D780 - Task 2 - Attempt 1

Silver Alcid - #011933973

9/15/2025

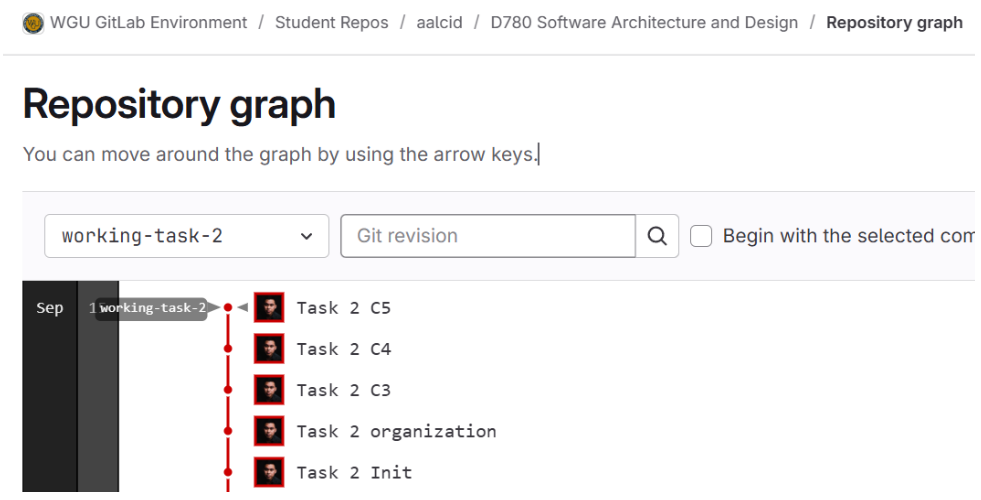
MBN1 Task 2: Software Architecture

A3. Submit a copy of the GitLab repository URL in the "Comments to Evaluator" section when you submit this assessment.

https://gitlab.com/wgu-gitlab-environment/student-repos/aalcid/d780-software-architecture-and-design/-/tree/2a99b432300a9a9e5c5cb28d4162c3d5d9318664/

A4. Commit with a message and push when you complete each requirement listed in parts C3 and C4.

https://gitlab.com/wgu-gitlab-environment/student-repos/aalcid/d780-software-architecture-and-design/-/network/working-task-2?ref_type=heads


WGU GitLab Environment / Student Repos / aalcid / D780 Software Architecture and Design / Repository graph
Repository graph
You can move around the graph by using the arrow keys.
working-task-2
Git revision
Begin with the selected com
Sep
1
working-task-2

```
Task 2 C5
Task 2 C4
Task 2 C3
Task 2 organization
Task 2 Init
```

B1. Identify the most appropriate architectural pattern from the list below for each of the three business scenarios provided in the attached "Business Scenarios" document and justify your selection.

Retail Environment

A microservices architecture is the best fit for the retail scenario. The system must support authentication, inventory, order processing, fulfillment, shipping, and notifications. Demand is high in fall and winter with additional spikes during Christmas. The business also changes suppliers and shipping vendors often and needs fast updates and redeployments. Independent services mapped to these domains allow selective scaling during peak periods, faster changes to vendor adapters, and smaller blast radius when failures occur.

Ticketing for Events

Microservices are also the most appropriate for ticketing. The system must handle authentication, search, event viewing, and seat booking under large, spiky traffic when new venues are announced. Users should always see event details and search should always return quickly. There must be no double booking. The business requires high reliability, global availability, and strong security. Splitting into focused services, such as search and indexing, event content, and a reservation service with strict concurrency controls, enables targeted scaling, specialized storage, and fault isolation.

Order Processing

An event driven architecture is the best fit for order processing. Each step is triggered by the previous step. A user creates an order, which triggers payment. A successful payment triggers fulfillment from inventory. Fulfillment triggers shipping. The customer must be notified at creation, payment, and shipping. Modeling these transitions as events decouples services, supports asynchronous work, improves resilience through retries and idempotency, and makes it easy to add new consumers such as fraud checks or analytics without changes to core logic.

B2. For each of the three business scenarios, discuss whether the software solution should be implemented in the cloud or on-premises. Support your discussion with evidence from the business requirements provided in the attached "Business Scenarios" document.

