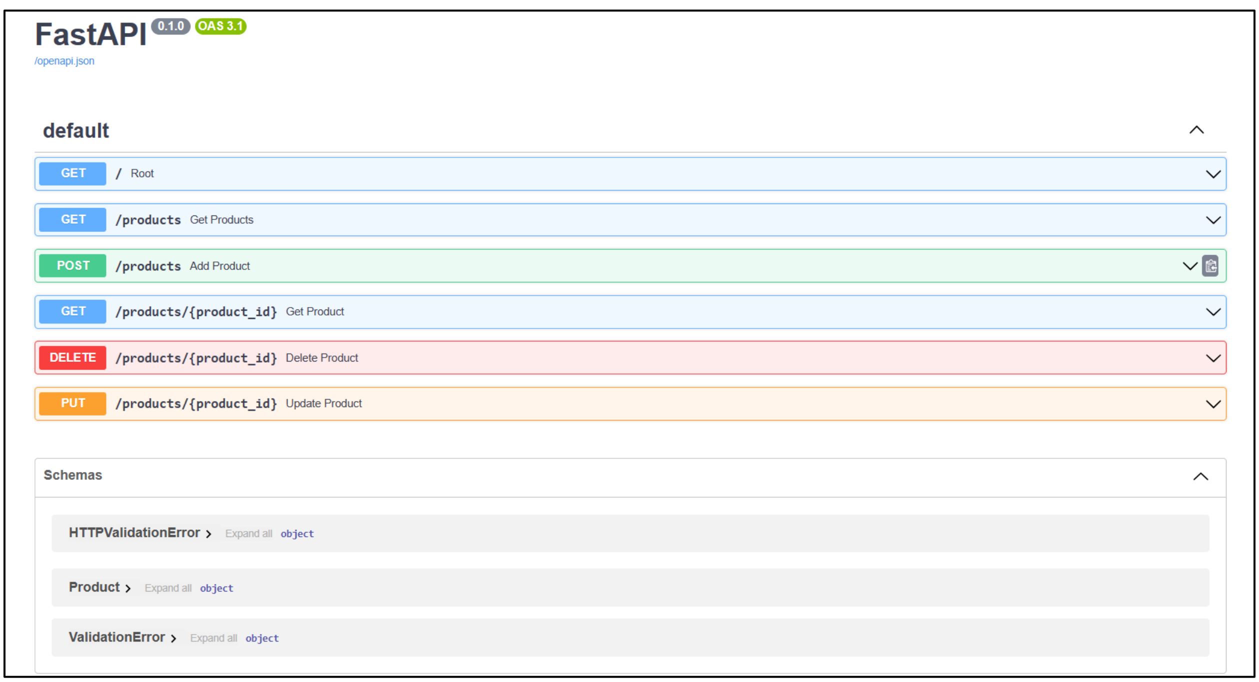


Figure 6

Link to Swagger UI: https://product_catalog-1-f3543029.deta.app/docs

Description: This microservice is responsible for managing the product inventory, details, and availability.

API Endpoints

- GET /products: Get all products.

Response

Response

`200 OK` : Successful operation. Returns a list of products.

```
@app.get("/products")
def get_products():
    cur.execute("""SELECT * FROM products""")
    products=cur.fetchall()
    # print(products)
    return {"data":products}
```

Figure 7

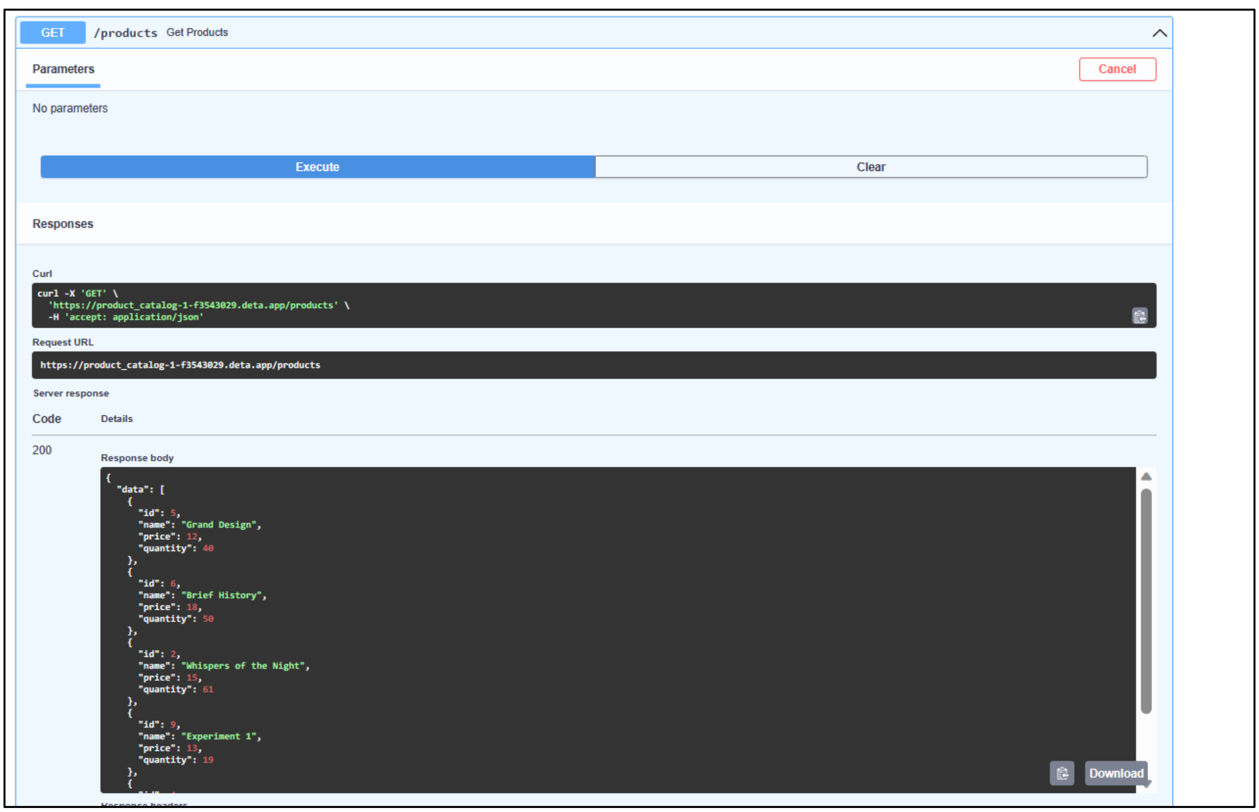


Figure 8

- GET /products/{product_id}: Returns the product with the specified ID.

Parameters

`product_id` : ID of the product to retrieve.

Response

`200 OK` : Successful operation. Returns the product with the specified ID.

`404 Not Found` : Product with the specified ID not found.

```
@app.get("/products/{product_id}")
def get_product(product_id: int):
    cur.execute("SELECT * FROM products WHERE id = %s", (product_id,))
    product = cur.fetchone()
    if product is None:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail=f"product with id: {product_id} was not found")
    return {"product_detail":product}
```

Figure 9

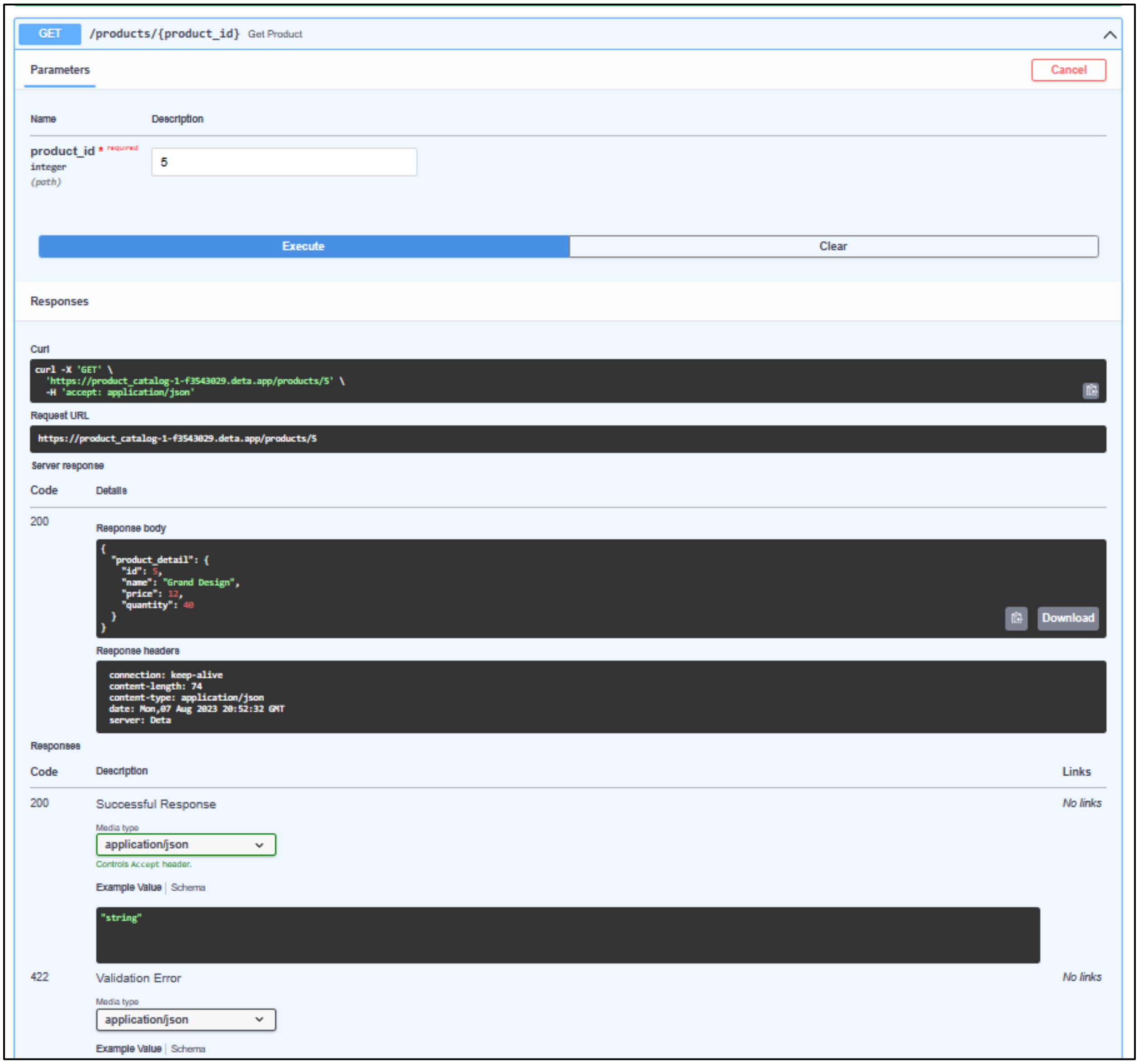


Figure 10

- POST /products: Creates a new product in the catalog.

Request Body

`name (string)`: Name of the product.

`price (integer)`: Price of the product.

`quantity (integer)`: Quantity of the product in the inventory.

Response

`201 Created`: Product created successfully.

`500 Internal Server Error`: Failed to create the product.

```
@app.post("/products", status_code=status.HTTP_201_CREATED )
def add_product(product: Product):
    cur.execute(
        """INSERT INTO products ( name, price, quantity) VALUES (%s, %s, %s) RETURNING * """,
        ( product.name, product.price, product.quantity)
    )
    new_product=cur.fetchone()
    conn.commit()
    return {"data": new_product}
```

Figure 11

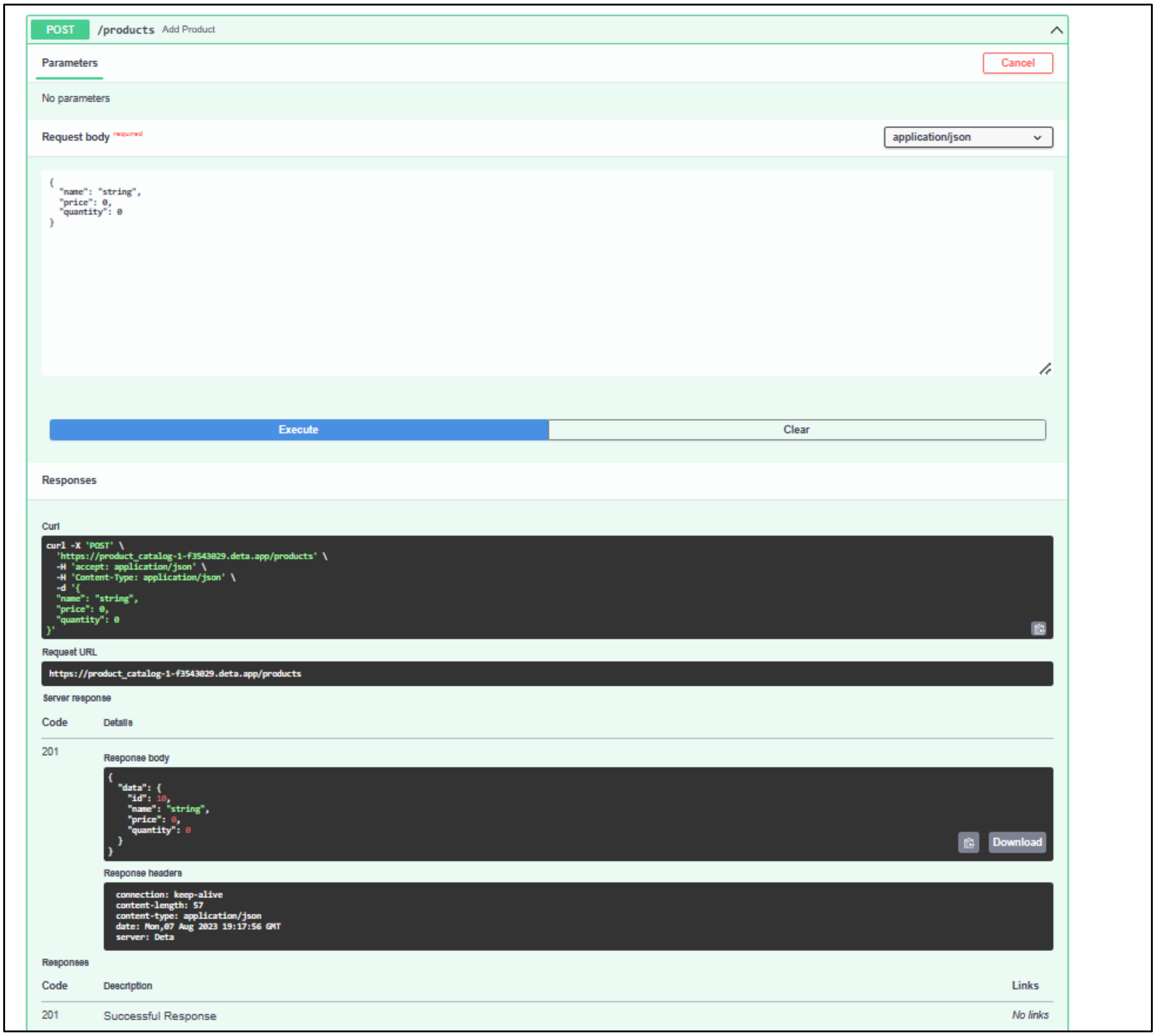


Figure 12

- PUT /products/{product_id}: Updates the specified product in the catalog.

