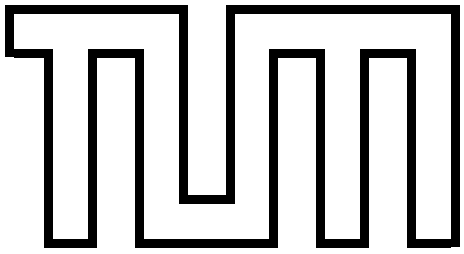


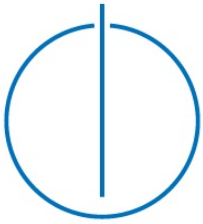
# Technische Universität München

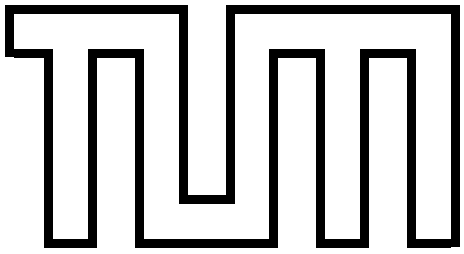
## Fakultät für Informatik

**Master's Thesis in Informatik**

PUBLISH/SUBSCRIBE SYSTEMS BASED ON SERVERLESS COMPUTING

Faisal Hafeez





# Technische Universität München

## Fakultät für Informatik

**Master's Thesis in Informatik**

PUBLISH/SUBSCRIBE SYSTEMS BASED ON SERVERLESS COMPUTING

PUBLISH/SUBSCRIBE SYSTEMS BASIEREND AUF SERVERLESS COMPUTING

**Author:** Faisal Hafeez

**Supervisor:** Prof. Dr. Hans-Arno Jacobsen

**Advisor:** Pezhman Nasirifard

**Submission:** 21.11.2018

I confirm that this master's thesis is my own work and I have documented all sources and material used.

München, 21.11.2018

*(Faisal Hafeez)*

# Abstract

Building reliable and scalable publish/subscribe (pub/sub) systems require tremendous development efforts. The serverless paradigm simplifies the development and deployment of highly available applications by delegating most of the operational concerns to the cloud providers. It describes a programming model, where the developers break the application downs into smaller microservices which run on the cloud in response to events. In this paper, we propose a design of a serverless pub/sub system based on the Amazon Web Services Lambdas and Microsoft Azure Functions. Our pub/sub system performs topic-based, content-based and function-based matching. The function-based matching is a novel matching approach where the subscribers can define highly customizable subscription function which the broker applies to the publications in the cloud. We developed an evaluation application which can create different scenarios involving multiple numbers of publishers and subscribers. The scenarios are evaluated on different cloud platforms which verify the correctness and scalability of designed pub/sub broker.

# Inhaltsangabe

Der Aufbau zuverlässiger und skalierbarer Publish / Subscribe (Pub / Sub) -Systeme erfordert enorme Entwicklungsanstrengungen. Das serverlose Paradigma vereinfacht die Entwicklung und Bereitstellung hochverfügbarer Anwendungen, indem die meisten betrieblichen Bedenken an die Cloud-Anbieter delegiert werden. Es beschreibt ein Programmiermodell, bei dem die Entwickler die Anwendungs-Downs in kleinere Microservices unterteilen, die als Reaktion auf Ereignisse in der Cloud ausgeführt werden. In diesem Dokument schlagen wir ein Design eines serverlosen Pub / Sub-Systems vor, das auf den Lambdas von Amazon Web Services und den Microsoft Azure-Funktionen basiert. Unser Pub / Sub-System führt themenspezifische, inhaltsbasierte und funktionsbasierte Übereinstimmungen durch. Das funktionsbasierte Matching ist ein neuartiger Matching-Ansatz, bei dem die Abonnenten eine stark anpassbare Abonnementfunktion definieren können, die der Broker auf die Publikationen in der Cloud anwendet. Wir haben eine Evaluierungsanwendung entwickelt, die verschiedene Szenarien mit einer Vielzahl von Publishern und Abonnenten erstellen kann. Die Szenarien werden auf verschiedenen Cloud-Plattformen bewertet, die die Korrektheit und Skalierbarkeit des entworfenen Pub / Sub-Brokers überprüfen.

# Acknowledgment

I would like to thank and express my gratitude to my advisor Pezhman Nasirifard. He provided his full support and steered me in the right direction during the course of my master thesis.

My acknowledgment would be incomplete without thanking my parents Muhammad Hafeez-ul-Naseer & Rubeena Hafeez, and my brothers Junaid & Jawad for providing me with unfailing support and continuous encouragement throughout my years of study. I would also like to thanks all of my friends at TUM who made university experience even more enjoyable. I would like to especially thank Sherjeel Sikander, Viktoriia Bakalova, Yesika Marlen Ramirez Cardenas, and Salma Farag Galal for making university life easy and fun.

# Contents

# List of Figures

# List of Tables

# Listings

# Chapter 1

# Introduction

## 1.1 Motivation

In the last decades, the influence of computers and mobiles on our lives has increased a lot. A lot of new programming paradigms are introduced and have created a lot of impact on both software and hardware level. It has also changed how we interact with computers and mobile devices these days.

Networking has begun a new era of computing. Earlier computers were used to work individually and were not that common to interact with other computers. But with the progress in networking, more and more applications focus on working in a distributed environment. Distributed computing paradigm allows different machines to work together and share their resources. Distributed systems also allow interaction of computers located in remote locations.

In the past decade, the cloud computing paradigm has shown a lot of growth and has created a big impact on distributed systems. Cloud computing provides a pool of configurable resources (storage, database, computational power, etc.) over the internet. Cloud providers take care of most of the management operations. With the evolution of cloud computing, many of the enterprises are moving away from architecture that is dependent on physical and on location servers. Amazon Web Services (AWS) and Microsoft Azure have the major market share in cloud computing [1]. While Google and IBM also have their own cloud solutions in the market, but with a limited market share. All cloud providers provide a lot of configurable services which developers can use in their applications.

## 1.2 Problem Statement

With the advent of distributed systems and cloud computing, messaging between devices plays an important part in the application flow. Messaging pattern is an architectural pattern which describes how systems/modules connect and communicate to each other. The Publish-Subscribe (pub/sub) pattern is a messaging pattern widely used in enterprise applications. It provides a middleware, known as the broker, that is responsible for communication between publishers and subscribers.

Pattern achieves loose coupling, flexibility, and clean design. Publishers and subscribers are unaware of each other and work separately. However, pattern's biggest advantage is also one of its disadvantages. Pattern's main goal is to ensure successful delivery of messages to subscribers. But due to loose coupling, middleware might not be able to identify successful delivery of messages. Middleware cannot ensure the state of both ends. Publications might not be sent to subscribers if they are not online. With the increase of publishers and subscribers, the number of messages increases over the network which may create some instabilities. Middleware applications can be scaled, but it requires extra effort to configure and manage resources effectively. Providing a highly scalable and effective pub/sub system is a big technical challenge.

The serverless computing represents a programming model where the developers decompose the applications into microservices, also known as functions, and deploy these functions to the cloud. And cloud providers are responsible for execution, automatic maintenance and scaling the resources. While serverless paradigm provides has several advantages, short-lived nature and statelessness nature of functions introduce new challenges in development.

### 1.2.1 Questions

In this thesis, we want to develop a pub/sub system using serverless platform which will answer the following questions.

- How to improve the scalability of the broker?
- How to overcome statelessness nature of serverless functions? How to store information in the broker?
- How to ensure successful delivery of publications to subscribers? How to handle and send publications to offline subscribers?

- Is it provided solution feasible enough to be used in the real enterprise environment? What are the benefits and limitations of the solution?

