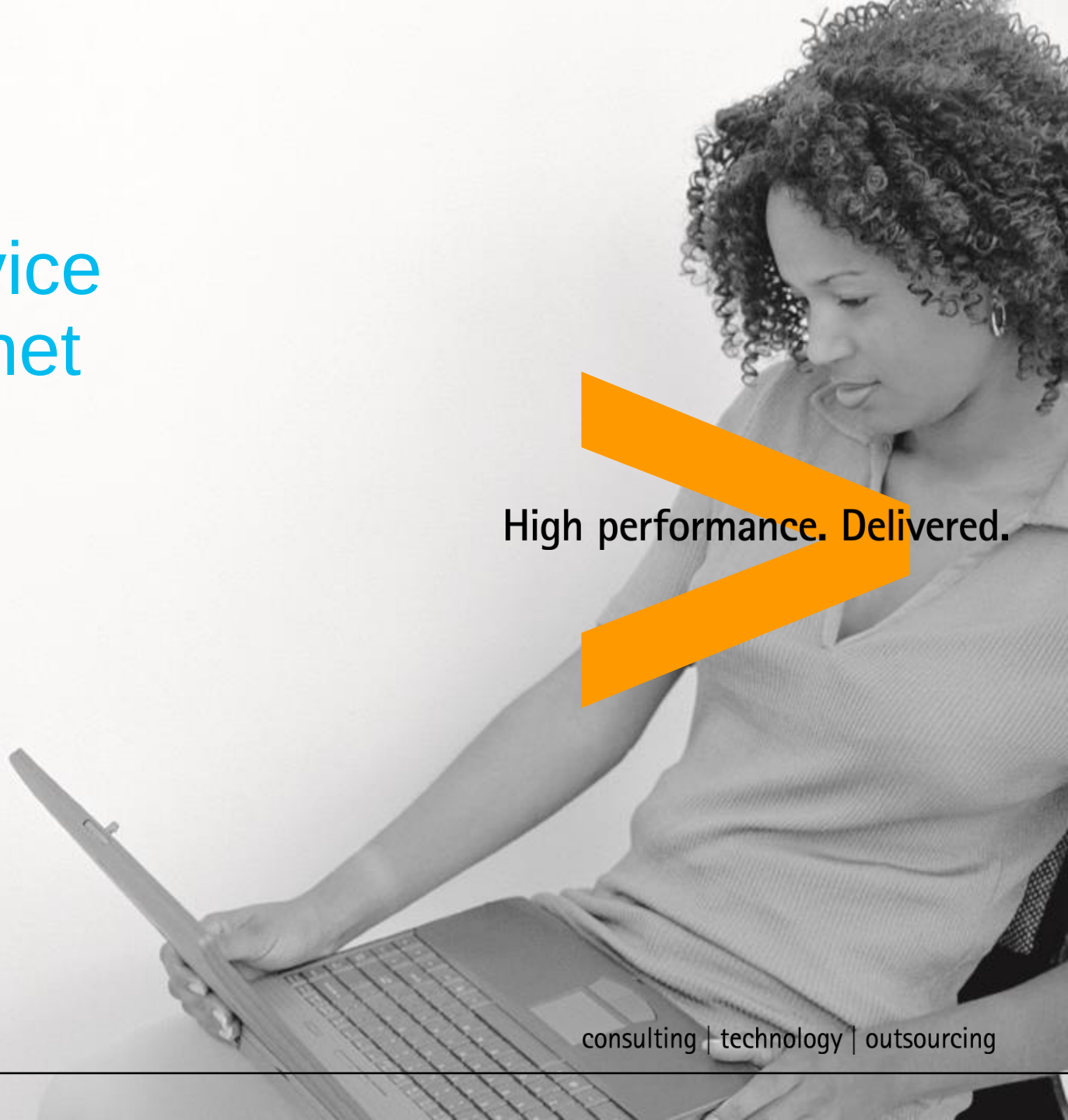
# Micro Service Using Dotnet




consulting | technology | outsourcing

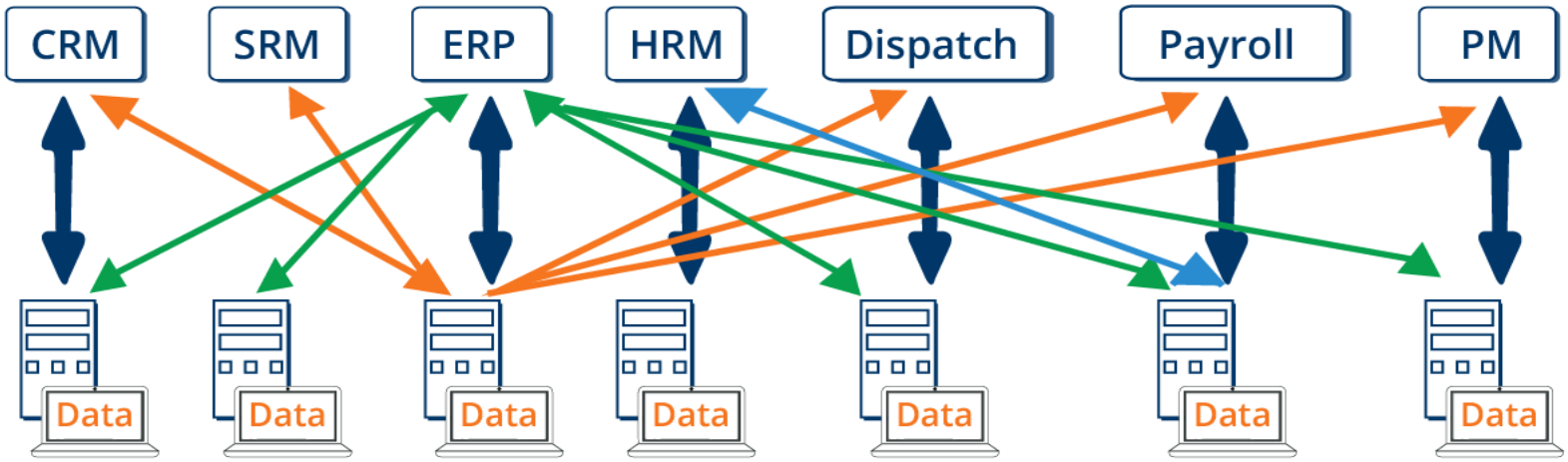
# What is Enterprise Integration and where is it needed?


- Organizations rely heavily on technologies – which are often of different nature to each other - and applications, running in different business units and likely based on different technologies.
- So, integrating these applications into a unified set of business processes has emerged as a priority.
- Enterprise Integration provides a mechanism to build a composite application and share processes and data.

# What is Enterprise Integration and where is it needed?


- Enterprise integration started as point-to-point integration and evolved to cloud based enterprise service bus architectures.
- In a point-to-point integration, integrating between two applications requires code development and creating information you could exchange between them.
- This may prove cost-saving and effective on small networks, but it results in exponential growth of cost and required effort for enterprises.
- Forrester Research estimates that up to 35% of development time is spent creating application interfaces.

