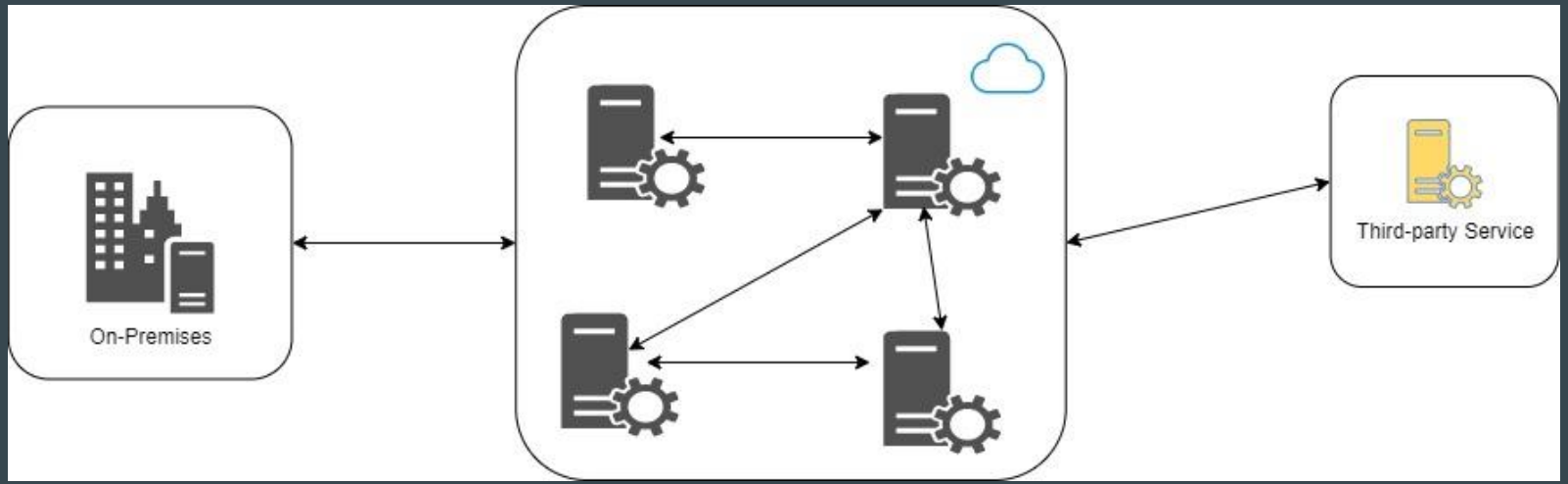
Cloud Applications Architecture

Course 10 - Integration Services

Integration

Integrate 2 or more systems.

Can be microservices, 3rd party APIs, or on-premises systems



Coupling

Degree of dependency between entities. I.e. how much entities must know about each other in order to function.

Strive for loose coupling (decoupled services/code).

Benefits

- Better maintenance and development/extensibility
- Higher cohesion
- More efficient work split (even among different teams)

Challenges

- Require additional tools/services
- Harder to plan, monitor and debug

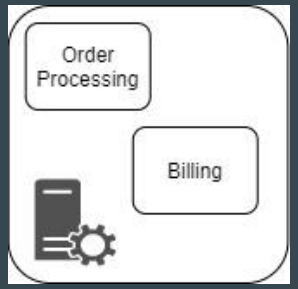
Decoupling Example - Code

```
function processTransaction(amount: number) {
  // the actual processing

  const notificationService = new EmailNotificationService();
  notificationService.send(`Sent ${amount} successfully`);
}
```

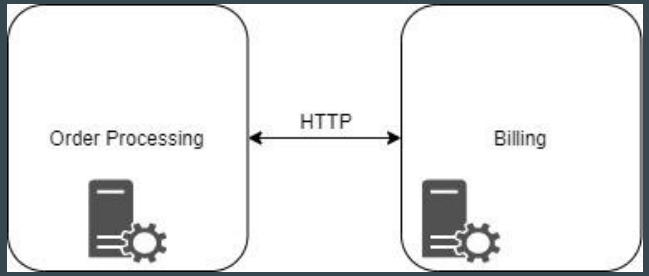
```
function processTransaction(amount: number, notificationService: NotificationService) {
  // the actual processing

  notificationService.send(`Sent ${amount} successfully`);
}
```

Our function is no longer responsible for managing the dependencies. We just program for a certain interface/contract.