Cloud deployment is recommended across the board for all three business scenarios.

Retail Environment

- Cloud is recommended because demand is seasonal with peaks in fall, winter, and during Christmas.
- Cloud elasticity supports scaling up for high traffic and scaling down in low seasons.
- Frequent changes to suppliers and shipping vendors require fast redeployment, which the cloud supports.
- Authentication, inventory, order processing, fulfillment, shipping, and notifications map well to cloud services.

Ticketing for Events

- Cloud is recommended because traffic is global and spikes when new events are announced.
- Users must always see event details, receive fast search results, and avoid double booking.
- The business requires high reliability, global 24/7 availability, and strong security.
- Cloud offers distributed infrastructure and security controls that meet these requirements.

Order Processing

- Cloud is recommended because the system depends on sequential triggers across order creation, payment, fulfillment, and shipping.
- Event-driven services in the cloud handle asynchronous processing and recovery.
- Independent scaling of payment, inventory, and shipping functions is supported in cloud environments.
- Cloud simplifies notifications and monitoring across each stage of the process.

C1. Explain how the code provided in the attached "Monolithic Retail System" document can be identified as a monolithic software architecture.

- All components are defined in one module: Cart and CartFactory, Inventory and its observer, payment strategies and processor, and the integrating RetailSystem. This indicates one executable unit rather than separately deployable services.

- The RetailSystem creates class instances and invokes their methods in memory. There are no network calls, service endpoints, or inter-process communication, which is typical for a monolith.
- Business logic crosses component boundaries through direct access to internal data. For example, RetailSystem.add_item_to_cart reads and writes inventory.stock directly, which couples cart logic to inventory internals.
- The RetailSystem class coordinates cart operations, inventory updates, and payment processing within the same runtime. There is no separate orchestrator service or API layer.
- The script instantiates RetailSystem and runs the workflow directly at the bottom of the file, which reinforces the single-package, single-process model.
- Components are classes in one program with shared memory and no external interfaces, they cannot be scaled or deployed independently. This is a core characteristic of a monolith, reflected by the lack of separate processes or service contracts in the code.

C2. Explain why a microservices-based architecture is an improvement over a monolithic architecture in each of the following areas: scalability, flexibility, and maintainability.

Scalability

Microservices improve scalability because each service can scale on its own. Components with heavy load such as checkout or inventory can run more instances without affecting others. This allows better use of compute and memory and reduces cost during low demand. It also reduces contention inside a single process since traffic is spread across many service processes.

In a monolith the entire application must scale together, which wastes resources and creates bottlenecks in shared code and shared data access.

Flexibility

Flexibility is improved by separating responsibilities into clear, deployable units. Each team can change code, upgrade libraries, and deploy on its own schedule. Services can use data stores and frameworks that fit their needs without forcing the entire system to adopt the same choice. Integrations with vendors can be added or replaced inside a single service boundary.

In a monolithic codebase, one change often requires a full application redeploy, and shared dependencies make technology choices harder to modify.

Maintainability

Maintainability is easier through smaller codebases with clear interfaces. Problems are easier to locate because logs, metrics, and failures are tied to a single service. Faults are contained, which reduces the blast radius and simplifies rollback. Tests run faster and continuous delivery pipelines are simpler because they target a limited scope.

As opposed to a monolith codebase where it grows in size and coupling, making changes harder to reason about and increasing the risk that one change breaks unrelated features.

C3. Refactor the monolithic components (i.e., cart, inventory, payment) into independent microservices such that each service (i.e., cart_service.py, inventory_service.py, payment_service.py) functions independently and handles its specific responsibility.

See GitLab repository.

C4. Correctly implement the orchestrator_service.py to manage communication between microservices.

See GitLab repository.

C5. Create a README file that includes the following: your name, your student ID, python version, a description of each function written in parts C3 and C4, and the steps to run each of the functions in parts C3 and C4

See GitLab repository.

D. Provide a Panopto video recording that includes a demonstration of the functionality of the code by executing and confirming successful output of each of the three test cases outlined in the attached "Test Cases" document.

https://wgu.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=988f4bf2-e1ca-4775-84a9-b35901011e98

E. Acknowledge sources, using in-text citations and references, for content that is quoted, paraphrased, or summarized.

No sources were needed or used.