Milestone 3: Practical Implementation of LowTech Gmbh Webshop in Azure

Wladymir Alexander Brborich Herrera (1437876)
wladymir.brborich-herrera@stud.fra-uas.de,
Vishwaben Pareshbhai Kakadiya (1471845)
vishwaben.kakadiya@stud.fra-uas.de,
Hellyben Bhaveshkumar Shah (1476905)
hellyben.shah@stud.fra-uas.de,
Heer Rakeshkumar Vankawala (1449039)
heer.vankawala@stud.fra-uas.de, and
Priyanka Dilipbhai Vadiwala (1481466)
priyanka.vadiwala@stud.fra-uas.de

Frankfurt University of Applied Sciences
(1971-2014: Fachhochschule Frankfurt am Main)
Nibelungenplatz 1
D-60318 Frankfurt am Main

Abstract This report presents a summary of all the planification and execution of the LowTech GmbH Cloud Transformation Project hosted in https://github.com/Helly2010/Cloud-Computing-Project. The final objective is to develop a proof of concept of the proposed infrastructure of a Cloud Native application, in this case, the Webshop. As detailed in milestone 2 the application will be hosted in Microsoft Azure. This report also discusses the advantages in system reliability, security, and scalability offered by the proposed approach. Moreover, we include a detailed description of the development techniques using tools like Terraform for infrastructure management and GitHub Actions for continuous integration and deployment (CI/CD). Special attention is given to the advantages in development time and repeatability of such tools and techniques. Furthermore, we evaluate the cost of the overall infrastructure identifying areas for optimization. The report also discusses future scalability strategies, ensuring that the application can handle different types of loads and it actually makes sense in a cloud environment Finally, the report provides a summary on the lessons learned and interesting findings while using the tools and techniques mentioned.

1 Introduction

LowTech GmbH, a medium-sized enterprise specializing in wooden furniture production, is modernizing its IT infrastructure as part of a comprehensive cloud transformation. Initially relying on traditional on-premises systems, the company faced challenges in scalability, security, and operational efficiency.

The transformation began with a thorough assessment of the existing infrastructure, which revealed the following key challenges:

- Limited scalability due to fixed hardware constraints.
- High operational costs and energy consumption of legacy systems.
- Outdated security measures, including basic firewall protection.
- Lack of automation, requiring manual interventions for maintenance and scaling.

To address these challenges, a private cloud migration strategy was adopted. The aim was to improve scalability, security, and cost-efficiency while ensuring minimal downtime and business continuity. The company transitioned its Webshop as a cloud native application to Microsoft Azure.

1.1 Overview of the Previous Milestones

1.1.1 Starting point LowTech GmbH, a small to medium-sized enterprise with 45 employees specializing in wooden furniture production, is facing a critical need for technological transformation. Initially relying on sales representatives, the company transitioned to an online store a few years ago, which has now become their primary selling platform. This shift has led to a significant increase in user numbers over the past two years, necessitating a modernization of their IT infrastructure. The CEO of LowTech GmbH, recognizing the need for modernization, has expressed interest in cloud computing to make the company future-ready.

1.1.2 Milestone 1: The objective of the technological transformation for LowTech GmbH is to modernize its server and application infrastructure by migrating to a private cloud environment. This initiative aims to create a scalable, secure, and efficient IT framework that aligns with the company's growing needs, particularly in response to increased online sales and user traffic. By adopting a private cloud strategy, LowTech GmbH will retain control over its data while outsourcing hardware maintenance to a third-party provider. This transition will involve leveraging newer technologies to enhance performance and availability, ensuring compliance with security standards, and reducing operational costs. Ultimately, the transformation seeks to provide a future-ready infrastructure that supports the company's evolving business model and operational requirements.

This was our assessment of the original infrastructure:

Scalability

- **Fixed Hardware & Inflexible Infrastructure**: Current infrastructure consists of 7 on-premises servers housed in a single 19-inch rack, along with 17 clients and 19 laptops all with predetermined, static configurations. Moreover, the physical constraints of the on-premises setup, with no additional space for expansion, severely limit scaling options. This inflexibility makes it challenging to accommodate growth or adapt to changing business needs.
- **Absence of Resource Utilization/pooling**: There's no apparent way to quickly scale resources up or down based on demand or user traffic fluctuation in the current infrastructure. Each application typically runs on a dedicated server with fixed resources. This approach leads to inefficient resource utilization, as some servers may be underutilized while others are overloaded which may lead to performance issues during peak times or resource waste during low-demand periods due to no dynamic resource allocation.
- **Manual Processes & High Cost**: Any changes in capacity would likely require manual hardware upgrades or replacements including hardware installation, and configuration, making the process time-consuming and potentially leading to downtime. Replacement of hardware is not only tedious but also financially burdensome due to high costs of new hardware.

Availability

- **Obsolete Hardware/OS & Runtime Environments**: Many components of the current infrastructure is based on very old hardware and outdated operating systems such as Windows XP SP3 (Finance clients), Windows 7 SP3 (HR clients and Customer Service laptops), Debian 5.0 Lenny (Warehouse clients and server), Ubuntu 16.04 LTS (Sales CRM Storage server) etc. Several applications are running on outdated software versions such as Java 1.7/1.8, MySQL 5.5/5.7, PHP 5.3 and Firefox 3.6 etc. which makes this whole infrastructure more susceptible to failure.
- **Lack of Redundancy and Backup Mechanism**: There's no mention of redundant systems or data backup solutions which might lead to significant service disruptions as well as data loss in case of any system failure.
- **Manual Maintainance**: It is impossible to meet high availability requirements without a robust failover mechanism due to manual maintenance operations. This increases the possiblity of human errors, leads to longer downtime and reduces overall reliability.

Security

- **Basic Windows Firewall**: It provides minimal outward traffic control, which may allow malware to interact easily. Without additional tools, centralized administration is impossible to maintain consistent security rules across all 36 client devices (17 clients and 19 laptops). It lacks advanced threat protection features, leaving the system vulnerable to sophisticated attacks.
- **pfSense for Network Packet Filtering**: There are no built-in antivirus features, which could let malicious payloads get through. Its effectiveness heavily relies on proper configuration, which may be challenging without dedicated IT staff. Moreover, advanced intrusion detection requires additional setup and maintenance.

As detailed in the analysis the infrastructure was becoming quite expensive and inefficient given the company objectives.

Plan and execution based on the original requirements: This milestone resulted in a migration and optimization plan for the entire infrastructure:

- Talk with server space providers, we need to stablish a contract and negotiate price and SLOs based on the server requirements
- Gather all the installation information, binaries, licenses and documentation for all applications.
- Configure a development environment to upload all artifacts such as configuration files and container images.
- Create Ansible configuration files for each application, to automate the installation and replication process.
- Develop scripts to automate application functionality and load testing. To determine if the configurations will be on par or better than the current infrastructure.
- Provision and install the private cloud management software, including the networking configuration
- Configure the storage
- Create the virtual machines for the base hosts
- Configure Prometheus to monitor the virtual machine installations
- Establish the network links between the different virtual domains
- Install all database servers
- Develop a migration process to replicate the current data into the new database servers.
- Review that data is up to parity with the legacy services
- Create a proxy getaway for the services, so we can redirect the traffic from the old services to the new ones.
- Deploy the new applications into the private cloud
- Check that the data replication is working
- Prepare the load shifting in sequence, and off hours
- Shift load in sequence, starting from the most isolated applications first, then the ones with the most dependencies.
  - Migrate finance and HR, since they are self-contained
  - Migrate operations
  - Migrate customer service
  - Migrate warehouse
  - Migrate webshop
  - Migrate sales

1.1.3 Milestone 2

Once the client for the project decided to actually migrate to a public cloud service provider, we developed a new strategy. Given that the Webshop in particular was going to be developed from scratch following modern approaches with new technologies.
We proposed a new standard for this kinds of applications:

Webshop Infrastructure, Tools, and Services: This application will be newly developed following the standard for new applications. As a special consideration, this application is the only public-facing application that needs to handle customer requests. Our standard for newly developed applications enables this critical piece of the business to easily scale and be secured with valid certificates with no extra configurations.

Cloud Context	Products And Technologies	Service Models
Public Cloud	Azure App Service	PaaS or CaaS
Public Cloud	Azure Static Web App	PaaS or CaaS
Public Cloud	Azure Database For PostgreSQL	PaaS

Table 1: Webshop: Website Deployment Strategy

General Considerations: Importantly, for all newly developed applications, we will:

- Set up a private GitHub repository
- Set up GitHub actions to enable continuous integration and continuous deployments in Azure
- Set up an infrastructure folder to hold terraform files. These files will define the infrastructure and will allow for easy repeatability
- Use Microsoft Entra ID in case of authentication with the organization is needed
- Use Docker to package the application with containers that will be deployed in Azure App Service in case of the backend, and Azure Static Web Apps in case of the frontend

Standard For a Cloud Native Application: When developing a cloud-native application Figure 1 shows the basic setup to integrate DevOps concepts into our workflow. The website is going to be developed under this standard.

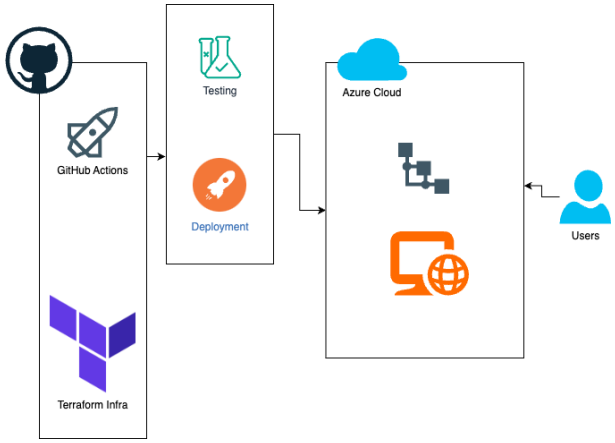


Figure 1: Standard Configuration To Deploy An Application

1.2 Objectives of the Cloud Implementation of Webshop

The Webshop is a critical component for LowTech GmbH, serving as the company's primary sales platform. The goal of its cloud implementation is to enhance performance, scalability, and security, ensuring a seamless user experience.