Decoupling Example - Systems

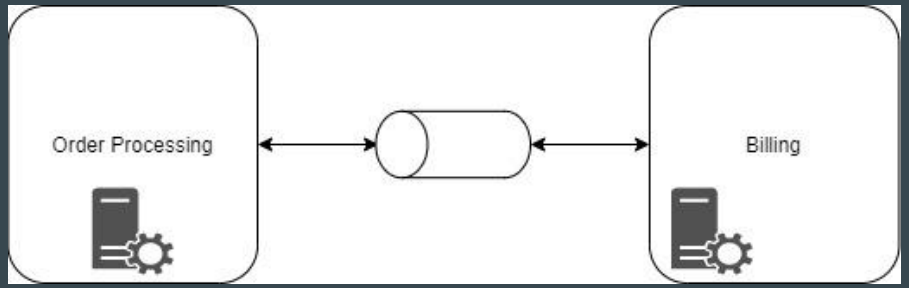


Everything built and deployed together

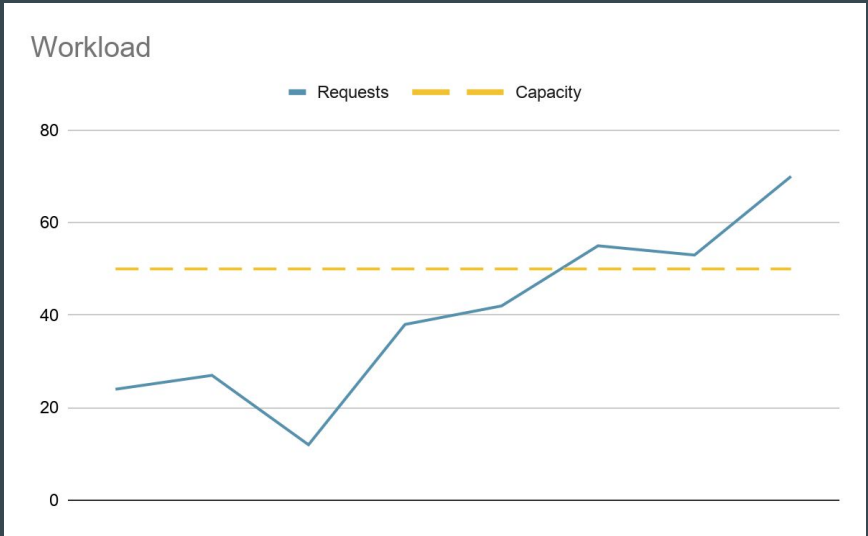


Independently built and deployed. Order service still must know how to communicate with the billing service

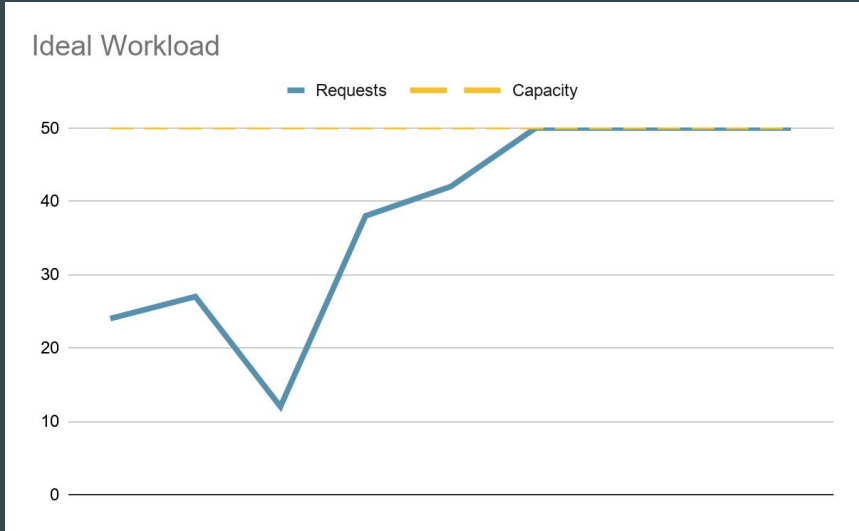
Independently built and deployed. Order service is no longer concerned with how the message reaches the billing service.