# What is Enterprise Integration and where is it needed?

# What is Enterprise Integration and where is it needed?

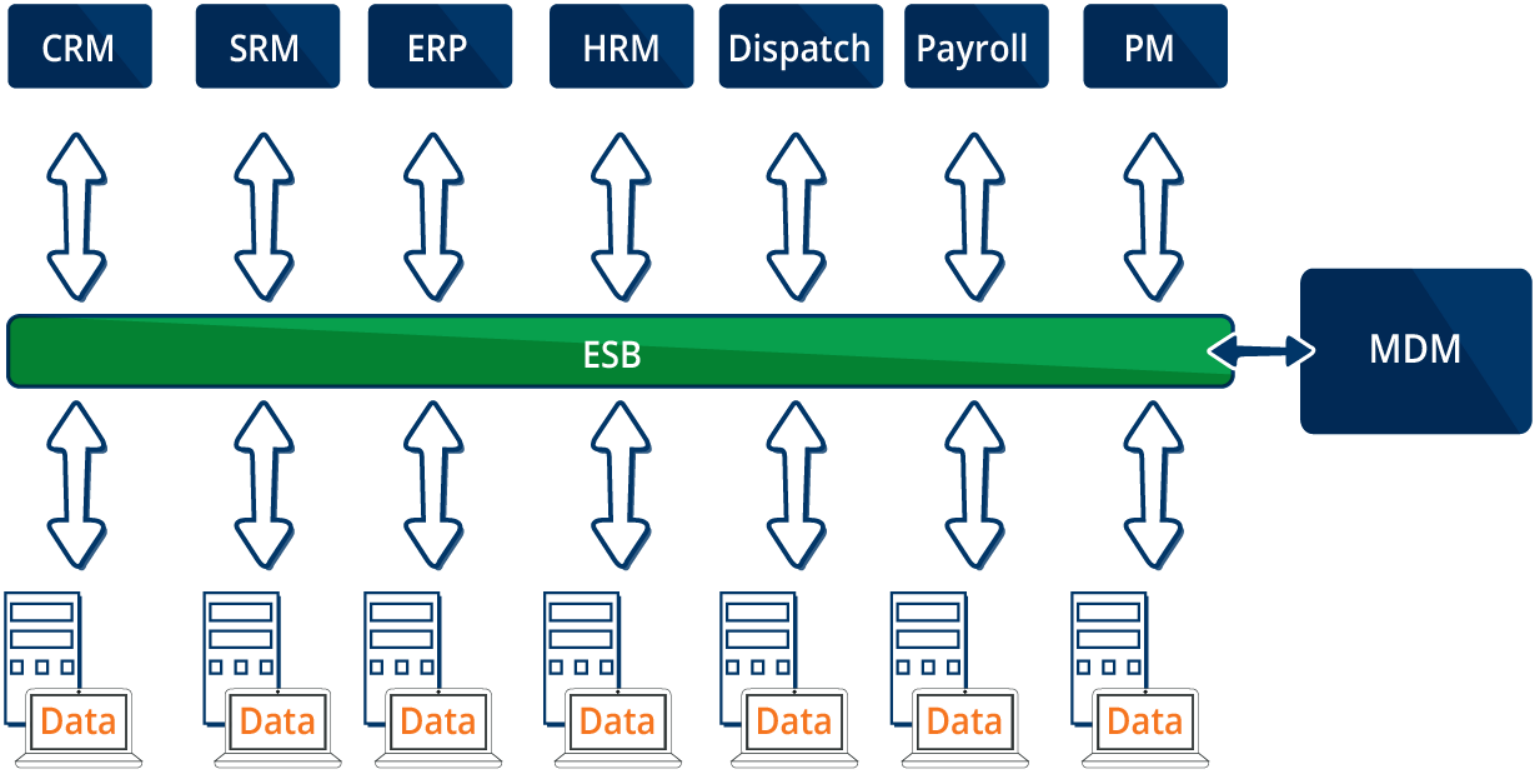

- In an Enterprise Integration platform, enterprises integrate between applications through an Enterprise Service Bus (a middleware platform) which orchestrates the information exchanged between the integration endpoints, reducing the number of interconnections.

# Enterprise Integration features


- Enterprise Integration platforms usually provide features such as
  - Scalability, high availability and sizing.
  - Use of industry standards as interface technologies (no custom development-universal connectivity).
  - Transactional processing, since the information should be consistent within multiple applications.
  - Process driven approach embedding logic into integration flows.
  - Code abstraction, lowering development cost.
  - Monitoring and managing components, making all transaction executions accessible and clear.
  - Deployment management components.

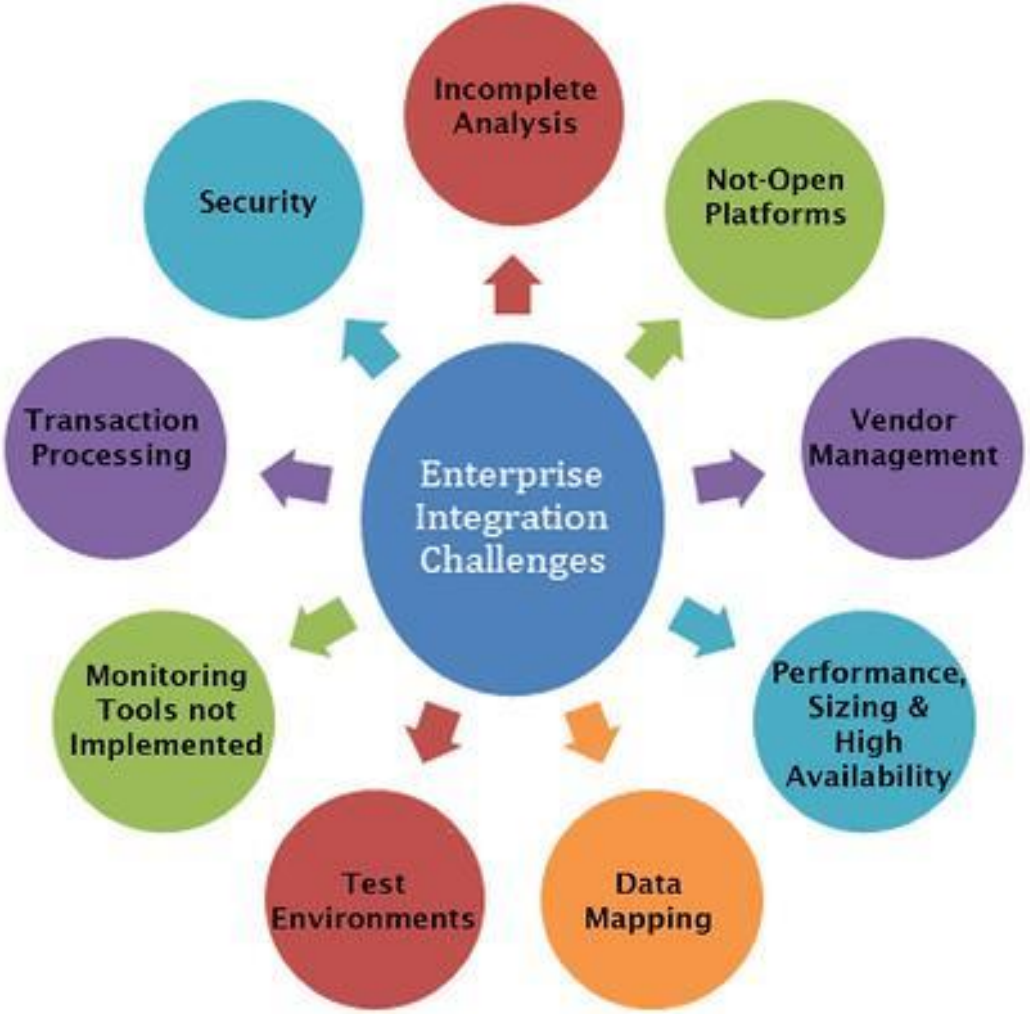
# Enterprise Integration features

Microservices



# Enterprise Integration features

Microservices



# The Global Bank Scenario


- Global Bank is a midsize, traditional bank that has acquired a complete range of financial services capabilities through a series of acquisitions.
- It has a limited online banking presence that is fragmented across its various divisions.
- As part of its strategy to expand with the limited cash it has available, Global Bank has decided to innovate in the online banking market by providing a host of value-added services in addition to a fully integrated financial management capability.

# High-level requirements and constraints


- Build a baseline architecture for a Web-based online banking portal that allows customers to pay bills online from their checking accounts.
- All account-related transactions will use the current system, which resides on an IBM mainframe (OS390) using Customer Information Control System (CICS) based transactions.
- The online bank system will reside in the corporate data center in Seattle, Washington.
- It will be connected to an acquired bank's data center in Los Angeles, California though a private leased line.

# High-level requirements and constraints

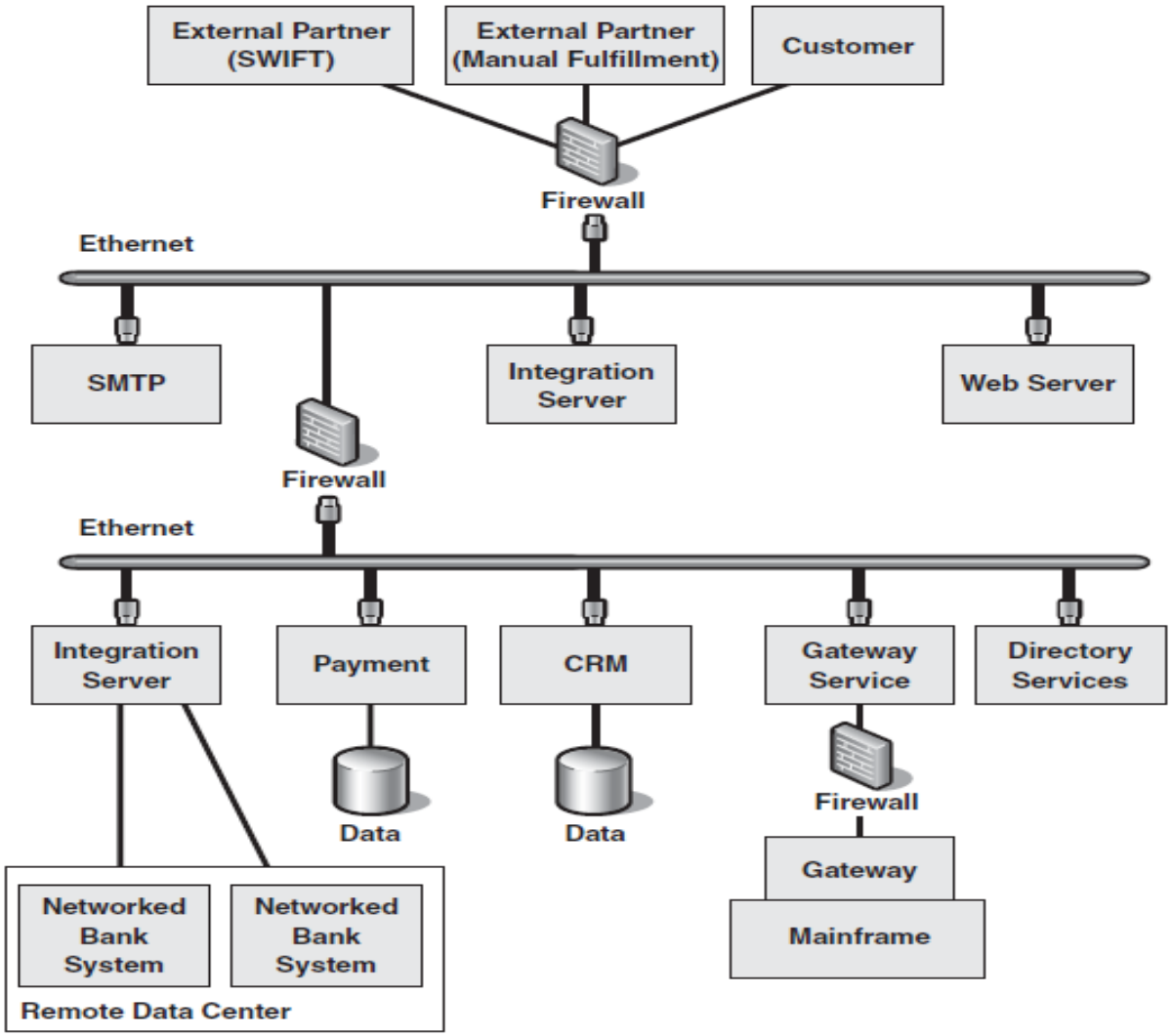

- Loan information will be pulled from the acquired bank's loan systems, which reside on systems that are based on IBM WebSphere J2EE.
- All customer profile information will use the current Customer Relationship Management (CRM) system.
- Domestic electronic payments will use the current payment system, and international electronic payments will use SWIFT-based transactions through an external payment gateway.