Parameters

`product_id` : ID of the product to update.

Request Body

`name (string)` : New name of the product.

`price (integer)` : New price of the product.

`quantity (integer)` : New quantity of the product in the inventory.

Response

`200 OK` : Successful operation. Product updated successfully.

`404 Not Found` : Product with the specified ID not found.

```
@app.put("/products/{product_id}")
def update_product(product_id: int, updated_product: Product):
    # check if product exists
    cur.execute("SELECT * FROM products WHERE id = %s", (product_id,))
    product = cur.fetchone()
    if product is None:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail=f"product with id: {product_id} was not found")

    # update the product
    cur.execute(
        """UPDATE products
           SET name = %s, price = %s, quantity = %s
           WHERE id = %s RETURNING *""",
        (updated_product.name, updated_product.price, updated_product.quantity, product_id)
    )
    conn.commit()
    updated_product = cur.fetchone()

    return {"message": "Product updated successfully", "updated_product": updated_product}
```

Figure 13

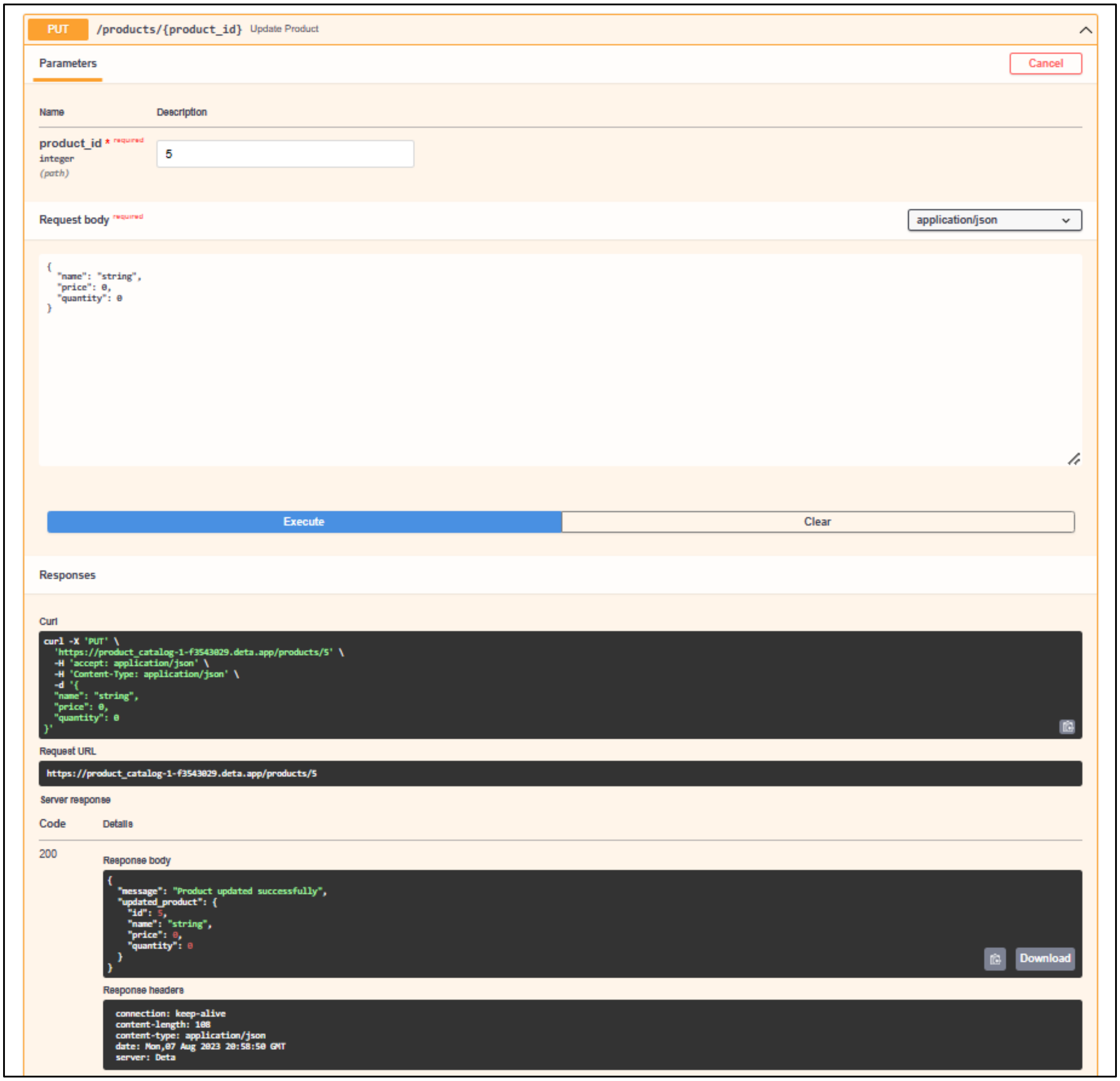


Figure 14

- DELETE /products/{product_id}: Deletes the specified product from the catalog.

Parameters

product_id: ID of the product to delete.

Response

`204 No Content` : Product deleted successfully.

`404 Not Found` : Product with the specified ID not found.

```
@app.delete("/products/{product_id}", status_code=status.HTTP_204_NO_CONTENT)
def delete_product(product_id: int):
    cur.execute("DELETE FROM products WHERE id = %s", (product_id,))
    conn.commit()
    if cur.rowcount == 0:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail=f"product with id: {product_id} does not exist")
    return Response(status_code=status.HTTP_204_NO_CONTENT)
```

Figure 15

Dependencies: This service is independent and does not have any dependencies.

Database Connection: This service uses PostgreSQL as its database which is deployed on Amazon RDS. It connects to the PostgreSQL instance using the psycopg2 library. The database connection parameters are specified within the application code, including the database name, user, password, host, and port.

Upon application start-up, the service automatically connects to the database using the provided credentials and host information. The code then checks for the existence of necessary tables (products) and creates them if they do not exist.

Database Tables

products: This table stores the details of all products in the catalog. It has four fields:

`id` : A unique identifier for the product.

`name` : The name of the product.

`price` : The price of the product.

`quantity` : The quantity of the product in the inventory.

```
try:
    cur = conn.cursor()

    create_table_query = '''
    CREATE TABLE IF NOT EXISTS products(
        id SERIAL PRIMARY KEY,
        name VARCHAR NOT NULL,
        price INTEGER NOT NULL,
        quantity INTEGER NOT NULL
    )
    '''

    cur.execute(create_table_query)
    conn.commit()
    print("Table created successfully")

except psycopg2.Error as e:
    print("An error occurred while creating the table:", e)
```

Figure 16

Service Deployment

This service is designed to be deployed to Deta Space, a cloud-based, scalable environment for running FastAPI applications. Step-by-step instructions on how to do this can be found on the read me file in GitHub repository:

https://github.com/ADA-GWU/guidedresearchproject-tmehtiyev2019/tree/main/app/product_catalog_microservice

Shopping Cart Microservice

Description: This microservice handles the management of customer shopping carts.

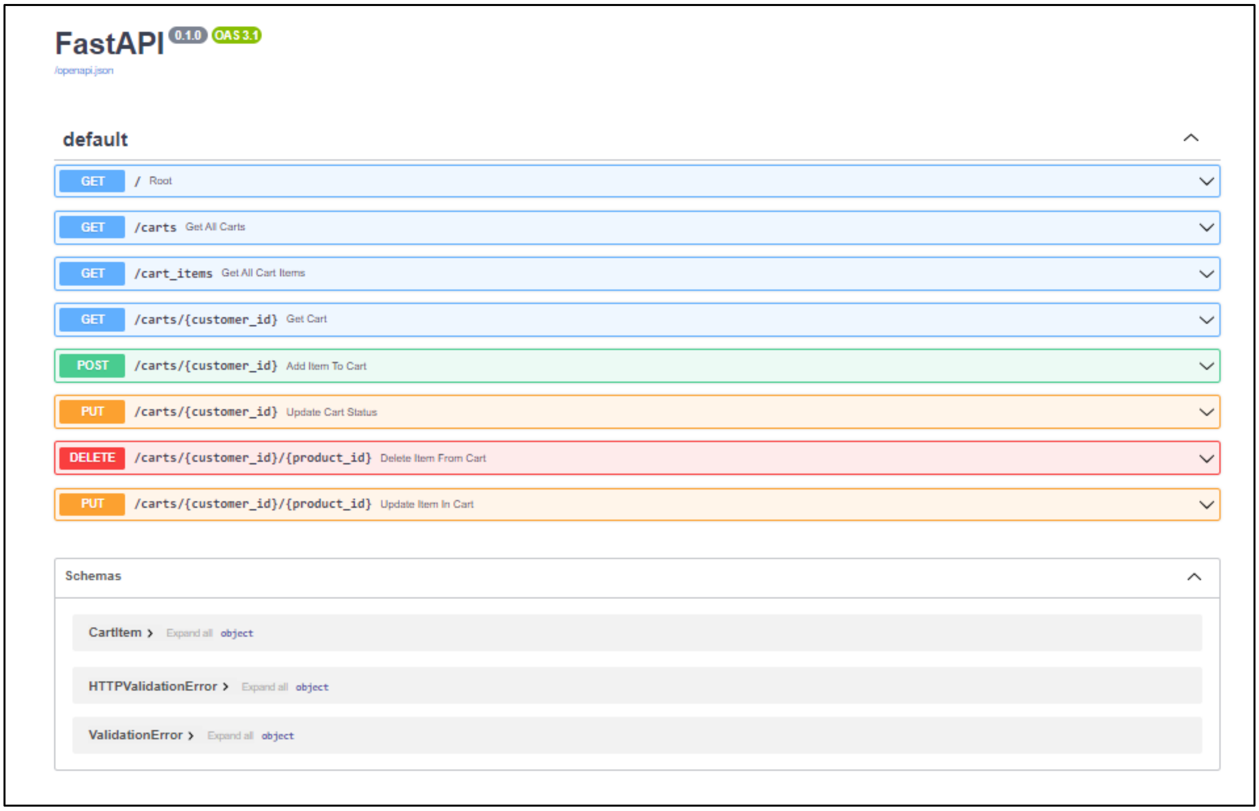


Figure 17

API Endpoints

- GET /products: Get all shopping carts.

Response:

`200 OK`: Successful operation. Returns a list of shopping carts. `404 Not` Found: No shopping carts found.

```
@app.get("/carts")
def get_all_carts():
    cur.execute("SELECT * FROM carts")
    all_carts = cur.fetchall()
    if not all_carts:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                            detail="No carts found")
    return {"carts": all_carts}
```

Figure 18

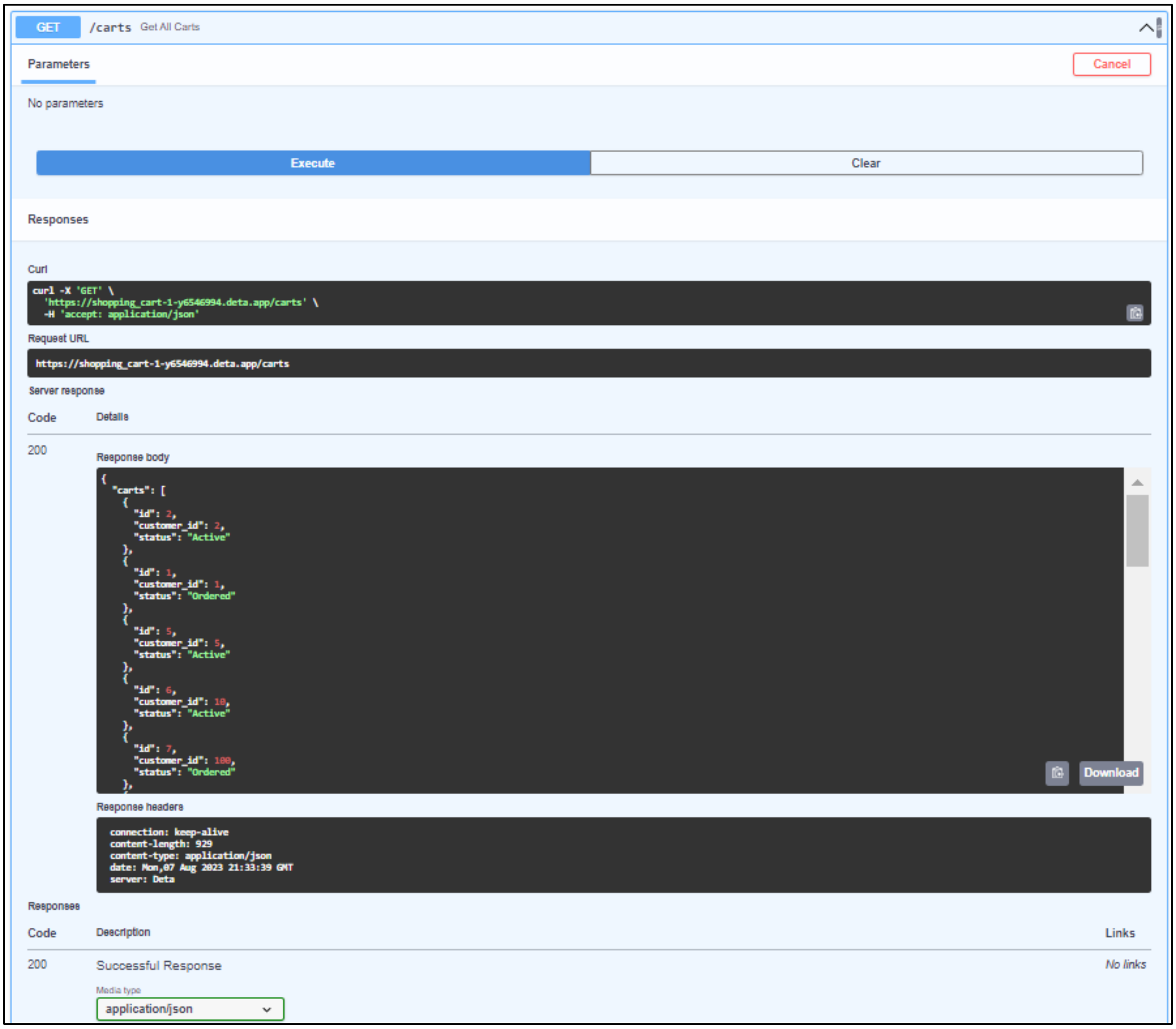


Figure 19

- GET /cart_items: Get all items in all shopping carts.