Unpredictable Workloads

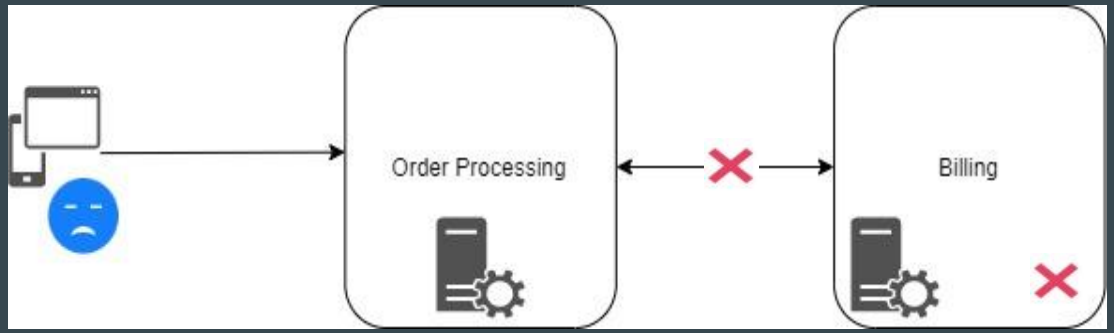


What happens when the demand exceeds the capacity?



Instead of handling directly all the incoming requests, have a buffer in front.

Temporary Issues



- Temporary connection issues
- Billing service is down (server issues, planned maintenance)
- Rate limiting/throttling

The client has to perform the action again (e.g. fill in a long form)

New Features

Considerations when delivering new features:

- Development (ripple effects/changes)
- Testing (how much to test to cover the new feature?)
- Deployment (speed, green/blue, canary)
- Minimizing the impact radius (in case something goes wrong)

(A)Synchronicity

From the user’s perspective, it’s about when the response is received

- Synchronous communication - immediate response
- Async communication - response is sent separately (e.g. notifications)

Synchronous communication works when everything works. In practice, things tend to break.

A good portion of use cases can be handled asynchronously

There are ways to overcome async communication downsides: e.g. server push, polling.

Flow Control

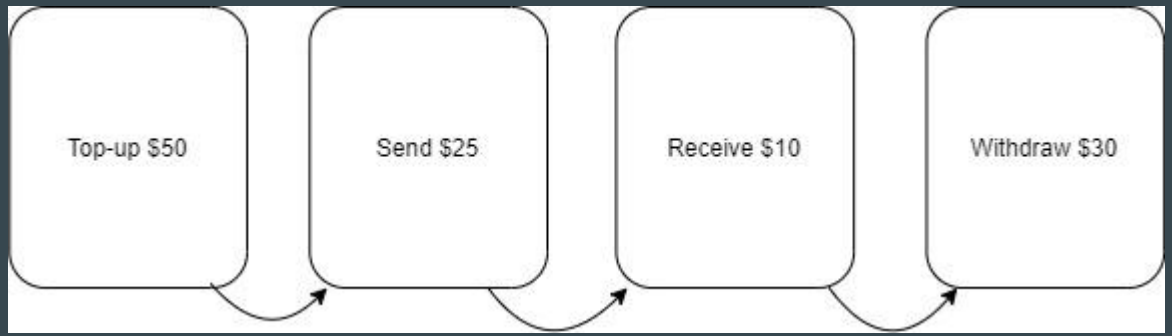
There might be cases in which requests must be processed in sequence.