## 1.3 Approach

In the serverless computing paradigm, application features are decomposed into smaller microservices, and these services are deployed on the cloud. To build a pub/sub system using the serverless platform, we need to identify which features will be moved to the cloud and which features will remain in the client application. In pub/sub model, the broker is the main entity which is responsible for handling communication between publishers and subscribers. So broker's whole functionality is divided into smaller functions and is moved on the serverless platform. There are two categories of these functions *Publisher Functions* and *Subscriber Functions*. Publishers and subscribers can invoke these functions via REST API. Subscribers can call these functions to register themselves to the broker and can subscribe to different kind of subscriptions. While publishers can submit publications. The broker will perform the matching of publications and will send to subscribers.

The broker needs to store information about subscribers and their subscriptions. Since serverless functions are stateless, so it is not possible to store information in them. To overcome this limitation, external services and resources should be used to complete broker functionality. And since broker is built on different cloud providers, so we need to be careful in choosing these resources to make sure similar kind of resources are available on different cloud providers. For pub/sub model, we need two additional resources. First one is messaging queues which will be used to send publications to subscribers. The second one is to store subscribers information along with their subscriptions. To store this information, databases services are required. These resources will be accessed from different serverless functions of the broker.

In pub/sub topic-based and content-based matching are performed by the broker to send publications to the subscribers. We have introduced a new kind of matching known as function-based matching. In function-based matching, subscribers provide their own functions which are executed by the broker for subscription matching. The output of function decides whether a subscriber will receive the publication or not. To achieve function-based matching, we will develop some custom functions and will host these functions on a cloud platform. Subscribers can use these function for function-based matching.

For visualizing results, an evaluation application will be built. The application will be able

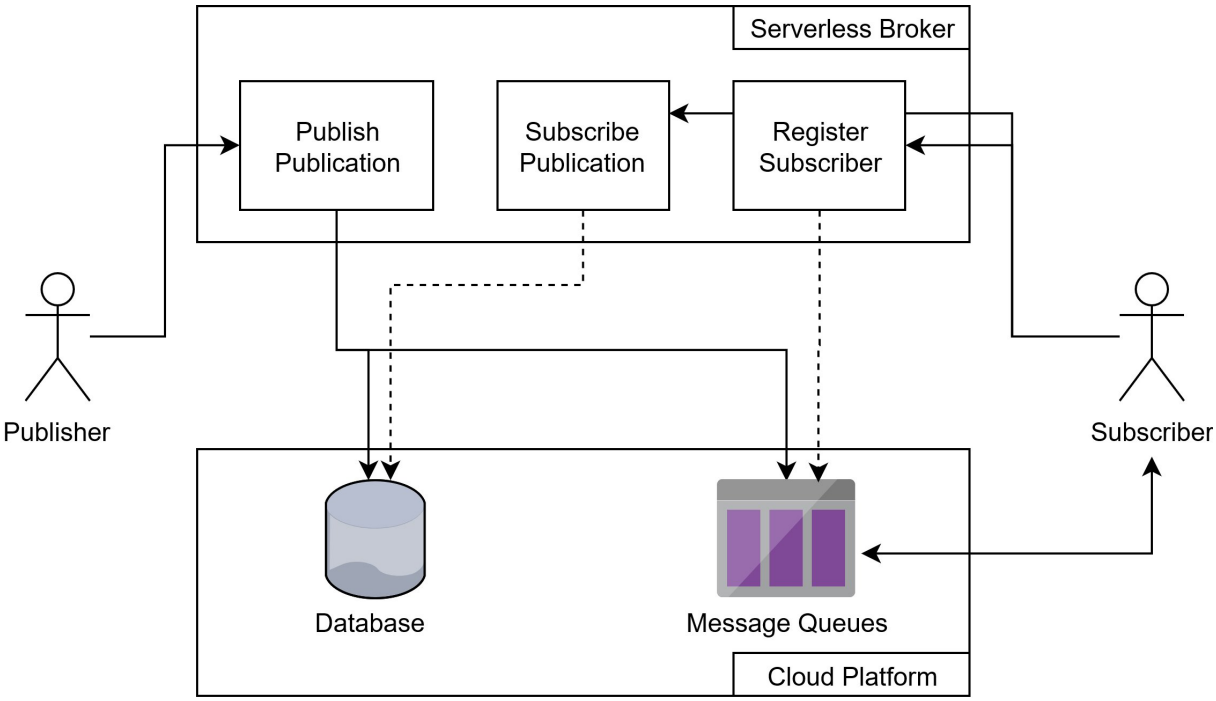



**Figure 1.1:** System Component Diagram

to create a simulation environment of publishers and subscribers via a User Interface (UI) or a configuration file. The application will provide functionality for both subscribers and publishers to communicate to the broker. Publishers will be able to send publications. Subscribers will be able to register to the broker and will be able to register different kind of subscriptions. The application will be used to test the correctness of the system. It will also be used to test the scalability of different cloud platforms.

## 1.4 Organization

Chapter 2 discusses a brief overview of pub/sub system, serverless computing, and microservices architecture.

Chapter 3 will share related work in the field of pub/sub systems and serverless computing paradigm.

Chapter 4 discusses the architecture of solution and technologies used to build pub/sub system using the serverless platform.

Chapter 5 contains the implementation of system. It gives detail of serverless functions, custom functions and evaluation application.

Chapter 6 evaluates the whole system. It discusses the use of an evaluation application and provides a comparison of the different serverless cloud platform.

Chapter 7 summaries our work. It discusses the current state of the system and describes possible future work.

# Chapter 2

# Background

## 2.1 Publish/Subscribe Pattern

The Publish-Subscribe (pub/sub) pattern is an architectural message based pattern. It describes how components connect to each other and communicate with each other. It consists of three components.

- Publishers
- Broker
- Subscribers

Components are loosely coupled with each other. Publishers and subscribers both are unaware of each other and don't communicate to each other directly. Subscribers register themselves to the broker and subscribe to different type of subscriptions. Broker maintains a list of subscribers and their subscriptions. Publishers send messages, also known as publications, to the broker along with publication type. Whenever broker receives a publication from the publisher, broker matches publication type with the saved subscriptions. If matching is successful, then broker sends the publication to the subscribed subscriber. Fig 2.1 shows a basic model overview of the publish-subscribe pattern.

pub/sub system enjoys the benefits of loose coupling as publishers are loosely coupled to subscribers. Broker design ensures separation of concerns and hence improves testing of system components. But it has drawbacks as well. Its biggest advantage is also one of its biggest disadvantage. Due to loose coupling, the broker might not be able to ensure successful delivery of publications to the subscribers. Publisher is unaware if publication

is received by subscribers or not. Subscription management adds extra overhead which can increase the latency of publication exchange between publishers and subscribers and hence effects performance.

