

Week2-Architecture & Communication

[Start Assignment](#)

- Due Wednesday by 11:59pm
- Points 20
- Submitting a text entry box or a website url

Due Date: Feb 18, 2026, Group Submission

Goal

Implement the same “campus food ordering” workflow in three ways:

1. Synchronous REST
2. Async messaging with RabbitMQ
3. Streaming with Kafka

Each part includes: build, run, test, and failure injection.

Repo Structure

cmpe273-comm-models-lab/

common/

 README.md

 ids.py

sync-rest/

 docker-compose.yml

 order_service/

 inventory_service/

 notification_service/

 tests/

async-rabbitmq/

 docker-compose.yml

 order_service/

 inventory_service/

 notification_service/

 broker/

 tests/

streaming-kafka/

 docker-compose.yml

```
producer_order/  
inventory_consumer/  
analytics_consumer/  
tests/
```

Keep each part self-contained with its own `docker-compose.yml`

Part A: Synchronous (REST or gRPC)

Implement

- `POST /order` on OrderService
- OrderService calls Inventory synchronously: `POST /reserve`
- If reserve succeeds, OrderService calls Notification synchronously: `POST /send`

Testing requirements

- Baseline latency test (N requests)
- Inject 2s delay into Inventory and measure impact on Order latency
- Inject Inventory failure and show how OrderService handles it (timeout + error response)

What to submit

- Simple latency table, plus reasoning on why the behavior happens

Part B: Async (RabbitMQ or equivalent)

Implement

- OrderService writes order to local store and publishes `OrderPlaced`
- InventoryService consumes `OrderPlaced`, reserves, publishes `InventoryReserved` or `InventoryFailed`
- NotificationService consumes `InventoryReserved` and sends confirmation

Testing requirements

- Kill InventoryService for 60 seconds, keep publishing orders, then restart and show backlog drain
- Demonstrate idempotency:
 - Re-deliver the same `OrderPlaced` message twice and ensure inventory does not double reserve
- Show DLQ or poison message handling for malformed event

What to submit

- Screenshots or logs showing backlog, then recovery
- Short explanation of the idempotency strategy

Part C: Streaming (Kafka or equivalent)

Implement

- Producer publishes `OrderEvents` stream: `OrderPlaced`
- Inventory consumes and emits `InventoryEvents`
- Analytics consumes streams and computes:
 - orders per minute
 - failure rate
- Demonstrate replay:
 - reset consumer offset and recompute metrics

Testing requirements

- Produce 10k events
- Show consumer lag under throttling
- Show replay producing consistent metrics (or explain why not)

What to submit

- A small metrics output file or printed report
- Evidence of replay (before and after)