**ThAmCo System Documentation**

**Student Name:** [Your Name]**Student ID:** [Your ID]**Date:** [Current Date]

**1. Introduction**

This document details the design, implementation, and deployment of a cloud-capable software system for ThreeAmigos Corp, named the ThAmCo System. The system aims to facilitate the purchase of products from third-party suppliers and their resale to registered customers via a web application. This assessment focuses on key aspects of the system's architecture, implementation, and the DevOps practices employed, as developed iteratively using an agile methodology.The system addresses the functional requirements for public users, registered customers, and staff, with a strong emphasis on automated testing, configuration management, security, resilience, and a robust DevOps workflow.

**2. System Architecture**

The ThAmCo System is designed as a distributed, multi-tiered application to ensure scalability and maintainability.

**2.1. System Context Diagram (Conceptual Description)**

The C4 Context Diagram provides a high-level view of the ThAmCo System, illustrating the interactions between external users (Public, Customers, Staff, Management) and the system. It also shows the connections to external systems such as the Email Service, Supplier APIs, and Payment Gateway.

**A diagram of a software system

AI-generated content may be incorrect.**

This diagram helps in understanding the overall system boundaries and external dependencies. The diagram is essential for visualizing how different components interact and the flow of data across the system. It highlights the importance of secure communication channels, such as HTTPS, to protect sensitive information during transmission.

**2.2. Container Diagram**

This diagram illustrates a simplified architecture of the ThAmCo system, highlighting how the frontend, backend, and external services communicate with each other.

A diagram of a web api

AI-generated content may be incorrect.

On the left, the Frontend Web Application is built using React and serves three types of users: the public, registered customers, and staff members. This frontend sends secure HTTPS requests to the Backend Web API, which is built using ASP.NET Core 8 and Entity Framework Core.

At the center, the Backend Web API handles core business logic and data access. It interacts with the ThAmCo Database, hosted on Azure SQL, to store and retrieve data. This communication likely uses a database connection (via EF Core), while SMTP is used to send emails through a mocked or faked Email Service.

To the right, the backend also integrates with Third-Party Supplier Services, which are currently faked for development purposes. These services are accessed using REST-like APIs, and connection strings are likely stored in the database or configuration files.

In summary, the diagram represents a typical client-server architecture where users access a React frontend that communicates with a secure .NET backend, which in turn interacts with databases, email services, and external APIs to complete its operations.

**2.3. Deployment Diagram (Conceptual Description)**

This diagram represents the architecture of the ThAmCo system deployed on Microsoft Azure, showing how various components interact to deliver and support the application’s functionality. At the core is the ThAmCo Web API, built with ASP.NET Core 8.0, which handles client requests and communicates with the backend. Alongside it are Background Services developed with .NET Worker Services that manage behind-the-scenes operations such as scheduled tasks or data processing. Both the Web API and Background Services use Entity Framework Core to interact with the Azure SQL Server, which stores the system’s data.

**A diagram of a software system

AI-generated content may be incorrect.**

Security-sensitive information such as database credentials is managed by Azure Key Vault, ensuring secure access to secrets and connection strings. The system also includes Application Insights, which provides telemetry and monitoring for performance and reliability. It collects logs and metrics and may trigger notifications or other services, such as the Email Service and Payment Gateway, for alerting or transaction processing.

Additionally, the backend connects to Supplier APIs to fetch or update product-related data. There's also an integration with Datamato Services, which appears to handle telemetry or data logging tasks. Overall, this architecture supports a modern, secure, and scalable cloud application environment using best practices in cloud deployment and microservice communication.

A screenshot of a computer

AI-generated content may be incorrect.

The screenshot showing the Azure SQL Database creation process (e.g., Create SQL Database, thamcosy | Networking, Connection strings) illustrates the cloud deployment aspect of the database. The appsettings.json screenshot shows the connection string being used to connect to the Azure SQL Database.

**3. System Implementation**

The implementation focused on delivering core functionalities across different user roles, ensuring testability, security, and maintainability.