Response:

`200 OK` : Successful operation. Returns a list of all items in all shopping carts. `404 Not Found` : No items found in any shopping cart.

```
@app.get("/cart_items")
def get_all_cart_items():
    cur.execute("SELECT * FROM cart_items")
    items = cur.fetchall()
    if items is None:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                            detail="No items found in any cart")
    return {"items": items}
```

Figure 20

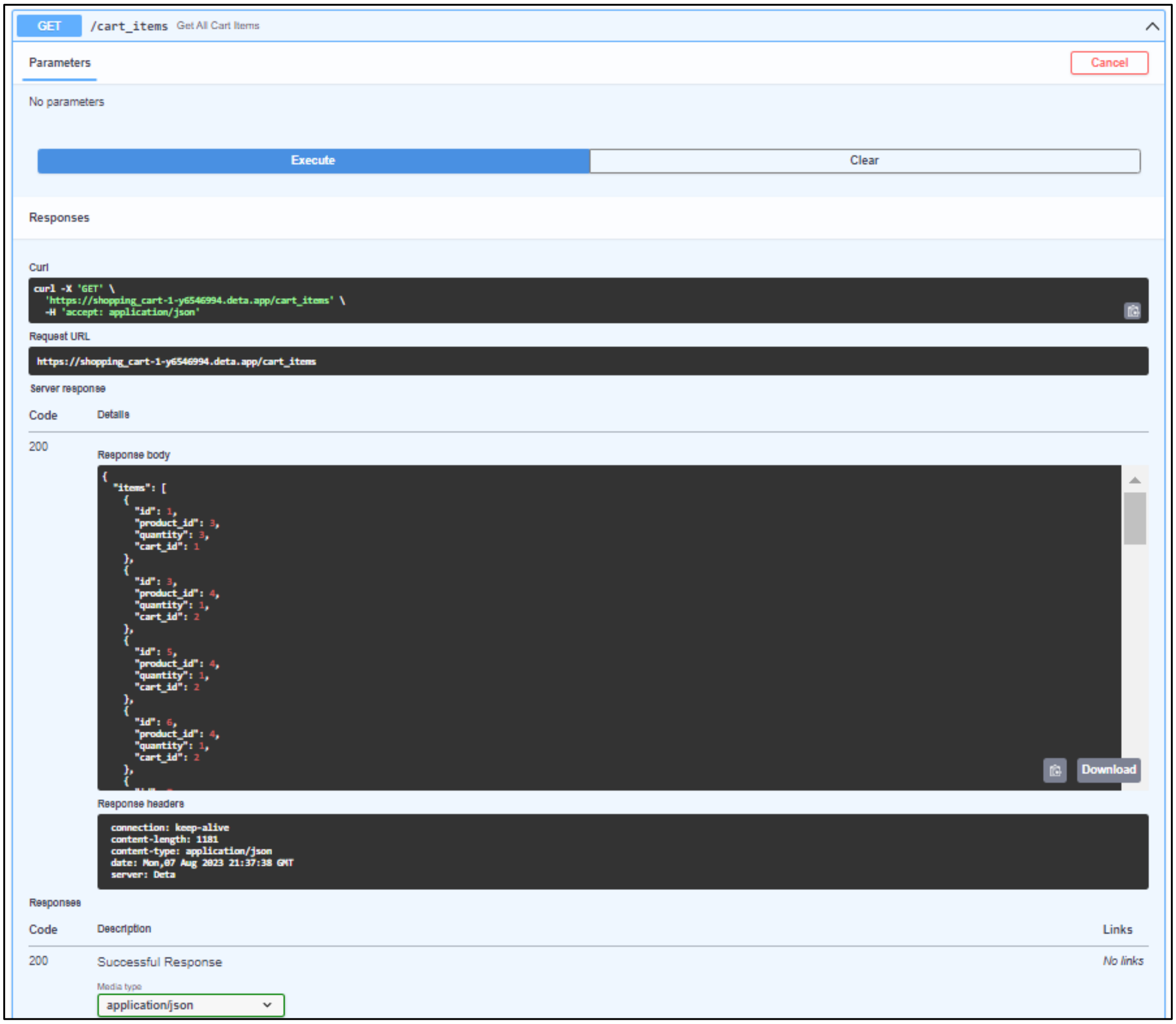


Figure 21

GET /carts/{customer_id}: Returns the active cart for the customer with the specified ID.

Parameters:

`customer_id` : ID of the customer whose cart is to be retrieved.

Response:

`200 OK` : Successful operation. Returns the active cart of the customer with the specified ID. `404 Not Found` : No active cart found for the customer with the specified ID.

```
@app.get("/carts/{customer_id}")
def get_cart(customer_id: int):
    cur.execute("SELECT * FROM carts WHERE customer_id = %s AND status = 'Active'", (customer_id,))
    cart = cur.fetchone()
    if cart is None:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                            detail=f"Active cart for customer with id: {customer_id} was not found")
    return {"cart": cart}
```

Figure 22

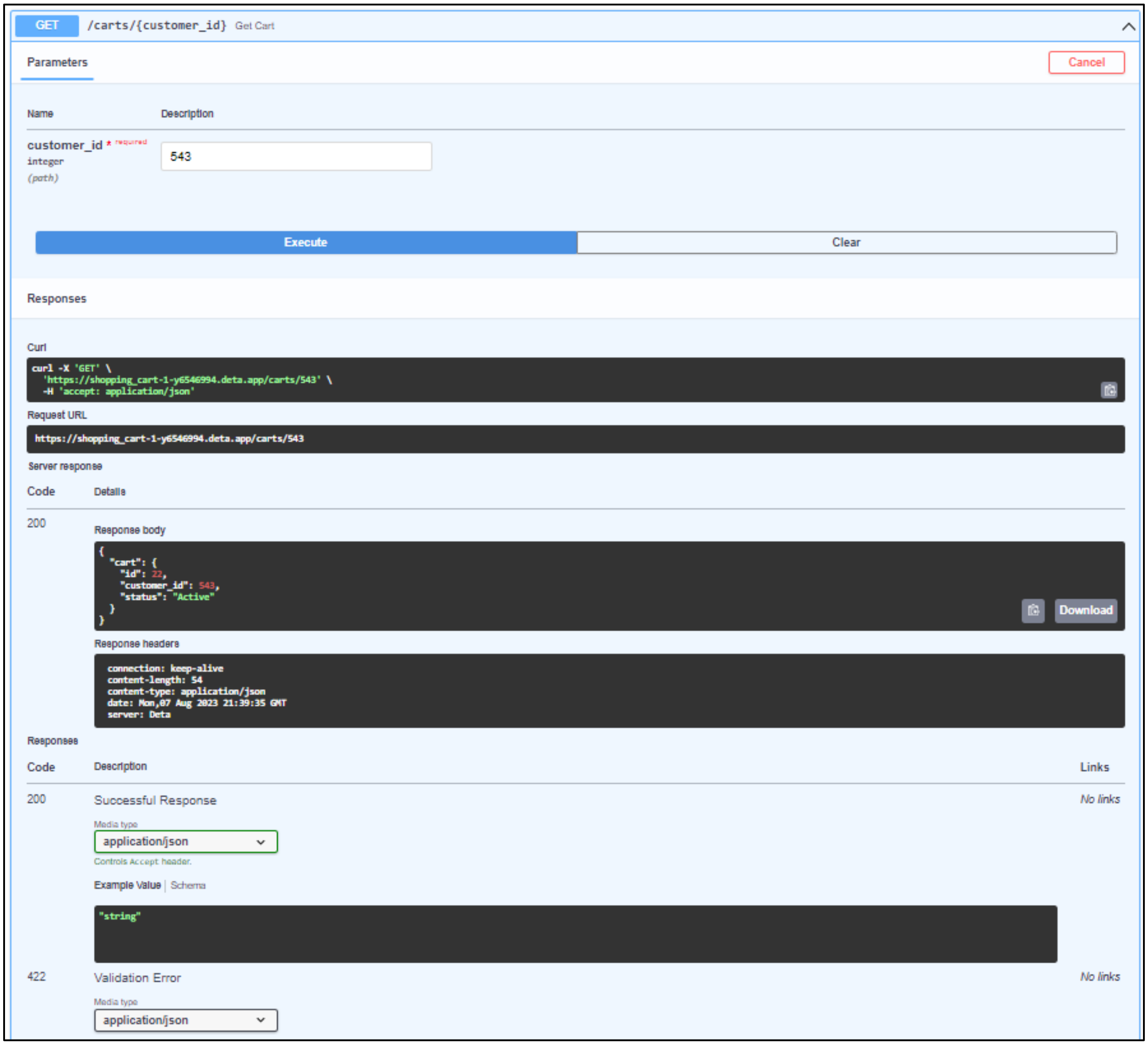


Figure 23

- POST /carts/{customer_id}: Adds an item to the active cart of the customer with the specified ID.

Parameters:

`customer_id` : ID of the customer whose cart the item is to be added to.

Request Body:

`product_id (integer)` : ID of the product to be added. `quantity (integer)` : Quantity of the product to be added.

Response:

`201 Created` : Item added successfully to the cart. `400 Bad Request` : Not enough product in stock. `404 Not Found` : Product with the specified ID not found.

```
@app.post("/carts/{customer_id}", status_code=status.HTTP_201_CREATED)
def add_item_to_cart(customer_id: int, cart_item: CartItem):
    response = requests.get(f"https://product_catalog-1-f3543029.deta.app/products/{cart_item.product_id}")
    if response.status_code == 200:
        product = response.json()["product_detail"]
        if product['quantity'] >= cart_item.quantity:
            cur.execute("SELECT id FROM carts WHERE customer_id = %s AND status = 'Active'", (customer_id,))
            cart = cur.fetchone()
            if cart is None:
                cur.execute("INSERT INTO carts (customer_id, status) VALUES (%s, 'Active') RETURNING id",
                            (customer_id,))
                cart_id = cur.fetchone()["id"]
            else:
                cart_id = cart["id"]
            cur.execute("INSERT INTO cart_items (cart_id, product_id, quantity) VALUES (%s, %s, %s)",
                        (cart_id, cart_item.product_id, cart_item.quantity))
            conn.commit()
            return {"message": "Item added to cart successfully"}
        else:
            raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST,
                                detail="Not enough product in stock")
    else:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                            detail=f"Product with id {cart_item.product_id} not found")
```

Figure 24

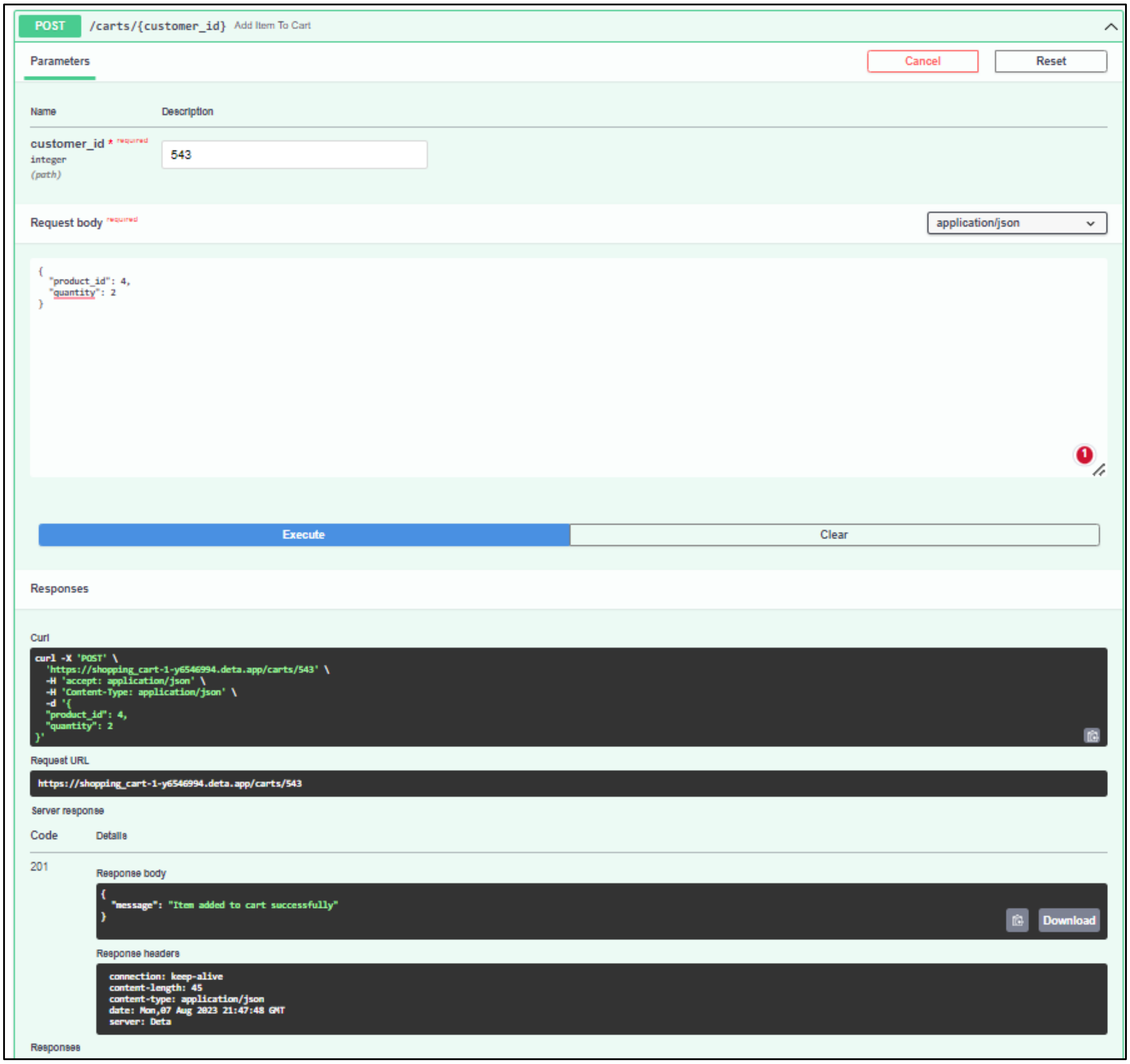


Figure 25

- DELETE /carts/{customer_id}/{product_id}: Deletes the specified product from the cart of the customer with the specified ID.

Parameters:

`customer_id` : ID of the customer whose cart the item is to be deleted from. `product_id` : ID of the product to be deleted.

