Migration from On-Prem Messaging System to The Cloud: What, How and Why

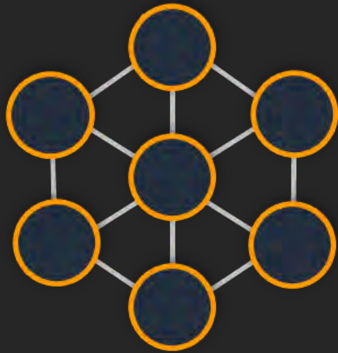
An introductory session about Messaging System Migration

Agenda

- Messaging System Overview
- Migration pathways for a messaging system
- Benefits of migrating messaging system from on-prem to serverless
- Strategy to move between the stages / key factors to consider
- Some Python Code Examples with RabbitMQ/SQS

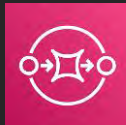
Messaging System Overview

Cloud Native

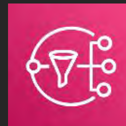


Microservice Design

Examples:



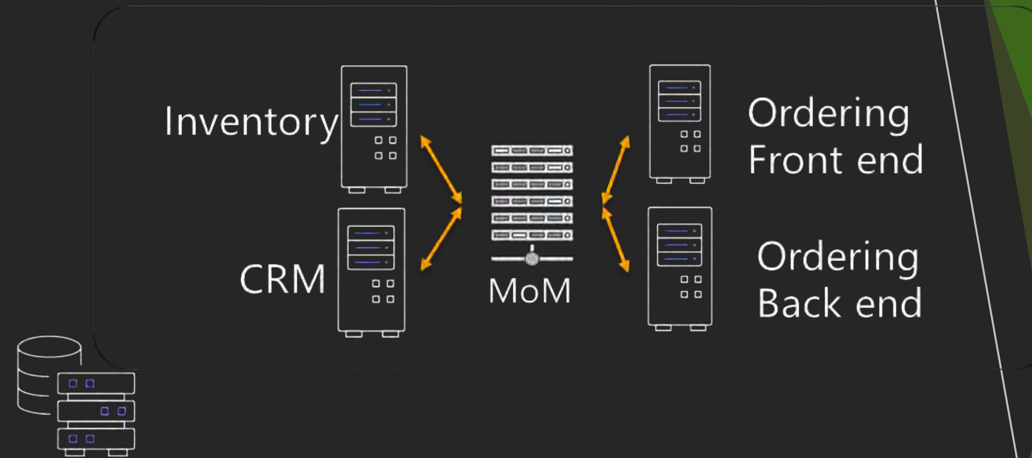
Amazon Simple Queue Services (Amazon SQS)



Amazon Simple Notification Service (Amazon SNS)