**3.1. Backend (src/ThAmCo.WebApi)**

* **Technologies**: ASP.NET Core 8 Web API, Entity Framework Core for ORM, SQL Server as the database provider. Dependency Injection is used extensively.
* **API Endpoints and Services**:
* **Authentication & Authorization (AuthController.cs, AuthService.cs)**:
* /api/auth/register-customer: Public registration.
* /api/auth/login: Customer login, issues JWT.
* /api/auth/staff-login: Staff login, issues JWT with staff role.
* /api/auth/refresh-token: Refreshes JWT.
* Implemented role-based authorization ([Authorize(Roles = "Staff,Administrator")]).
* **Customer Management (CustomersController.cs, CustomerService.cs)**:
* /api/customers/profile (GET, PUT): View and update customer profile.
* /api/customers/funds (GET): View customer account funds.
* /api/customers/account (DELETE): Request account deletion.
* /api/customers (GET, Staff/Admin): Get all customers.
* /api/customers/{id} (GET, Staff/Admin): Get specific customer details.
* /api/customers/add-funds (POST, Staff/Admin): Add funds to a customer's account.
* /api/customers/{id} (DELETE, Staff/Admin): Delete (anonymize) a customer account.
* **Order Management (OrdersController.cs, OrderService.cs)**:
* /api/orders (POST): Create a new order (customer).
* /api/orders/history (GET): Get customer's order history.
* /api/orders/{id} (GET): Get specific order details.
* /api/orders/pending-dispatch (GET, Staff/Admin): Get orders awaiting dispatch.
* /api/orders/{id}/dispatch (PUT, Staff/Admin): Mark an order as dispatched.
* /api/orders (GET, Staff/Admin): Get all orders.
* **Product Management (ProductsController.cs, ProductService.cs)**:
* /api/products (GET): Browse and filter products.
* /api/products/{id} (GET): Get product details.
* /api/products/search (GET): Search products.
* /api/products/categories (GET): Get available product categories.
* **Supplier Integration (SupplierService.cs)**: Mocked for purchasing from suppliers and updating stock/prices.
* **Email Service (EmailService.cs)**: Mocked for sending order confirmations/status updates.
* **Database Interactions**: Utilizes Entity Framework Core for all database operations, including migrations for schema management and seeding initial data (ThAmCoContext.cs).
* **Error Handling**: Centralized error handling to ensure consistent JSON responses for API errors (e.g., ErrorResponseDto for "Insufficient funds" and other bad requests).
* **Configuration Management**: appsettings.json is used for application configuration, including database connection strings, JWT settings, and email settings. Separate appsettings.

A screenshot of a computer program

AI-generated content may be incorrect.

* Development.json and appsettings.Production.json are used for environment-specific settings.

**3.2. Frontend (frontend)**

* **Technologies**: React.js, Material-UI for UI components, Vite for development build.
* **Key Components**:
* src/App.jsx: Main application routing and layout.
* src/AuthContext.jsx: Manages user authentication state and JWT tokens, including login/logout and user data updates.
* src/config.js: Centralized API base URL configuration.
* src/components/Auth/Login.jsx, Register.jsx: User authentication forms.
* src/components/Customer/Profile.jsx: Customer profile viewing and updating, including PaymentAddress, DeliveryAddress, and PhoneNumber. The updateUser function ensures the AuthContext user object is refreshed after profile updates.
* src/components/Customer/CreateOrder.jsx: Logic for creating orders, including checks for delivery address and phone number.
* src/components/Customer/OrderHistory.jsx: Displays customer's past orders.
* src/components/Staff/StaffOrders.jsx: Staff view and dispatch orders.
* src/components/Staff/StaffCustomers.jsx: Staff view, delete, and add funds to customer accounts.
* src/components/PrivateRoute.jsx, StaffPrivateRoute.jsx: Route guards for authorization.
* **UI Design**: Leverages Material-UI for a modern, responsive, and consistent user interface. ThemeProvider and CssBaseline are used for global styling.
* **Connectivity**: Uses fetch API to interact with the Backend Web API, configured via API\_BASE\_URL from src/config.js.
* **Product Display**: Products are fetched and displayed, with filtering by category and search functionality.

A screen shot of a computer

AI-generated content may be incorrect.

**4. DevOps Workflow and Practices**

The development process employed several DevOps practices to ensure efficient development, testing, and deployment.

* **Source Control**: Git is used for version control, with frequent commits to track changes and facilitate collaboration.
* **Automated Building and Testing**:
* **Backend**: dotnet test is used for automated unit and integration testing of the backend services. Tests were created for CustomerService and OrderService (e.g., src/ThAmCo.WebApi.Tests/Services/CustomerServiceTests.cs, OrderServiceTests.cs). Mocks and test doubles (Moq) were used to isolate units under test.
* **Frontend**: npm run dev for local development, and build processes (e.g., npm run build) for production deployments.
* **Configuration Management**:
* appsettings.json (backend) and config.js (frontend) are used to manage environment-specific configurations (e.g., database connection strings, API base URLs). This allows for easy switching between development, testing, and production environments without code changes.
* The appsettings.json screenshot demonstrates how the connection string is managed.
* **Deployment (Local Iteration)**:
* Backend: dotnet run --project src/ThAmCo.WebApi/ThAmCo.WebApi.csproj is used to launch the backend. Commands were executed in the background using is\_background=True parameter of the run\_terminal\_cmd tool.
* Frontend: npm run dev in the frontend directory launches the React development server.
* **Challenges and Solutions**:
* **PowerShell Syntax**: Initial attempts to chain dotnet commands using && failed due to PowerShell's syntax. This was resolved by using the ; operator for sequential command execution (e.g., dotnet clean ... ; dotnet restore ... ; dotnet run ...).
* **Dynamic Ports**: The backend initially launched on dynamic ports, requiring manual updates to the frontend's API base URL. This was later addressed by explicitly setting the backend to run on http://localhost:5000 (though sometimes dynamic ports still occurred). The src/frontend/src/config.js file was crucial for centralized API URL management.
* **Database Sync Issues**: Multiple database resets (removing migrations, dropping the database, creating new migrations, updating the database) were necessary to resolve schema mismatches after model changes and seed data additions (e.g., Staff.cs password, Product.cs ImageUrl, ThAmCoContext.cs seed data). This highlighted the importance of a well-defined database migration strategy in a true DevOps pipeline.
* **Frontend-Backend Communication**: Errors like "Unexpected token 'I', 'Invalid em'... is not valid JSON" were resolved by ensuring the backend consistently returned structured JSON error responses (using ErrorResponseDto). The "Failed to fetch" errors due to incorrect API URLs were resolved by centralizing the API\_BASE\_URL in src/frontend/src/config.js and updating all frontend components.