Response:

`204 No Content` : Product deleted successfully from the cart. `404 Not Found` : Product with the specified ID not found in the cart of the customer with the specified ID.

```
@app.delete("/carts/{customer_id}/{product_id}", status_code=status.HTTP_204_NO_CONTENT)
def delete_item_from_cart(customer_id: int, product_id: int):
    cur.execute("DELETE FROM cart_items WHERE cart_id IN (SELECT id FROM carts WHERE customer_id = %s) AND product_id = %s",
                (customer_id, product_id))
    conn.commit()
    if cur.rowcount == 0:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                            detail=f"product with id: {product_id} does not exist in the cart of customer with id: {customer_id
    return Response(status_code=status.HTTP_204_NO_CONTENT)
```

Figure 26

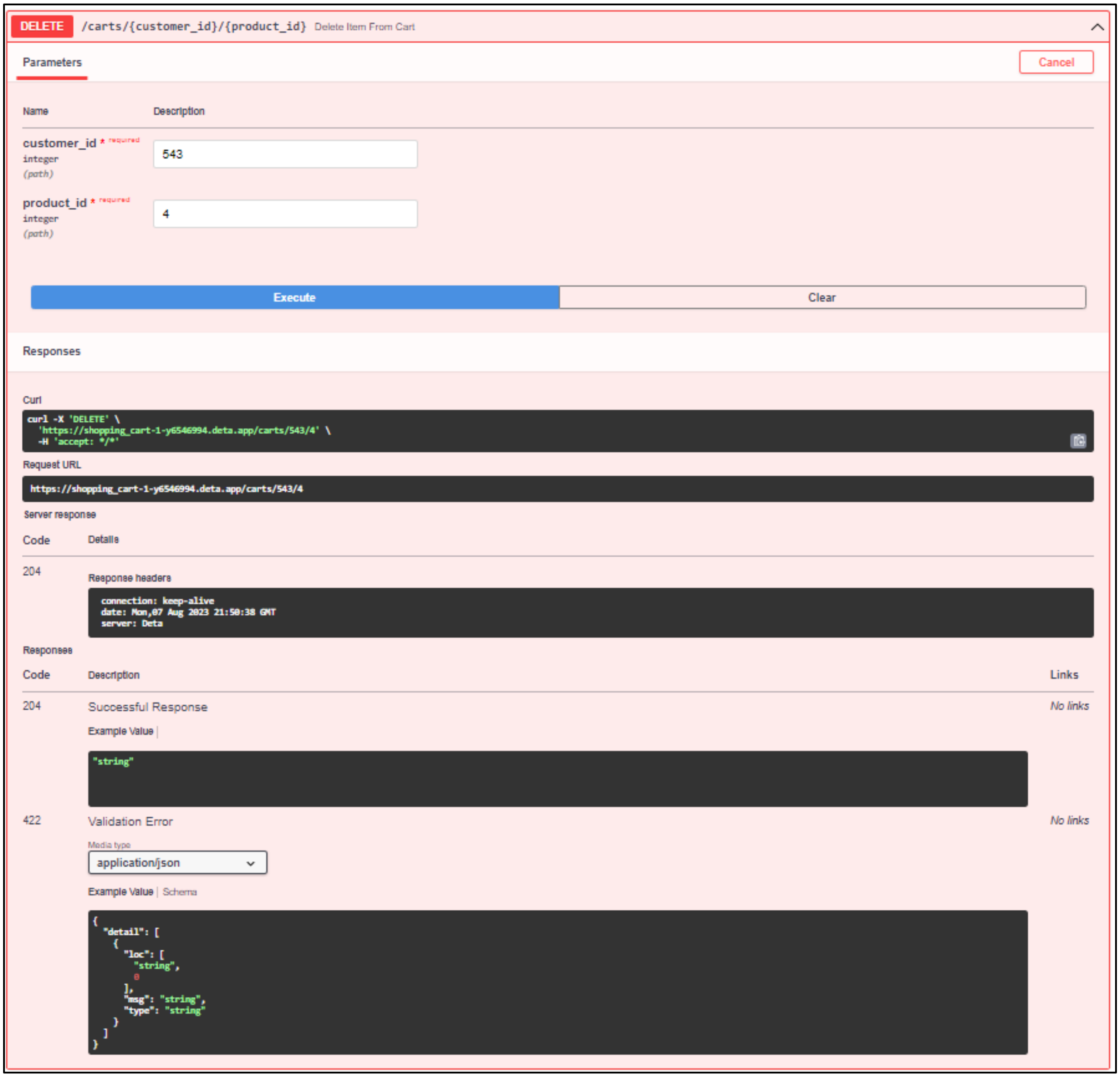


Figure 27

- PUT /carts/{customer_id}/{product_id}: Updates the quantity of a specific item in the active cart of the customer with the specified ID.

Parameters:

`customer_id` : ID of the customer whose cart the item is in. `product_id` : ID of the product whose quantity is to be updated.

Request Body:

`quantity (integer)` : New quantity of the product in the cart.

Response:

`200 OK` : Successful operation. Item quantity updated successfully. `400 Bad Request` : Not enough product in stock. `404 Not Found` : Product with the specified ID not found in the cart of the customer with the specified ID.

```
@app.put("/carts/{customer_id}/{product_id}")
def update_item_in_cart(customer_id: int, product_id: int, updated_cart_item: CartItem):
    response = requests.get(f"https://product_catalog-1-f3543029.deta.app/products/{product_id}")
    if response.status_code == 200:
        product = response.json()["product_detail"]
        if product['quantity'] >= updated_cart_item.quantity:
            cur.execute("UPDATE cart_items SET quantity = %s WHERE cart_id IN (SELECT id FROM carts WHERE customer_id = %s AND :
                        (updated_cart_item.quantity, customer_id, product_id))
            conn.commit()
            if cur.rowcount == 0:
                raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                                    detail=f"Product with id: {product_id} does not exist in the cart of customer with id: {cus
            return {"message": "Item quantity updated successfully"}
        else:
            raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST,
                                detail="Not enough product in stock")
    else:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                            detail=f"Product with id {product_id} not found")
```

Figure 28

PUT /carts/{customer_id}: Updates the status of the active cart of the customer with the specified ID.

Parameters:

`customer_id` : ID of the customer whose cart status is to be updated.

Request Body:

`status (string)` : New status of the cart ('Ordered' by default).

Response:

`202 Accepted` : Cart status updated successfully. `404 Not Found` : No active cart found for the customer with the specified ID.

```
@app.put("/carts/{customer_id}", status_code=status.HTTP_202_ACCEPTED)
def update_cart_status(customer_id: int, status: str = 'Ordered'):
    cur.execute("UPDATE carts SET status = %s WHERE customer_id = %s AND status = 'Active'", (status, customer_id))
    conn.commit()
    if cur.rowcount == 0:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                            detail=f"No active cart found for customer with id: {customer_id}")
    return {"message": "Cart status updated successfully"}
```

Figure 29

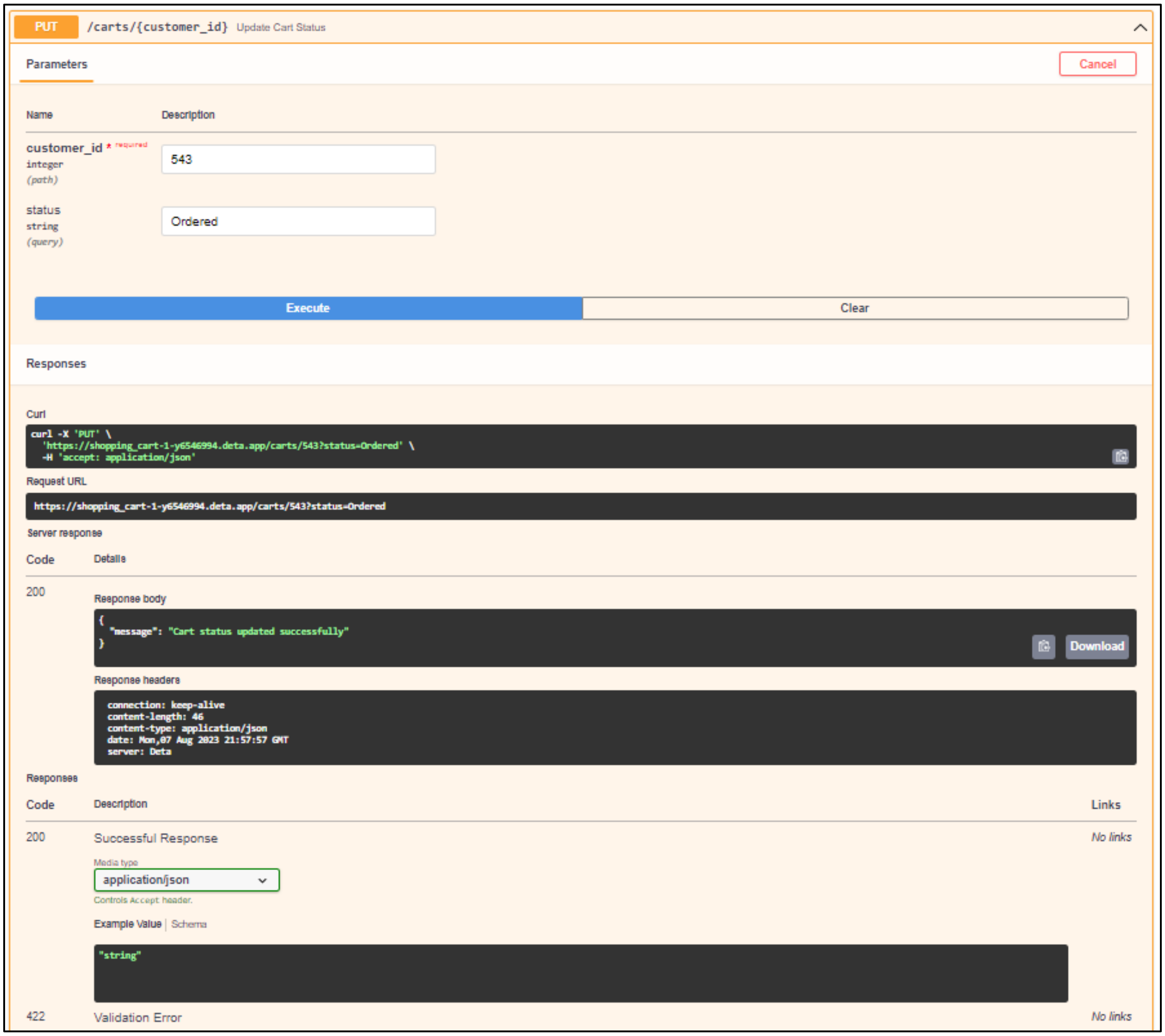


Figure 30

Dependencies

This service is dependent on the Product Catalog Microservice for product data.

The Shopping Cart Microservice is dependent on the Product Catalog Microservice for information about the products. Here's what this dependency involves:

When a customer wants to add a product to their shopping cart, the Shopping Cart Microservice needs to check the Product Catalog Microservice to ensure the product exists and to get its details. This means there is a call from the Shopping Cart Microservice to the Product Catalog Microservice.

Similarly, when a customer updates the quantity of a product in their cart, the Shopping Cart Microservice has to ensure that enough quantity of the product is available. This information is retrieved from the Product Catalog Microservice.

These dependencies are managed through API calls between the two services as described in the codes above.

Database Connection

This service uses PostgreSQL as its database which is deployed on Amazon RDS. It connects to the PostgreSQL instance using the psycopg2 library. The database connection parameters are specified within the application code, including the database name, user, password, host, and port.

Upon application start-up, the service automatically connects to the database using the provided credentials and host information. The code then checks for the existence of necessary tables (carts and cart_items) and creates them if they do not exist.

Database Tables

```python
try:
    cur = conn.cursor()

    # Create a 'carts' table
    create_carts_table_query = '''
    CREATE TABLE IF NOT EXISTS carts(
        id SERIAL PRIMARY KEY,
        customer_id INT NOT NULL,
        status VARCHAR NOT NULL
    )
    '''

    # Create a 'cart_items' table
    create_cart_items_table_query = '''
    CREATE TABLE IF NOT EXISTS cart_items(
        id SERIAL PRIMARY KEY,
        product_id INT NOT NULL,
        quantity INT NOT NULL,
        cart_id INT,
        FOREIGN KEY (cart_id) REFERENCES carts (id)
    )
    '''

    cur.execute(create_carts_table_query)
    cur.execute(create_cart_items_table_query)

    conn.commit()
    print("Tables created successfully")

except psycopg2.Error as e:
    print("An error occurred while creating the tables:", e)



class CartItem(BaseModel):
    product_id: int
    quantity: int


class Cart(BaseModel):
    id: int
    customer_id: int
    status: str
    items: list[CartItem]
```

Figure 31

Service Deployment

This service is designed to be deployed to Deta Space, a cloud-based, scalable environment for running FastAPI applications. The step-by-step instructions on how to do this are given in the Read Me file in GitHub repository:

https://github.com/ADA-GWU/guidedresearchproject-tmehtiyev2019/tree/main/app/shopping_cart_microservice#readme

Order Management Microservice

Order Management Microservice

Description: This microservice manages the processing of customer orders.

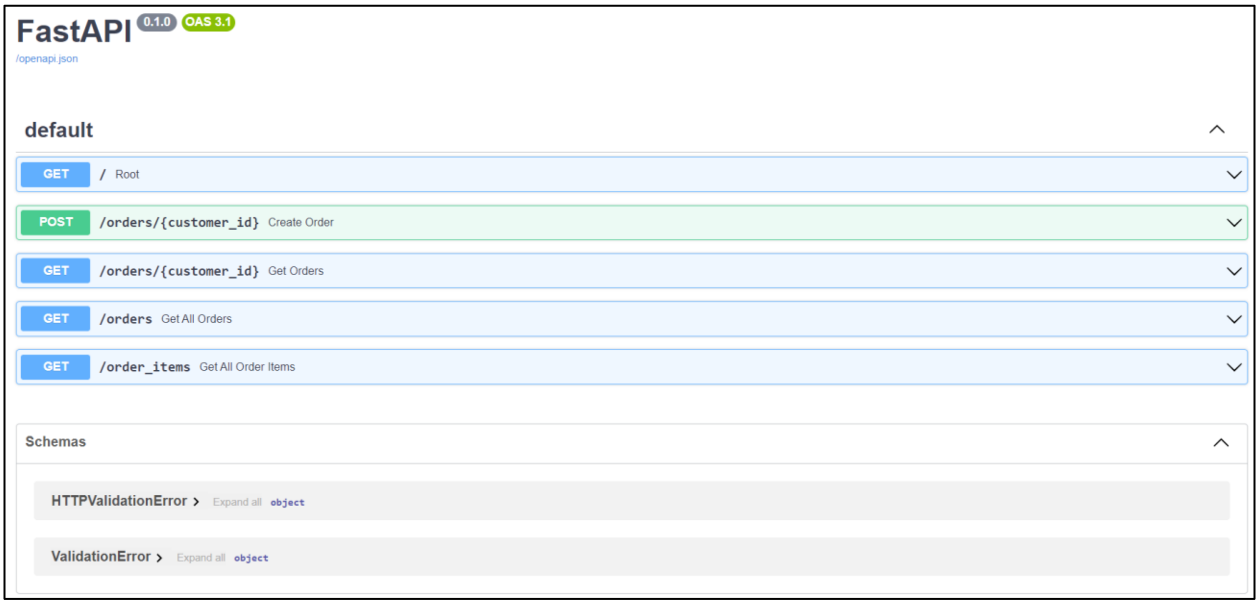
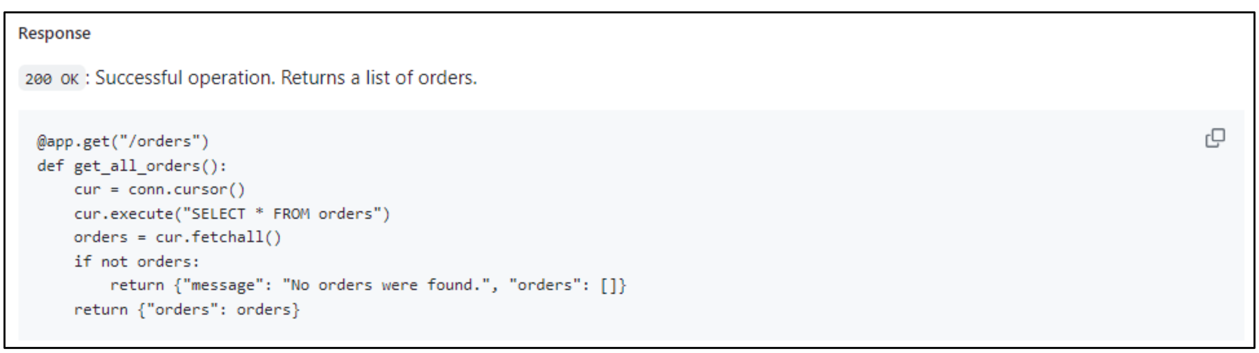


Figure 32

API Endpoints

- GET /orders: Get all orders.