Message queuing service
Publish / subscribe messaging and notification service

Traditional Applications

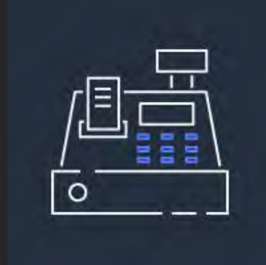


Examples: IBM MQ TIBCO ActiveMQ RabbitMQ

Message-oriented middleware (MoM) or Message Broker

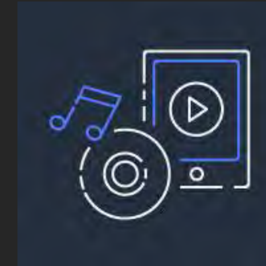
Messaging System Applications

Financial services
Trades and transactions

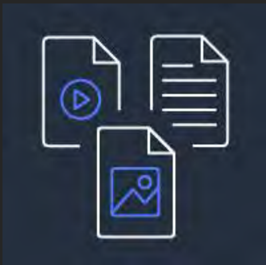


Retail
Order processing and fulfillment

Health care
Clinical data exchange

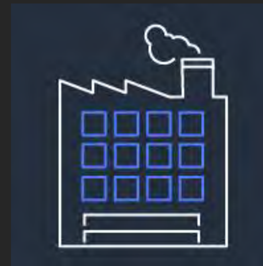


Media and entertainment
Image, or video processing

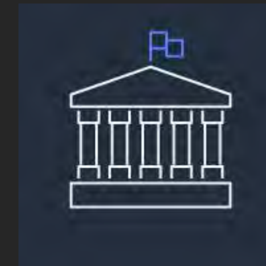


Publication
Document capture and publication search

Education
Learning and engagement



Manufacturing
Multi-step process



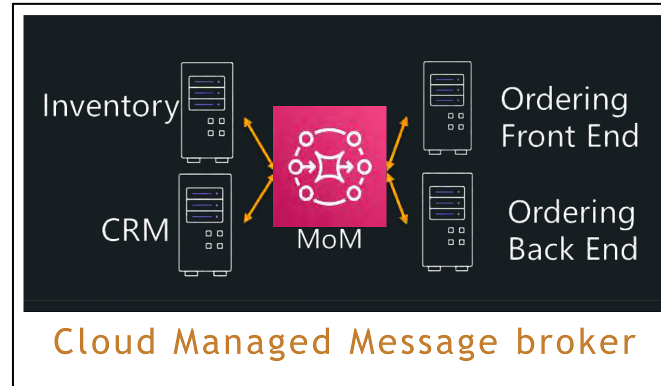
Government
Citizen engagement

Loosely coupled, larger scale, more fault-tolerant systems

Messaging System Migration Journey

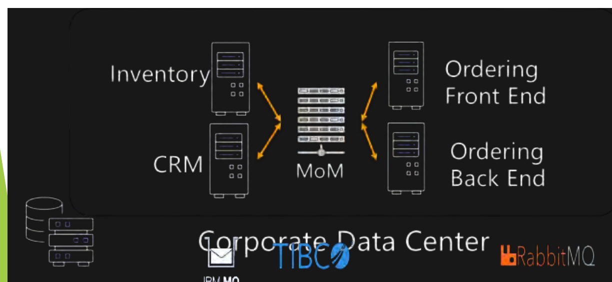
Lift and Shift, Quick Migration

Cost Savings, Operational Efficiencies, & Availability



Higher Scalability, Reliability & Decoupling

On-premise message broker



- Expensive (Capacity and Operation Costs)
- Specialized skillset
- Infrastructure Maintenance and Management
- Difficult to scale

Responsibility Shift by Migration

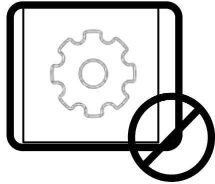
	Stages of Migration			
	ON-PREMISES	INFRASTRUCTURE SERVICES (i.e. EC2)	Managed Services (i.e. Amazon MQ)	CLOUD NATIVE / Serverless (SQS/SNS)
Application code	✓ Customer	✓ Customer	✓ Customer	✓ Customer
Capacity planning and scaling	✓ Customer	✓ Customer	✓ Customer	✓ AWS
Software install and maintenance	✓ Customer	✓ Customer	✓ AWS	✓ AWS
Infrastructure-Level Configuration	✓ Customer	✓ Customer	✓ AWS	✓ AWS
Physical server, storage, networking, and facilities	✓ Customer	✓ AWS	✓ AWS	✓ AWS
Security and network configuration	✓ Customer	✓ AWS ✓ Customer	✓ AWS ✓ Customer	✓ AWS ✓ Customer

MANAGED BY

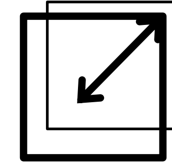
CUSTOMER

Cloud Service Provider (i.e. AWS)

Serverless Architecture Benefits



Decouple infrastructure
from business



Automatically scale
by unit of consumption

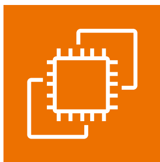
Pay for what you use

Highly available
and durable

Rehost: Lift and Shift to the Cloud

On-Premise

Cloud Instance

Example:  Amazon Elastic Compute Cloud (Amazon EC2)

Pros:

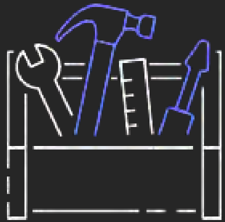
- Quick and Straight Forward Migration Strategy
- Minimal changes to current application logics
- High level of customization over the Cloud instances

Cons:

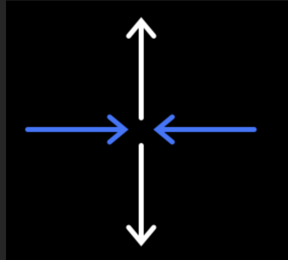
- Infrastructure and Operational tasks remain on the customer's team
- Difficulty integrating with other Cloud-Native Services
- No native built-in observability over the messaging system applications

Replatform: Migrate to Managed Services

Example: Amazon MQ for RabbitMQ

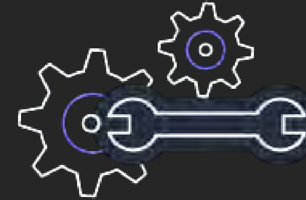


- Setup
- Automatic provisioning of single-node and clustered brokers with best practices by default

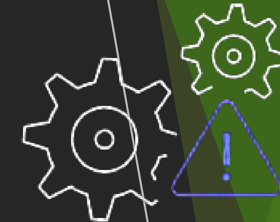


- Scaling
- Vertically scale between Amazon MQ broker instance types

- Security
- Encryption in transit over TLS and encryption at rest using KMS keys



- Upgrades
- Managed administrative tasks, such as software upgrades



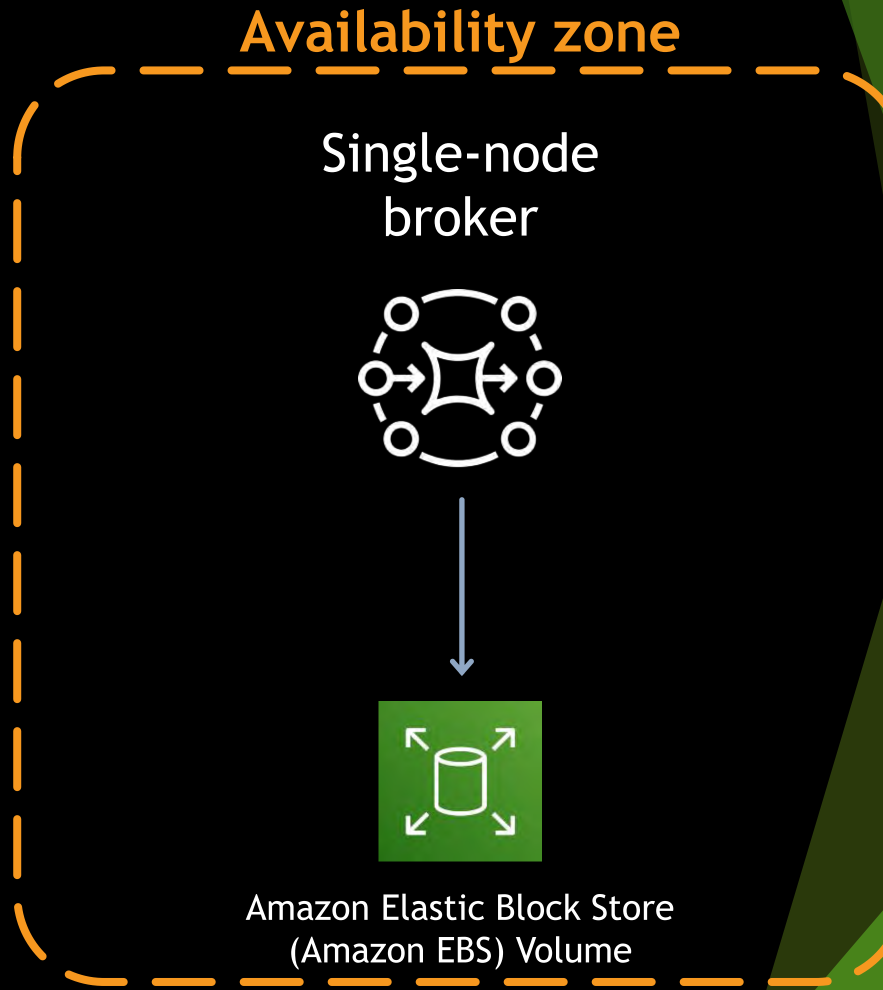
- Maintenance
- Customized maintenance window schedule