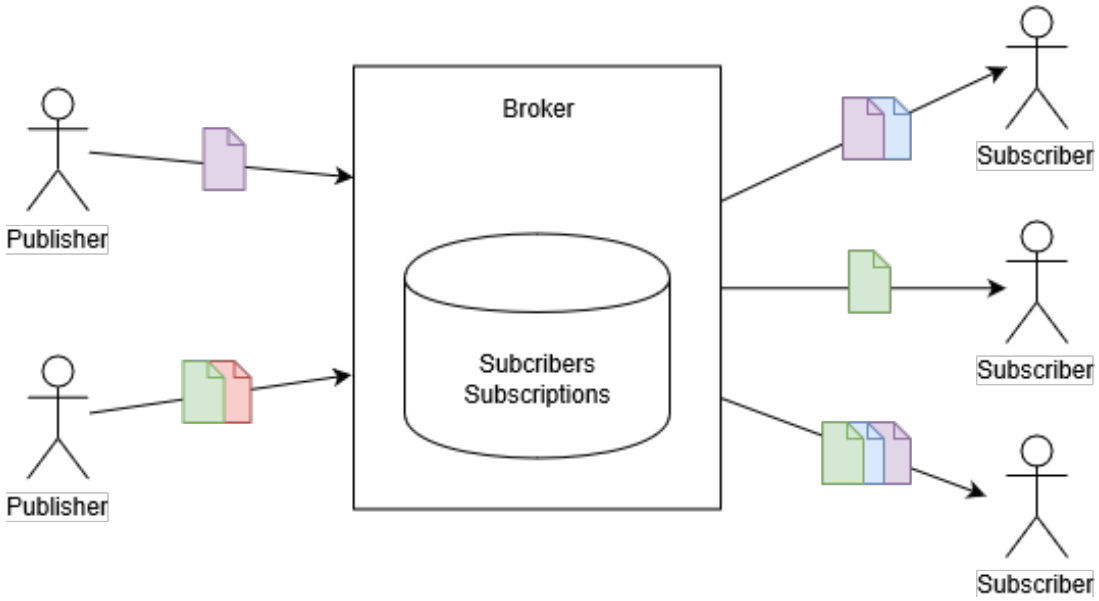



**Figure 2.1:** Simple Publish Subscribe Model

### 2.1.1 Publication matching

In the pub/sub model, subscribers don't receive all publications. They only receive a subset of publications. They subscribe to subscriptions and only receive publications for these subscriptions. This process is called publication matching. Two most common types are topic-based matching and content-based matching. We have introduced a new type of matching known as function-based matching.

#### Topic-based Matching

In topic-based matching, subscribers provide a list of *topics* to the broker. This information is saved by the broker. When a publisher publishes a publication, it also provides a list of *topics* associated with the publication to the broker. Broker matches the list of topics associated with the publication with the topics provided by subscribers. If matching is successful, then publication is sent to the subscriber.

#### Content-based Matching

In content-based matching, subscribers upon subscription provides a list of *values* and *conditions* pair. When a publication is received, the broker applies value-condition to the publication. If publication data matches the condition, then the broker sends the publication to the subscriber.

#### Function-based Matching

Function-based matching is a novel idea which allows the subscriber to provider their own custom logic for matching a publication. The subscriber provides a custom function to the broker. This function can be invoked by the broker. Whenever a publication is published by the publisher, the broker invokes the custom function with the publication provided by the publisher. The custom function runs the custom logic. If the publication fulfills custom function criteria, then the broker passes the publication to the subscriber.

## 2.2 Monolithic Architecture

Monolithic architecture is one of the traditional design models in software engineering. In monolithic architecture everything is composed in one piece. In monolithic applications, everything is designed as self-contained. Components are connected and dependent on each other and are tightly coupled. They are deployed and packaged as a monolith. It is also known as single-tiered software applications which are designed without modularity.

### 2.2.1 Advantages

- Easy to develop.
- Easy to test. End to end test can be executed easily.
- Simple to deploy as everything is bundled in one package.
- They benefit horizontal scaling as whole package can be replicated.

### 2.2.2 Disadvantages

- Have bigger size due to one package.

- Continuous deployment is not easy.
- Deploy whole package on update
- Dependency requirements of different modules
- Difficult to upgrade frameworks/technologies used in modules

Figure 2.2 shows monolithic application structure. It can be seen that all services are hosted in a single process (monolith). For scaling purposes, the monolith is replicated on different servers.

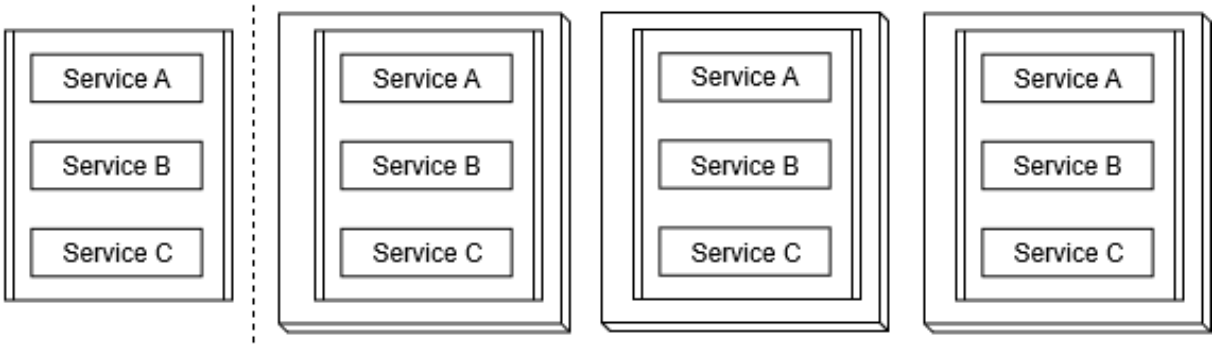



**Figure 2.2:** Monolith Architecture

## 2.3 Microservice Architecture

Microservice architecture is an approach to software development where the application is built using small separable modules, also known as services. Each service is responsible for a specific task and is exposed via an interface. These services are deployed separately and run in their own processes. It supports loose coupling, separation of concerns, and decentralized approach.

### 2.3.1 Advantages

- Decomposes big logic into smaller units (services)
- Services can be deployed independently
- Services can be used by different systems
- Only selected service is updated upon update

- Easy to use new technologies in services

### 2.3.2 Disadvantages

- Adds extra complexity to the system.
- Difficult to test whole system.
- Adds overhead as services has to communicate to each other.
- Difficult to implement changes which span different services.
- Difficult to deploy and effort required for maintenance.

Figure 2.3 shows microservice application architecture. All services are hosted on different processes. And microservices are scaled by distributing different services on different servers. These services are not equally replicated. They are replicated depending on the usage and requirement.

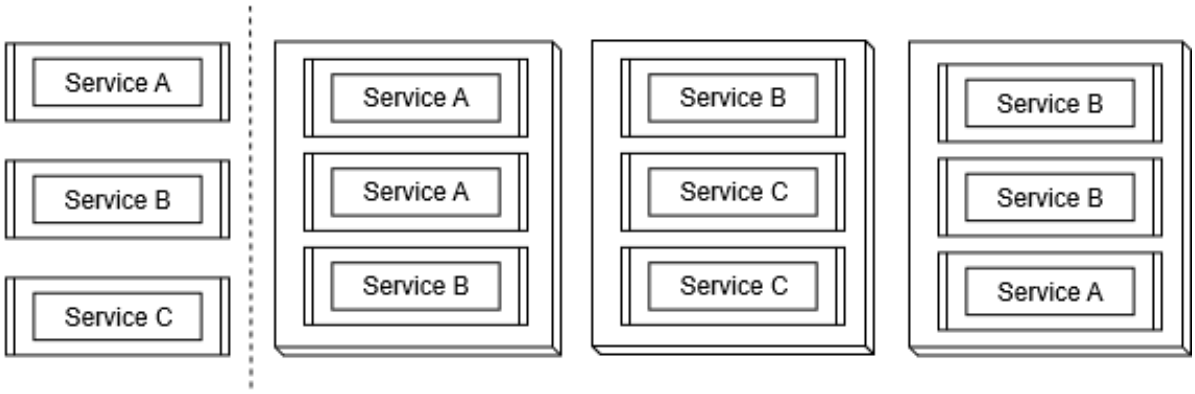



**Figure 2.3:** Microservice architecture

## 2.4 Serverless Computing

Serverless computing, also known as Function as a Service (FAAS), is a cloud computing paradigm where developer breaks down their application into smaller microservices, also known as functions, and deploy these functions to the cloud platform.

Cloud providers take responsibility for the management of all resources. Cloud providers take care of resource allocation, dynamic scaling, availability, etc. of resources. It completely hides the underlying infrastructure from developers and hence removes a

significant amount of effort from the developers end. Provider allocates resources on the fly, and pricing is based on usage of resources by the application. Serverless computing is highly compatible with microservice architecture.