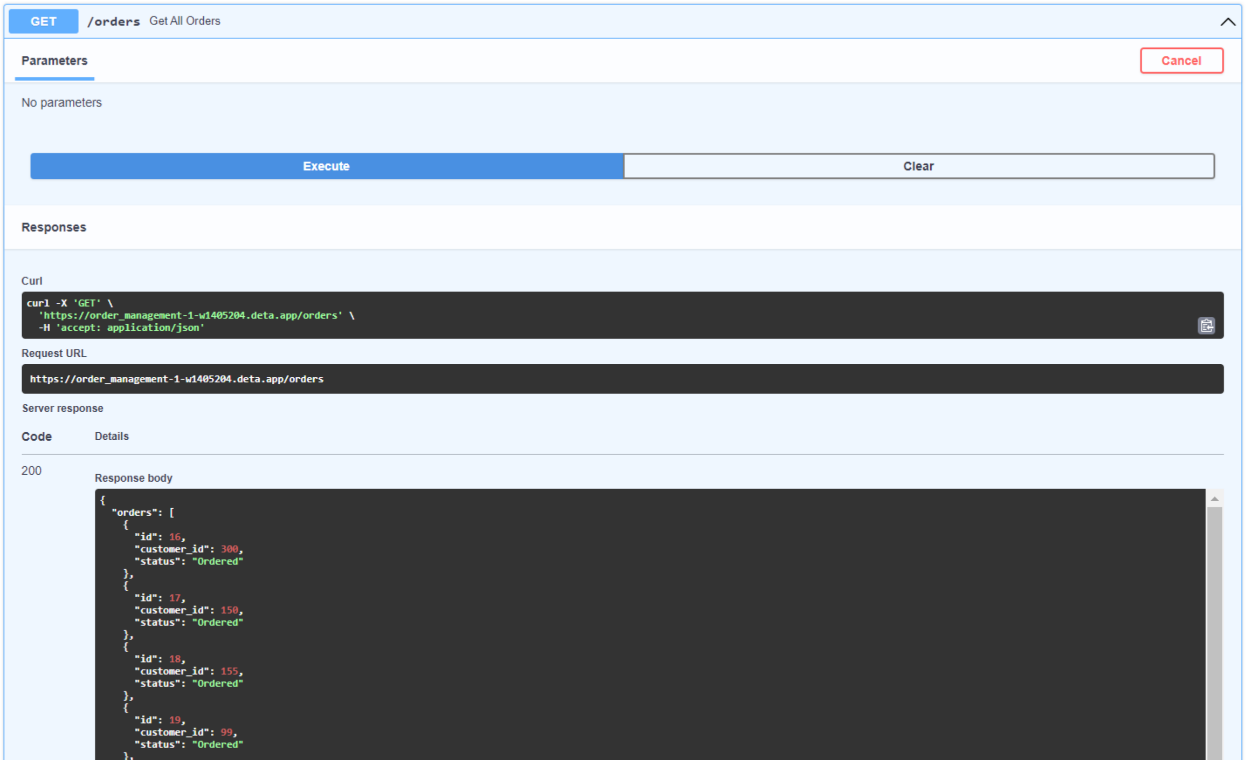


Response

`200 OK`: Successful operation. Returns a list of orders.

```
@app.get("/orders")
def get_all_orders():
    cur = conn.cursor()
    cur.execute("SELECT * FROM orders")
    orders = cur.fetchall()
    if not orders:
        return {"message": "No orders were found.", "orders": []}
    return {"orders": orders}
```

Figure 33



- GET /orders/{customer_id}: Fetch all orders for the specified customer.

Parameters

`customer_id` : ID of the customer for whom to retrieve the orders.

Response

`200 OK` : Successful operation. Returns a list of orders.

`404 Not Found` : No orders found for the specified customer.

```
@app.get("/orders/{customer_id}")
def get_orders(customer_id: int):
    cur.execute("SELECT * FROM orders WHERE customer_id = %s", (customer_id,))
    orders = cur.fetchall()
    if not orders:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                            detail=f"Orders for customer with id: {customer_id} were not found")
    return {"orders": orders}
```

Figure 34

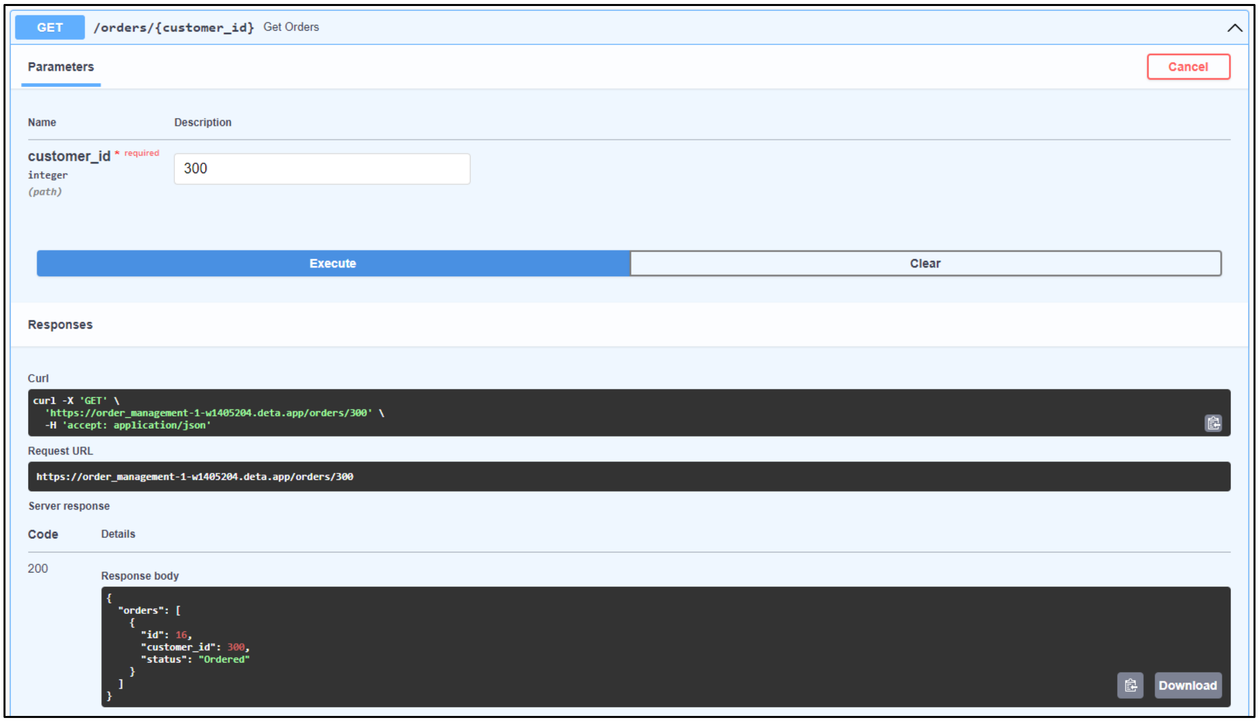


Figure 35

- POST /orders/{customer_id}: Creates a new order for the specified customer.

Parameters

`customer_id` : ID of the customer for whom the order is to be created.

Request Body

`items (array of objects)` : Array of order items.

`product_id (integer)` : ID of the product.

`quantity (integer)` : Quantity of the product.

Response

`201 Created` : Order created successfully.

`400 Bad Request` : Insufficient inventory for one or more items.

`500 Internal Server Error` : Failed to create the order.

```
@app.post("/orders/{customer_id}")
def create_order(customer_id: int):
    # Retrieve the customer's shopping cart from the Shopping Cart Microservice
    response = requests.get(f"https://product_catalog-1-f3543029.deta.app/carts/{customer_id}")
    if response.status_code == 200:
        cart = response.json()

        # Confirm that sufficient inventory is available for each item in the cart
        for item in cart['items']:
            product = requests.get(f"https://product_catalog-1-f3543029.deta.app/products/{item['product_id']}").json()
            if product['quantity'] < item['quantity']:
                return {"message": f"Not enough product in stock for product id {item['product_id']}"}, 400

        # If we reach here, it means we have enough inventory for all items in the cart.
        # Now, decrease the product quantity in the Product Catalog Service and create the order
        cur.execute("INSERT INTO orders (customer_id, status) VALUES (%s, 'Ordered') RETURNING id",
                    (customer_id,))
        order_id = cur.fetchone()['id']
        for item in cart['items']:
            product = requests.get(f"https://product_catalog-1-f3543029.deta.app/products/{item['product_id']}").json()
            product['quantity'] -= item['quantity']
            requests.put(f"https://product_catalog-1-f3543029.deta.app/products/{item['product_id']}", json=product)

            cur.execute("INSERT INTO order_items (product_id, quantity, order_id) VALUES (%s, %s, %s)",
                        (item['product_id'], item['quantity'], order_id))
        conn.commit()

        # Updating the order status in the Shopping Cart Microservice
        response = requests.put(f"https://product_catalog-1-f3543029.deta.app/carts/{customer_id}", json={"status": "Ordered"})
        if response.status_code != 200:
            return {"message": "Failed to update cart status"}, 500

        return {"message": "Order created successfully"}, 201

    return {"message": "Failed to retrieve the customer's shopping cart"}, 500
```

Figure 36

Dependencies

This service is dependent on two other services:

Product Catalog Service: This service is used to retrieve product details and update inventory levels when an order is placed. The order management service makes GET and PUT requests to this service.

Shopping Cart Service: This service is used to retrieve the customer's shopping cart when placing an order. The order management service makes GET and PUT requests to this service.

Database Connection

This service uses PostgreSQL as its database which is deployed on Amazon RDS. It connects to the PostgreSQL instance using the psycopg2 library. The database connection parameters are specified within the application code, including the database name, user, password, host, and port.

Upon application start-up, the service automatically connects to the database using the provided credentials and host information. The code then checks for the existence of necessary tables (orders and order_items) and creates them if they do not exist.

Database Tables

There are two tables created within this service:

orders: This table keeps track of all customer orders. It has three fields:

`id` : A unique identifier for the order.

`customer_id` : The identifier for the customer who placed the order.

`status` : The status of the order (e.g., 'Ordered').

order_items: This table stores details of all items in each order. It has four fields:

`id` : A unique identifier for the order item.

`product_id` : The identifier for the product ordered.

`quantity` : The number of units of the product ordered.

`order_id` : The identifier of the order in which the item was ordered. This is a foreign key referencing the id field in the orders table.

```
try:
    cur = conn.cursor()

    # Create 'orders' table
    create_orders_table_query = '''
    CREATE TABLE IF NOT EXISTS orders(
        id SERIAL PRIMARY KEY,
        customer_id INT NOT NULL,
        status VARCHAR NOT NULL
    )
    '''

    # Create 'order_items' table
    create_order_items_table_query = '''
    CREATE TABLE IF NOT EXISTS order_items(
        id SERIAL PRIMARY KEY,
        product_id INT NOT NULL,
        quantity INT NOT NULL,
        order_id INT,
        FOREIGN KEY (order_id) REFERENCES orders (id)
    )
    '''

    cur.execute(create_orders_table_query)
    cur.execute(create_order_items_table_query)

    conn.commit()
    print("Tables created successfully")

except psycopg2.Error as e:
    print("An error occurred while creating the tables:", e)
```

```
class OrderItem(BaseModel):
    product_id: int
    quantity: int


class Order(BaseModel):
    id: int
    customer_id: int
    items: list[OrderItem]
    status: str
```

Figure 37

Formulation of Test Case Scenarios

Utilizing Apache JMeter, I have designed test scenarios that would provide valuable insights into the **effects of concurrent users on the application's response rate**. The primary objective was to understand how the application behaves under different loads and the corresponding response times.

Scenario 1:

<u>Objective</u>: Understand the application's behavior with a light load.

Start Threads Count: **10 users**

This signifies that 10 users will start simultaneously to send requests to the application.

Initial Delay: 0 seconds

The test begins without any initial delay.

Startup Time: 10 seconds

This parameter ensures that all users become active over a span of 10 seconds.

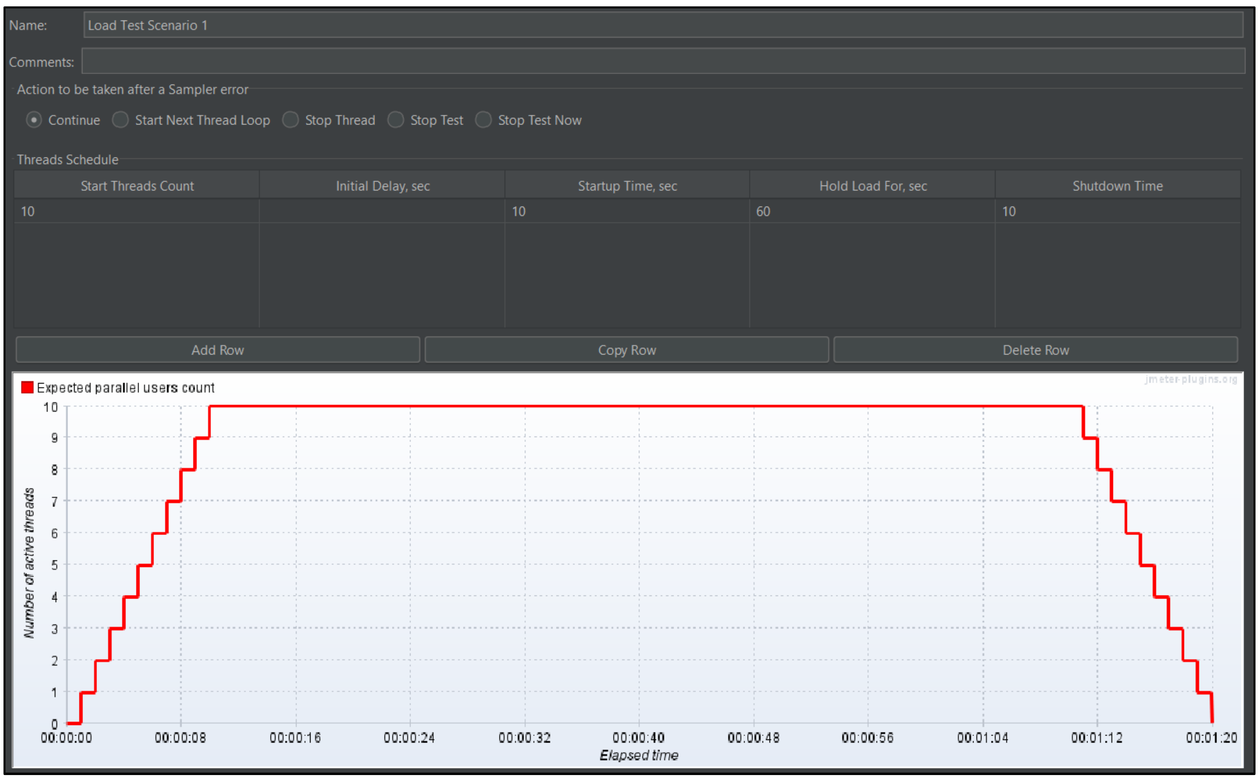


Figure 38

Scenario 2:

Objective: Assess the application's capability to manage increased load.