A Deeper Look into Amazon MQ for RabbitMQ

- AMQP 0-9-1 and 1.0 support
- Management, Shovel, and Federation plugins installed
- AWS CloudWatch metrics for broker monitoring
- AWS CloudWatch Logs for broker logging

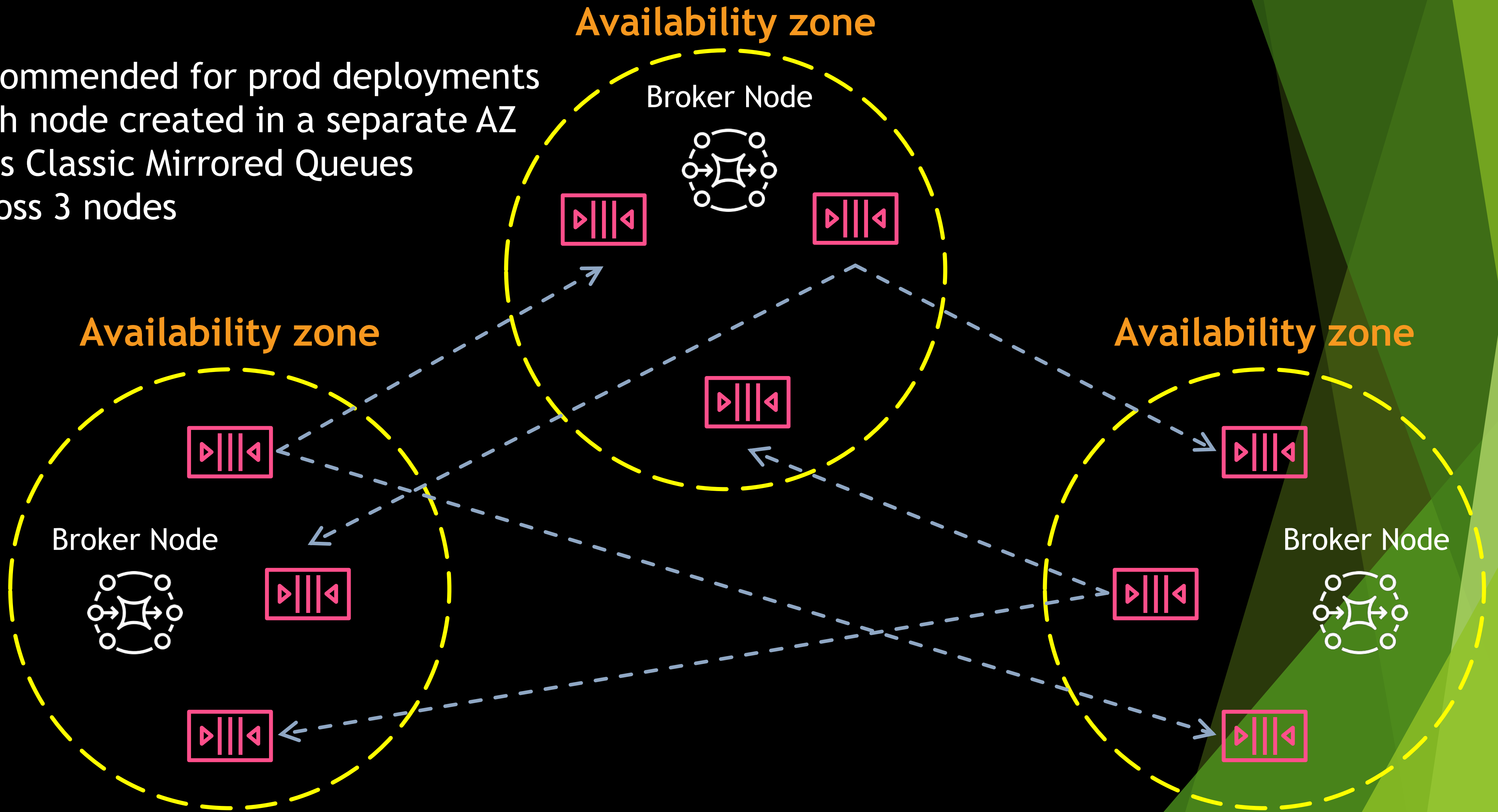
Amazon MQ for RabbitMQ Single-Node Deployment