We might want to temporary stop processing requests.

Certain requests might have higher priority.

Persistence of Change

We are used to persisting the state.

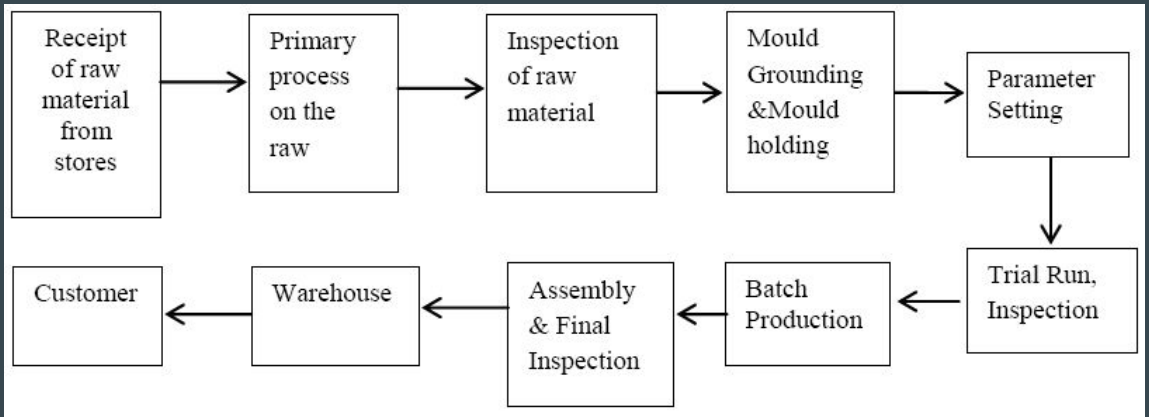


We can reconstruct the state.

We can analyze the history.

A.k.a. **Event-sourcing**

Workflow Orchestration



Combine different services to achieve one result.

Maintain state.

Support different triggers (schedule, on a certain action, under certain conditions)

Contract Definition

Separating the interface from the implementation might be beneficial:

- Easier refactoring/reworking/changing the architecture (e.g. transition from monolith to microservices)
- Other services are concerned only with the contract.
- Easier to achieve a common/unified interface.
- More in course 13 (API Design)