Start Threads Count: **50 users**

Initial Delay: 0 seconds

Startup Time: 10 seconds

Hold Load Time: 60 seconds

Shutdown Time: 10 seconds

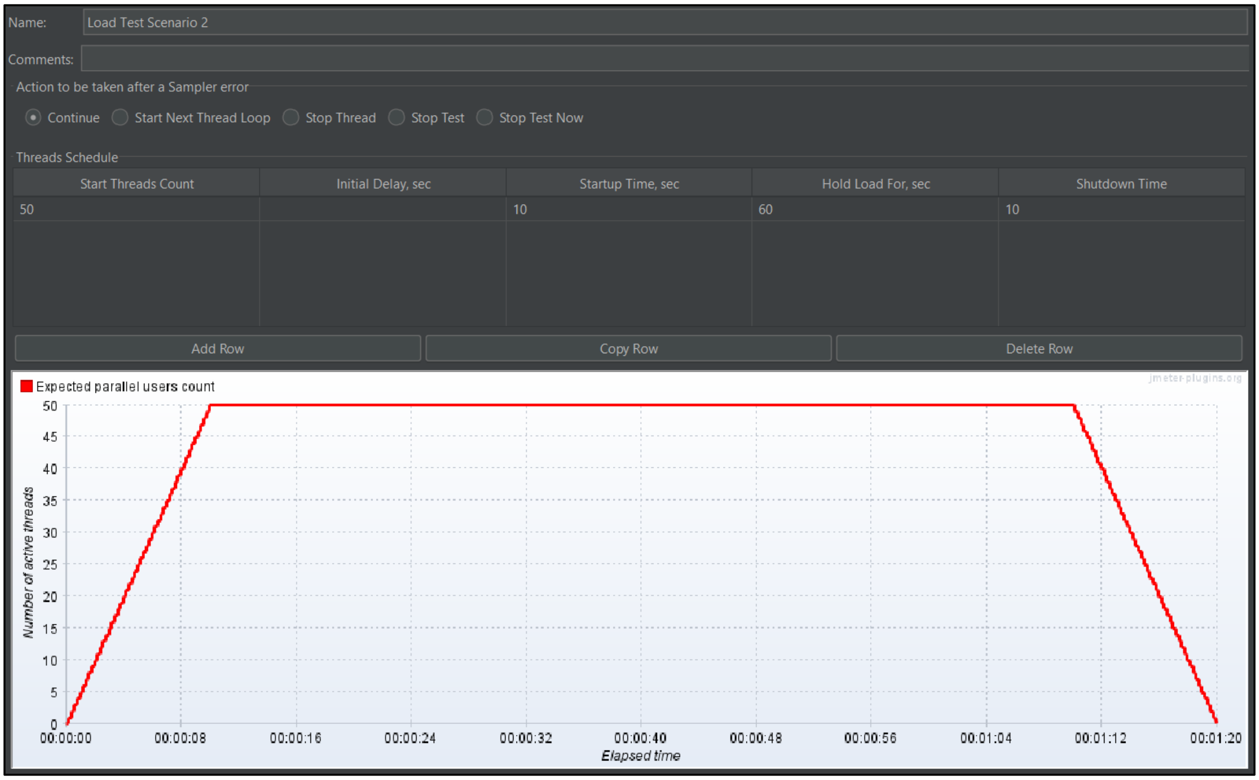


Figure 39

Scenario 3:

Objective: Examine the application's robustness under a heavy user load.

Start Threads Count: **100 users**

Initial Delay: 0 seconds

Startup Time: 10 seconds

Hold Load Time: 60 seconds

Shutdown Time: 10 seconds

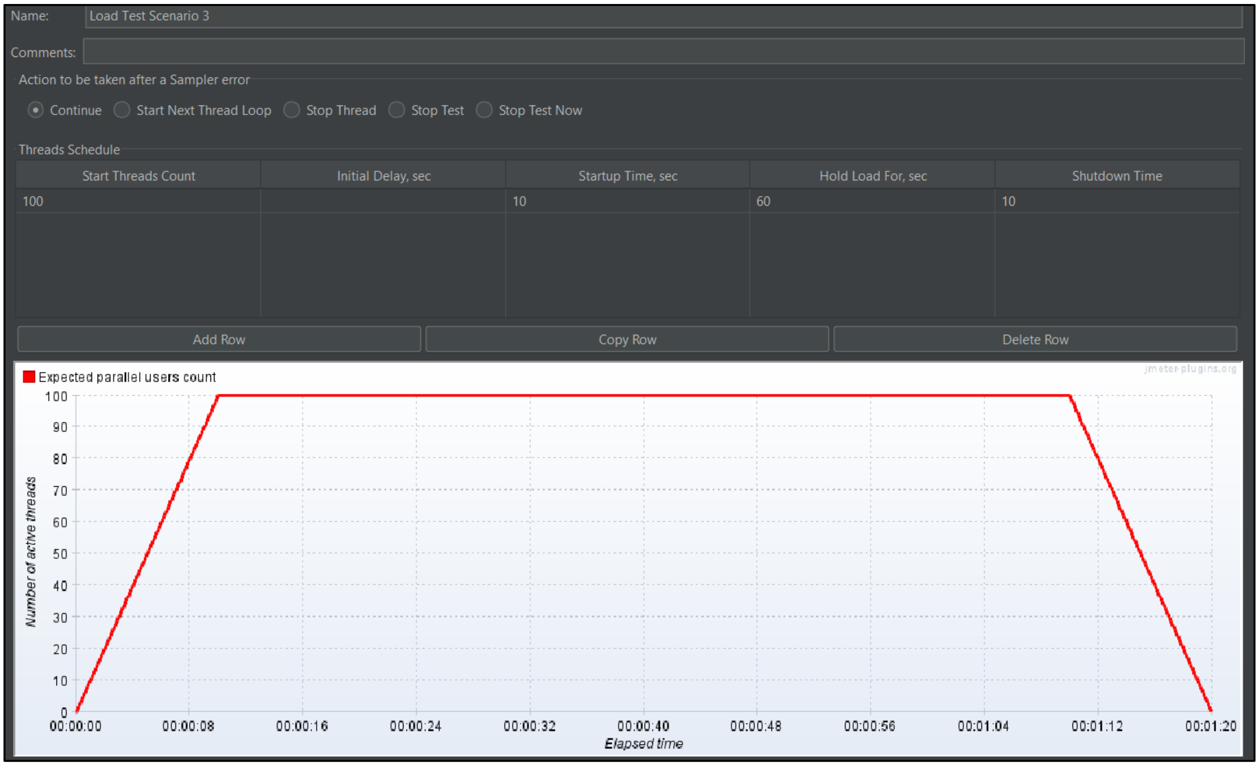


Figure 40

Data Collection

After the formulation and execution of the scenarios, Apache JMeter's listeners were employed to capture a wide array of data points. During my testing phase, I have employed many JMeter listeners that were instrumental in both data collection and visualization. Below I give descriptive information about some of them:

View Results Tree:

Detailed view of all request-response pairs, essential for debugging purposes. I have collected several data (**Sample time, Latency, Connect Time, Bytes sent/received, Success/error status**) about each request.

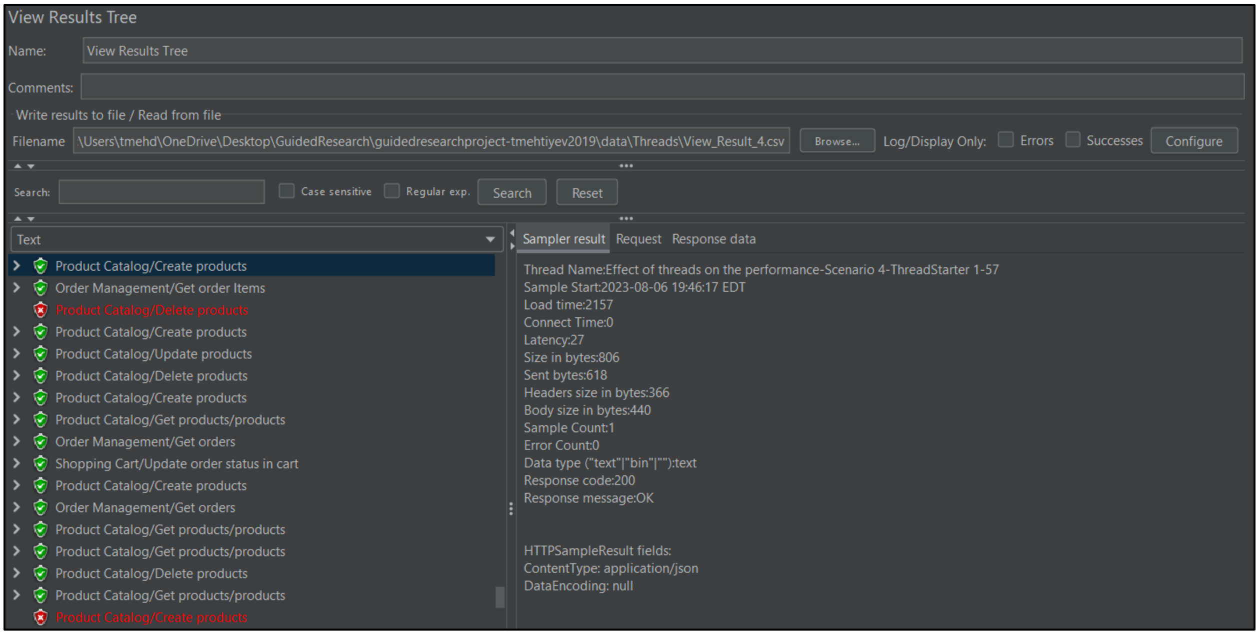
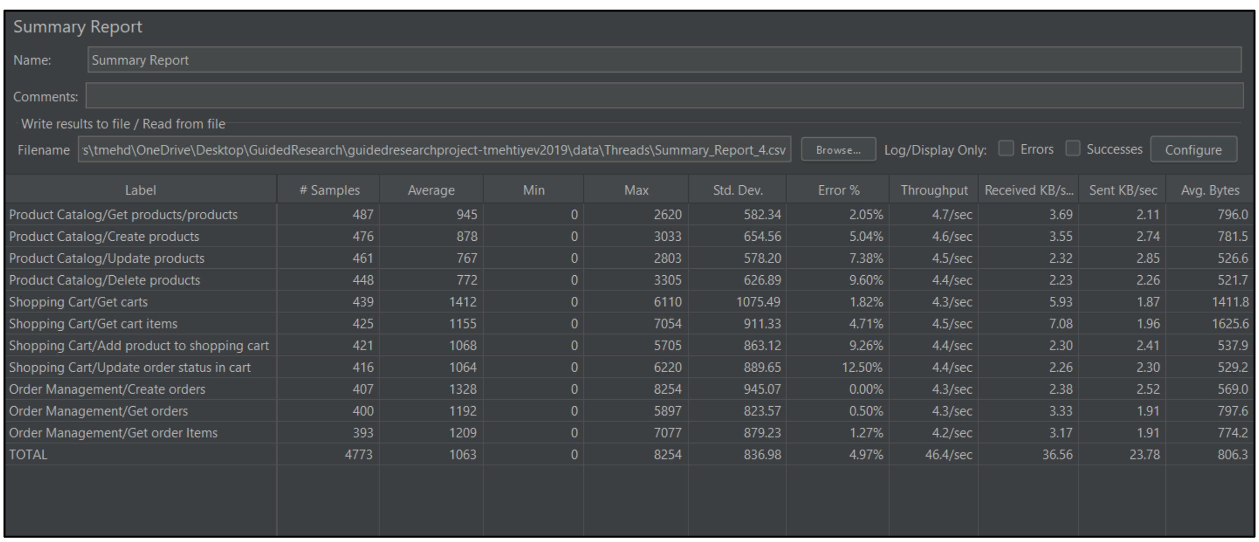


Figure 41

Summary Report & Aggregate Report:

Tabulated comprehensive statistics, including Sample count, Average response time, Median response time, 90th percentile response time, Minimum and Maximum response times, Error percentage, Throughput (requests per second), Received KB/sec & Sent KB/sec, providing a holistic overview of performance.



Summary Report

Name: Summary Report

Comments:

Write results to file / Read from file

Filename s\tmehd\OneDrive\Desktop\GuidedResearch\guidedresearchproject-tmehtiyev2019\data\Threads\Summary_Report_4.csv Browse... Log/Display Only: Errors Successes Configure

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/s...	Sent KB/sec	Avg. Bytes
Product Catalog/Get products/products	487	945	0	2620	582.34	2.05%	4.7/sec	3.69	2.11	796.0
Product Catalog/Create products	476	878	0	3033	654.56	5.04%	4.6/sec	3.55	2.74	781.5
Product Catalog/Update products	461	767	0	2803	578.20	7.38%	4.5/sec	2.32	2.85	526.6
Product Catalog/Delete products	448	772	0	3305	626.89	9.60%	4.4/sec	2.23	2.26	521.7
Shopping Cart/Get carts	439	1412	0	6110	1075.49	1.82%	4.3/sec	5.93	1.87	1411.8
Shopping Cart/Get cart items	425	1155	0	7054	911.33	4.71%	4.5/sec	7.08	1.96	1625.6
Shopping Cart/Add product to shopping cart	421	1068	0	5705	863.12	9.26%	4.4/sec	2.30	2.41	537.9
Shopping Cart/Update order status in cart	416	1064	0	6220	889.65	12.50%	4.4/sec	2.26	2.30	529.2
Order Management/Create orders	407	1328	0	8254	945.07	0.00%	4.3/sec	2.38	2.52	569.0
Order Management/Get orders	400	1192	0	5897	823.57	0.50%	4.3/sec	3.33	1.91	797.6
Order Management/Get order Items	393	1209	0	7077	879.23	1.27%	4.2/sec	3.17	1.91	774.2
TOTAL	4773	1063	0	8254	836.98	4.97%	46.4/sec	36.56	23.78	806.3

Figure 42

Aggregate Graph:

A visual representation of key metrics like Average, Median, 90th percentile, and Min/Max response times, Throughput, Standard deviation, providing graphical insights for quick trend analysis.