Key objectives for the Webshop's cloud-based deployment include:

- **Scalability and Performance Optimization:** The Webshop is deployed on Azure App Service with auto-scaling capabilities, enabling efficient traffic handling during peak periods. A load balancer ensures even traffic distribution across multiple instances.

- **High Availability and Reliability:** Azure Virtual Machine Scale Sets provide fault tolerance with automatic failover. Azure Blob Storage is used to securely store digital assets with high availability.
- **Security and Compliance:** The Webshop integrates with Microsoft Entra ID for user authentication and Azure Security Center for enhanced threat protection. Encryption and Role-Based Access Control (RBAC) are employed to safeguard sensitive customer data.
- **Continuous Deployment and DevOps Automation:** A CI/CD pipeline powered by GitHub Actions automates code deployments, improving deployment speed and reducing manual intervention.
- **Cost Efficiency and Resource Optimization:** The dynamic allocation of resources optimizes compute and storage usage, reducing operational expenses. Azure's pay-as-you-go model aids in cost forecasting and budget management.
- **Future-Proofing and Cloud-Native Development:** The Webshop follows cloud-native best practicesn such as infrastructure as code, and continuous deployments.

2 Application Design

2.1 Architectural Overview

As requested by the project. This application wil have 3 tiers. The presentation or user interface, the backend or business logic, and the storage tier. By our plan in milestone II we will deploy each tier in a different azure service tailored for its needs. Each of the services provides autoscaling features, and load balancing. Ensuring that we are leveraging the elasticity charactersitics of the cloud.

2.1.1 Presentation-Tier (Frontend) - User Interface (UI) It is important to note that the majority of the presentation was forked from https://github.com/DebasmitaMallick/React-e-Commerce-Website. This allowed our team to focus on backend and cloud deployment tasks. The original project was modified to interact with our backend API, integrate with payment providers and provide detailed information about products.

Technology Stack

- Frontend Framework: React.js (JavaScript)
- State Management: React Context API
- Hosting & Deployment: Azure Static Web Apps

Functionality Overview

1. Navigation & Routing Users can navigate between different sections of application, such as viewing product lists, accessing product details, and managing their shopping cart. React Router enables seamless client-side navigation.
   - React Router Setup
     - React Router (react-router-dom) to handle the routing of different components, allowing users to navigate through pages like the homepage (/), product detail pages (/product/:id), and potentially a shopping cart or checkout page.
     - For product detail pages, you're using dynamic routes with product/:id to fetch and display specific product data based on the product's unique id. For instance:
     - When a user clicks on a product in the catalog, the URL changes to something like /product/123, and the ProductDetail component is rendered with data for product 123.
     - This is achieved using useNavigate and useParams hooks provided by React Router to capture the dynamic part of the URL.

  - Navigation Links
    - On the homepage (Home.js component), displaying a list of products. Each product has an image and name that users can click. When clicked, the app navigates to the product detail page using the navigate(/product/$prod.id) function
2. API Communication
   - The product data, which includes the list of products with details such as images, names, prices, and categories, is fetched when the homepage or product catalog page loads. This is done using useEffect to trigger an API call and update the state with the fetched data.
   - API communication in app is responsible for sending and receiving data from the backend server. This includes fetching product data to populate catalog, and creating orders after checkout.
   - For payment, we use PayPal and Stripe in a sandbox mode. As this is not the final product, we are just testing the actual integration with the providers.

Key Features of the UI

1. UI Components & Design
   - Key Features
     - Reusable Components: Components like SingleProduct.js, ProductDetail.js, and Cart.js.
     - Bootstrap Integration: react-bootstrap is used to style UI components to enable responsive behavior.
     - Dark & Light Theme Support: A theme context (ThemeContextProvider) dynamically adjusts UI styles based on user preference.
2. Product Display & Interaction The UI displays products with details, allowing users to browse, select, and view product specifications before making a purchase.
   - Key Features
     - Product Catalog: Displays a grid of available products with images, names, and prices.
     - Product Cards (SingleProduct.js):
       * Clickable product images navigate to the product details page.
       * Shows product information such as name, category, description, and price.
       * “Add to Cart” and ”Remove from Cart” buttons enable quick cart management.
     - Product Details Page (ProductDetail.js):
       * Displays full product details, including stock availability.
       * Users can add/remove items from the cart.
3. Shopping Cart UI & Checkout The cart and checkout sections provide a way for users to review their selected products and manage quantities.
   - Key Features
     - Cart UI (Cart.js)
       * Displays a list of selected products with prices and a ”Remove from Cart” button.
       * Updates total price dynamically based on the cart’s contents.
       * Navigates users to checkout when ready.
     - Checkout UI (CheckoutForm.js)
       * Collects user details (name, email, and shipping information).
       * Integrates Stripe and PayPal for secure payment processing.
       * Displays success messages with reference numbers for placed orders.
4. Notifications The app uses toast notifications and error messages to keep users informed.
   - Key Features
     - Toast Notifications (react-toastify)
       * Displays success messages when products are added/removed from the cart.
       * Shows error messages if something goes wrong (e.g., out-of-stock products, payment failures).

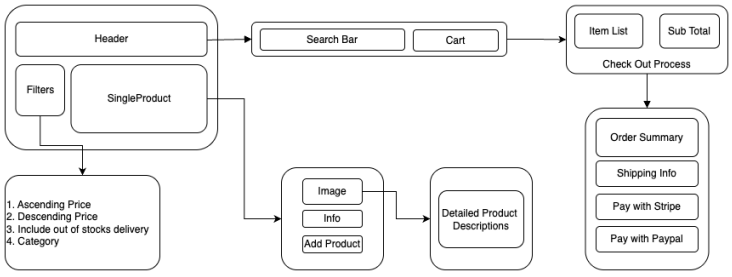


Figure 2: Component Diagram

2.1.2 Application-Tier (Backend) - Business Logic

Technology Stack

- FastAPI
- SQLAlchemy

Product Management

The Product Management module handles the lifecycle of products in the system. This includes operations for adding new products, updating product details, and retrieving product information. The core business logic ensures that products are categorized correctly, their stock levels are managed, and prices are updated as needed.

- Key Operations:
  - Add, update, and delete products.
  - Manage product details such as descriptions, prices, and categories.
  - Track stock availability and reorder levels.

Order Management

The Order Management module manages customer orders, from order creation, and status changes. It tracks the order status and ensures that orders are processed correctly.

- Key Operations:
  - Create and update orders with customer and product information.
  - Monitor order statuses (e.g., processing, dispatched, delivered).
  - Validate inventory and ensure product availability during order processing.

Inventory Management

The Inventory Management module tracks the stock levels of all products, ensuring that inventory can be updated.

- Key Operations:
  - Monitor and update product stock levels after each sale.
  - Track the supplier's stock.

Email Notification

The Email Notification module sends emails to customers and administrators for various events in the system, such as order confirmations, shipment tracking updates, and payment status notifications. It integrates with FastMail to send HTML-formatted emails with dynamic content.

- Key Operations:
  - Send order confirmation emails to customers.
  - Notify customers of order status updates, such as dispatched or delivered.
  - Alert suppliers of low stock levels and send reorder requests.
  - Provide a user-friendly email template system for various types of notifications.

2.1.3 Data-Tier (Database) - Databases

Technology Stack

- PostgreSQL
- Alembic
- Azure Database for PostgreSQL

Database

1. Product Data
   - Product
     - Stores the core product information and links to categories, suppliers, and inventory.
     - Key Attributes:
       * id: Unique product identifier (Primary Key).
       * name: Product name.
       * description: Product description.
       * category_id: Foreign Key linking to the Categories table.
       * supplier_id: Foreign Key linking to the Suppliers table.
       * stock_id: Links to the Stock table for inventory tracking.
       * public_unit_price: Price displayed to customers.
       * supplier_unit_price: Price from the supplier (for internal calculations).
       * ean_code: Unique product code for identification.
       * reorder_level: Threshold for restocking.
       * img_link: URL to the product image.
       * extra_info: JSON field to store additional product attributes. Like the ratings, or if the product has fast delivery.
   - Catagories
     - This table categorizes products for better organization and searchability.
     - Key Attributes:
       * id: Unique category identifier (Primary Key).
       * name: Product name.
       * description: Category description.
       * extra_info: JSON field for additional metadata.
2. Order Data
   - Order
     - This table records general order details and customer information.
     - Key Attributes:
       * id: Unique order identifier (Primary Key).
       * customer_name: Name of the customer.
       * customer_email: Customer's email address for notifications.
       * customer_phone: Contact number.
       * order_total: Total value of the order.
       * status: Enum representing the order state (e.g., "active", "cancelled").
       * tracking_status: Enum tracking delivery progress (e.g., "dispatched", "delivered").
       * payment_method: JSON field storing payment details (e.g., card type).
       * customer_shipping_info: JSON field for customer delivery address.
       * created_at: Timestamp when the order was created.
       * updated_at: Timestamp when the order was last updated.
   - Order Details
     - This table provides itemized details of each product within an order.
     - Key Attributes:
       * id: Unique identifier for each order detail record (Primary Key).
       * order_id: Foreign Key linking to the Orders table.
       * product_id: Foreign Key linking to the Products table.
       * quantity: Number of units of the product ordered.
       * product_price: Unit price at the time of the order.
       * subtotal: Calculated subtotal for the item.
3. Inventory Data

  - Stocks
    - Monitors the available quantity of each product and manages restocking.
    - Key Attributes:
      * id: Unique stock identifier (Primary Key).
      * quantity: Number of available units for a product.
      * updated_at: Timestamp of the last stock update.
      * created_at: Timestamp when the stock record was created.
  - Suppliers
    - Manages supplier-related information and links to products for procurement.
    - Key Attributes:
      * id: Unique supplier identifier (Primary Key).
      * name: Supplier name.
      * address: Supplier's physical address.
      * phone: Contact number.
      * email: Contact email address.
4. Database Migrations with Alembic
  - It allows us to track database changes, apply updates to production, and roll back changes when needed.
  - Key Operations with Alembic:
    - *Schema Migration:* Automatically generates migration scripts to apply changes to the database (e.g., adding new fields to a table).
    - *Version Control:* Each migration is versioned, allowing us to track and audit changes.
    - *Rollback Support:* Provides the ability to downgrade to previous schema versions if issues arise.