Common Integration Services

Common Integration Services

Queues

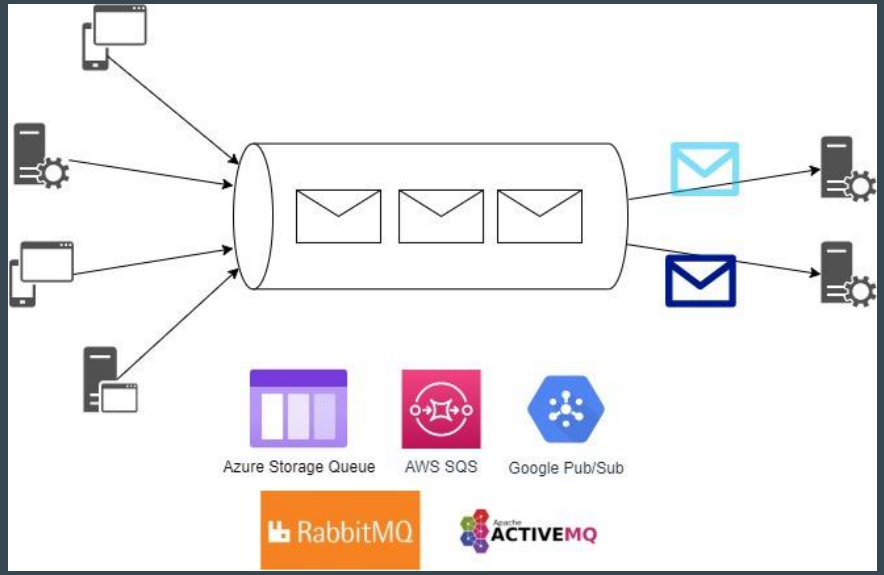
Publish/Subscribe Services

Streaming Services

API Gateways

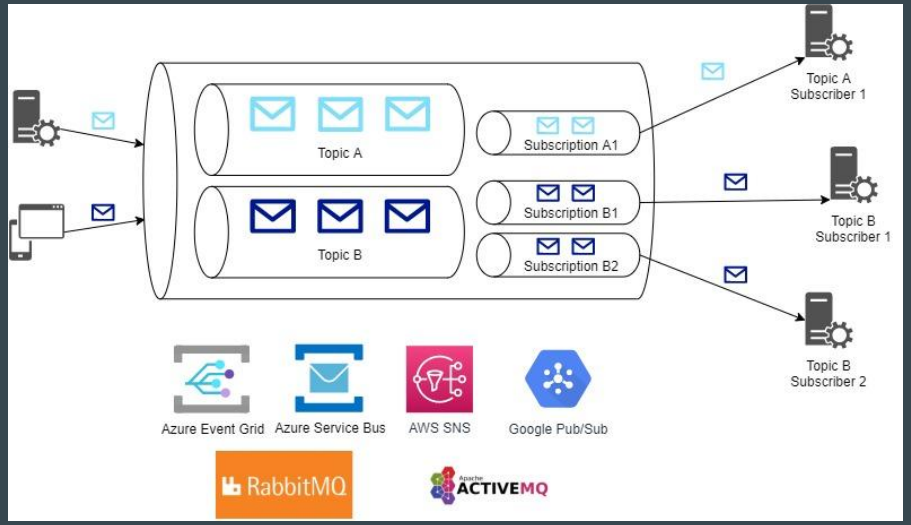
Workflow Services

Queues



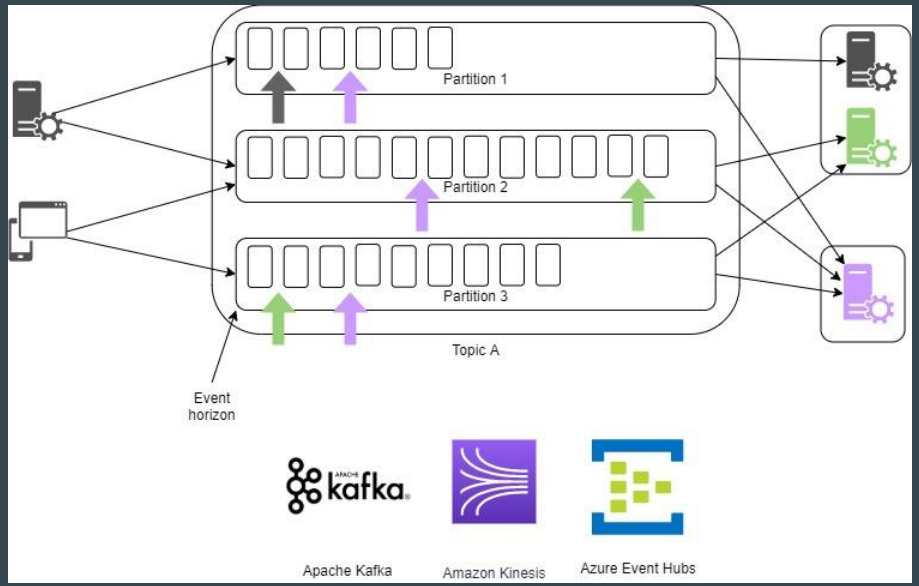
- **Producers** send messages to the queue
- **Brokers** manage the messages
- **Consumers poll the queue (pull)**
- Any consumer can receive any message(s) (batch)
- Once processed, **the consumer sends the acknowledgement** (usually deletes the message)
- Messages might be out of order.
- Usually **at least once** delivery.
- Queues help smoothing out the traffic and avoiding data loss
- Unprocessed message usually go to a **dead-letter queue**