Figure 2.4 shows a simple workflow of serverless computing. A trigger is required to invoke serverless functions. Once a trigger is invoked, resources are allocated for the function. Different cloud providers provide different types of triggers. The most widely used trigger is via Application programming interface (API) Gateway. Each function is bound to an endpoint. Users make a call to the API endpoint. After this function gets executed. These function may update some resources upon execution. Once functions complete their execution, the response is sent to users. Resources are deallocated by cloud providers if these resources are not used for a specific amount of time.

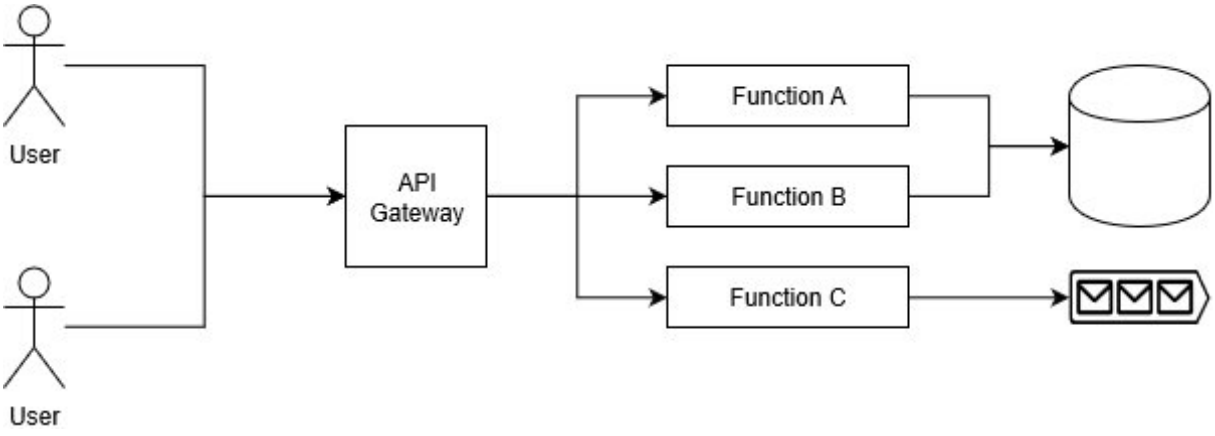



**Figure 2.4:** Serverless computing Workflow

Amazon, Microsoft, Google, and IBM all provide serverless platforms. Amazon was the pioneer in serverless computing and introduced Amazon Web Services (AWS) Lambda in 2014. Soon other providers came up with their own serverless platform. The serverless platform provides a new way of developing and deployment of microservices. Organizations are trying to move their existing applications to the serverless platform, but it is not easy to migrate legacy applications to the serverless platform. Applications need to be redesigned and refactored before moving to the serverless platform.

### 2.4.1 Advantages

#### Easy Deployment

Serverless computing removes the pain of managing resources from the developer end. Developer focus only on their functions. It is quite easy to deploy and update functions to the serverless platform.

#### Scalability

Cloud providers take care of the scalability of resources. Serverless applications can automatically scale up/down according to the usage of service.

#### Cost

Serverless computing works on a pay per use model. Cloud provider allocates resources only when a function is executed. So customers only pay when their function is executed. They only pay for the resources which application consumes. It saves from paying for Virtual Machines (VMs) or containers, which are up and running all the time but are not being used by applications.

### 2.4.2 Disadvantages

#### Statelessness

Serverless functions are stateless and short-lived of nature. Additional resources are required to maintain state. Performance is affected when accessing these additional resources. Serverless functions support caching, but it requires additional setup and may not be viable and reliable.

#### Vendor Lock-in

Every cloud provider has their own features for the serverless platform. Each provider has their own configuration, interactions, triggers, etc. So it is not easy to switch between vendors. Moving from one provider to another provider will require additional effort. And in some cases, it can be impossible to move from one cloud provider to another.

### Debugging

Since resource are allocated on the fly, and developers don't have access to the running environment, so it is quite difficult to test and debug functions in the production environment. Although cloud providers provide monitoring and logging, still proper debugging tools are either not available or not good enough.

### Performance

The serverless platform can have different performance depending on the state of resources. Since provider deallocates resources if there is no usage, so cold start can be a problem for some applications.

# Chapter 3

# Related Work

Serverless computing paradigm is quite new, and there are quite limited studies on building a pub/sub model using serverless computing. However, there are some studies related to the performance of serverless computing.

McGrath et al. [2] worked on the design, implementation, and performance of the serverless platform built on Microsoft Azure using .NET. They focused on the challenges of development, deployment, scaling, and lifecycle of functions. They also compared results using different serverless providers with different metrics.James [3, 4] performed a deep comparison of AWS Lambdas and Azure Functions. He concluded that AWS Lambdas perform better than Azure Functions. However, Microsoft worked on the performance of Azure Functions and results became better with updates.

In April 2018, Lloyd et al. [5] also worked on the factors that influence microservice performance in the serverless platform. They examined hosting implications related to infrastructure elasticity, load balancing, provisioning variation, infrastructure retention, and memory reservation size. They identified different states of the process execution and how it affects the performance of serverless functions.

Fourth International Workshop on Serverless Computing (WoSC) 2018 [6] will take place on December 21, 2018. Researchers have worked on different aspects of serverless computing, and these results will be shared in the workshop. Das et al. worked on the benchmarking of Edge computing using Amazon AWS Greengrass and Azure Internet of Things (IoT) Edge. Manner et al. will discuss on the impact of the cold start in functions as a service. Similarly, Jackson et al. investigated the impact of language runtime on the performance of serverless functions.

Some use cases are also built on the serverless platform, and there are studies which

investigate different aspects of the serverless platform for a big scale application development.

Fouladi et at. [7] present a video processing system for executing thousands of functions in parallel to perform edit, transform, and encode a video with low latency. They implemented a video encoder which allows computations to divided into thousands of smaller tasks which can run in parallel. Their results indicate that by using serverless architecture, these small tasks can be executed in parallel and can decrease the overall latency of these tasks. Another application of serverless as a practical tool is presented by Yan et al. [8]. They built a chatbot using the serverless platform. They made a sequence of functions for the backend on the serverless platform. These functions coordinate with each other and communicate with external devices. Their results indicate improvements in extensibility of chatbot with different kind of bot services.

Currently, all serverless platform are stateless and short-lived. It is an open problem to persist state in serverless functions. [9]. Hendrickson et al. [10] proposed a solution of stateless nature of serverless computing. They suggested using cookies to persist the state of sessions. They also used a database to store the state of sessions. Similarly, Spillner [11] also suggest using stateful services such as key-value stores and file storages to persist the state of serverless functions. Both recommended using Database-as-a-service to persist state in serverless paradigm.

Monolith and Microservice architecture are the two traditional architectural patterns used for server-side application development. Lewis [12] and Kharenko [13] discussed both architectures in detail. They compared different aspects like deployment, development, scaling, etc. of these architectures. Both architectures have their own advantages and disadvantages. They discussed the factors which should be considered when developing a complex application.

Migrating existing applications to the serverless platform is not quite easy. Existing applications are not built with microservice architecture in mind. Cui [14, 15] and Villalba [16] discussed some important things to be considered while migrating existing applications to the serverless platform.

Some studies have been done on making the pub/sub model using cloud computing paradigm. Happ et al. [17] built a pub/sub model for cloud-based IoT devices and discussed its implications. They discussed the challenges of the discovery of subscribers and the successful delivery of information to a certain number of subscribers. Nasirifard et al. [18] built a pub/sub broker that performs topic-based and content-based matching using IBM Cloud Functions. They used CloudantDB and Watson IoT to maintain state

and delivery of publications to subscribers.