# High-level requirements and constraints


- Payees that cannot receive electronic payments will be paid using electronic transactions to a manual fulfillment center, which will then make the payments manually through the U.S. mail.
- Except for the systems previously identified, the system will be based on the Microsoft platform.
- The system's overall transaction rates, concurrent users, and response time must meet the first year's projected usage plus an engineering safety factor of 3x (or three times the first year's projected usage) to handle burst load.
- The system must meet or exceed the service level agreement (SLA) for our current online system.

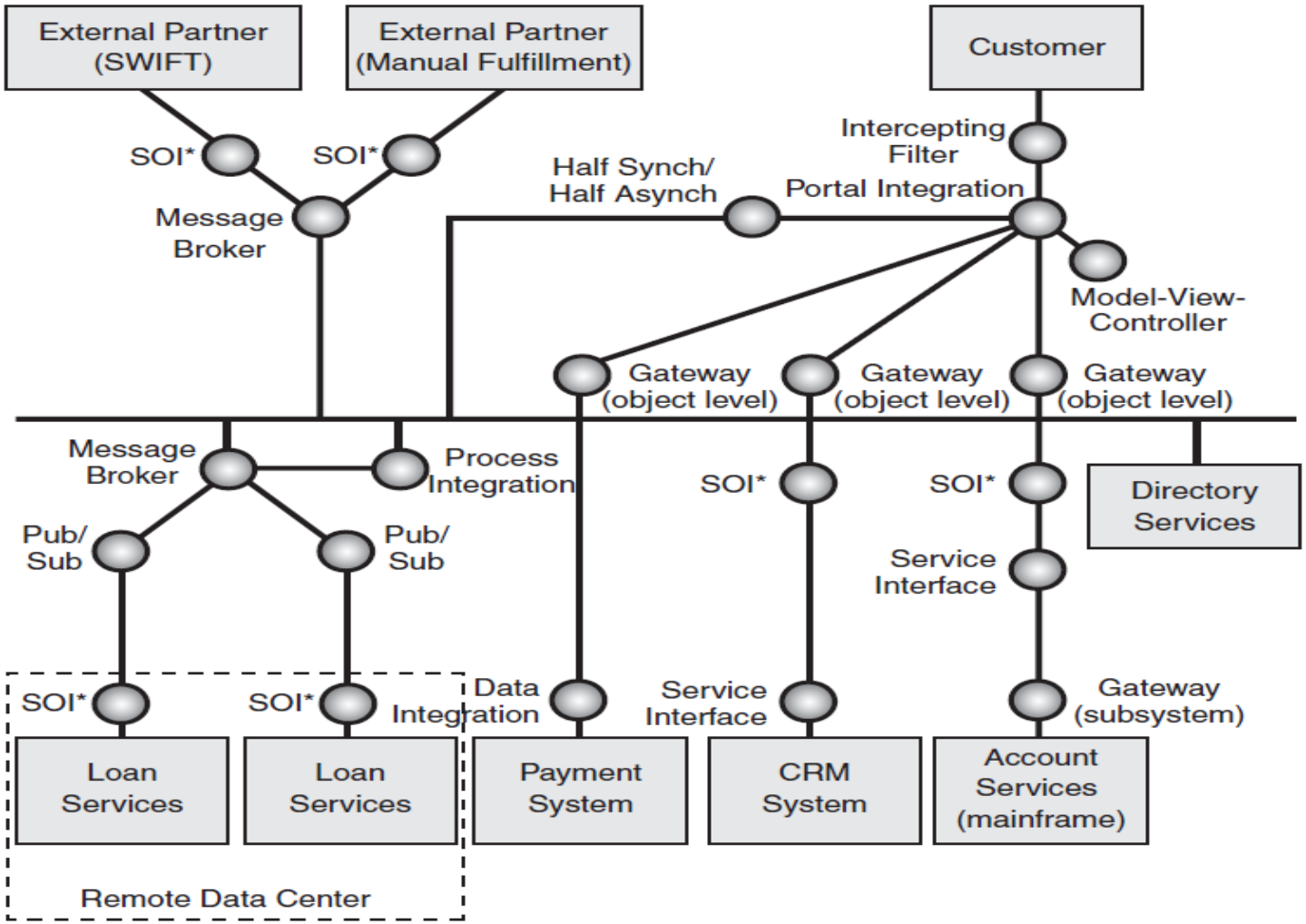
# Patterns at Global Bank(Initial Netwok Setup)

Microservices



# Patterns at Global Bank

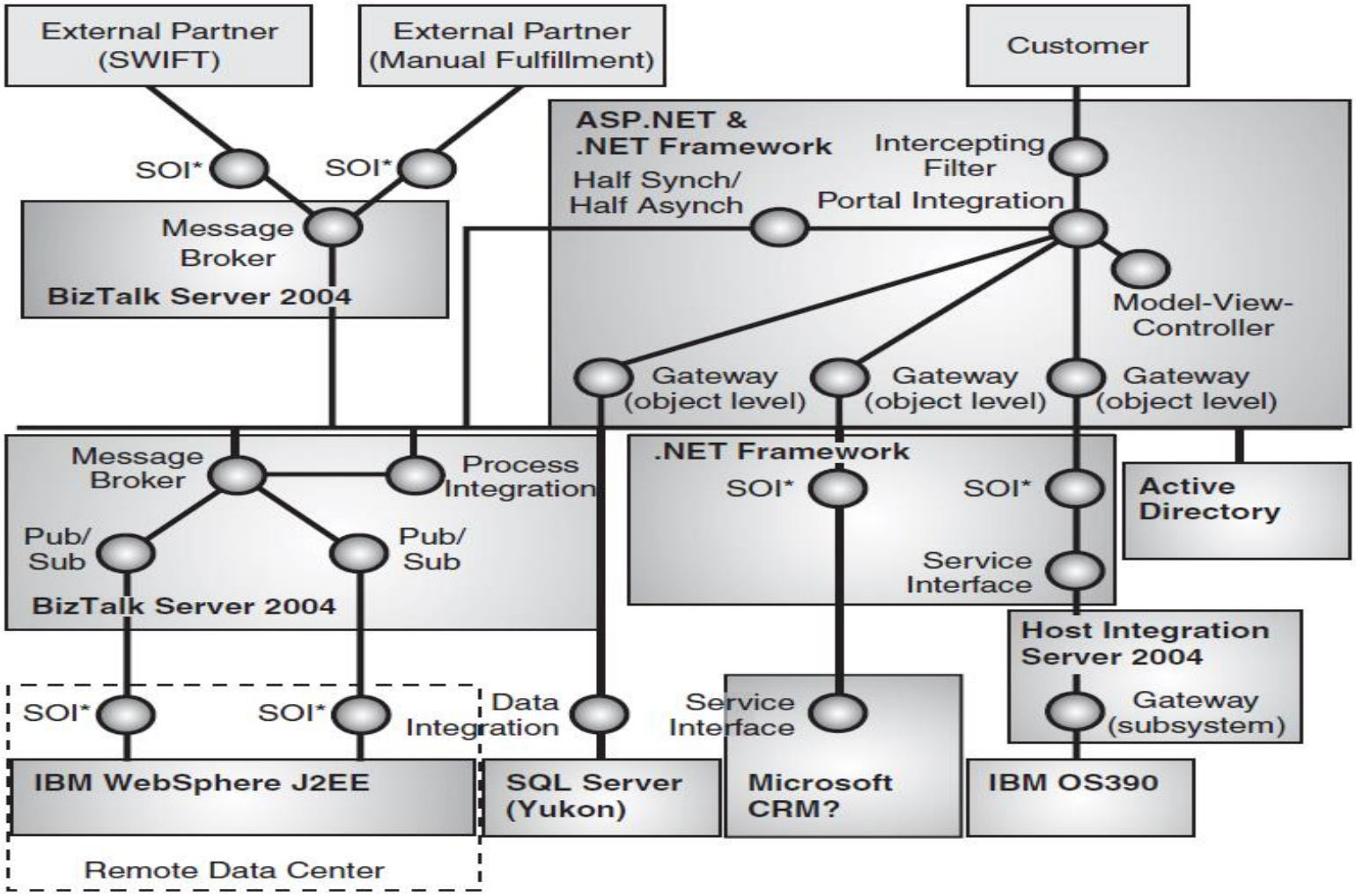
Microservices




SOI* = Service-Oriented Integration

# Patterns at Global Bank

Microservices



# New Demands for Microservices Integration


- While most of the requirements for integrating microservices still reflects existing enterprise integration patterns.
- Their highly distributed nature creates new demands for decentralized messaging, based on smart endpoints and dumb pipes.
- Instead of a central, unique integration bus, each group of microservices (usually within the same bounded context) will choose its own messaging implementation, depending on the needs and characteristics of each use case.