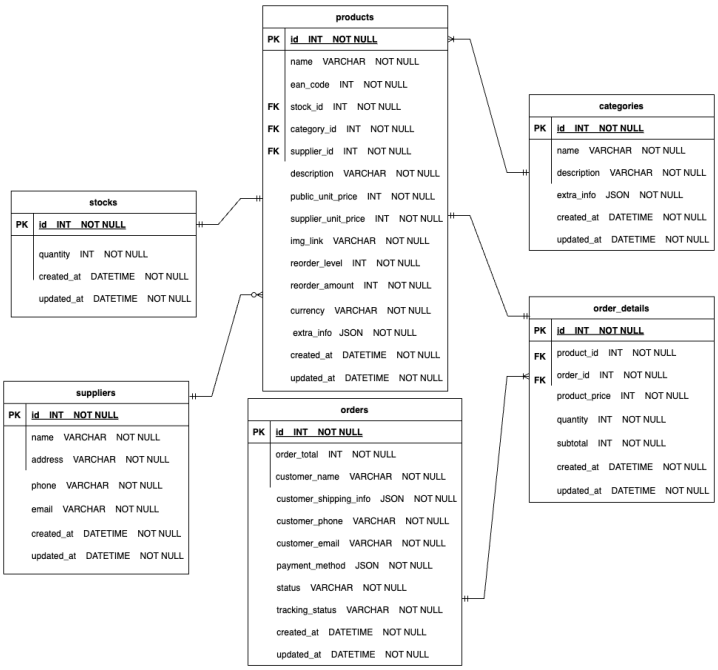


Figure 3: ERD Diagram of the Webshop Database

2.2 Development Process

For the development of the application, the team created a Trello board. Using a kamban-like approach, we scoped, divided, and prioritized all the necesary development and documentation tasks. Internally two teams were formed, frontend and backend. Each working in a different layer of the webshop. The backend team was also in charge of the data layer just for convenience. The most important tasks were to upgrade the existing UI code to fit the new requirements and integrate with the backend, and to actually develop the backend. Frameworks like FastAPI were chosen due to the flexibility benefits, rapid development, and

automatic documentation generation. Which allowed our team members to quickly review the spec and make the integration. During the development timeline, we scheduled weekly meetings to report progress and address concearns with the development environment.

For example, we encountered issues while running the new ‘psycopg‘ libary in asynchronous mode in windows. Which caused some issues while configuring the development environment. This could have been an oportunity to build a docker image to standardize our development environments, but given the timeline, we decided against it and changed to ‘asyncpg‘ in windows

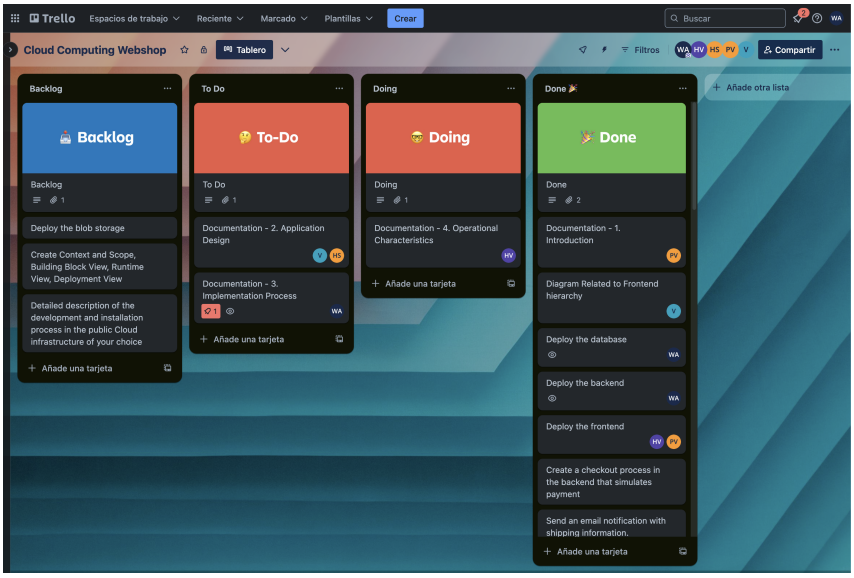


Figure 4: Trello Board

3 System Diagrams

3.1 Context and Scope

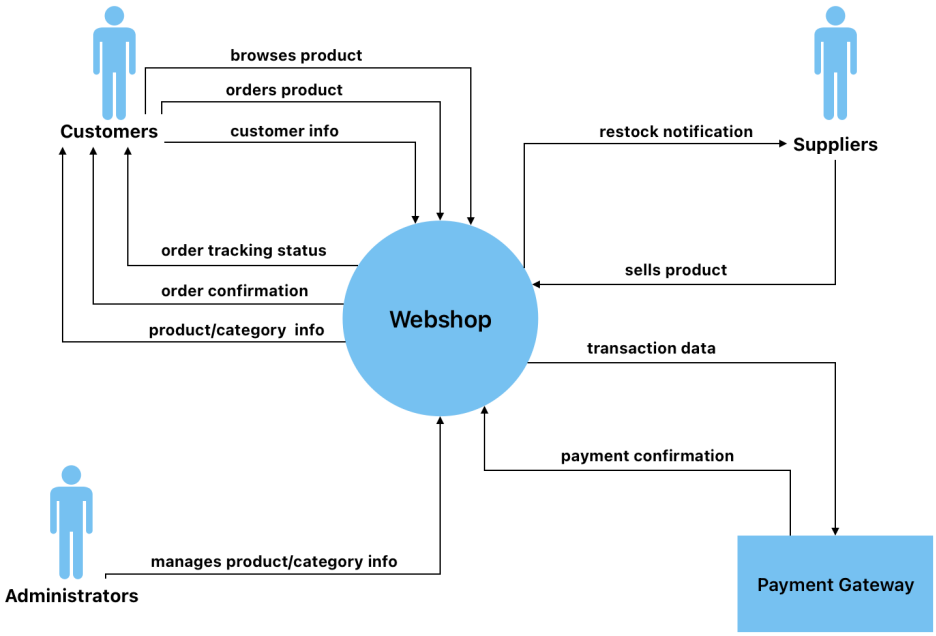


Figure 5: Context and Scope Diagram

3.2 Building Block View

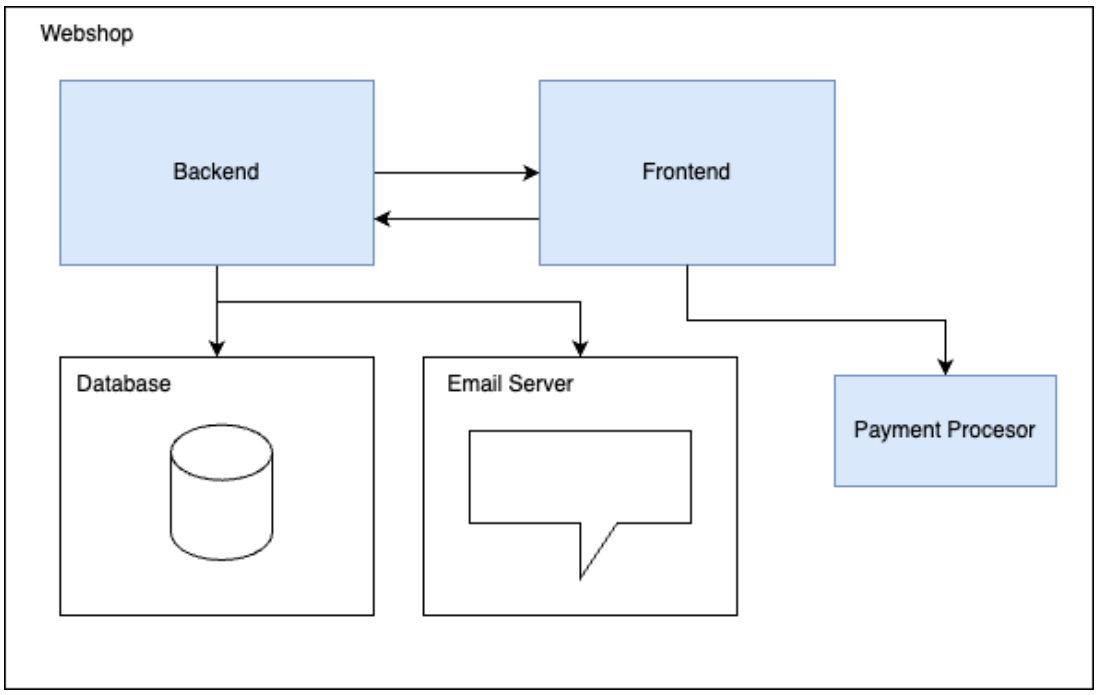


Figure 6: Building Block View Level 1

The building block view makes emphasis on the backend, since we have provided a component diagram for the frontend.