Amazon introduced Pub/Sub Messaging [19] which allows developers to set up a pub/sub model on AWS. And recently they introduced Amazon Simple Notification Service (SNS) [20] which is a fully managed pub/sub messaging services built on Amazon Lambdas [21]. They both provide topic-based matching and use Amazon Simple Message Queues [22] to send publications to subscribers. Microsoft also introduced Service Bus [23] which is used to send messages between different entities.

Both Amazon and Microsoft introduced services which can be used to make a customizable pub/sub model on the serverless platform. We extend Nasirifard's work to AWS Lambdas and Azure Functions. We use designated services in our pub/sub model to get better results. Also, we introduce new function-based matching in the broker. Function-based matching is a new concept which allows a subscriber to provide their own custom function for publication matching. We also used Amazon Web Services and Microsoft Azure serverless providers to overcome the limitations of IBM Cloud Functions.

# Chapter 4

# Architecture

## 4.1 Architecture

The project architecture can be divided into three layers. These layers are the presentation layer, serverless layer, and data layer. Figure 4.1 shows a basic overview of architectural components of the system.

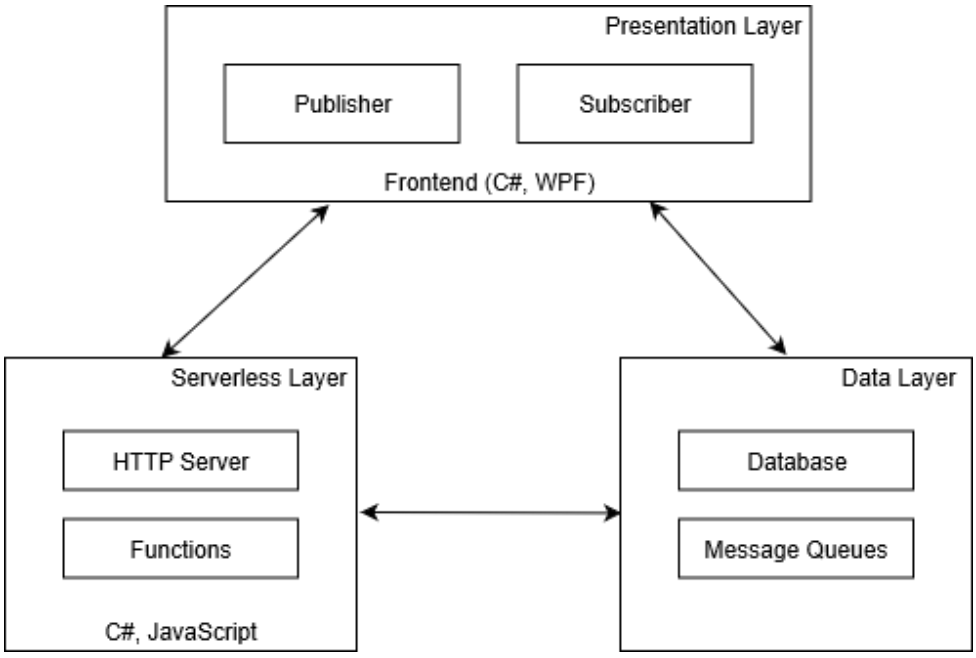



**Figure 4.1:** Basic Application Architecture

### 4.1.1 Presentation Layer

The presentation layer consists of a user interface for publishers and subscribers. This interface provides commands for publishers and subscribers to interact with the broker. Publishers can send publications to the broker. Publishers only communicate with the serverless layer. Subscribers can register to the system and can register for subscriptions. Subscribers also can view publications when received from message queues. Subscribers communicate both with the serverless layer and the data layer.

### 4.1.2 Serverless Layer

The serverless layer is the main layer of the system. It hosts pub/sub broker and has two main components: serverless functions and HTTP REST API. Functions implement the core functionality of a broker. These functions can be accessed by the presentation layer via HTTP REST API. Each function is linked with one REST endpoint. Serverless layers communicate with the data layer to store information and to publish publications.

Cloud providers support multiple languages for development of serverless functions.

### 4.1.3 Data Layer

The data layer is used to store states of serverless functions. It is composed of two components. The first component is a database which is used to store subscribers and their subscription information. The second component is messaging queues. Messaging queues are used to send publications to subscribers. Every subscriber has their own dedicated messaging queue.

## 4.2 Resources

Amazon and Microsoft are two of the biggest cloud platform provider. We have built a pub/sub model on both providers. As discussed in the architecture section, the pub/sub model requires three main services to function properly. Table 4.1 shows the services we used to build a serverless pub/sub model.

| Cloud Provider | Serverless Platform | Database | Message Queue |
|---|---|---|---|
| Amazon Web Services | Lambdas | DynamoDB | Simple Queue Service |
| Microsoft Azure | Functions | Tables | Service Bus Namespace |

**Table 4.1:** Cloud providers and services

### 4.2.1 Serverless Platform

The serverless broker is built using AWS Lambdas and Azure Functions. Both providers support multiple languages and allow a serverless project of functions written in different languages. They also support environment variables which can be used to save metadata information related to functions.

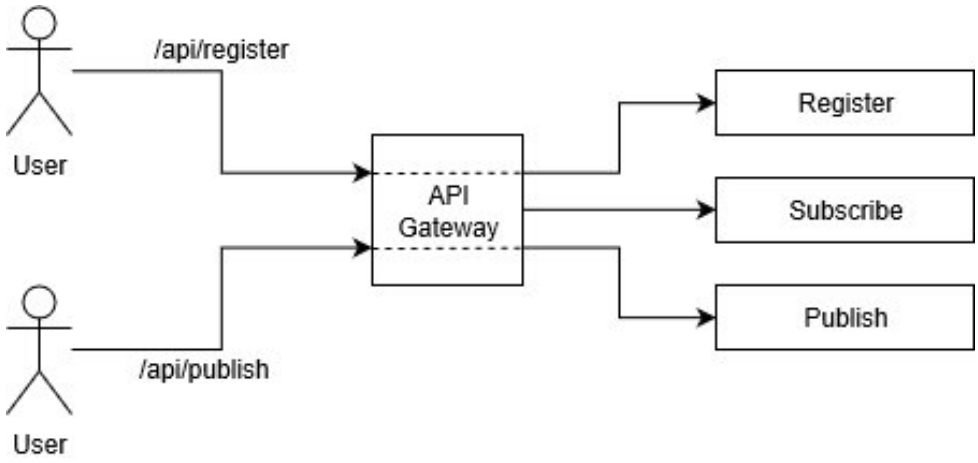



**Figure 4.2:** Serverless functions workflow

#### AWS Lambdas

AWS Lambdas can be developed either using online console [24] or locally using AWS Command Line Interface (CLI) [25]. Once a function is uploaded to server Console or CLI can be used to test out these functions. AWS recently introduced beta tools for testing lambdas locally [26].

Lambdas are invoked by triggers. AWS provides different kinds of triggers, but for our workflow, we need HTTP triggers. We used API Gateway [27] to generate public endpoints and associated these endpoints with different lambdas. Publishers and subscribers can invoke lambdas using these endpoints.

#### Azure Functions

Similar to Lambdas, Azure functions can be developed using online console [28] or locally using Visual Studio [29]. Microsoft provides a great debugging environment for Azure Functions. It deploys all serverless functions on the local machine and can be used to test the functionality.

Azure functions also need HTTP trigger to invoke functions, but they don't require additional steps like making API Gateway in AWS Lambdas. Upon function creation, Azure itself create a public endpoint for the function.

### 4.2.2 Database

To store information of subscribers and their subscriptions we need a database. For AWS, we used DynamoDB [30] and for Azure we used Table Storage [31]. Both database are NoSQL databases. For each type of matching, we have one table. Table schema is same for both DynamoDB and Table Storage.