Recommended for non-production deployments and low-latency use cases with non-persistent messages



Amazon MQ for RabbitMQ Cluster Deployment

- Recommended for prod deployments
- Each node created in a separate AZ
- Uses Classic Mirrored Queues across 3 nodes



Python Example (RabbitMQ)

- Step 1: Setup RabbitMQ Base Client (Common Code Base for Both Publisher and Consumer)

```
import ssl
import pika

class BasicPikaClient:
    def __init__(self, rabbitmq_broker_id, rabbitmq_user, rabbitmq_password, region):
        # SSL Context for TLS configuration of Amazon MQ for RabbitMQ
        ssl_context = ssl.SSLContext(ssl.PROTOCOL_TLSv1_2)
        ssl_context.set_ciphers('ECDHE+AESGCM:!ECDSA')

        url = f"amqps://{rabbitmq_user}:{rabbitmq_password}@{rabbitmq_broker_id}.mq.{region}.amazonaws.com:5671"
        parameters = pika.URLParameters(url)
        parameters.ssl_options = pika.SSLOptions(context=ssl_context)

        self.connection = pika.BlockingConnection(parameters)
        self.channel = self.connection.channel()
```

Python Example (RabbitMQ)

- Step 2: Setup Publisher Client

```
from basicClient import BasicPikaClient

class BasicMessageSender(BasicPikaClient):

    def declare_queue(self, queue_name):
        print(f"Trying to declare queue({queue_name})...")
        self.channel.queue_declare(queue=queue_name)

    def send_message(self, exchange, routing_key, body):
        channel = self.connection.channel()
        channel.basic_publish(exchange=exchange, routing_key=routing_key, body=body)
        print(f"Sent message. Exchange: {exchange}, Routing Key: {routing_key}, Body: {body}")

    def close(self):
        self.channel.close()
        self.connection.close()

if __name__ == "__main__":

    # Initialize Basic Message Sender which creates a connection and channel for sending messages.
    basic_message_sender = BasicMessageSender("<broker-id>","<username>","<password>","<region>")
    # Declare a queue
    basic_message_sender.declare_queue("hello world queue")
    # Send a message to the queue.
    basic_message_sender.send_message(exchange="", routing_key="hello world queue", body=b'Hello World!')
    # Close connections.
    basic_message_sender.close()
```

Python Example (RabbitMQ)

- Step 3: Setup Consumer Client

```
from basicClient import BasicPikaClient

class BasicMessageReceiver(BasicPikaClient):

    def get_message(self, queue):
        method_frame, header_frame, body = self.channel.basic_get(queue)
        if method_frame:
            print(method_frame, header_frame, body)
            self.channel.basic_ack(method_frame.delivery_tag)
            return method_frame, header_frame, body
        else:
            print('No message returned')

    def close(self):
        self.channel.close()
        self.connection.close()


if __name__ == "__main__":

    # Create Basic Message Receiver which creates a connection and channel for consuming messages.
    basic_message_receiver = BasicMessageReceiver("<broker-id>","<username>","<password>","<region>")
    # Consume the message that was sent.
    basic_message_receiver.get_message("hello world queue")
    # Close connections.
    basic_message_receiver.close()
```

Refactor: Moving into Serverless

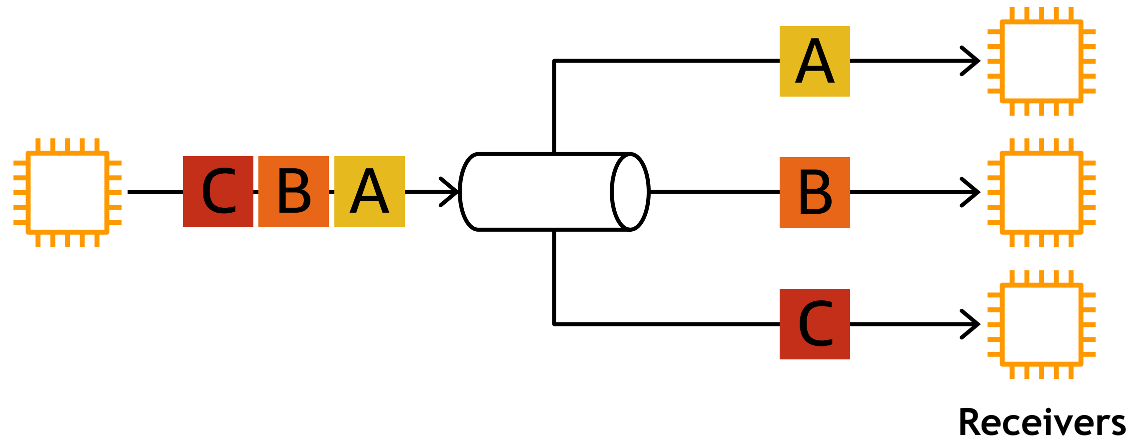
Categorize your messaging system components