Publish/Subscribe



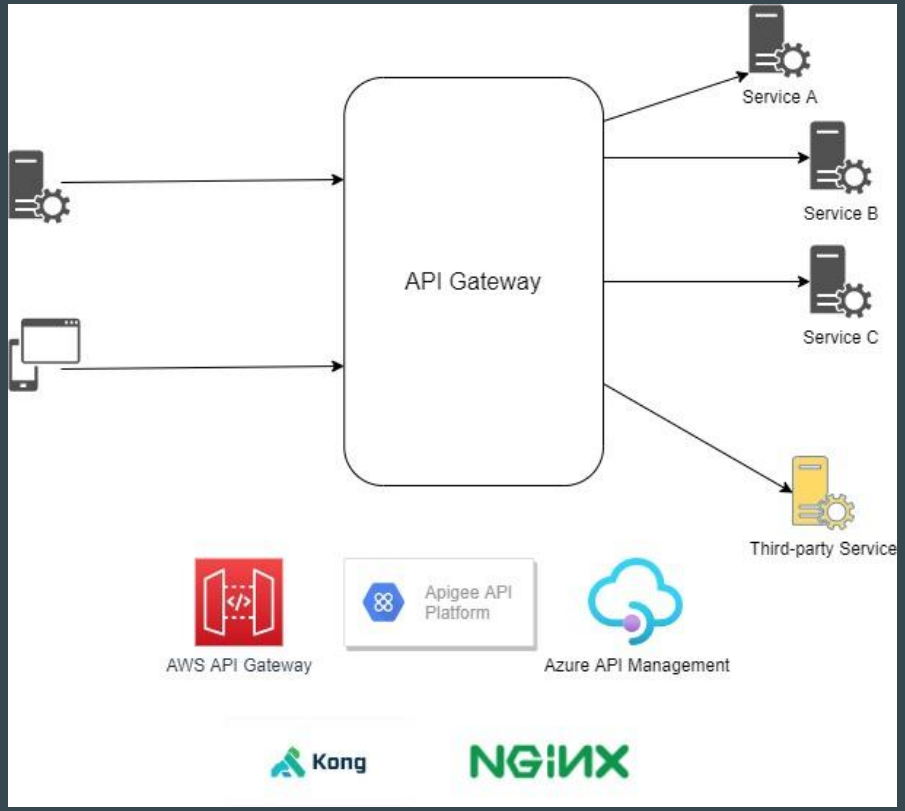
- **Publishers** publish messages
- Brokers manage messages
- **Subscribers receive** messages (**push**)
- All subscribers receive all messages (relevant to them)
- Some services store the messages, some don't (e.g. if there are no subscribers for AWS SNS, the message is lost)
- Some services offer message deduplication (to achieve **at most once delivery**) and strict ordering (e.g. Azure Service Bus)
- Usually offer dead-letter queues

Streaming Services



- **Producers** publish messages
- Brokers manage messages
- **Consumers** read messages from topics
- Each **consumer group** receives a copy of the message. Partitions are distributed across the consumers within a group.
- Each consumer is responsible to keep track of the messages (checkpoints)
- Partitions can be seen as **append-only files**
- Messages are not deleted, they expire (or they might be kept indefinitely - e.g. New York Times keeps all posts ever created in Kafka)

API Gateways



- Usually based on **HTTP**
- Act as an entrypoint/front door/reverse proxy
- Handle request routing and protocol translation if necessary
- Usually provide additional features such as rate limiting and caching
- Common design pattern when working with microservices
- Also useful when monetizing APIs
  - API Management services

Workflow Services

- Orchestrate processes involving multiple services
- Can be seen as a **finite state machine**
- Usually come with lots of connectors/integrations. E.g. dropbox, slack, stripe, google suite, office365
- Can span various time intervals (from seconds to years)
- Are usually based on FaaS and leverage other data stores for state tracking