#### Topics Table

For topic-based matching, we have *Topics* table. Each data entry stores information of one subscription. Listing 4.1 shows table schema for topics table.

```
[DynamoDBTable("topics")]
public class TopicTable
{
    [DynamoDBHashKey]
    public string TopicName { get; set; }

    [DynamoDBRangeKey]
    public string QueueId { get; set; }
}
```

**Listing 4.1:** Topics Table Schema

It consists of only two columns. *QueueId* stores information of subscriber's queue and *TopicName* and stores the name of the topic which subscriber subscribed. Both columns are primary keys which enable multiple topic subscriptions for subscribers.

### Content Table

*Content* table stores information for content-based matching. Listing 4.2 shows database schema of Content database table.

```
[DynamoDBTable("content")]
public class ContentTable
{
    [DynamoDBHashKey]
    public string ContentKey { get; set; }


    [DynamoDBRangeKey]
    public string QueueId { get; set; }

    public string Value { get; set; }

    public string Condition { get; set; }
}
```

**Listing 4.2:** Content Table Schema

*QueueId* stores information of subscriber's queue id and *ContentKey* stores subscription key. *Value* and *Condition* are the constraints which are required by broker to apply content-based matching. *ContentKey* and *QueueId* are primary keys for the table.

### Functions Table

*Functions* table stores information for function-based matching. Listing 4.3 shows database schema of functions database table.

```
[DynamoDBTable("functions")]
public class FunctionsTable
{
    [DynamoDBHashKey]
    public string SubscriptionTopic { get; set; }

    [DynamoDBRangeKey]
    public string QueueId { get; set; }

    public string FunctionType { get; set; }
```

```
    public string MatchingFunction { get; set; }
}
```

**Listing 4.3:** Functions Table Schema

*QueueId* stores information of subscriber's queue id and *SubscriptionTopic* and stores the topic of the subscription. *FunctionType* the type of function subscriber provides during subscription. *MatchingFunction* store the url of custom function provided by the subscriber. *SubscriptionTopic* and *QueueId* are primary keys for the table.

### 4.2.3 Message Queues

Broker uses message queues to send publications to the subscribers. Every subscriber has their own dedicated message queue. Message queues are hosted on the cloud platform and save publications to the queues. For AWS, we used Simple Queue Service. And for Azure, we used Service Bus namespace.

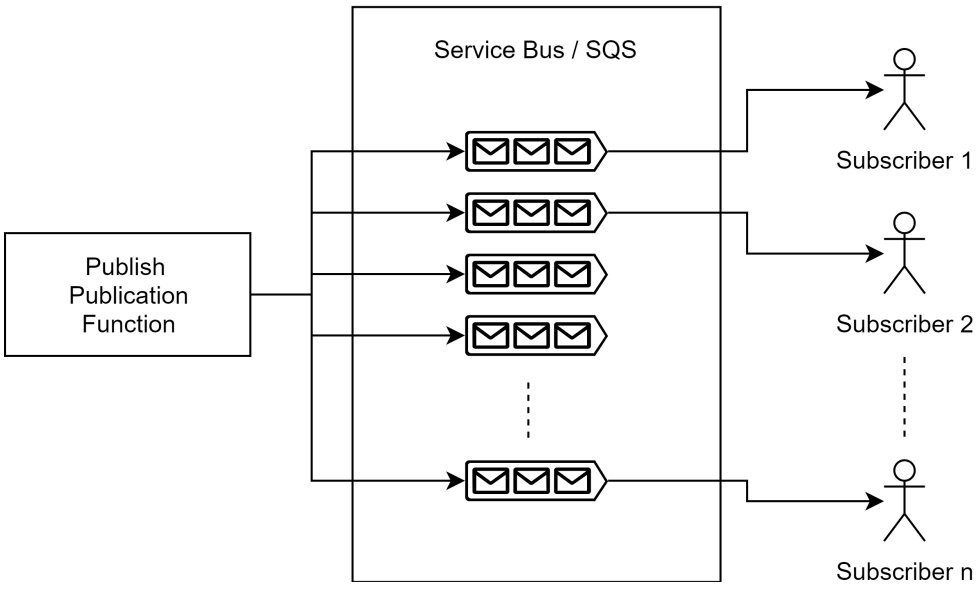



**Figure 4.3:** Message Queues Workflow

### 4.2.4 Web API

For function-based matching, subscribers need to provide their own custom functions. We made some custom functions which subscribers can use in the function-based matching

scenario. These functions are accessible to the broker and take some predefined parameters as input. We used Azure API Apps [32] to host these custom functions.

### 4.2.5 Evaluation Application

We built a windows application using Windows Presentation Foundation (WPF) to simulate the pub/sub model. The application is capable of creating publishers and subscribers. The application is used to test the validation and performance of the whole system.

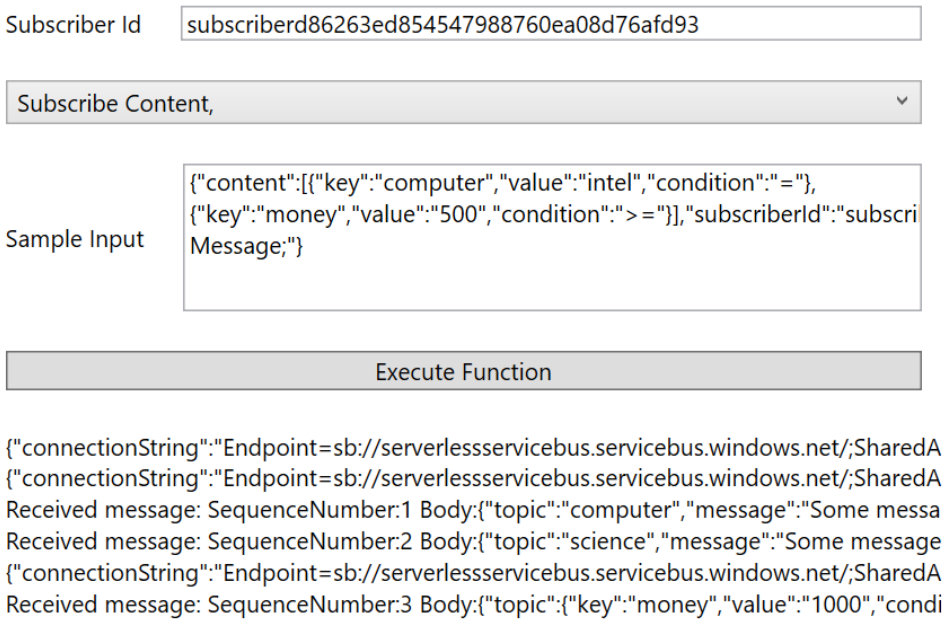



**Figure 4.4:** Subscriber Interface

# Chapter 5

# Implementation

## 5.1 Serverless Broker

Publish-Subscribe (pub/sub) broker is the main component of the system. We built the broker on the serverless platform. To built serverless broker, we first need to break down functionality of broker into smaller independent functions. Once we identified these functions, we developed these functions and hosted these on the serverless platform. We break down broker functionality into two main categories.

1. Subscriber's Functions
2. Publisher's Functions

As the name suggests, the subscriber will call subscriber's functions to register to the system and subscribe to different kind of subscriptions. The publisher will call the publisher's function to publish different kinds of publications. These functions are independent of each other. But they use shared resources.

### 5.1.1 Subscriber's Functions

Subscriber functions are divided into the following categories.