# Communication types


- **Synchronous protocol**
  - HTTP is a synchronous protocol.
  - The client sends a request and waits for a response from the service.
  - That's independent of the client code execution that could be synchronous (thread is blocked) or asynchronous (thread isn't blocked, and the response will reach a callback eventually).
  - The important point here is that the protocol (HTTP/HTTPS) is synchronous and the client code can only continue its task when it receives the HTTP server response.

# Communication types


- **Asynchronous protocol**
  - Other protocols like AMQP (a protocol supported by many operating systems and cloud environments) use asynchronous messages.
  - The client code or message sender usually doesn't wait for a response.
  - It just sends the message as when sending a message to a RabbitMQ queue or any other message broker.

# Communication types

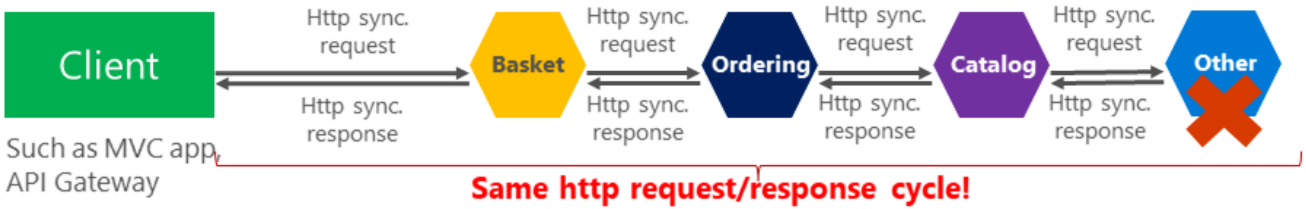

- **Single receiver.**
  - Each request must be processed by exactly one receiver or service. An example of this communication is the Command pattern.
- **Multiple receivers**
  - Each request can be processed by zero to multiple receivers.
  - This type of communication must be asynchronous.
  - An example is the publish/subscribe mechanism used in patterns like **Event-driven architecture**.
  - This is based on an event-bus interface or message broker when propagating data updates between multiple microservices through events;
  - it's usually implemented through a service bus or similar artifact like **Azure Service Bus by using topics and subscriptions.**

# Anti-patterns and patterns in communication between microservices

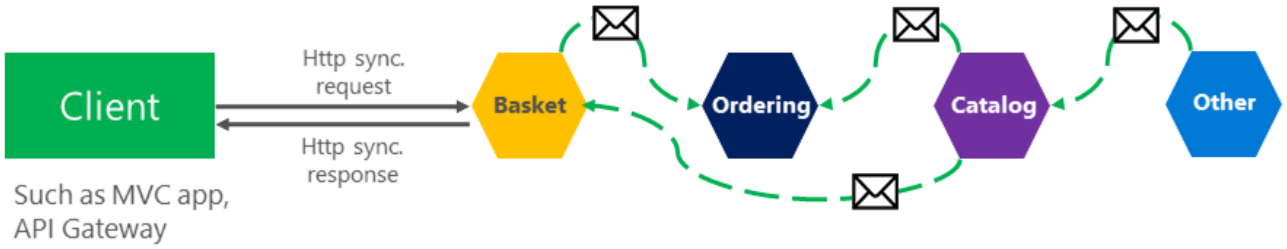

## Synchronous vs. async communication across microservices

**Anti-pattern**

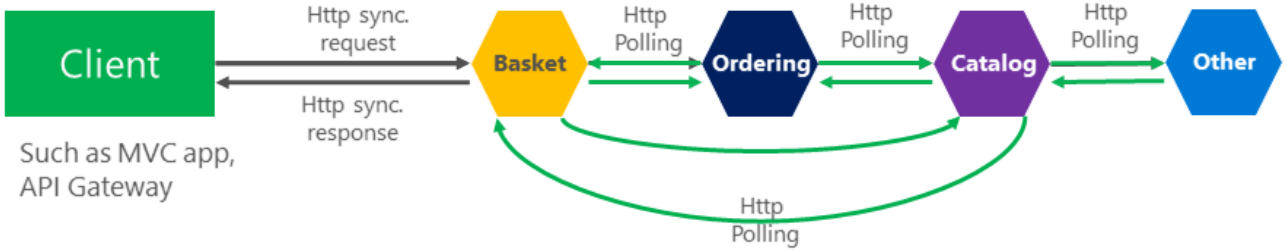
**Synchronous**
all request/response cycle




**Asynchronous**
Comm. across internal microservices
(EventBus: like **AMQP**)



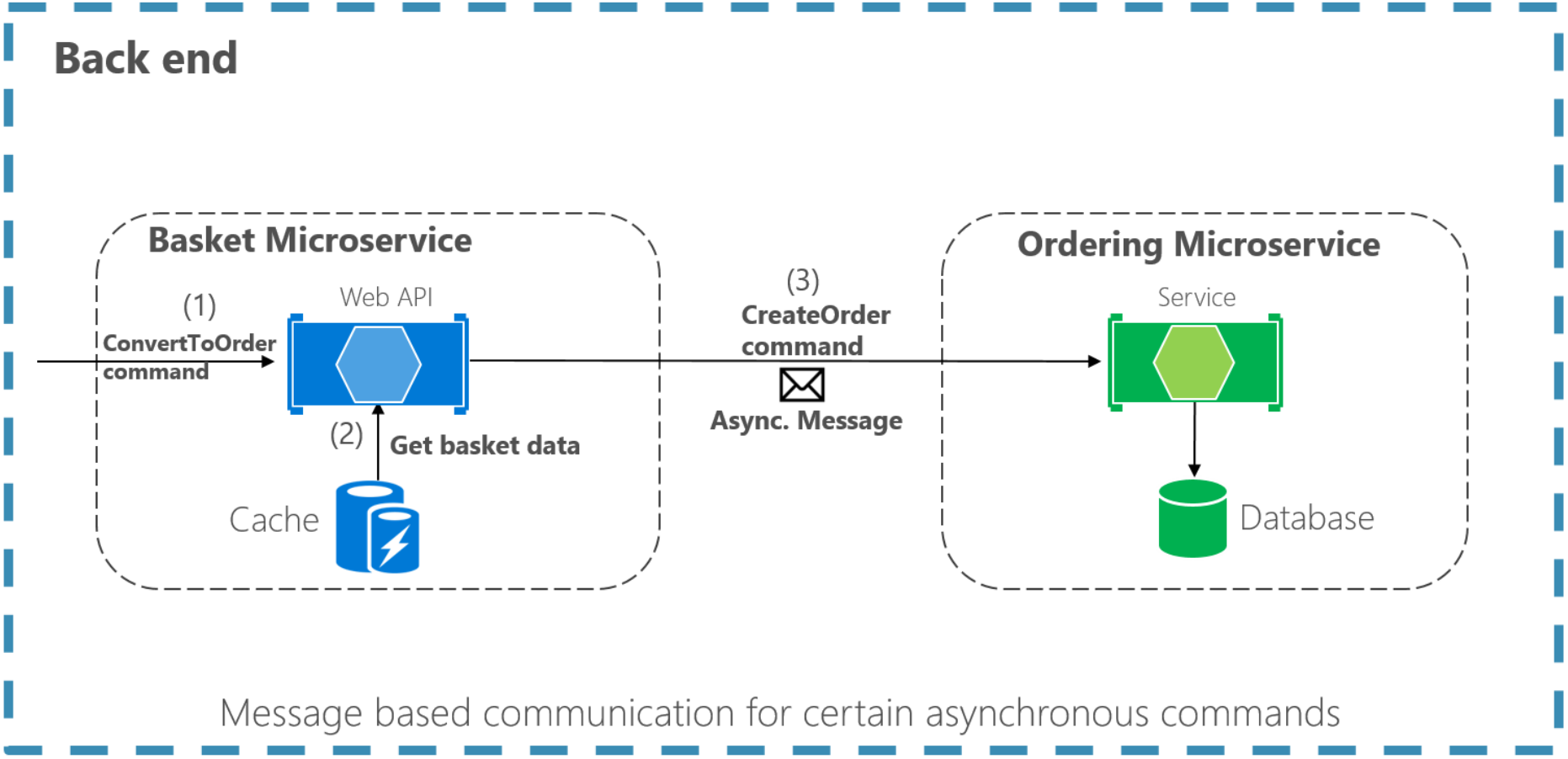

**"Asynchronous"**
Comm. across internal microservices
(Polling: **Http**)



# A single microservice receiving an asynchronous message


## Single receiver message-based communication

(i.e. Message-based Commands)