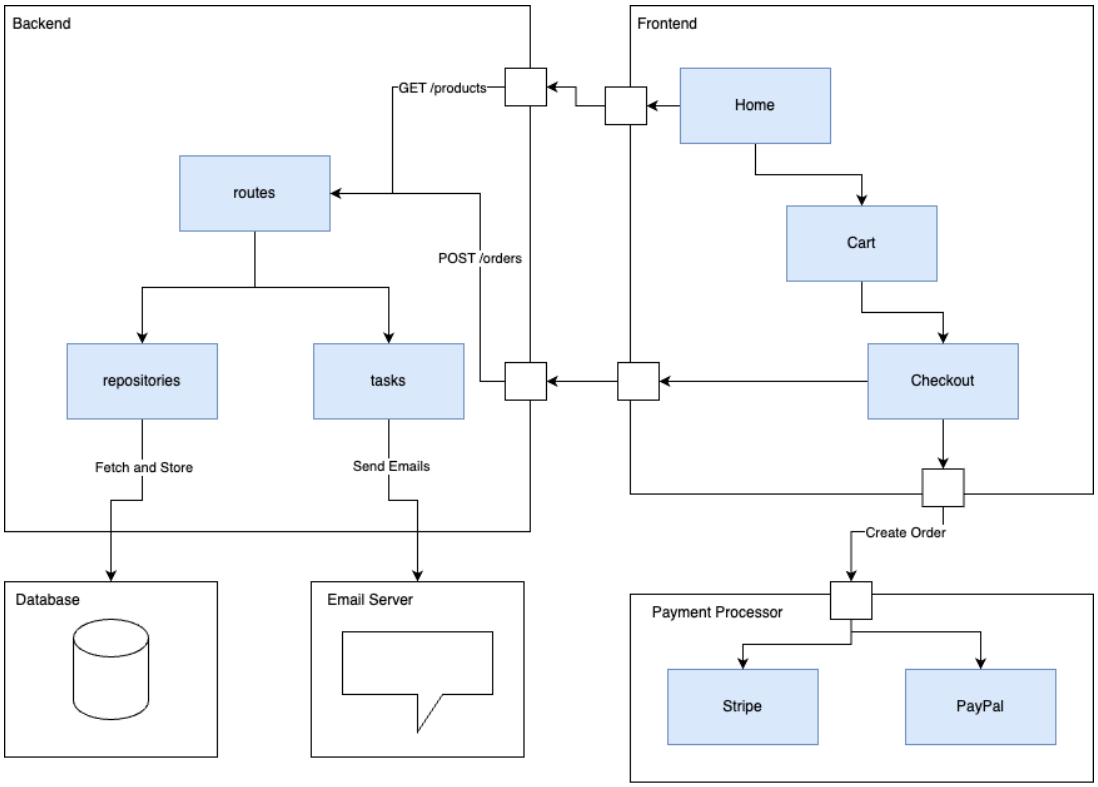


Figure 7: Building Block View Level 2

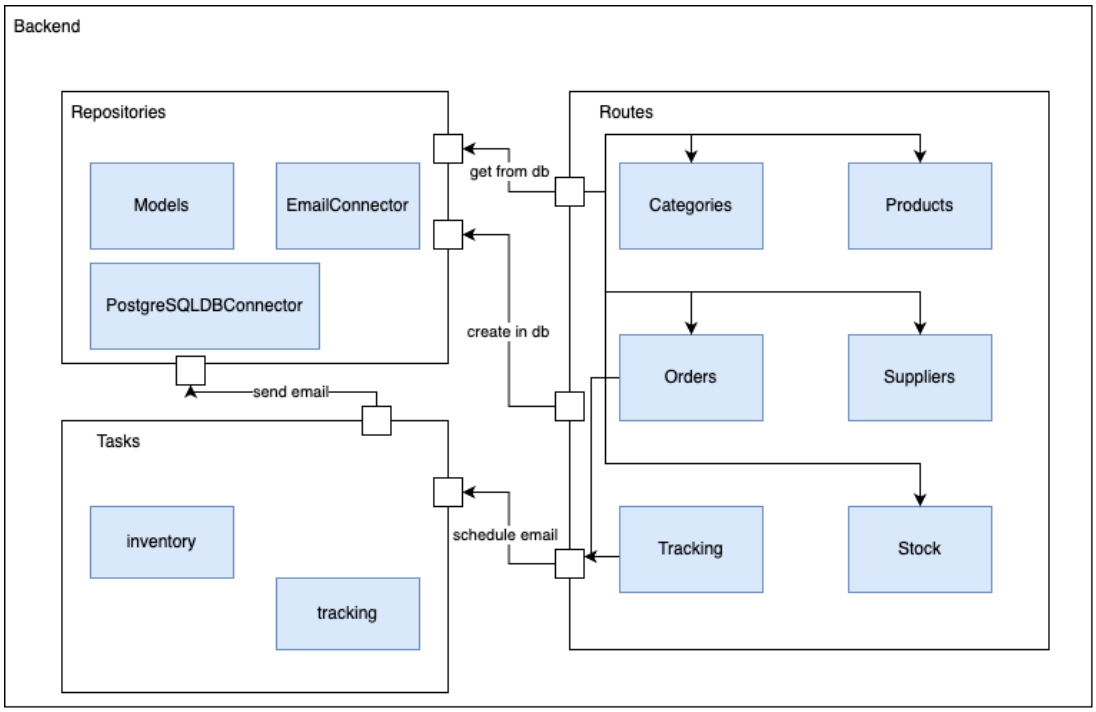


Figure 8: Building Block View Level 3

3.3 Runtime View

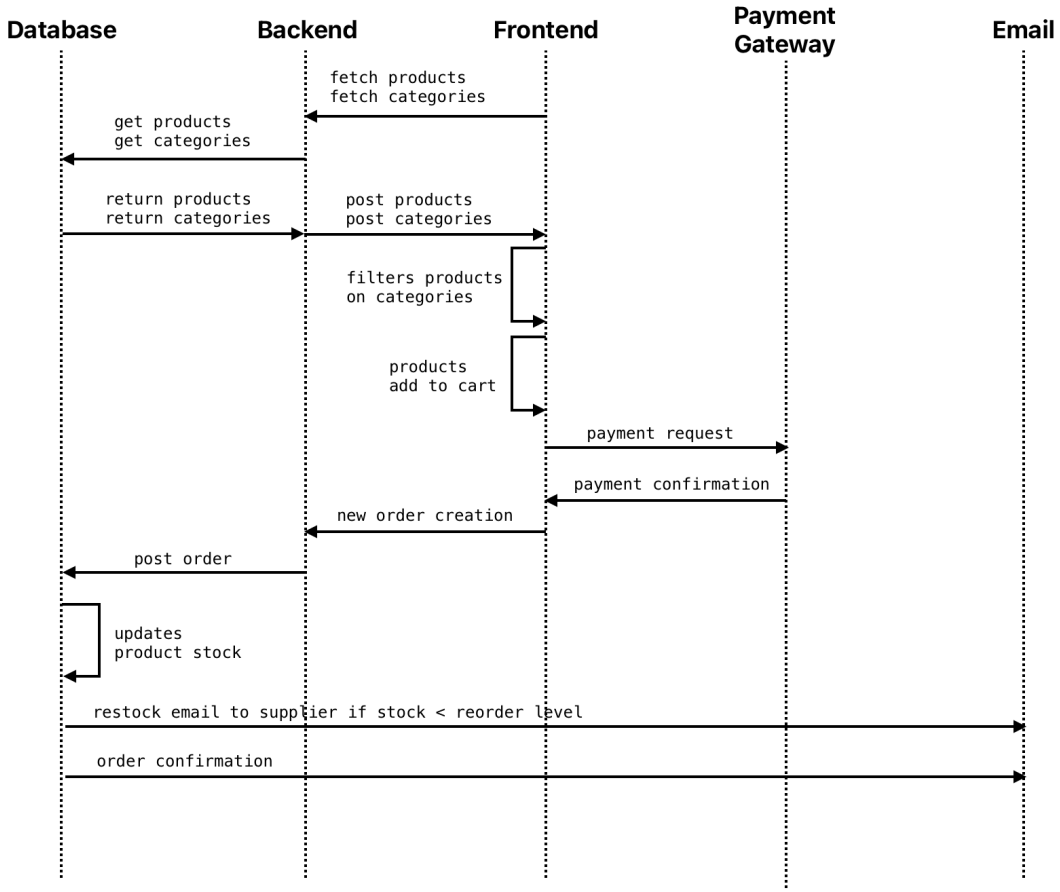


Figure 9: Runtime View Diagram

3.4 Deployment View

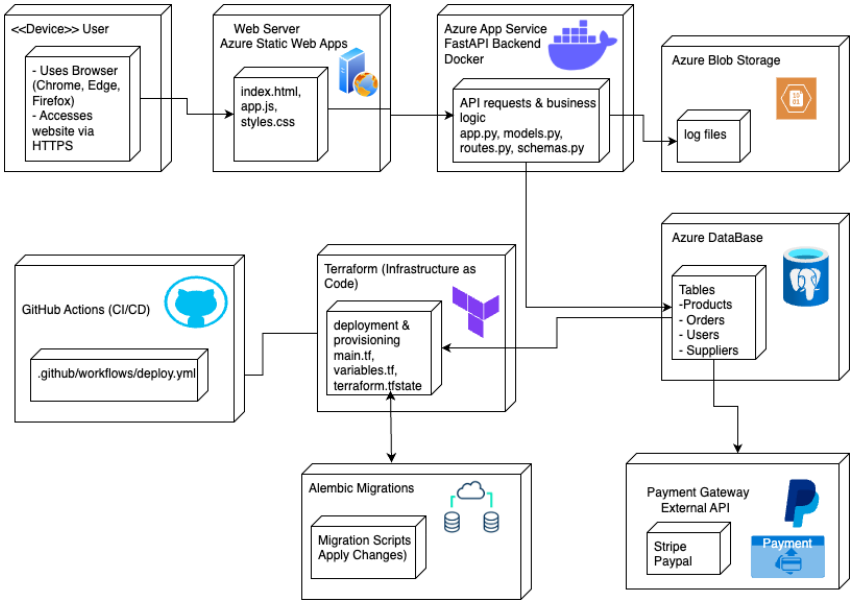


Figure 10: Deployment View Diagram

4 Implementation Process

To make the implementation easier and repeatable in the public cloud provider we used terraform. Each of the different services was encapsulated in a module, allowing us to make the main template readable. So far we did not setup any CI implementation for the infrastructure deploy, only for the code changes, since that the functionality was provided by Azure.

In the following code snippets we can see the specific infrastructure template we used for our deploy.