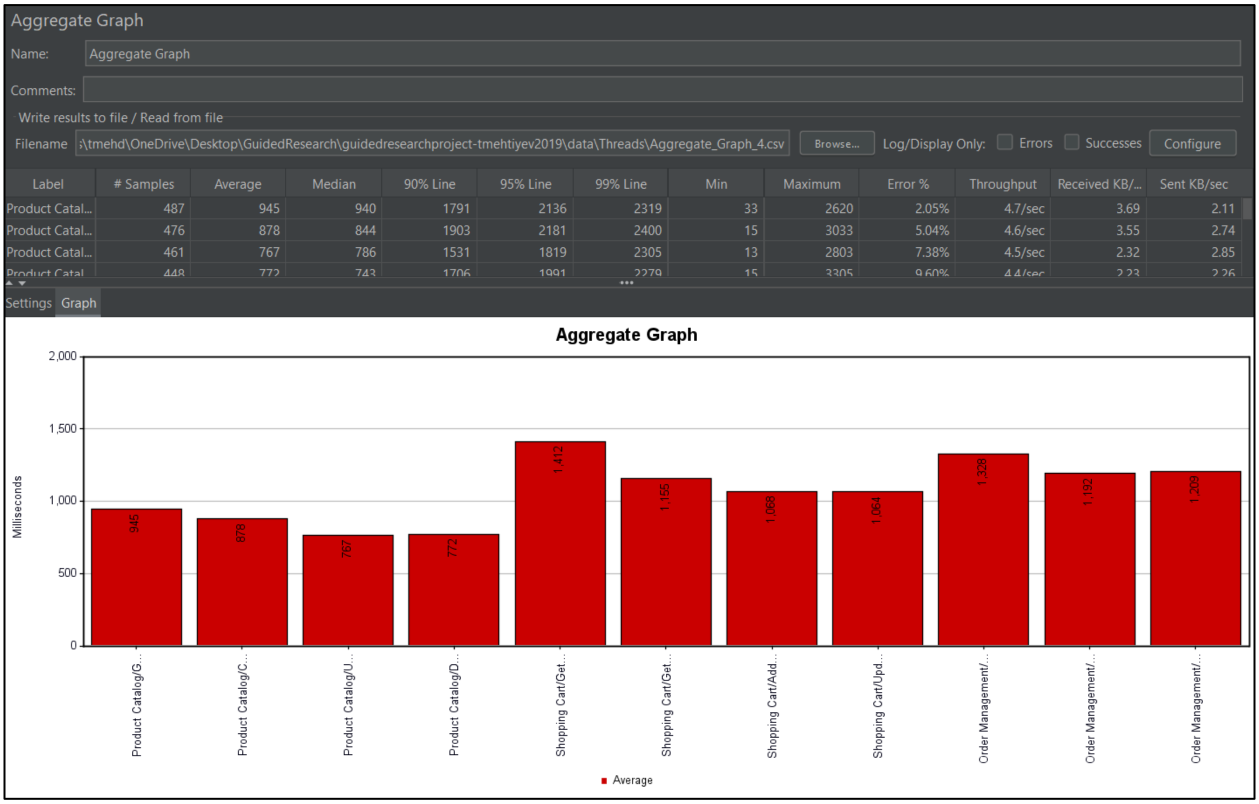


Figure 43

Response Time Graph:

Graphical portrayal of Response time against time or sample number, Deviation, Throughput, Median, Average, and 90th percentile values, aiding in visual detection of patterns or anomalies in response time.

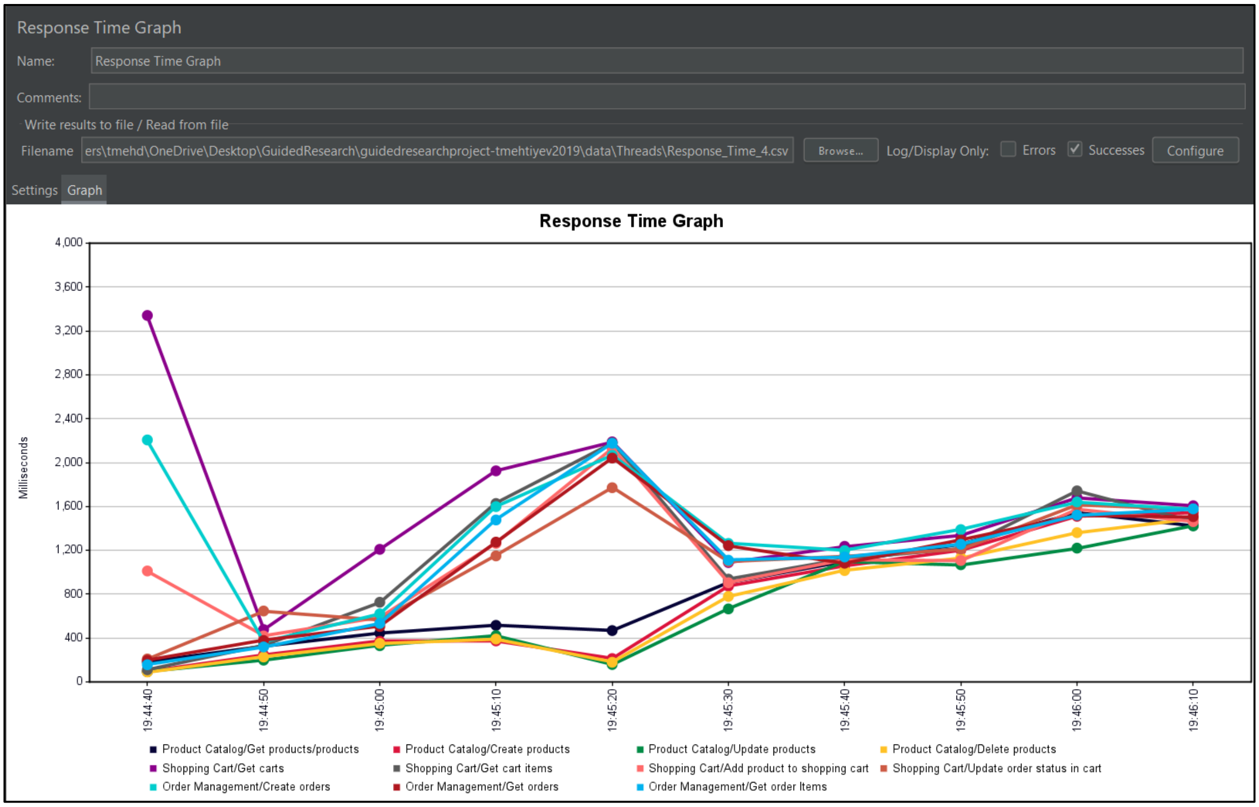
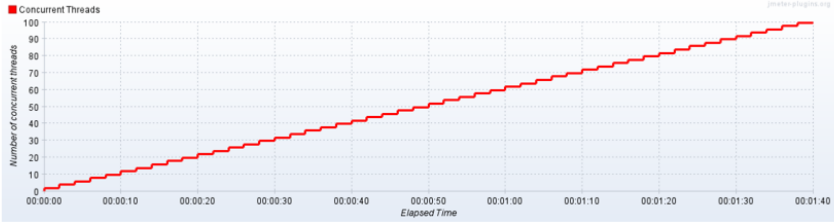


Figure 44

Visual Description | Effects of Ramp-Up Steps Count on Response Time

Effects of Ramp-Up Steps Count on Response Time



Metrics	Value
Target Concurrency Count	100
Ramp-Up Time, sec	100
Ramp-Up Steps Count, sec	50
Hold Load For, sec	0

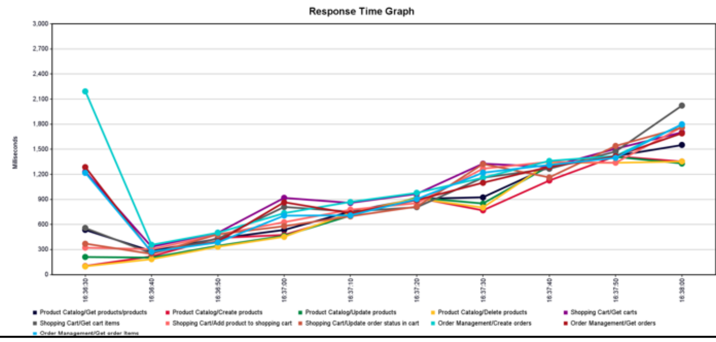
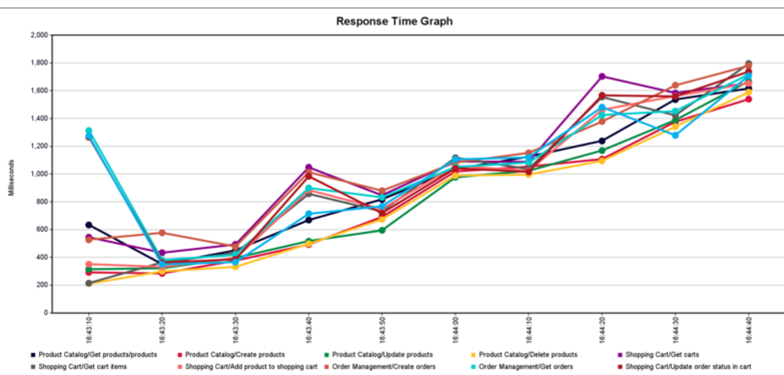


Figure 45

Effects of Ramp-Up Steps Count on Response Time

Metrics	Value
Target Concurrency Count	100
Ramp-Up Time, sec	100
Ramp-Up Steps Count, sec	5
Hold Load For, sec	0



16

Figure 46

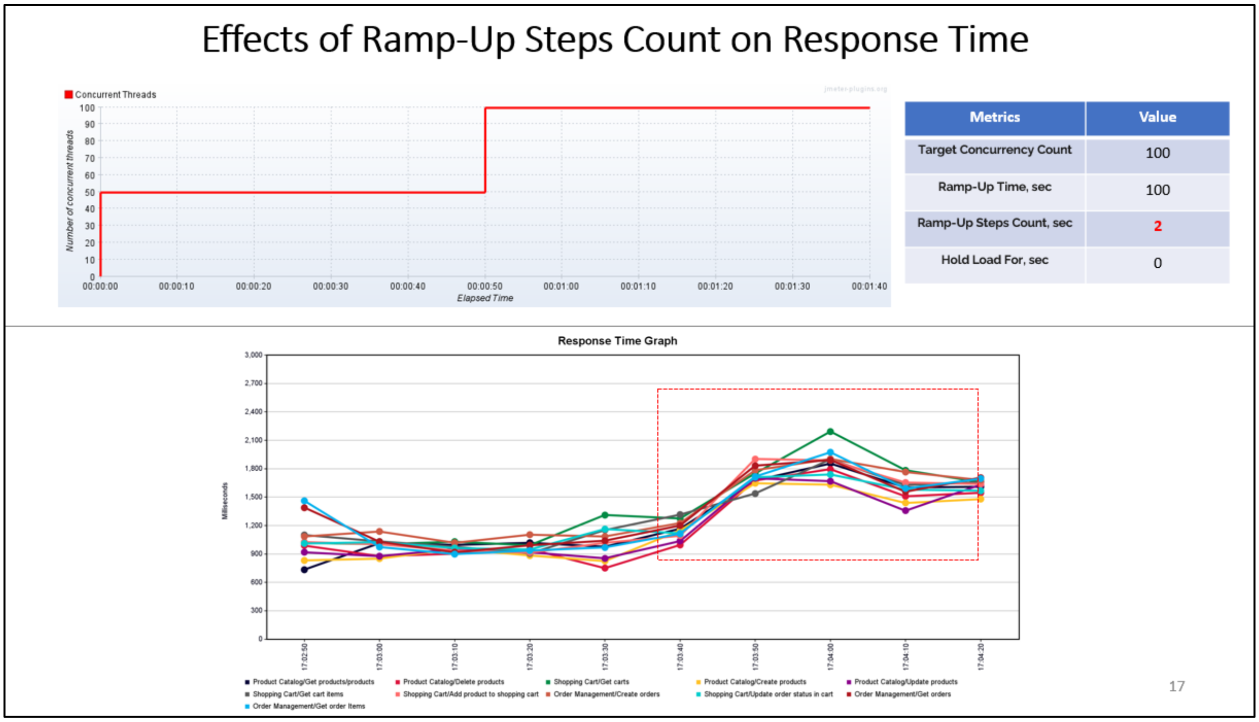


Figure 47

Statistical Analysis | Effects of Threads Count on Response Rate

With the refined data, I have applied various statistical techniques to extract meaning from the numbers. By leveraging the **Central Limit Theorem**, **hypothesis testing**, I have gained a deeper understanding of the application's performance metrics (mainly response rate) under different user loads.

Below are the details of the response rate distribution for each test scenario:

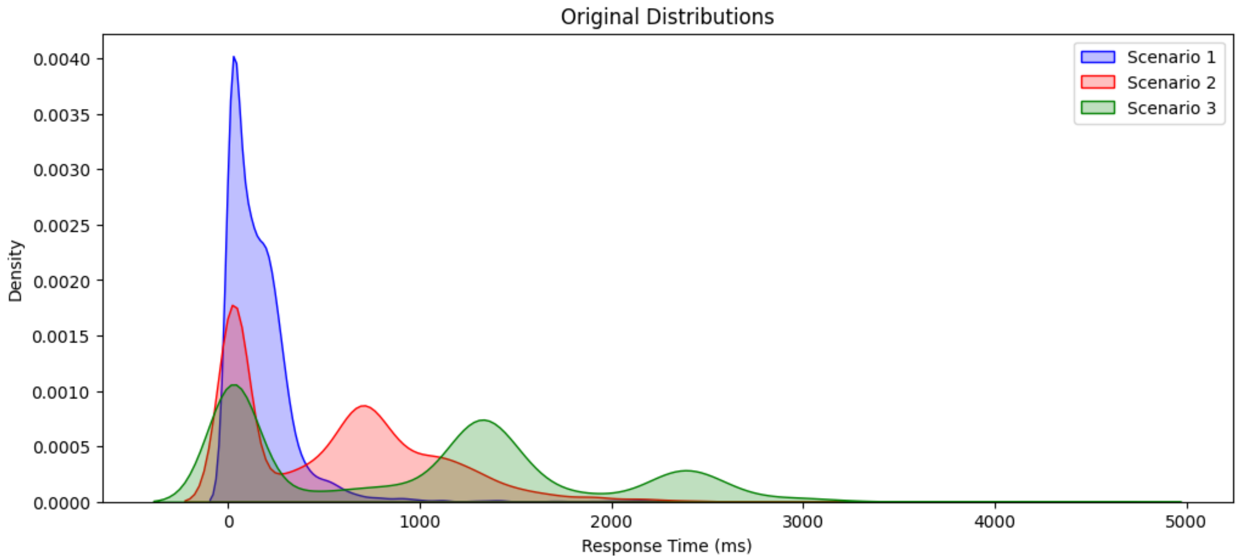


Figure 48

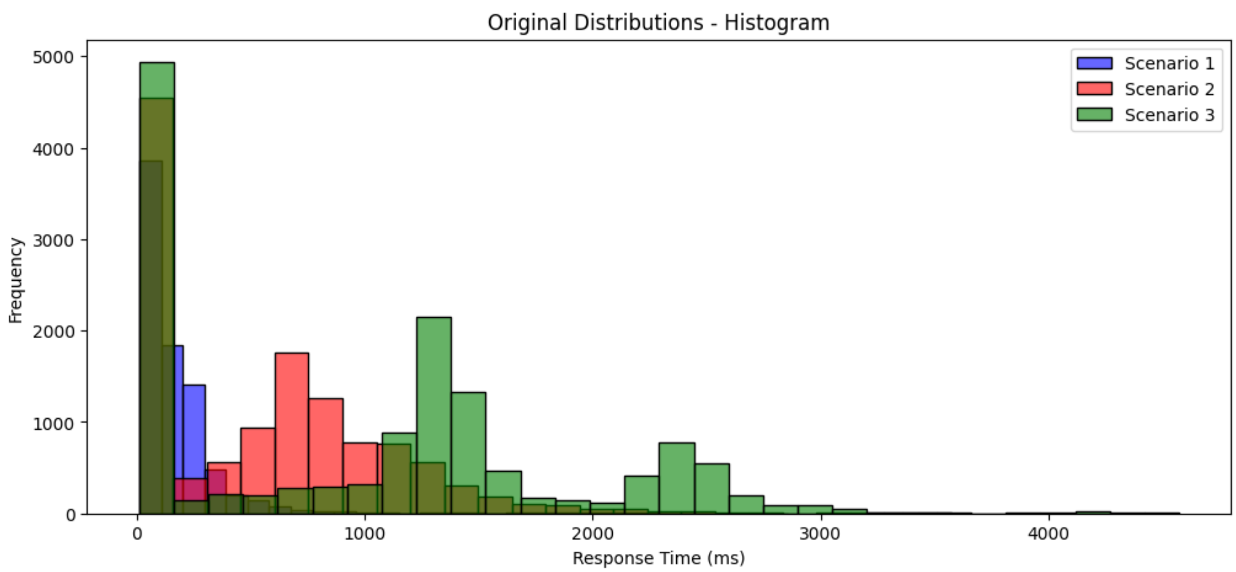


Figure 49

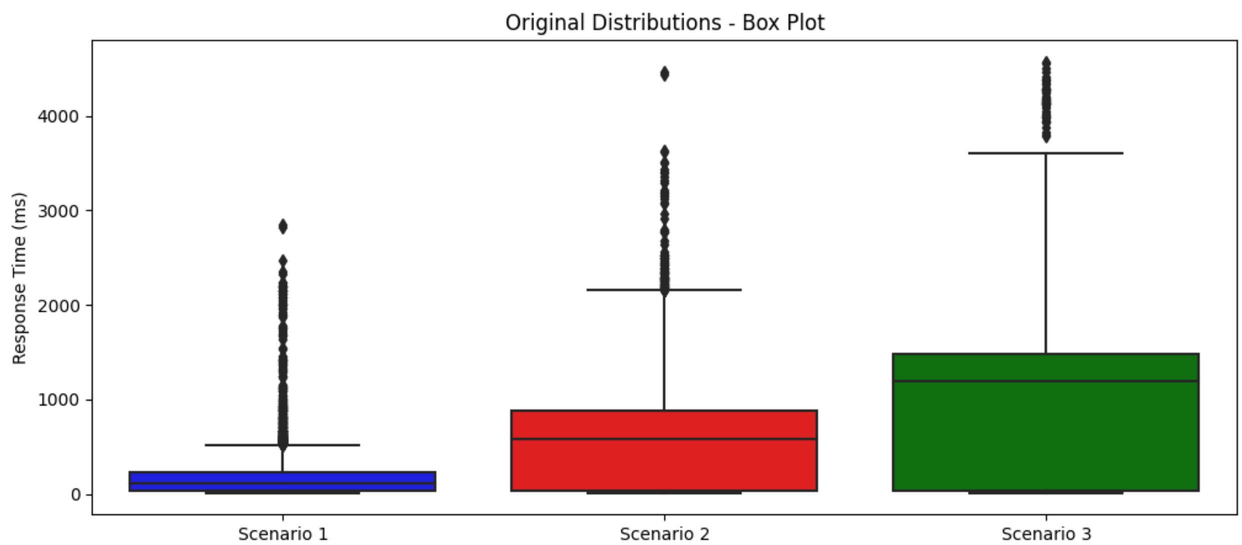


Figure 50

Application of the Central Limit Theorem

The Central Limit Theorem (CLT):

The Central Limit Theorem is a statistical principle stating that, given a sufficiently large sample size, the distribution of the sample means of independent and identically distributed random variables will be approximately normal, irrespective of the original distribution of the variables.

Why Did I Apply the Central Limit Theorem?

- **Normality Assumption**: Many statistical techniques and tests assume the data to be normally distributed. If the original data is not normal, applying the CLT helps meet this assumption by working with the distribution of sample means instead.

- **Statistical Robustness**: By working with a normally distributed dataset (the distribution of sample means), my hypothesis tests and statistical inferences become more reliable and robust.

- **Simplification**: The normal distribution is well-understood, and its properties are widely used in statistics. By ensuring that my dataset adheres to a normal distribution (via the CLT), analysis and interpretation become more straightforward.

Details from the Application:

Sample Size: In the sampling process, the sample size (number of observations in each sample) is set as 50.

Number of Samples: I've collected 1,000 samples from each scenario. This means that I've drawn 1,000 separate samples, each of 50 observations, and then calculated their means.

Random Sampling: I've drawn random samples from the original data, ensuring that each sample is drawn independently.

The application of the Central Limit Theorem allowed me to make statistical inferences using the normal distribution, which has desirable properties. By applying the CLT, I'm aiming to leverage these properties to make more reliable decisions and interpretations based on my JMeter test results.

Below you can find the distribution for each test scenario after the application of CLT.

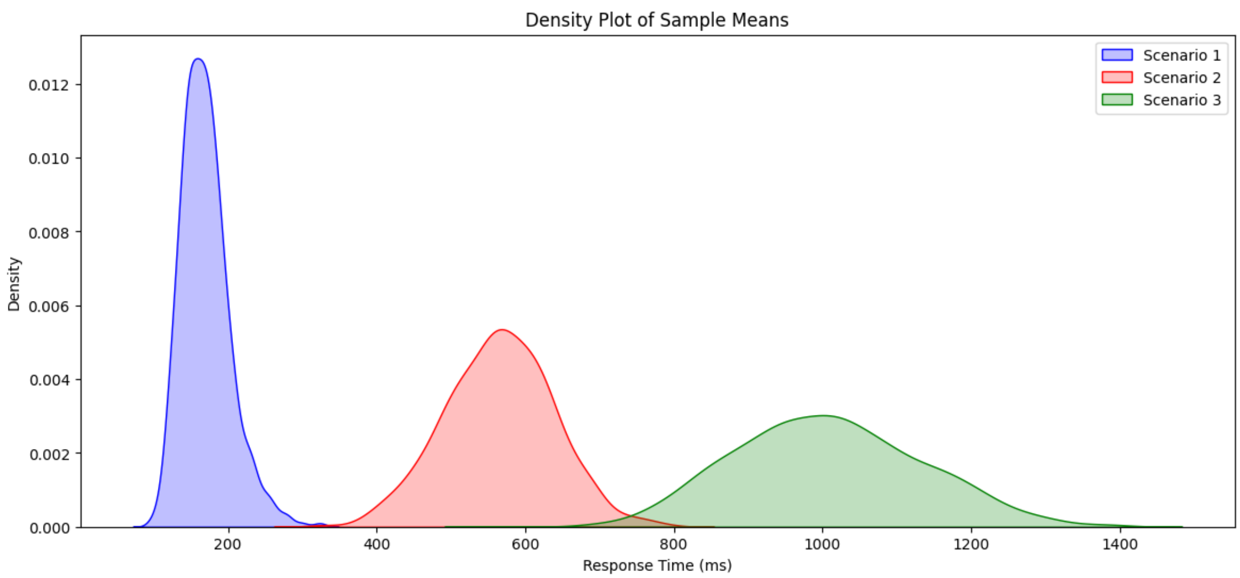


Figure 51

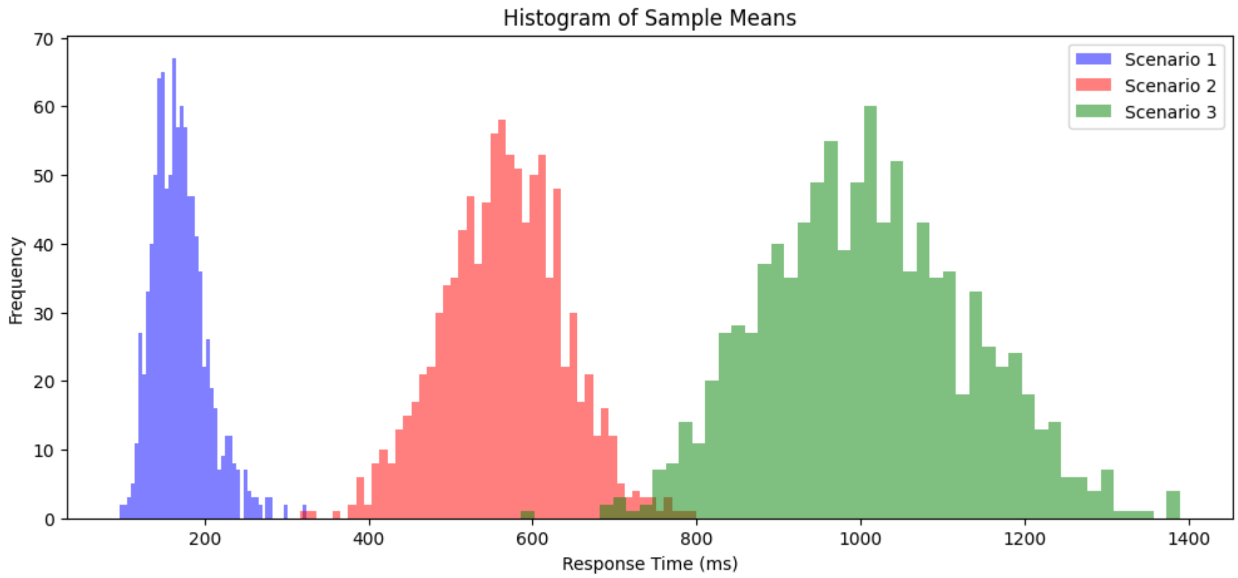


Figure 52

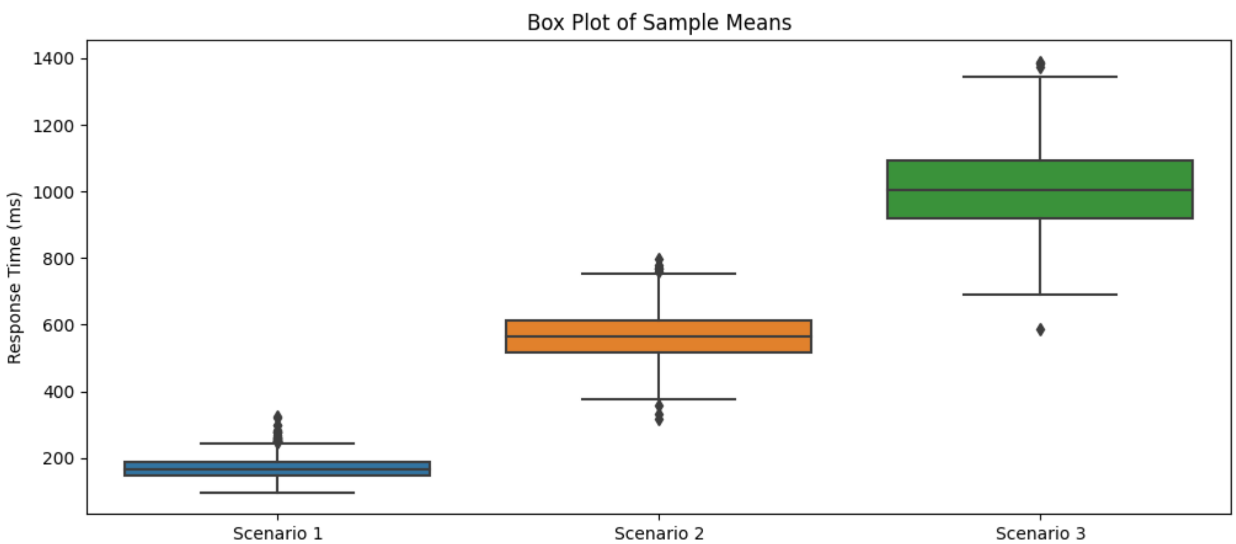


Figure 53

Hypothesis Testing:

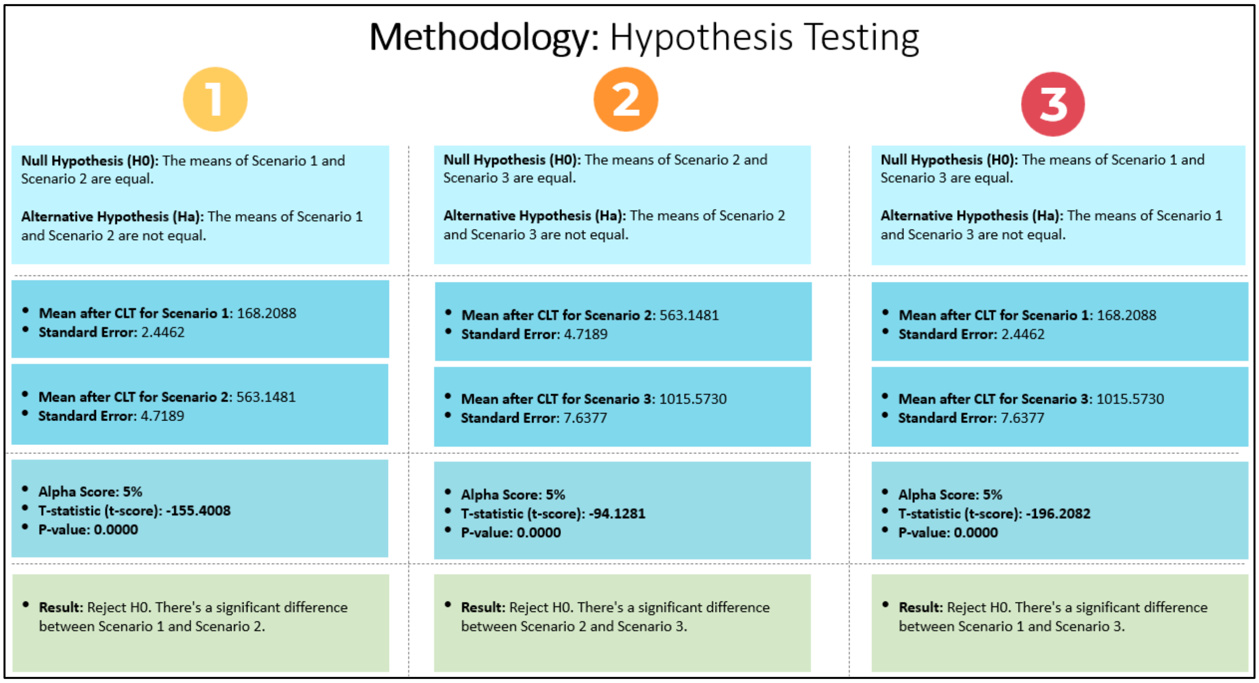


Figure 54