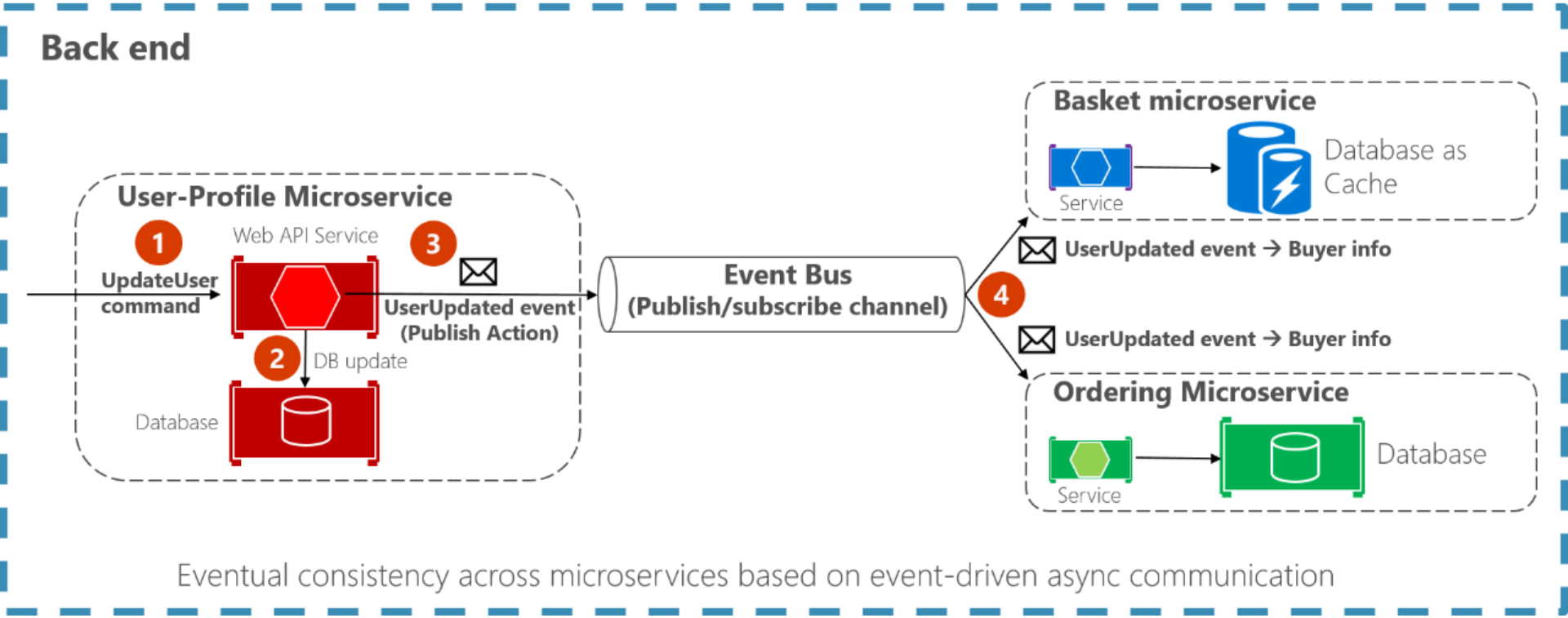


# Asynchronous event-driven message communication

Microservices

## Asynchronous event-driven communication

Multiple receivers




Eventual consistency across microservices based on event-driven async communication

# Mass Transit


## A DISTRIBUTED SYSTEM. AN INSURANCE COMPANY









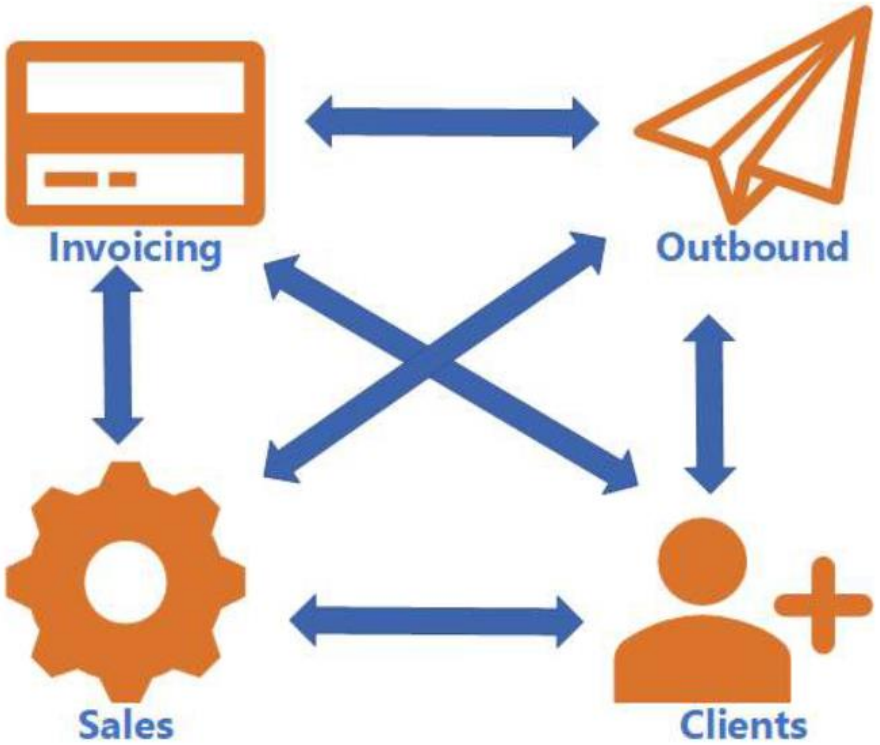


# Mass Transit

Microservices

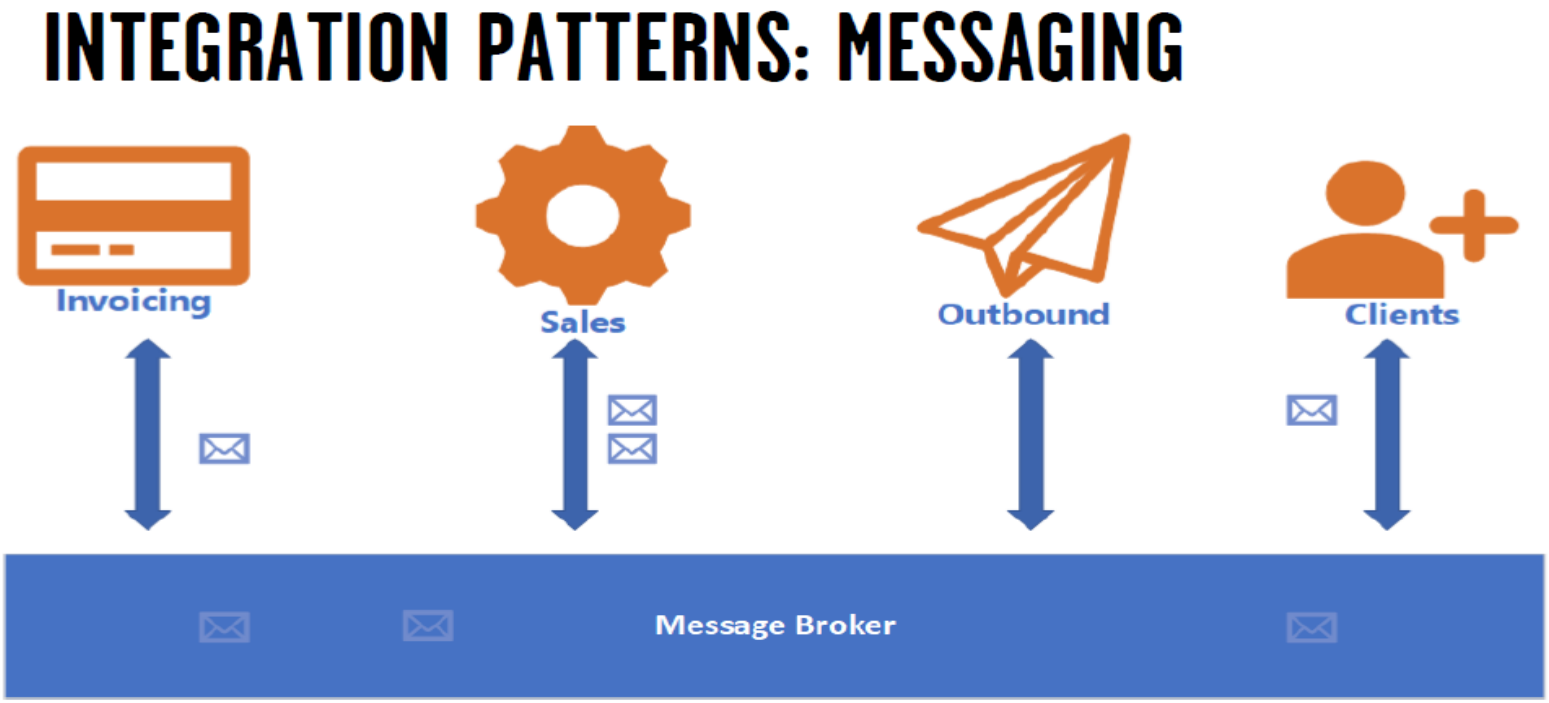
## INTEGRATION PATTERNS: RPC (SOAP/REST)

- Tightly coupled
- Complexity grows
- Need to modify existing code, when a new system appears
- What if system X is down for maintenance?