- Configuration of the azure terraform provider:

```
terraform {
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "=4.18.0"
    }
  }
}

provider "azurerm" {
  features {}

  client_id       = var.TF_AZURE_APP_ID
  client_secret   = var.TF_AZURE_PASSWORD
  tenant_id       = var.TF_AZURE_TENANT
  subscription_id = var.TF_AZURE_SUBSCRIPTION_ID

}
```

- Creation of a general resource group in Germany West Central (This zone is allowed by the student subscription)

```
resource "azurerm_resource_group" "main_rg" {
  name     = "lowtechgmbh-project-rg"
  location = "Germany West Central"
}
```

- Creation of a virtual network to comunicate the database and backend. This is required by the postgresql flexible server

```

resource "azurerm_virtual_network" "vn" {
  name                = "lowtechgmbh-vn"
  location            = azurerm_resource_group.main_rg.location
  resource_group_name = azurerm_resource_group.main_rg.name
  address_space       = ["10.0.0.0/16"]
}

resource "azurerm_subnet" "vn_subnet" {
  name                 = "lowtechgmbh-sn"
  resource_group_name  = azurerm_resource_group.main_rg.name
  virtual_network_name = azurerm_virtual_network.vn.name
  address_prefixes     = ["10.0.2.0/24"]
```

```
    service_endpoints    = ["Microsoft.Storage"]
    delegation {
      name = "fs"
      service_delegation {
        name = "Microsoft.DBforPostgreSQL/flexibleServers"
        actions = [
          "Microsoft.Network/virtualNetworks/subnets/join/action",
        ]
      }
    }
    depends_on = [azurerm_virtual_network.vn]
  }


  resource "azurerm_subnet" "vn_subnet_backend" {
    name                 = "lowtechgmbh-sn-backend"
    resource_group_name  = azurerm_resource_group.main_rg.name
    virtual_network_name = azurerm_virtual_network.vn.name
    address_prefixes     = ["10.0.3.0/24"]
    service_endpoints    = ["Microsoft.Storage"]
    delegation {
      name = "fs"
      service_delegation {
        name = "Microsoft.Web/serverFarms"
        actions = [
          "Microsoft.Network/virtualNetworks/subnets/action",
        ]
      }
    }
    depends_on = [azurerm_virtual_network.vn]
  }
```

– Creation of a private DNS zone for the database. This is required to get an internal DNS name avoiding the use of a single public IP.

```

  resource "azurerm_private_dns_zone" "private_dns_zone" {
    name                = "lowtechgmbh.postgres.database.azure.com"
    resource_group_name = azurerm_resource_group.main_rg.name
    depends_on          = [azurerm_resource_group.main_rg]
  }

  resource "azurerm_private_dns_zone_virtual_network_link" "vnet_link" {
    name                  = "lowtechgmbh.com"
    private_dns_zone_name = azurerm_private_dns_zone.private_dns_zone.name
    virtual_network_id    = azurerm_virtual_network.vn.id
    resource_group_name   = azurerm_resource_group.main_rg.name
    depends_on            = [azurerm_subnet.vn_subnet]
  }
```

– Creation of the DB, frontend and backend modules. All necesary code is included in the /infra folder of the repository.

```


  module "db" {
    source      = "./postgresql"
    username    = var.POSTGRESQL_USERNAME
    password    = var.POSTGRESQL_PASSWORD
    location    = azurerm_resource_group.main_rg.location
    rg_name     = azurerm_resource_group.main_rg.name
    dns_zone_id = azurerm_private_dns_zone.private_dns_zone.id
    subnet_id   = azurerm_subnet.vn_subnet.id
    depends_on  = [azurerm_private_dns_zone_virtual_network_link.vnet_link]
  }


  module "backend" {
    source                          = "./appservice"
    location                        = azurerm_resource_group.main_rg.location
    rg_name                         = azurerm_resource_group.main_rg.name
    subnet_id                       = azurerm_subnet.vn_subnet_backend.id
    db_url                          =
          "${var.POSTGRESQL_USERNAME}:${var.POSTGRESQL_PASSWORD}@${module.db.db_url}:5432"
    mail_from                       = var.MAIL_FROM
    mail_password                   = var.MAIL_PASSWORD
    mail_port                       = var.MAIL_PORT
    mail_server                     = var.MAIL_SERVER
    mail_username                   = var.MAIL_USERNAME
    scm_do_build_during_deployment = "1"
    depends_on                      = [module.db]

  }

  module "frontend" {
    source     = "./staticwebapp"
    depends_on = [module.backend]
  }
```

To get the appropriate credentials, we created a student account with 100USD in credits, after running the `az ad sp create-for-rbac --role="Contributor" --scopes="/subscriptions/<SUBSCRIPTION_ID>"` command the team was able to populate the respective variables file. Finally we executed the terraform plan to create the infrastructure.

Listing 1.1: Azure Credentials Example

```
{
    "appId": "XXXXX-XXXX-XXX-XXX-XXXXXXX",
    "displayName": "azure-cli-2025-02-09-15-07-35",
    "password": "SECRET_PASSWORD",
    "tenant": "TENANT_UUID"
}
```

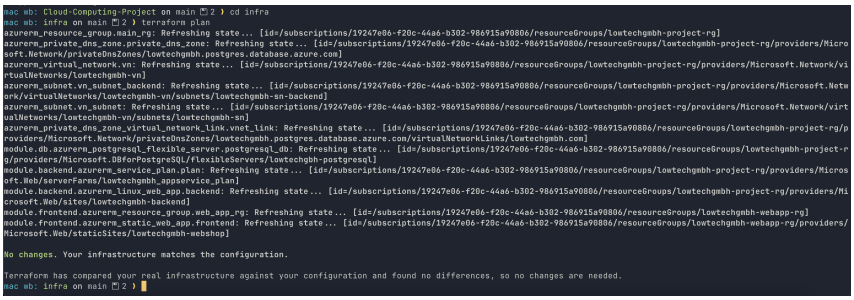


Figure 11: Terraform Plan Results

4.1 Cloud Environment Setup

Once we had all resources created in Azure, the team had to perform further manual configurations to enable the CI pipelines in azure webapps and static websites.

- To enable CI for the backend we used the provided azure functionality which created a GitHub actions file wich packages and uploads all changes once they are merged into main
- To enable CI for the frontend we used the provided azure functionality, which esentially provides the same features as in the backend.
- We also created an standalone virtual machine with access to the database, in this virtual machine we ran all migrations and seeded the database si all data was available for testing.

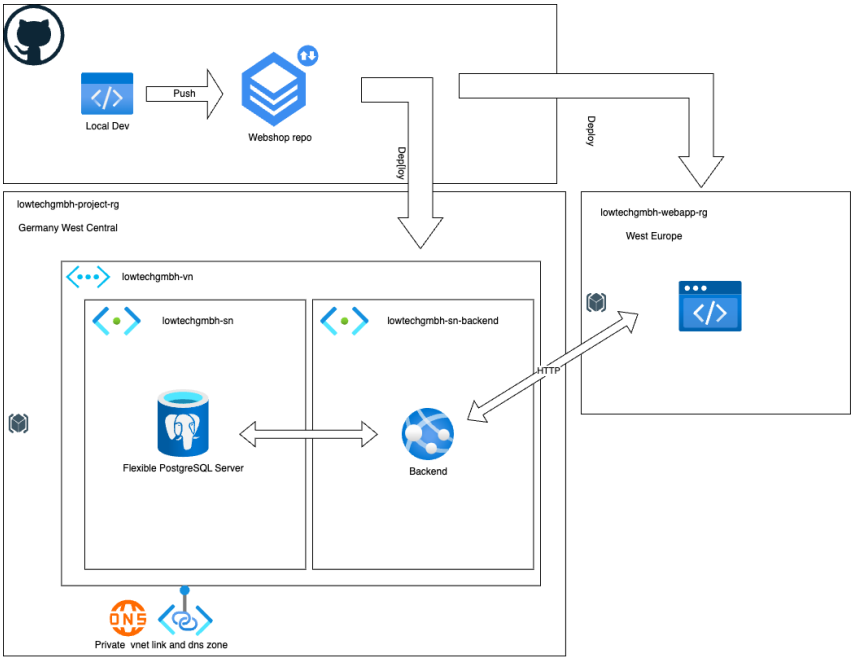


Figure 12: Cloud Environment Setup Diagram

4.2 Development Challenges

- Sometimes the azure terraform provider encountered race conditions and produced errors while creating or destroying the infrastructure.
- Some functionalities are not available for the student subscription, so we had to change availavility zones until we got one that was allowed

– By default our applications don't have any autoscaling configured, given the choico of SKUs and to save costs. Nevertheless the functionallity can be enabled by changing one configuration in the dashboard

5 Application Functionality and Characteristics

5.0.1 Functional Test Cases

Test Case ID	Test Scenario	Expected Outcome	Status (Pass/Fail)
TC-001	Load Homepage and verify product listing	Homepage loads with products displayed correctly.	Pass, see Figure 13
TC-002	Apply Price filter (Ascending/Descending)	Products reorder correctly based on selected price.	Pass, see Figure 13
TC-003	Apply Out of Stock filter	Only out-of-stock items are displayed.	Pass, see Figure 14
TC-004	Apply Fast Delivery filter	Only products eligible for fast delivery show up.	Pass, see demo video
TC-005	Filter by Category	Products are filtered correctly by selected category.	Pass, see demo video
TC-006	Search Product by Name or Category	Products matching search are shown correctly.	Pass, see demo video
TC-007	Clear Filter functionality	All filters are removed, showing the full product list.	Pass, see demo video
TC-008	Product Detail Page - Click Product	Clicking a product opens its detailed page.	Pass, see demo video
TC-009	Product Detail Page - Load Product Details	Product details (name, description, price) are displayed.	Pass, see demo video
TC-010	Add a Product to Cart	Product appears in cart with correct details.	Pass, see Figure 15
TC-011	Increase product quantity in cart	Quantity updates and is reflected in the cart.	Pass, see Figure 16
TC-012	Remove product from cart	Product is removed from the cart immediately.	Pass, see Figure 17
TC-013	Proceed to checkout	Checkout page loads with the correct order summary.	Pass, see demo video
TC-014	Select payment method - Stripe	Stripe payment option is selected and processed.	Pass, see Figure 21
TC-015	Select payment method - PayPal	PayPal payment option is selected and processed.	Pass, see Figure 18
TC-016	Complete order processing	Order confirmation message is displayed.	Pass, see demo video
TC-017	Order confirmation email is received	Email is sent after order is placed.	Pass, see Figure 21
TC-018	Shipment notification email is received	Email is sent when the order is shipped.	Pass, see Figure 19
TC-019	Product Dispatched email is received	Email is sent when the order is dispatched.	Pass, see Figure 20