1. Register Subscriber
2. Subscribe to Subscriptions
   (a) Subscribe Topics

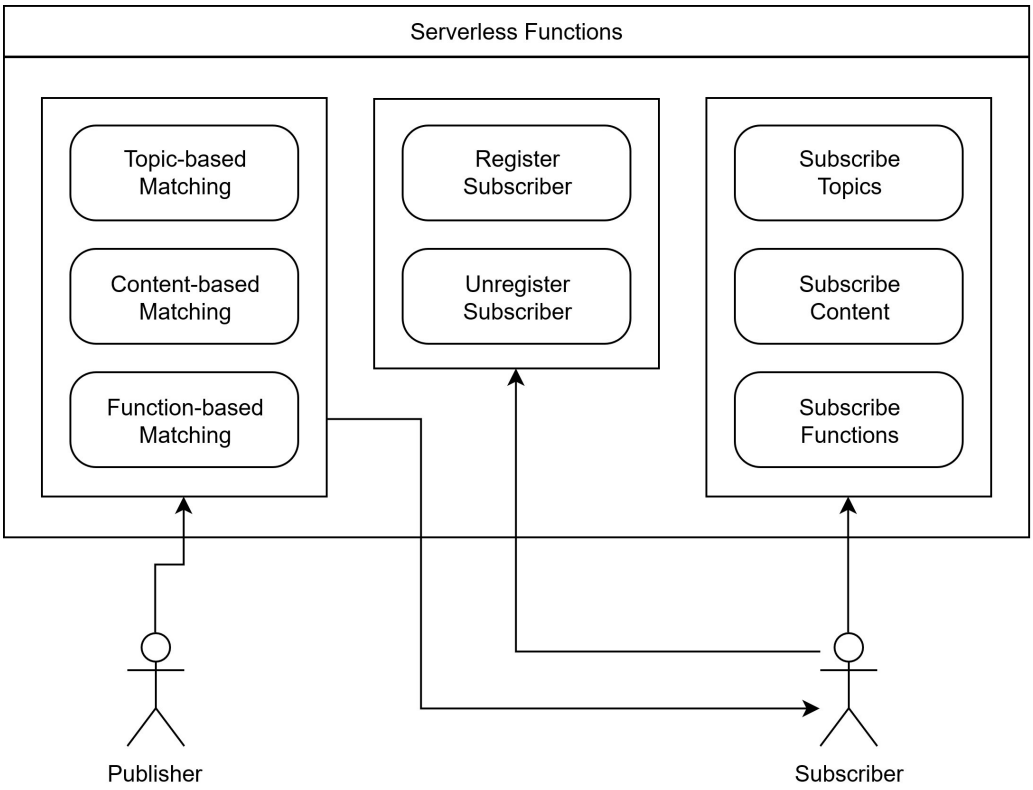



**Figure 5.1:** Serverless functions for Pub/Bub Broker

(b) Subscribe Content

(c) Subscribe Function

3. Unregister Subscriber

### Register Subscriber

To receive publications from publishers, a subscriber first needs to register to the broker. This function is responsible for creating all the required resources for the subscriber. These resources include a unique id and messaging queue.

Figure 5.2 shows the system sequence diagram for register subscriber function.

This function doesn't require any input from the subscriber. It returns a unique id to the subscriber. Subscriber will require this id for other serverless functions. Also, the subscriber needs to make a connection to the message queue using the connection string.

Return output can be different depending on the cloud provider. We used Simple Service Queues (SQS) for AWS and Service Bus Namespace for Microsoft Azure. Sample output

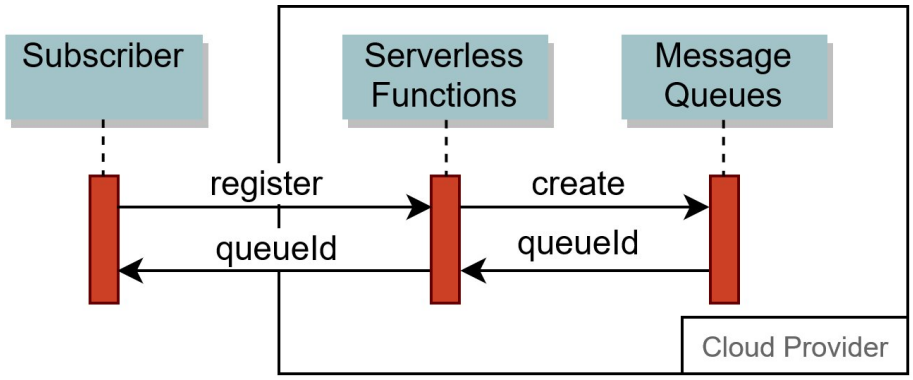



**Figure 5.2:** Register Subscriber Sequence Diagram

for register subscriber is shown in listing 5.1.

```
{
    "connectionString": "Endpoint=sb://..../;...,
    "queueName": "subscriber32",
    "queueUrl": "subscriber32",
    "subscriberId": "subscriber32"
}
```

**Listing 5.1:** Sample Subscriber Register Output

### Subscribe To Subscriptions

The subscriber can subscribe to different kind of subscriptions. For each type of subscription, there is one serverless function. This model is extendable can support other types of subscriptions as well. For each type of matching, there is one table and function adds data to the table.

Figure 5.3 represents the sequence of steps used for subscribe to subscription functions.

### Subscribe Topics

Topic-based matching is one of the basic types of matching in a pub/sub model. Subscribe topic function is responsible for saving subscriber information along with subscribed topics to the database.

Subscriber pass *subscriberId* along with a list of *topics* to the function. Listing 5.2 shows sample input for subscribe topics function.

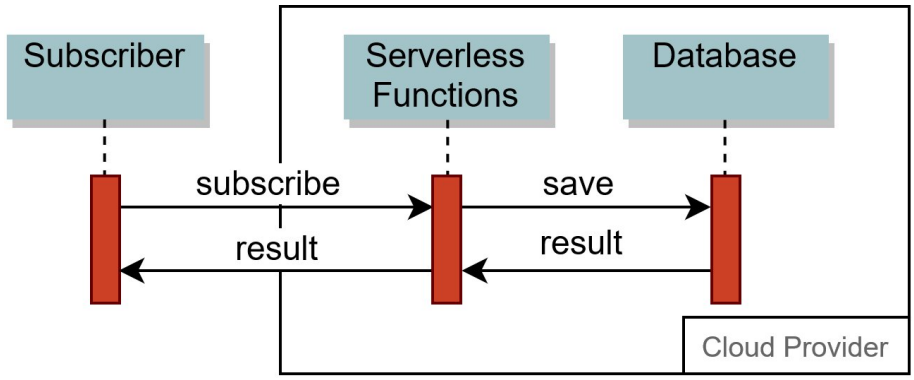



**Figure 5.3:** Subscribe to Subscription Sequence Diagram

```
{
    "subscriberId": "subscriber32",
    "topics": [
        "computer",
        "science"
    ]
}
```

**Listing 5.2:** Sample Subscribe Topics Input

Subscription info for topic-based matching is stored in *topics* table in the database. A subscriber can subscribe to as many topics as desired. Table 5.1 shows sample data for topics table.

| **TopicName** | **SubscruberId** |
|---|---|
| Computer | Subscriber32 |
| Science | Subscriber32 |
| Computer | Subscriber33 |
| Blockchain | Subscriber34 |

**Table 5.1:** Topics Table Sample Data

The function performs the following steps during execution.

- Extracts subscriber id and list of topics from function input
- For each topic, save the topic name and subscriber information into topics table

### Subscribe Content

Content-based matching is also one of the basic types of matching in a pub/sub model. Subscribe content function is responsible for saving subscriber information along with content information in the database.

Subscriber pass *subscriberId* along with a list of *content* to the function. Each content has a *key* which serves as an identifier for content. Along with key, there are *value* and *condition*. Both value and condition are used for publication matching. Listing 5.3 shows sample input for subscribe content function.

```
{
    "subscriberId": "subscriber32",
    "content": [
        {
            "key": "computer",
            "value": "Intel",
            "condition": "="
        },
        {
            "key": "money",
            "value": "1000",
            "condition": ">="
        }
    ]
}
```

**Listing 5.3:** Sample Subscribe Topics Input

Subscription info for content-based matching is stored in *content* table in the database. Similar to topic, a subscriber can have many content subscriptions. Table 5.2 shows sample data of Content database table.