# Mass Transit

Microservices

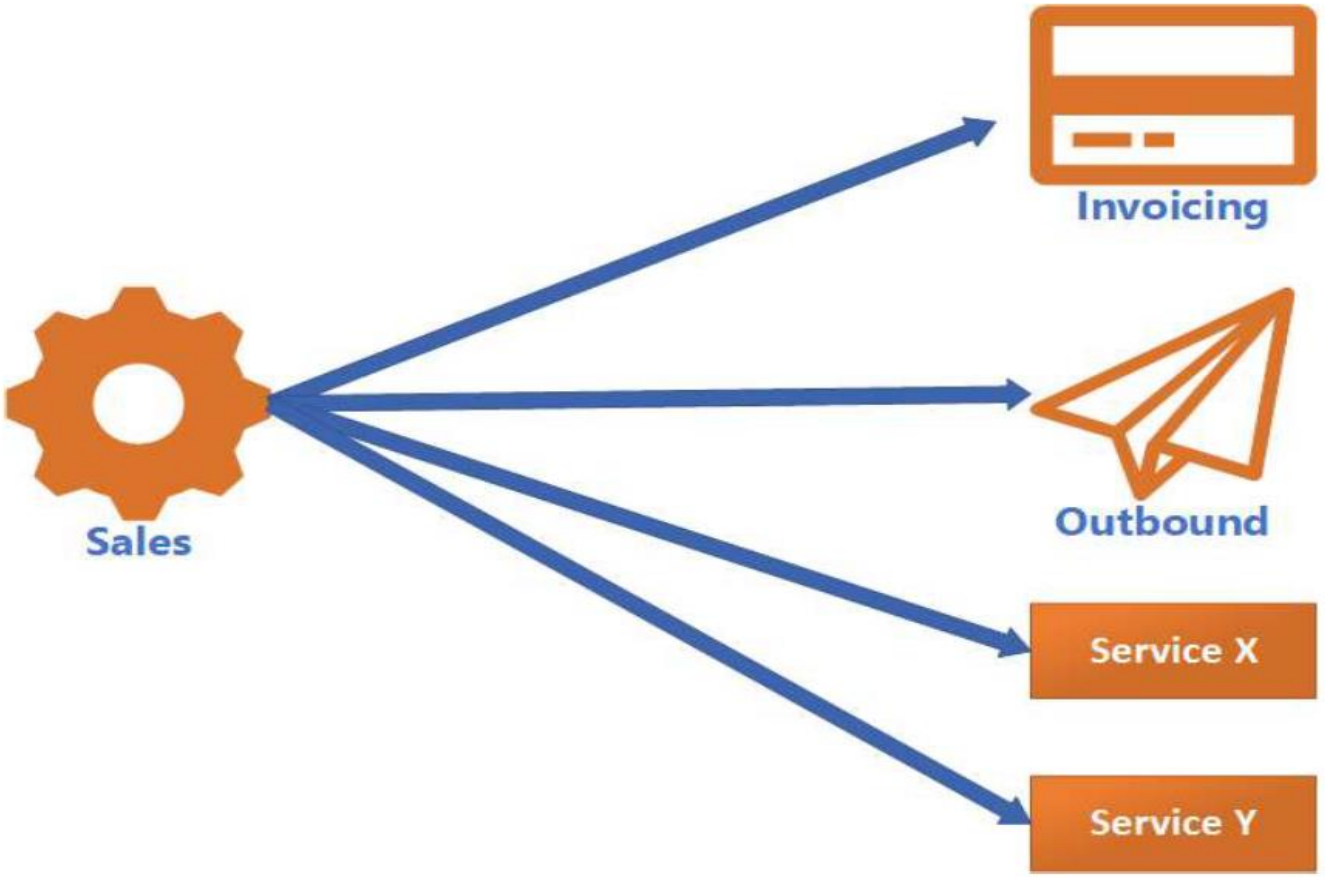



1. Decoupled
2. Share only messaging contract
3. Truly async communication: the message will reach the destination

# Mass Transit

Microservices

## MESSAGING PATTERNS: BROADCAST/EVENT

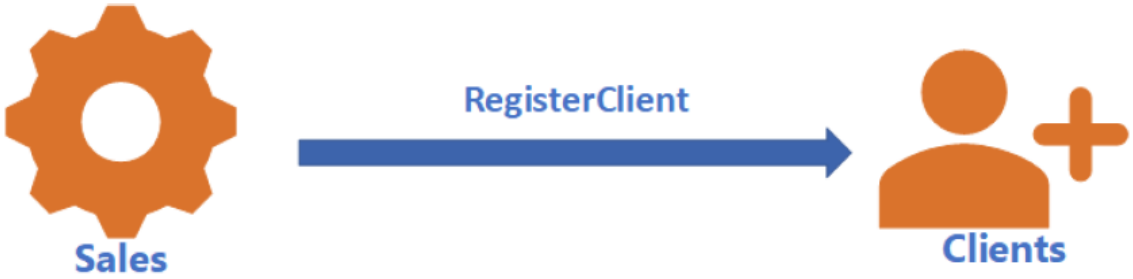


# MESSAGING PATTERNS: BROADCAST/EVENT

Events should be expressed in a noun-verb (past tense) sequence:

- PolicyIssued
- ClientUpdated
- OrderShipped
- PhoneChanged

# MESSAGING PATTERNS: A COMMAND



# MESSAGING PATTERNS: A COMMAND

Commands should be expressed in a verb-noun sequence, following the *tell* style

- RegisterClient
- ChangeAddress
- SubmitOrder

# Mass Transit


## MESSAGING PATTERNS: RPC REQUEST/RESPONSE

# MESSAGE BROKER VS MESSAGE BUS

Another layer of abstraction can solve any problem in IT

- Message bus is broker agnostic (an abstraction above message brokers). Like ORM DB providers in .NET or Log4net, etc.
- Message bus takes care of low-level details like
  - Queues/Exchanges/Bindings
  - Acknowledgements
  - Serialization/Deserialization
  - Error handling
  - Retries/Resilience
  - Auditing
- Developers can concentrate on sending and receiving messages.
- Message bus – usually is language specific

# Azure Service Bus - Working With Queue In A Real World Scenario

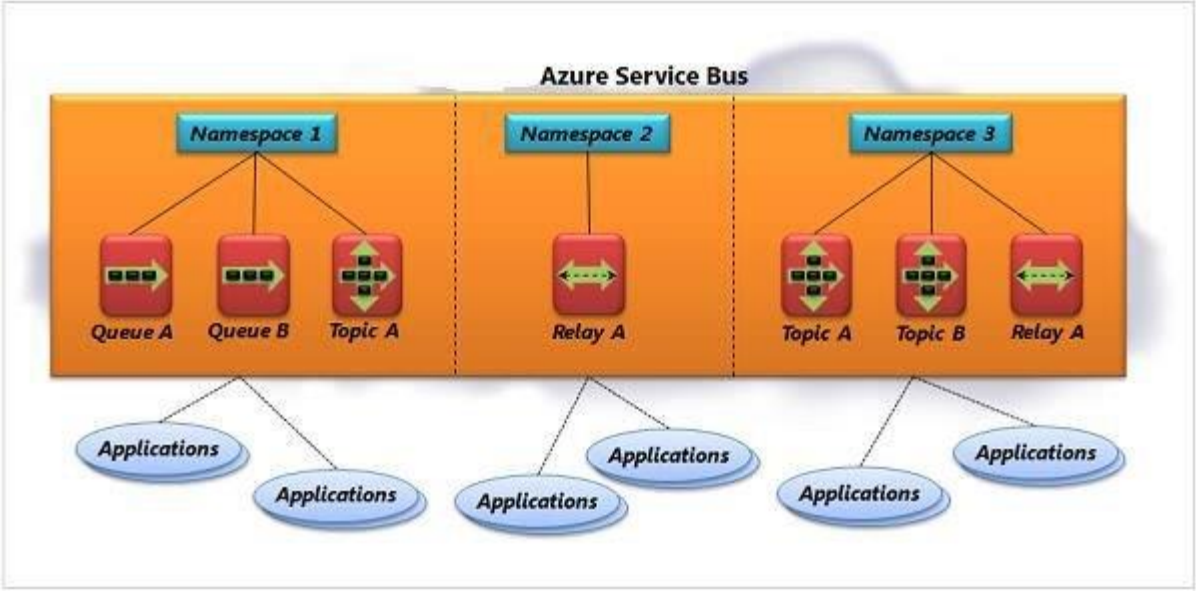

- Microsoft Azure Service Bus is a reliable information delivery service.
- The main agenda is to make the communication easier.
- When two or more parties want to exchange information, they need a communication facilitator.
- Service Bus is a brokered, or third-party communication mechanism.

# Azure Service Bus - Working With Queue In A Real World Scenario


- Microsoft Azure Service Bus is a reliable information delivery service.
- The main agenda is to make the communication easier.
- When two or more parties want to exchange information, they need a communication facilitator.
- Service Bus is a brokered, or third-party communication mechanism.

# Azure Service Bus: Concept


- Azure service bus is a multi- tenant cloud service. multiple users share the services. The application developer who creates a namespace, then defines the communication mechanisms needed within that namespace.



# Azure Service Bus Fundamentals


- Queues
  - Queues allows one way communication.
  - Each queue acts a broker that stores sent messages until they are
  - A single recipient receives every single message.

# Azure Service Bus Fundamentals


- Topics
  - The Topics help in one-way communication
  - The Topic is similar to a broker, but they also help you to filter the received message to match the specific criteria.