Table 2: Functional Test Cases for Webshop

Screenshots of the webshop

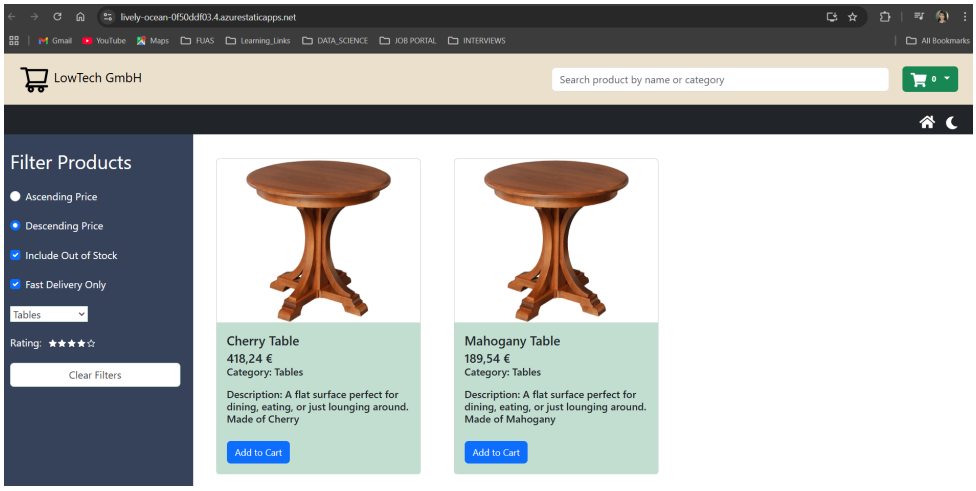


Figure 13: Homepage with All Filters Applied to Products

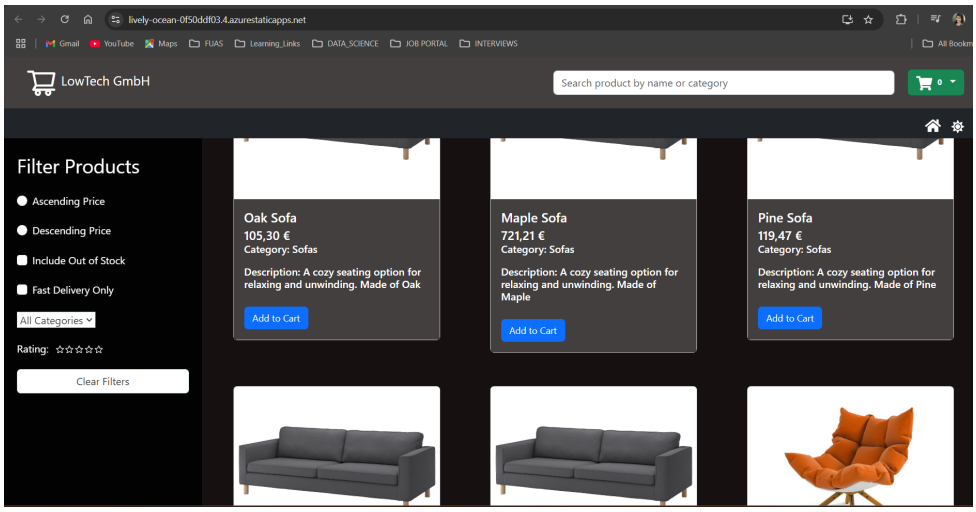


Figure 14: Homepage with All Products Listed, Dark Theme Enabled, and Clear Filter Applied

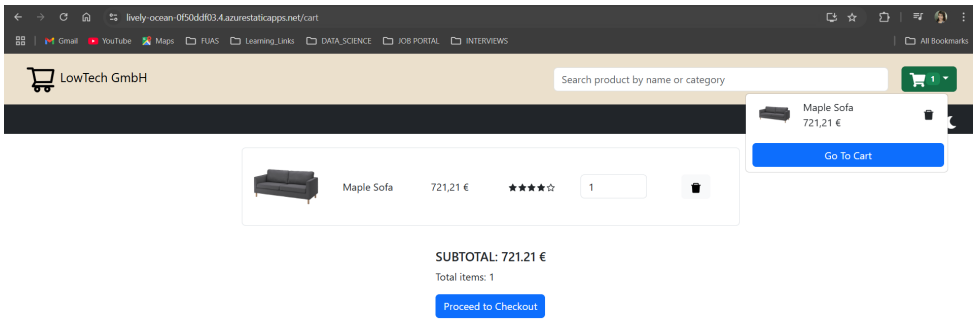


Figure 15: Product Added to Cart, Cart View Displayed with Selected Item

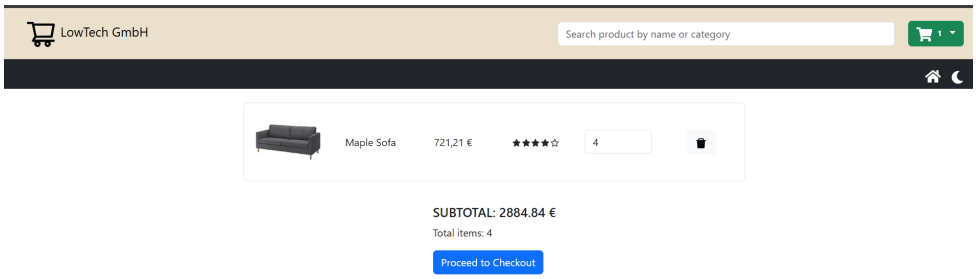


Figure 16: Product Quantity Updated in Cart

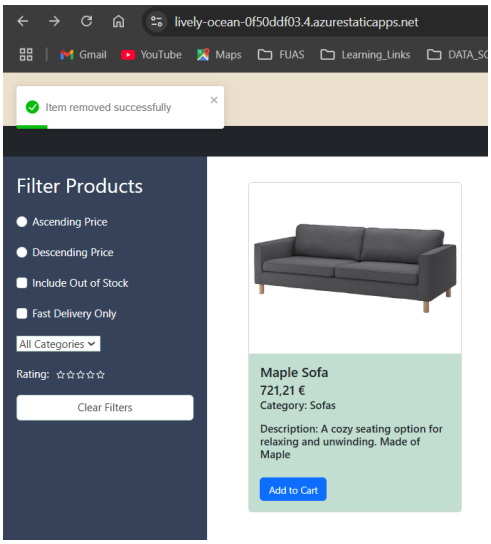
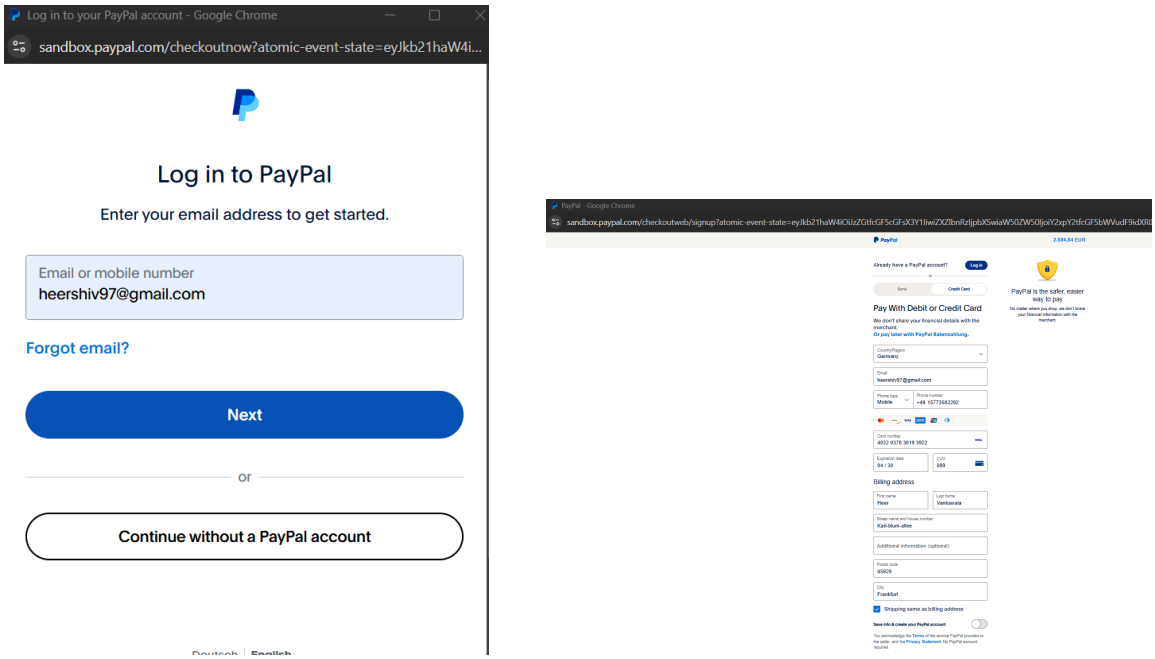