**5. Narrative: Consequences of Technology Choices**

The selection of ASP.NET Core for the backend and React with Material-UI for the frontend was driven by the need for a modern, scalable, and maintainable enterprise-style system, as implied by the ICA scenario.

* **ASP.NET Core (Backend)**:
* **Pros**: Strong typing (C#), robust framework, excellent performance, built-in dependency injection, good ecosystem for API development, and strong integration with SQL Server and Entity Framework Core. This choice aligns well with the need for a well-engineered and growing system.
* **Cons/Consequences**: C# compilation errors can be verbose and sometimes tricky to debug, especially when dealing with interface mismatches or missing references. The strictness, while beneficial for long-term stability, can slow down rapid prototyping if not managed carefully. The repeated compilation errors regarding ICustomerService definitions, despite seemingly correct code, highlighted the importance of understanding the build process and potential caching issues.
* **React with Material-UI (Frontend)**:
* **Pros**: React provides a component-based architecture for building complex UIs, promoting reusability and maintainability. Material-UI offers a comprehensive set of pre-designed, accessible UI components, accelerating development and ensuring a good user experience with minimal custom CSS. Vite offers a fast development server and build process.
* **Cons/Consequences**: The asynchronous nature of API calls in React can lead to "Failed to fetch" errors if API URLs are incorrect or if backend responses are not as expected (e.g., non-JSON errors). Debugging frontend state management (like the AuthContext not refreshing customer data after profile updates) required careful tracing and explicit updateUser calls. The dynamic nature of localhost ports during backend restarts also introduced friction, necessitating a centralized configuration for the API base URL.
* **Azure SQL Database**:
* **Pros**: A managed cloud database service offers high availability, scalability, and built-in security features, reducing operational overhead. It aligns with the "cloud-capable" requirement.
* **Cons/Consequences**: Database schema changes (migrations) can be complex and error-prone, especially during active development. The need for multiple database resets (dropping and recreating) due to schema mismatches and seed data changes underscored the importance of careful migration management in a CI/CD pipeline. Also, managing firewall rules for development access (as shown in the Azure networking image) can be an extra step.
* **DevOps Workflow (Challenges)**: The continuous integration and deployment (CI/CD) aspect was iterative.
* **PowerShell vs. Bash**: The initial struggle with && vs. ; for command chaining highlighted the need for environment-specific scripting knowledge in cross-platform development. This was a direct consequence of working in a Windows environment with PowerShell, which differs from common Linux/Bash CI environments.
* **Automated Testing**: While implemented, ensuring comprehensive test coverage and continuous integration of tests remains an ongoing process, crucial for system resilience.

In retrospect, the primary challenges encountered were often related to the interplay between different components (frontend to backend communication), environment-specific scripting, and maintaining database schema consistency during rapid development. These experiences reinforced the value of robust error handling, centralized configuration, and a well-defined, automated CI/CD pipeline to mitigate such issues in a production environment. The use of AI as a companion proved invaluable for quickly identifying and resolving coding problems, especially with syntax and API usage, which directly contributed to problem-solving during development.

**6. Demonstration Media (References)**

Please refer to the following screenshots provided separately as part of the submission:

* **Azure SQL Database Creation**: Image showing the steps to create the Azure SQL Database, including resource group, database details, server, and networking settings.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

* **Azure SQL Database Connection Strings**: Image displaying the ADO.NET connection strings for the Azure SQL Database.
* A screenshot of a computer

  AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

* **appsettings.json Configuration**: Screenshot of the appsettings.json file, showing ConnectionStrings, JwtSettings, EmailSettings, and SupplierSettings.
* **Product Listing (Frontend)**: Screenshot of the frontend displaying products, their prices, and stock quantities, with quantity selectors. This demonstrates product browsing and the "Create New Order" functionality.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

* **localhost:5000/api/products JSON Response**: Screenshot showing the raw JSON response from the backend /api/products endpoint, confirming product data structure.
* **Customer Login Page**: Screenshot of the frontend customer login interface, including the "Staff Login" toggle.

A screenshot of a login screen

AI-generated content may be incorrect.

* **Staff Login Page**: Screenshot of the frontend staff login interface.
* A screenshot of a login screen

  AI-generated content may be incorrect.
* **Sign Up Page**: Screenshot of the frontend registration form.

A screenshot of a login form

AI-generated content may be incorrect.