- A producer publishes an event to an event router
- An event router filters and sends the events to consumers.
- A consumer receives an event from a router and processes it.
- Producer services and consumer services should be decoupled, which allows them to be scaled, updated, and deployed independently.

Refactor: Moving into Serverless

Understand message delivery model

Point-to-point (queue)

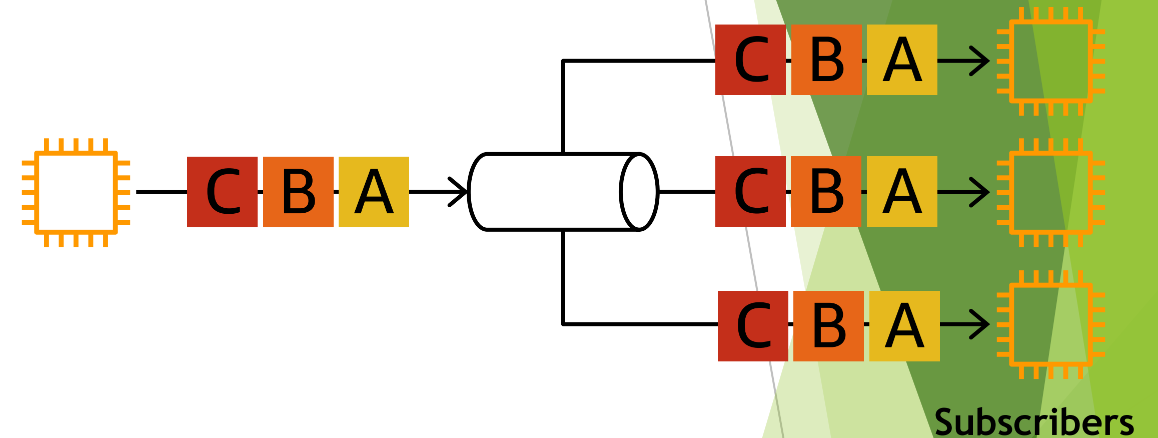


Each message consumed by one receiver

Consumers easy to scale

Buffering can flatten peak loads

Publish-subscribe (topic)



Each message consumed by each subscriber

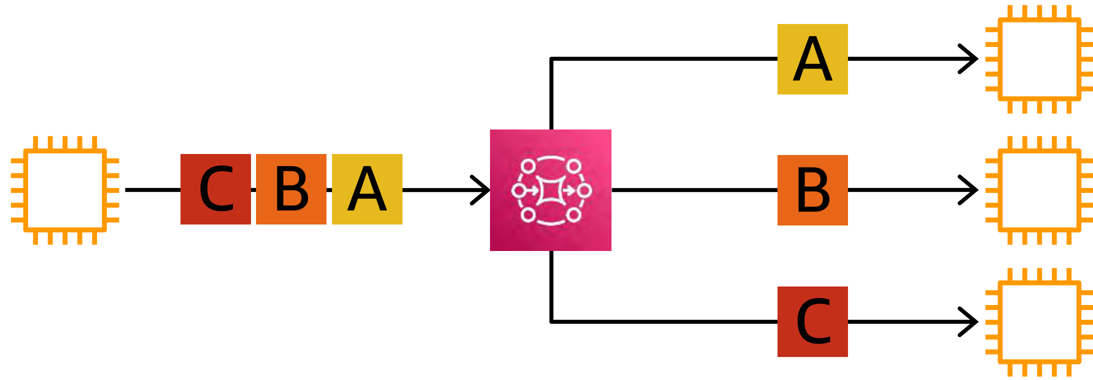
Scale consumers with new subscriptions

Conceptually no buffering

Refactor: Moving into Serverless

Replatform into Managed Service (e.g. Amazon MQ)