Figure 17: Product Removed from Cart Successfully



(a) Page Load (b) Credit Card Input

Figure 18: Paypal Payment Processing

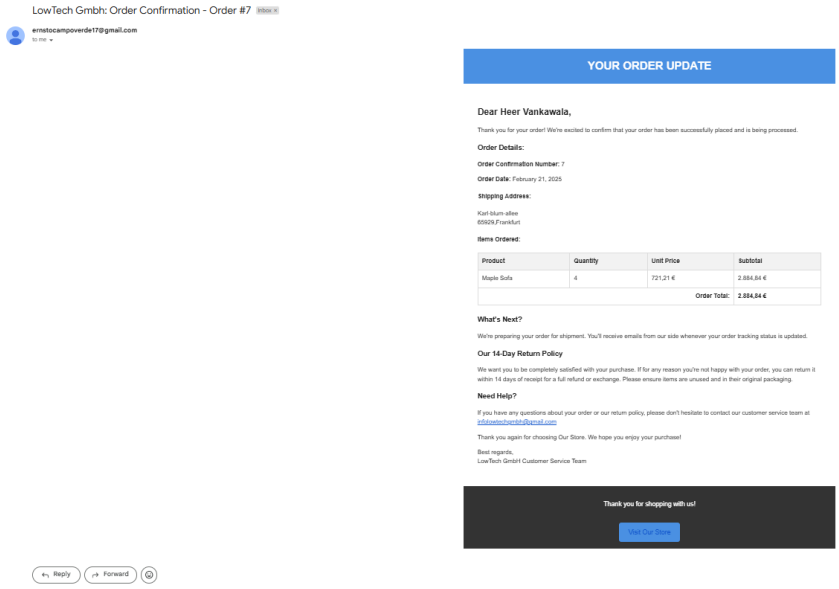


Figure 19: Shipment Notification Email Received for Order

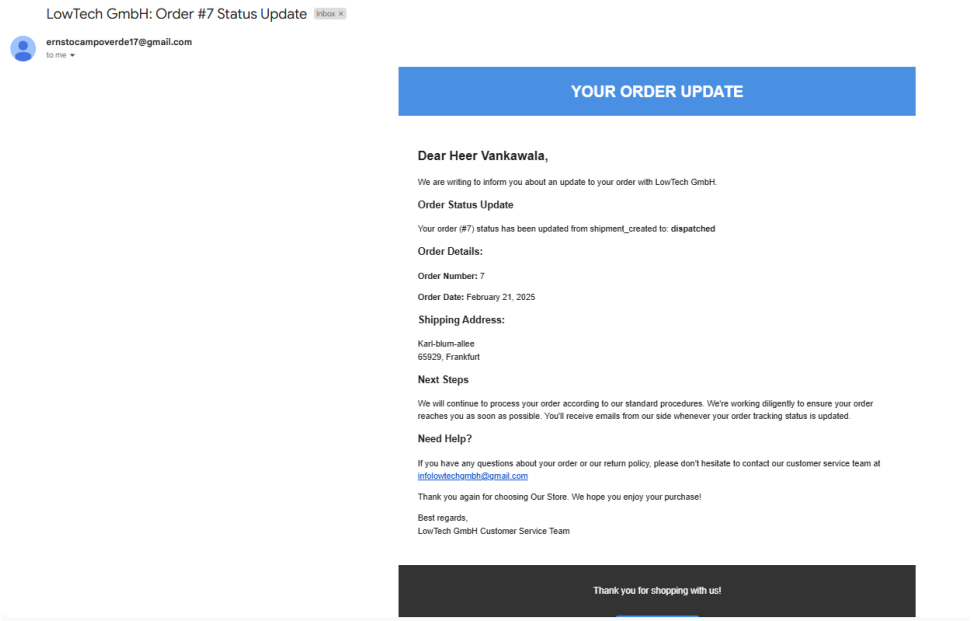


Figure 20: Order Dispatched Email Received After Status Change

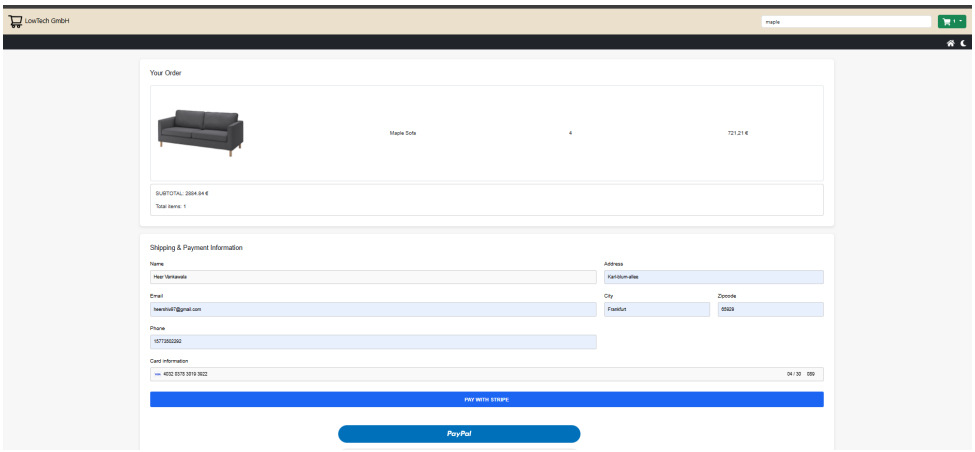


Figure 21: Order Placed with Stripe Payment Method

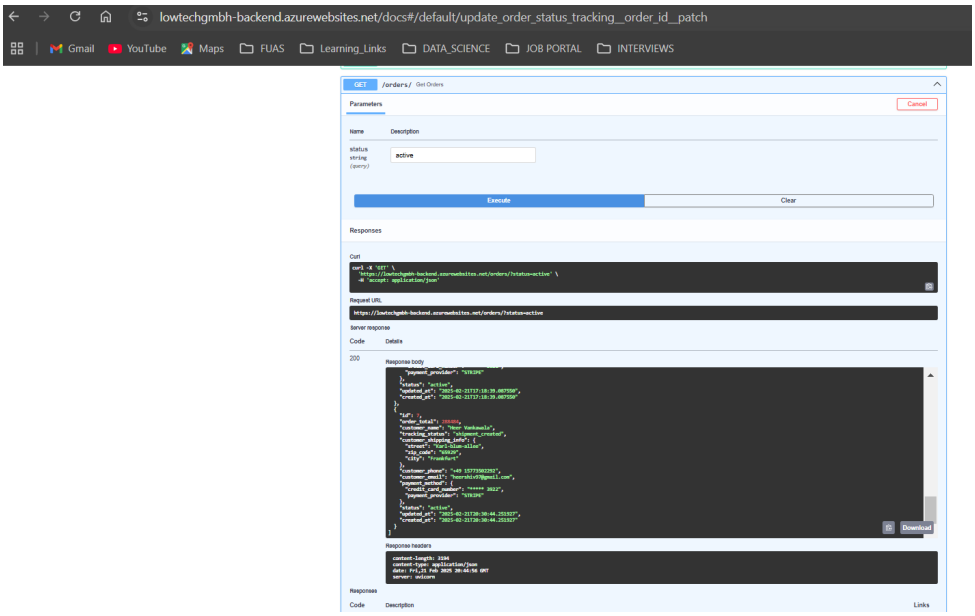


Figure 22: Order Data Retrieved Through Fast API for All Placed Orders

6 Critical Analysis

In general the application conforms to the specification of the client. Neverhteless, in hindsight, some areas of the code, infrastructure, and development process could have been improved.

- The base project for the front end used vanila JavaScript and no state management library. Initially to get the store up and running fast is perfect. But, once we need to add more functionality and the state management becomes more complex, we could have beneffited from the inclusion of a stronger type system. And a dedicated state management solution. Even on the first stages of integration, the lack of types makes it more difficult to integrate with the backend.
- The choice of 3 different services to run the application might have been more complex than what was needed. In theory either Azure App Service and Static Web apps could have easily handled the entirety of the application.
- Once we start scaling up, having three different services scaling independently even thouhg is more flexible, could result in bigger expenses, since all network traffic and DNS queries, even internally are billed. This is a situation that becomes less of a problem if all components are in a single service.

– The management of the database becomes more difficult once you host the server on a private virtual cloud. We had to create a virtual machine to act as a proxy for migrations and seeding the database.

6.1 Cloud Service Evaluation

In general the configuration of the services was simple enough. Nevbertheless there were some setbvacks associated to the type of subscription we were using. All services have auto-scaling capabilities, which can be enabled on demand, and configured acording to the aplications needs.

6.1.1 Azure Static Web Apps In this case we are using the Free SKU. The main disadvantage of the Free SKU is that is very limited in terms of throughput and connections to the client. Nevertheless this service can be upgraded to use ënterprise-grade egemmeaning that Azure will automatically cache the website on edge servers making it fast and easy to deliver to clients.

This service does not support traditional autoscaling since we are not rendering on the server. The client connects to the backend using http.

One of the disadvantages is thet even though it allows for integration options with APIs and even databases, it is clear that we will have to architect the application to only be tailored to use this services under Azyres paradigms.

Another disadvantage is that it requires its own resource group and own subnet.

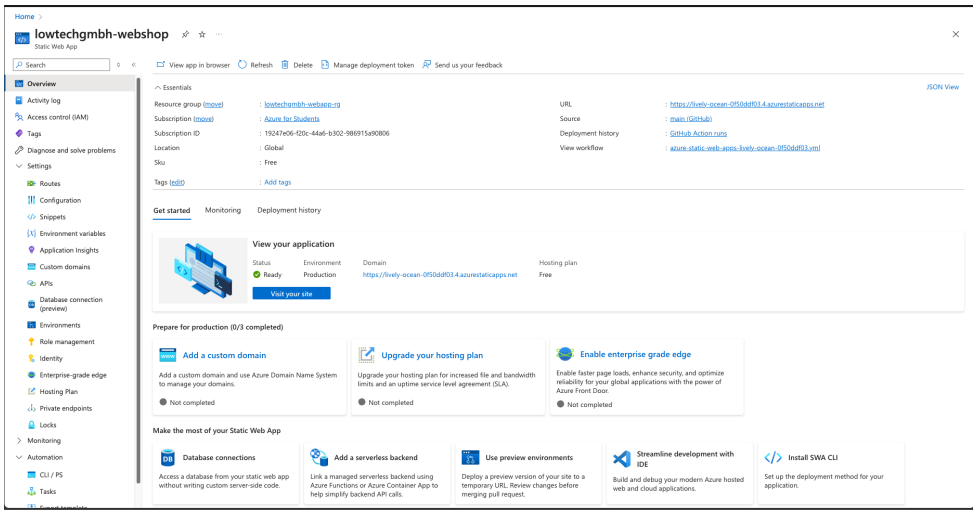


Figure 23: Azure Static Web Apps Dashboard

6.1.2 Azure App Service This service was used due to its versatility in deploying any type of web applications. To avoid extra costs associated to a private image registry, to handle deploys we are packaging all the code and artifacts in the github runner and publishing to our application.