# Azure Service Bus Fundamentals


- Relays
  - Relays provide bi-directional communication.
  - When compared to queues and topics relays don’t act like a broker.
  - Instead, it just passes them on to the destination application.

# Azure Service Bus: Queues


- When the user decides to connect two application is using a Service Bus queue.
- The process is carried out when the sender sends a message to a Service Bus Queue and a receiver picks up the message at some point of time.
- It is to be noted that a queue can have just a single receiver.
  - **Every individual message has two parts**
  - **A set of properties each of which is a key and a message payload.**

# Azure Service Bus: Queues


- A receiver can read the message from a Service Bus queue in two different ways
  - **Receive & delete**: Receives a message from the queue and deletes it immediately
  - **Peek Lock** : It removes the message from the queue, it does not delete the message instead it locks the message making it invisible to other

# Azure Service Bus: Relay


- A one- way asynchronous communication is provided by both queues and topics through a broker.
  - In a relay there the traffic flows in just one direction and there is no direct connection between senders and receivers.
  - You don’t need a broker to store a message when your application needs to both send and receive messages

# Azure Service Bus:Standard vs Premium Messaging


| Premium | Standard |
| --- | --- |
| High throughput | Variable throughput |
| Predictable performance | Variable latency |
| Fixed pricing | Pay as you go variable pricing |
| Ability to scale workload up and down | N/A |
| Message size up to 1 MB | Message size up to 256 KB |

# Azure Function


- Azure Function is a serverless compute service that enables user to run event-triggered code without having to provision or manage infrastructure.
- Being as a trigger-based service, it runs a script or piece of code in response to a variety of events.
- Azure Functions can be used to achieve decoupling, high throughput, reusability and shared. Being more reliable, it can also be used for the production environments.

# Azure Function Features


- Serverless applications
- Choice of language
- Pay-per-use pricing model
- Bring your own dependencies
- Integrated security
- Simplified integration
- Flexible development
- Stateful serverless architecture
- Open-source

# What can I do with Azure Functions?


- **HTTP**: Run code based on HTTP requests
- **Timer**: Schedule code to run at predefined times
- **Azure Cosmos DB**: Process new and modified Azure Cosmos DB documents
- **Blob storage**: Process new and modified Azure Storage blobs
- **Queue storage**: Respond to Azure Storage queue messages
- **Event Grid**: Respond to Azure Event Grid events via subscriptions and filters
- **Event Hub**: Respond to high-volumes of Azure Event Hub events
- **Service Bus Queue**: Connect to other Azure or on-premises services by responding Service Bus queue messages
- **Service Bus Topic**: Connect other Azure services or on-premises services by responding to Service Bus topic messages

# Azure Functions vs Web Jobs

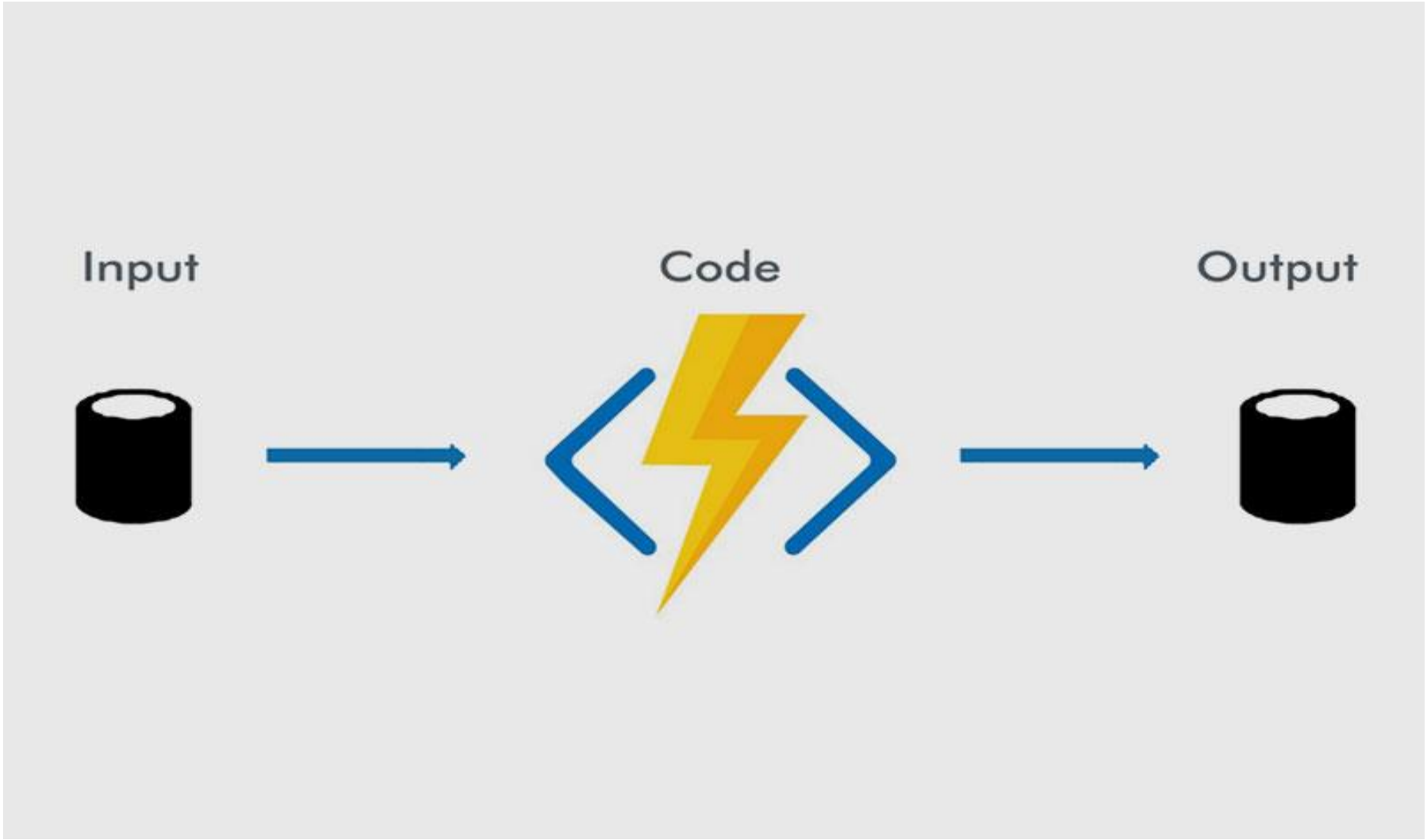

| | Azure Functions | Web Jobs |
|---|---|---|
| Trigger | Azure Functions can be triggered with any of the configured trigger, but it doesn't run continuously | Web Jobs is of two types, Triggered Web Jobs and Continuous Web Jobs |
| Supported languages | Azure Functions support various languages like C#, F#, JavaScript, node.js and more. | Web Jobs also support various languages like C#, F#, JavaScript and more. |
| Deployment | Azure Functions is a separate App Service that run in the App Service Plan | Web Jobs run as a background service for the App services like Web App, API Apps and mobile Apps |

# Azure Functions Vs Logic Apps


| Azure Functions | Azure Logic Apps | |
|---|---|---|
| Trigger | Azure Functions can be triggered with the configured trigger like HTTPTrigger, TimerTrigger, QueueTrigger and more | Azure Logic Apps can be triggered with the API as connectors. It can also have multiple triggers in a workflow. |
| Defining Workflow | Workflow in Azure Functions can be defined using Azure Durable Function. It consists of Orchestrator Function that has the workflow defined with Several Activity Functions | Workflow on Azure Logic Apps can be defined with Logic App designer using various APIs as Connector. |
| Monitoring | Azure Functions can be monitored using Application Insights and Azure Monitor | Azure Logic Apps can be monitored using Log Analytics and Azure Monitor |
| | Serverless360 can monitor both Azure Function Apps and Logic Apps | |