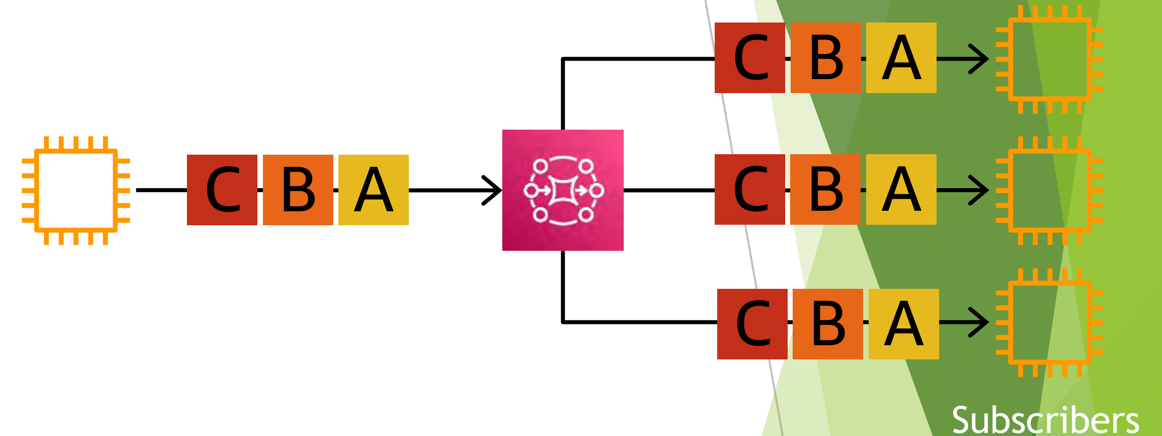
Point-to-point (queue)



Amazon MQ – managed RabbitMQ or Apache ActiveMQ

For applications constrained to industry-standard protocols like JMS or AMQP

Publish-subscribe (topic)



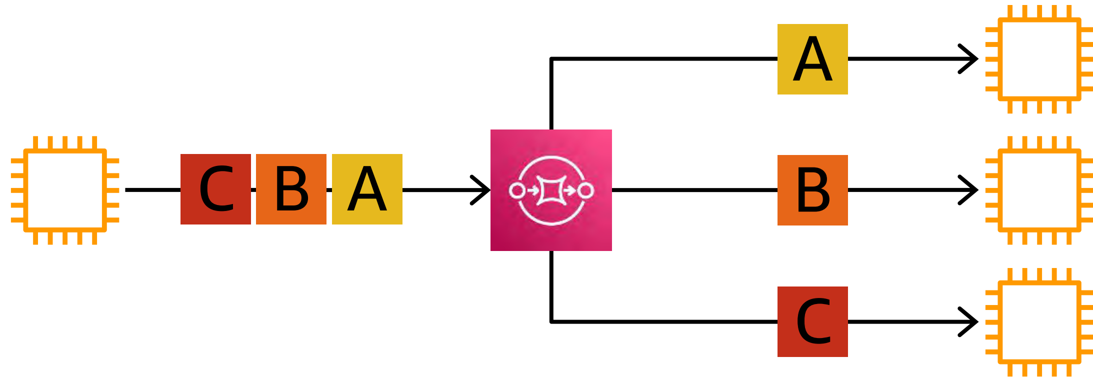
Amazon MQ – managed RabbitMQ or Apache ActiveMQ