One of the disadvantages is the price, once we start using the more advanced features and more compute, the service scales rapidly in cost.

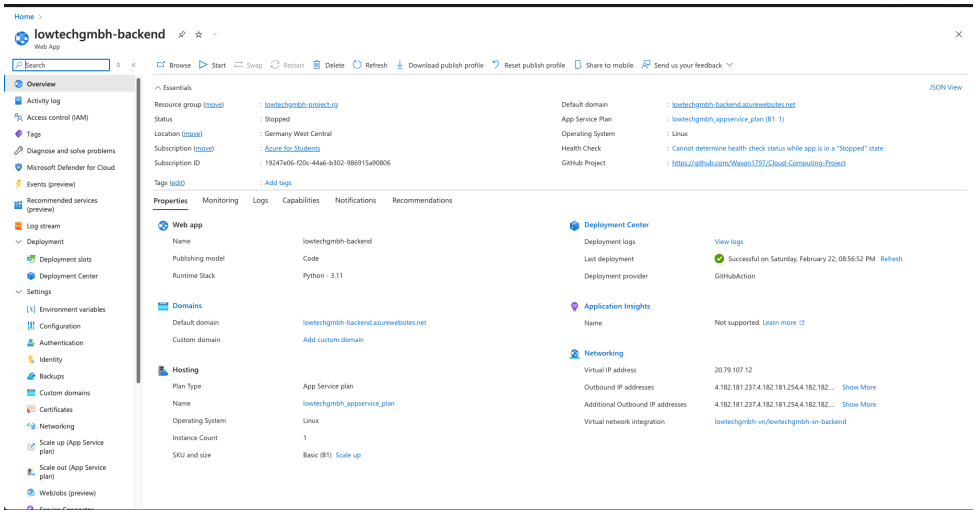


Figure 24: Azure App Service Dashboard

6.1.3 Azure PostgreSQL Flexible Server This is one of the more reasonable options to deploy a database server in Azure, given that the Free tier is quite capable. It also offers automatic maintenance and backups.

In this case, one of the disadvantages is the access complexity if we decide to host it on a VPC and not have a public IP.

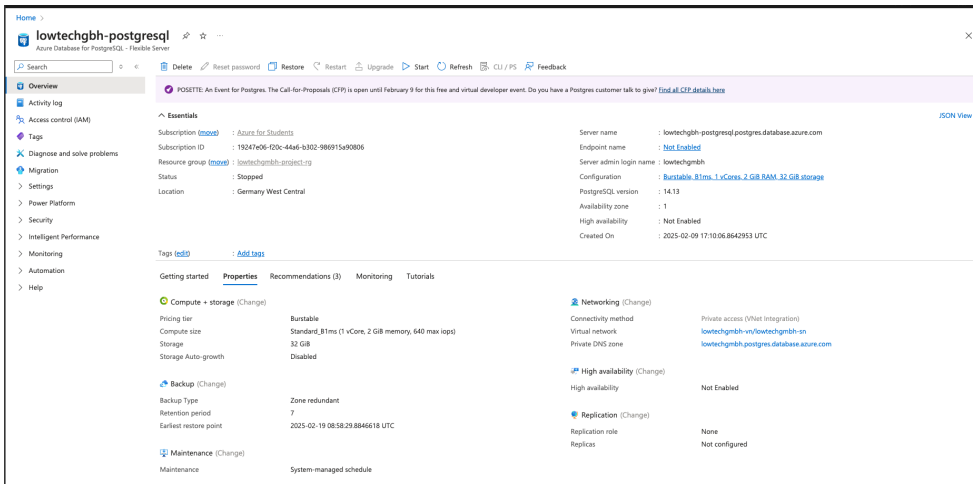


Figure 25: Azure PostgreSQL Flexible Server Dashboard

6.2 Architectural Decisions

- The frontend application uses a really basic component driven approach and segregates all API comunication logic to its own folder.
- In the backend we decided to have 2 simple layers, the database access layer, and the presentation layer i.e. The endpoints. Given the time constraints, and the scope of the project we did not divide the code further.
- We used FastAPI in the backend given its asynchronous nature, meaning a more efficient use of CPU resources, and the ability to schedule background tasks to send emails out of the box.
- In general this application is stil limited by the scale of the database, given that we are not using a microservice approach or have a database that supports sharding. Nevertheless, we feel that for a webshop which needs to handle orders, transaction integrity is more important than data replication. Furthermore, given the flexible nature of the cloud, we could scale the server vertically if there is the need.

7 Repository Documentation

7.1 GitHub Structure

The team configured the following repository: https://github.com/Helly2010/Cloud-Computing-Project.

1. **Branching Strategy:**
   The project follows a simple main branch strategy where all contributors work directly on the main branch. This ensures seamless integration without managing multiple branches. All changes should be committed with meaningful messages. Code should be reviewed and tested before pushing to the main branch.
2. **Continuous Integration/Continuous Deployment:**
   - **Continuous Deployment (CD):**
     Once the CI stage passes successfully, the application is automatically deployed to the appropriate environment. This process includes:
     - Deploying to a staging environment for final testing.
     - Running automated acceptance tests before production deployment.
     - Deploying to production with rollback mechanisms in case of failure.

7.2 Contributions

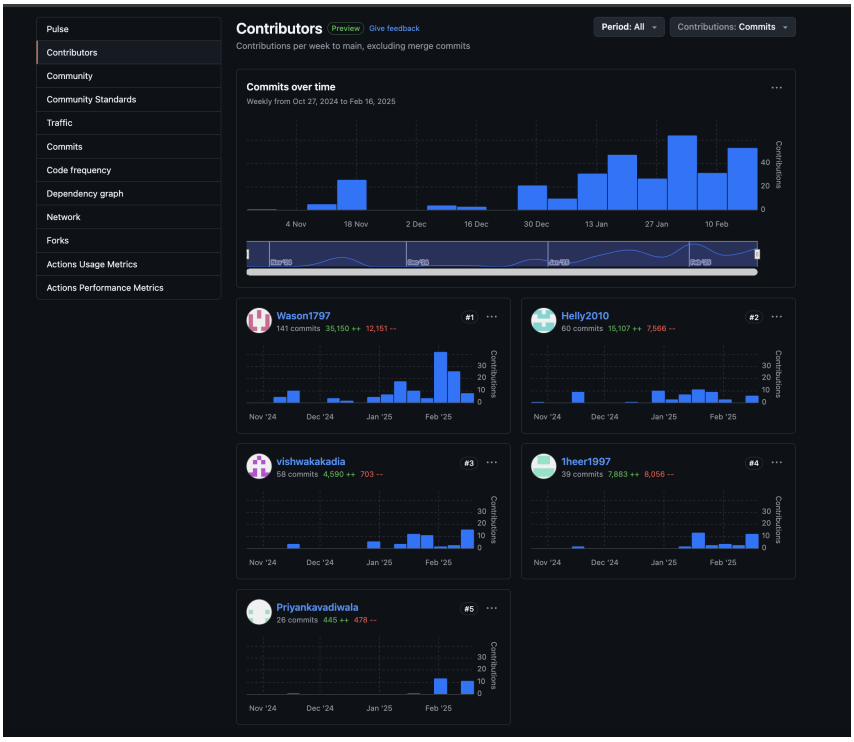


Figure 26: Contribution Stats in GitHub

8 Conclusion

8.1 Project Outcomes

At the end of the project we had a 3 tier cloud native application running. Furthermore, we have infrastructure templates to recreate this particular setup with minimal manual configuration. The team also

Topic	Contributor
Introduction and Objectives of Cloud Implementation Repository Documentation	Priyanka Vadiwala
Implementation Process And Critical Analysis	Wladymir Brborich-Herrera
Application Functionality and Characteristics Presentation slides And Project Demo	Heer Vankawala
System Diagrams	Hellyben Shah
Application Design And System Diagrams	Vishwaben Kakadiya

Table 3: Contribution Table

implemented a database migration system to track changes to the tables and to actually seed the initial data without much effort. All services have the possibility to scale horizontally, and are easily configurable. Nevertheless the architecture of the project could have been much simpler. As it was pointed out in the critical analysis of our project, services like Azure static web apps offer the possibility to deploy all 3 layers in one single service. Some of them even include their own MySQL database. We believe that it could be a more cost effective way, but at the cost of flexibility. Since all the application will be tailored to a single service.

8.2 Future Enhancements

For production readiness a lot of polish work is needed, specially in the order handling and tracking, not to mention security and user management. Nevertheless, in the infrastructure side, it is a matter of changing the SKUs and autoscaling configurations to something more appropriate depending on the expected load. Inherently the system runs in a private network inside the cloud provider, and it provides https certificates out of the box. If in the future the cost assesment reveals that the choice of servies was on the expensive side, we could at least consolidate the presentation and business logic layers in one service.

References

1. Microsoft, *Azure App Service plan overview*, 2024, Available: `https://learn.microsoft.com/en-us/azure/app-service/overview-hosting-plans`.
2. Microsoft, *Azure subscription and service limits, quotas, and constraints*, 2024, Available: `https://learn.microsoft.com/en-us/azure/azure-resource-manager/management/azure-subscription-service-limits`.
3. Microsoft, Available: *Azure Pricing Calculator*, 2024, `https://azure.microsoft.com/de-de/pricing/calculator/`.
4. Microsoft, *Configure Linux Python apps - Azure App Service*, 2024, Available: `https://learn.microsoft.com/en-us/azure/app-service/configure-language-python`
5. Microsoft, *What is Azure Static Web Apps?*, 2024, Available: `https://learn.microsoft.com/en-us/azure/static-web-apps/overview`.
6. Microsoft, *Azure Database for PostgreSQL – Flexible Servers*, 2024, Available: `https://learn.microsoft.com/de-de/azure/postgresql/flexible-server/overview`.
7. Microsoft, *What are Virtual Machine Scale Sets?*, 2024, Available: `https://learn.microsoft.com/en-us/azure/virtual-machine-scale-sets/overview`.