# ConsumerFunction

Microservices



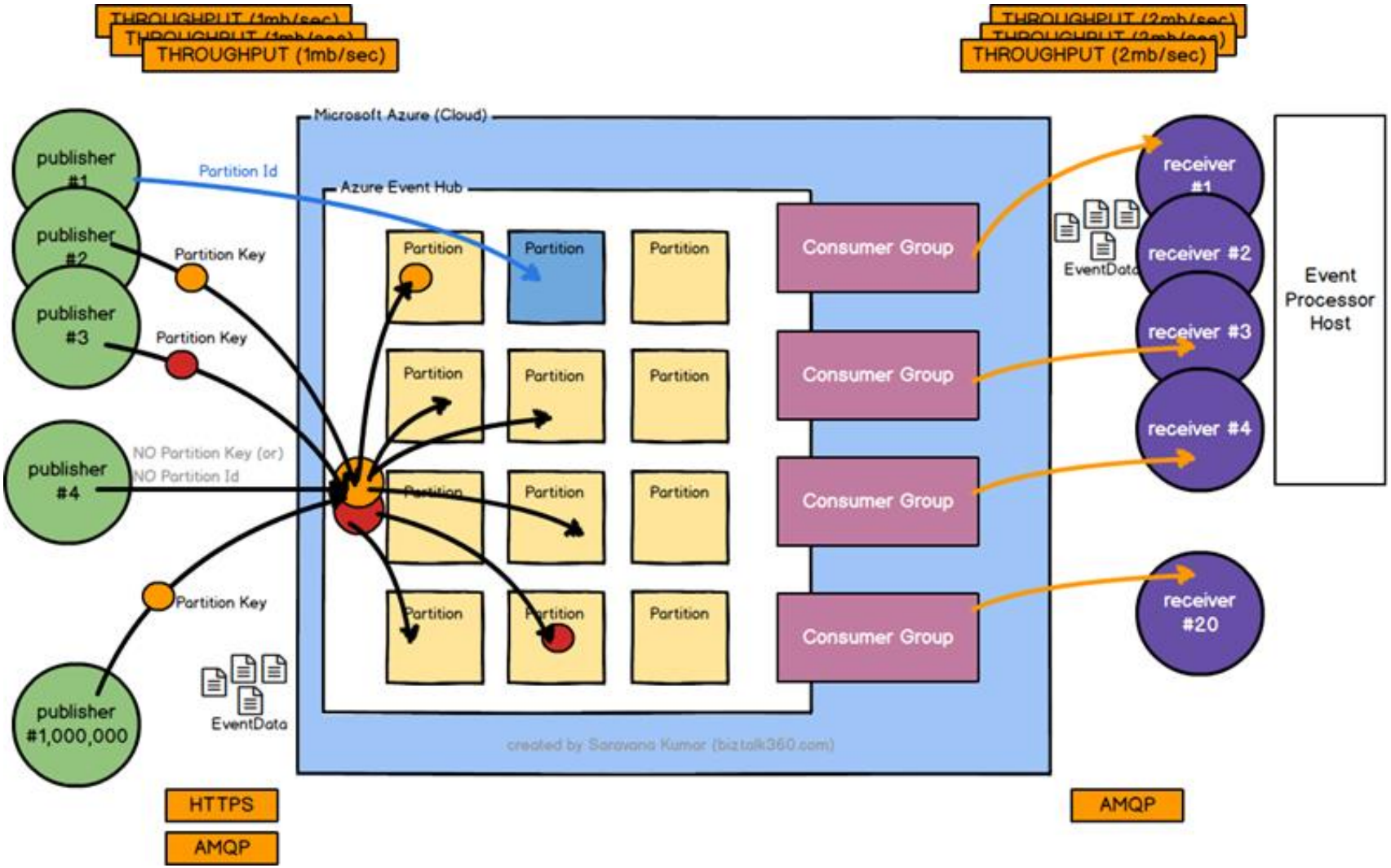
# Azure Event Hub


- Azure Event Hubs is a big data streaming platform and event ingestion service.
- It can receive and process millions of events per second.
- Data sent to an event hub can be transformed and stored by using any real-time analytics provider or batching/storage adapters.

# Azure Event Hub


- The following scenarios are some of the scenarios where you can use Event Hubs:
  - Anomaly detection (fraud/outliers)
  - Application logging
  - Analytics pipelines, such as clickstreams
  - Live dashboarding
  - Archiving data
  - Transaction processing
  - User telemetry processing
  - Device telemetry streaming

# Azure Event Hub



# Why use Event Hubs?


- Data is valuable only when there is an easy way to process and get timely insights from data sources.
- Event Hubs provides a distributed stream processing platform with low latency and seamless integration, with data and analytics services inside and outside Azure to build your complete big data pipeline.

# Why use Event Hubs?


- Event Hubs represents the "front door" for an event pipeline, often called an event ingestor in solution architectures.
- An event ingestor is a component or service that sits between event publishers and event consumers to decouple the production of an event stream from the consumption of those events.
- Event Hubs provides a unified streaming platform with time retention buffer, decoupling event producers from event consumers.

# Key architecture components

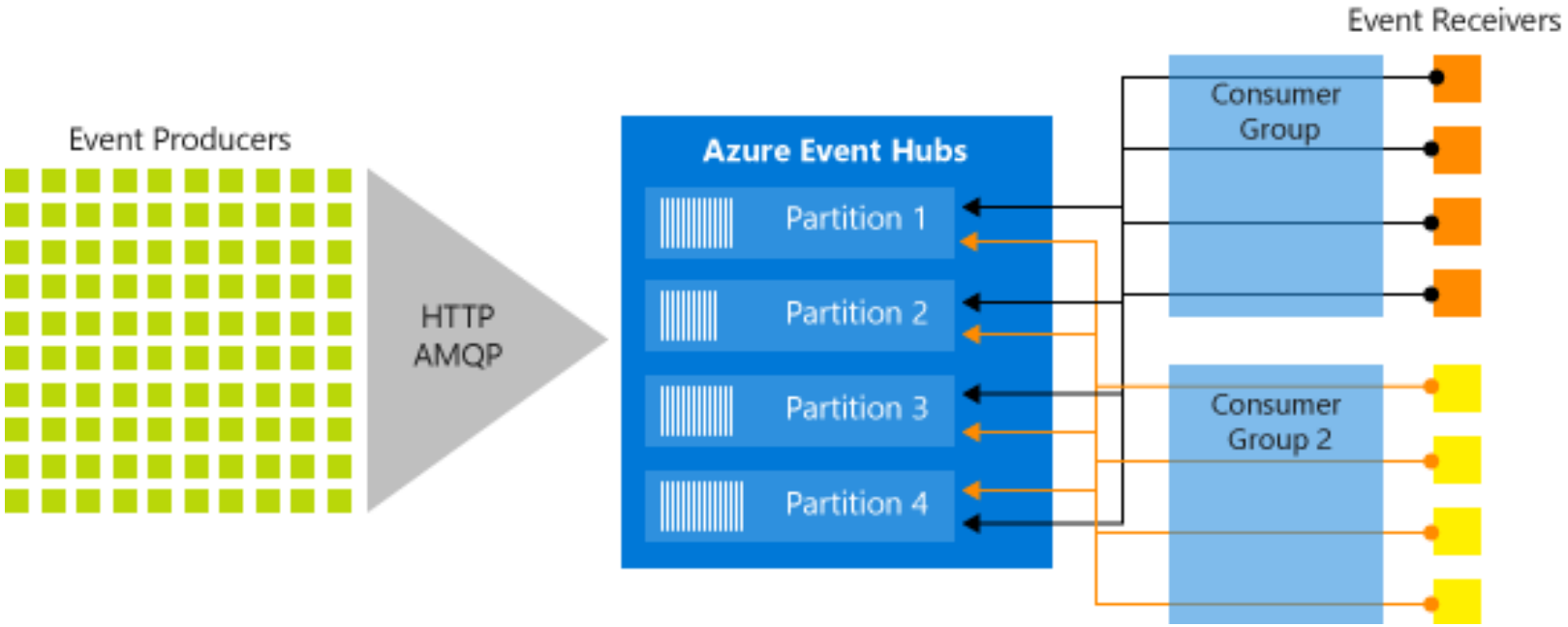

- Event Hubs contains the following key components:
  - **Event producers**: Any entity that sends data to an event hub. Event publishers can publish events using HTTPS or AMQP 1.0 or Apache Kafka (1.0 and above)
  - **Partitions**: Each consumer only reads a specific subset, or partition, of the message stream.
  - **Consumer groups**: A view (state, position, or offset) of an entire event hub. Consumer groups enable consuming applications to each have a separate view of the event stream. They read the stream independently at their own pace and with their own offsets.

# Key architecture components


- Event Hubs contains the following key components:
  - **Throughput units:** Pre-purchased units of capacity that control the throughput capacity of Event Hubs.
  - **Event receivers:** Any entity that reads event data from an event hub. All Event Hubs consumers connect via the AMQP 1.0 session. The Event Hubs service delivers events through a session as they become available. All Kafka consumers connect via the Kafka protocol 1.0 and later.

# Key architecture components

Microservices



# Mass Transit


- What is MassTransit?
- MassTransit is a .NET Friendly abstraction over message-broker technologies like RabbitMQ.
- It makes it easier to work with RabbitMQ by providing a lots of dev-friendly configurations.
- It essentially helps developers to route messages over Messaging Service Buses, with native support for RabbitMQ.
- MassTransit does not have a specific implementation.
- It basically works like an interface, an abstraction over the whole message bus concept.

# Mass Transit


- **What is MassTransit?**
  - MassTransit is a .NET Friendly abstraction over message-broker technologies like RabbitMQ.
  - It makes it easier to work with RabbitMQ by providing a lots of dev-friendly configurations.
  - It essentially helps developers to route messages over Messaging Service Buses, with native support for RabbitMQ.
  - MassTransit does not have a specific implementation.
  - It basically works like an interface, an abstraction over the whole message bus concept.

# Mass Transit


- **What is MassTransit?**
  - Remember how Entity Framework Core provides an abstraction over the data access layers? Similary MassTransit supports all the Major Messaging Bus Implementations like RabbitMQ, Kafka and more.
  - Since we are using RabbitMQ, we could have installed the client packages of RabbitMQ like RabbitMQ.Client and so on, that enables us to interact with the RabbitMQ Server.
  - But, for a cleaner and future-proof solution, it is always advisable to go with an Abstraction. Makes your life much easier on the longer run.

# Mass Transit


- An open-source lightweight message bus framework for .NET. MassTransit is useful for routing messages over MSMQ, RabbitMQ, TIBCO, and ActiveMQ service busses, with native support for MSMQ and RabbitMQ.
- MassTransit also supports multicast, versioning, encryption, sagas, retries, transactions, distributed systems, and other features.

# Questions

# Module Summary

- Microservices using docker and Kubernets.
- Message, Channel and Adapter
- Understood the different Component Integration
- Understood the Event-Driven Architecture