For applications constrained to industry-standard protocols like JMS or AMQP

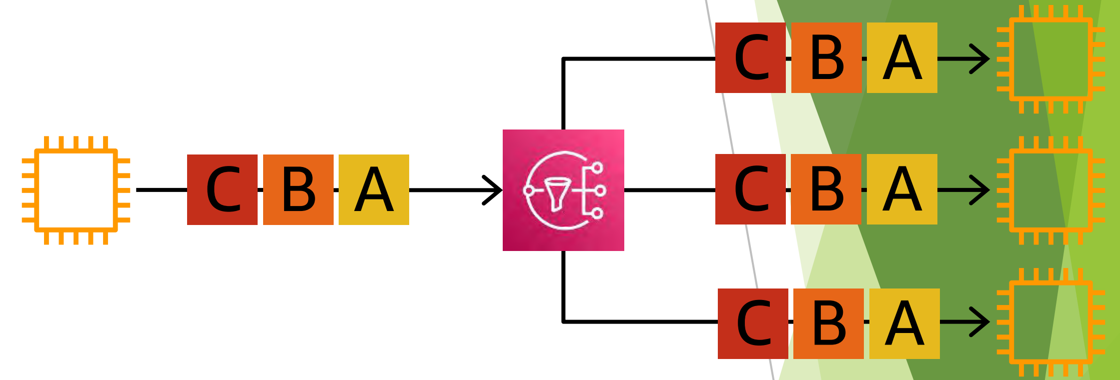
Refactor: Moving into Serverless

Refactor application logic to adopt cloud-native messaging services

Point-to-point (queue)



Publish-subscribe (topic)



Amazon Simple Queue Service (Amazon SQS)

Cloud-native & serverless

Amazon Simple Notification Service (Amazon SNS)

Cloud-native & serverless

Python Example (SQS on Boto3)

- Queue Management (Create, Delete, Get)

```
def create_queue(name, attributes=None):

    if not attributes:
        attributes = {}

    try:
        queue = sqs.create_queue(QueueName=name, Attributes=attributes)
        logger.info("Created queue '%s' with URL=%s", name, queue.url)

    except ClientError as error:
        logger.exception("Couldn't create queue named '%s'.", name)
        raise error

    else:
        return queue
```

Python Example (SQS on Boto3)

- Queue Management (Create, Delete, Get)

```
def get_queue(name):

    try:
        queue = sqs.get_queue_by_name(QueueName=name)
        logger.info("Got queue '%s' with URL=%s", name, queue.url)

    except ClientError as error:
        logger.exception("Couldn't get queue named %s.", name)
        raise error

    else:
        return queue

def remove_queue(queue):

    try:
        queue.delete()
        logger.info("Deleted queue with URL=%s.", queue.url)

    except ClientError as error:
        logger.exception("Couldn't delete queue with URL=%s!", queue.url)
        raise error
```

Python Example (SQS on Boto3)

- Send Message

```
def send_message(queue, message_body, message_attributes=None):

    if not message_attributes:
        message_attributes = {}

    try:
        response = queue.send_message(
            MessageBody=message_body, MessageAttributes=message_attributes
        )
    except ClientError as error:
        logger.exception("Send message failed: %s", message_body)
        raise error
    else:
        return response
```

Python Example (SQS on Boto3)

- Receive Message

```
def receive_messages(queue, max_number, wait_time):
    try:
        messages = queue.receive_messages(
            MessageAttributeNames=["All"],
            MaxNumberOfMessages=max_number,
            WaitTimeSeconds=wait_time,
        )
        for msg in messages:
            logger.info("Received message: %s: %s", msg.message_id, msg.body)

    except ClientError as error:
        logger.exception("Couldn't receive messages from queue: %s", queue)
        raise error

    else:
        return messages
```

Python Example (SQS on Boto3)

- Delete Message

```
def delete_message(message):

    try:
        message.delete()
        logger.info("Deleted message: %s", message.message_id)
    except ClientError as error:
        logger.exception("Couldn't delete message: %s", message.message_id)
        raise error
```

Refactor: Moving into Serverless

Refactor Considerations

- Protocol Support – from AMQP/WebSocket/OpenWire/STOMP/MQTT to HTTPS/JMS based communication
- Message Order Preservation/FIFO: Standard vs. FIFO type of Queues/Topics
- AWS CloudTrail Logs* for Request-Level Logging
- Native integration with Authentication/Authorization through IAM/Identity Center
- Large Payload Extension for JAVA and Python (Up to 2GB per message, through S3)

Key Factors to Consider…

- Nuances of each technology
- Authentication/Authorization
- Deployment
- Operations and monitoring

In Summary:

- By migrating messaging system to the cloud, the overall architecture becomes more resilient and highly available
- There are multiple stages for migration
- A successful migration requires changes at both application level and business/operations level

THANK YOU