The function performs the following steps during execution.

- Extracts subscriber queue information and list of contents from function input
- For each content, save content key, value, condition and subscriber information into the content table

| Key | SubscriberId | Value | Condition |
|---|---|---|---|
| Computer | Subscriber32 | intel | = |
| Currency | Subscriber32 | 1000 | >= |
| Computer | Subscriber33 | AMD | != |
| Currency | Subscriber34 | 500 | = |

**Table 5.2:** Content Table Sample Data

### Subscribe Function

Function-based matching is a new kind of matching in a pub/sub model which we introduced in this research thesis. Function-based matching allows the subscriber to upload its own custom function. The broker will execute this function for matching and depending on its output publication will be sent to the subscriber. Subscribe Function is responsible for saving subscriber information along with custom function provided to the database.

Subscriber pass *subscriberId* along with *functionType*, *subscriptionTopic*, and *matchingFunction*. SubscriptionTopic is key, while functionType describes type of function. Right now we support only url based functions where subscriber will provider a url in *matchingFunction*. Listing 5.4 shows sample input for subscribe content function.

```
{
    "subscriberId": "subscriber32",
    "functionType": "url",
    "subscriptionTopic": "vision",
    "matchingFunction": "www.customfunction.com/vision",
}
```

**Listing 5.4:** Sample Subscribe Topics Input

Subscription info for function-based matching is stored in *functions* table in the database. Similar to topic, a subscriber can have many function-based subscriptions. Table 5.3 shows sample data for functions table.

The function performs the following steps during execution.

- Extracts subscriber queue information and list of functions from function input

| SubscriptionTopic | SubscriberId | FunctionType | MatchingFunction |
|---|---|---|---|
| Vision | Subscriber32 | url | www.customfunctions.com/vision |
| Text | Subscriber32 | url | www.customfunctions.com/text |
| Vision | Subscriber33 | url | www.customfunctions.com/vision |
| Text | Subscriber34 | url | www.customfunctions.com/text |

**Table 5.3:** Functions Table Sample Data

- For each function, save subscription type, matching function, and subscriber information into the function table

#### Unregister Subscriber

This function removes subscriber from the system. This involves deleting message queue for the subscriber and subscriptions from the database tables.

Function requires *subscriberId* and *queueUrl* as input. Listing 5.5 shows sample message input for the function.

```
{
    "subscriberId": "subscriber32",
    "queueUrl": "subscriber32",
}
```

**Listing 5.5:** Sample Unregister Subscriber Input

The function performs the following steps during execution.

- Deletes message queue for the given queueUrl
- Deletes subscriber subscription from topics, content and functions table

SQS and Service Bus messaging queues functionality is quite similar, but smaller changes are required to set up and remove message queues for subscribers.

### 5.1.2 Publisher's Function

Publishers can execute the following functions.

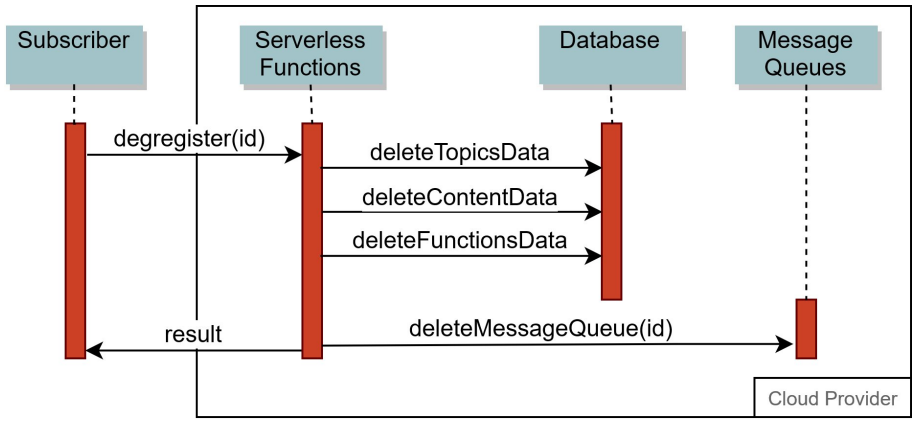



**Figure 5.4:** Unregister Subscriber Sequence Diagram

1. Publish Topics
2. Publish Content
3. Publish Functions

Publishers don't need to register to the system. They only publish publications for different kinds of matching. Figure 5.5 shows common sequence diagram for different kinds of publishing.

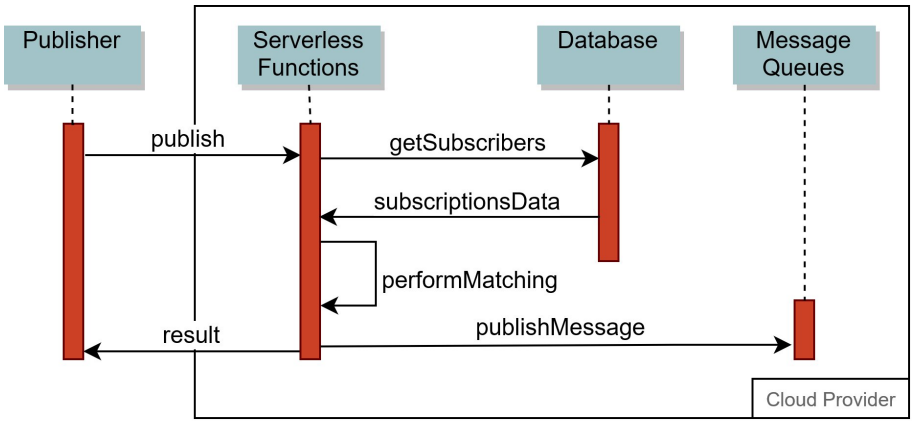



**Figure 5.5:** Publish Publication Sequence Diagram

### Publish Topics

Publish topics function publishes publications to subscribers by using topic-based matching. This function retrieves data from the topics table and sends publications to subscribers if matching is successful.

Publisher pass *message* along with a list of *topics* to the function. Listing 5.6 shows sample input for publish topics function.

```
{
    "message": "Publication text.",
    "topics": [
        "computer",
        "science"
    ]
}
```

**Listing 5.6:** Sample Publish Topics Input

The function performs the following steps during execution.

- Extracts list of topics from function input
- For each topic retrieves subscribers information from the topic table
- For each subscriber, send the publication to the subscriber's queue

In the end, all subscribers will receive publication who are subscribed to the topics provided by the publisher. It is possible that a subscriber will receive the same publication multiple times as a publication can belong to multiple topics, and the subscriber can subscribe to multiple topics.

### Publish Content

Publish content function performs content-based matching and sends publications to subscribers. This function retrieves subscribers information from the content table and performs content matching as provided by subscribers.

Publisher pass *message* along with a list of *content* to the function. Unlike Subscribe Content function, list of contents doesn't have a condition. The condition is only provided by the subscriber. Listing 5.7 shows sample input for publish content function.

```
{
    "message": "Publication text.",
    "content": [
        {
            "key": "computer",
            "value": "Intel",
```

```
        },
        {
            "key": "money",
            "value": "1000",
        }
    ]
}
```

**Listing 5.7:** Sample Subscribe Topics Input

The function performs the following steps during execution.

- Extracts list of topics from function input
- For each content key, retrieves subscribers information from the content table
- For each subscriber, the function performs condition operation on values provided by publisher and subscriber.
- If condition passes, send publication message to the subscriber

We provide basic arithmetic operations and string matching for content matching.

### Publish Functions

Publish functions function performs function-based matching and sends publications to subscribers. This function retrieves subscribers custom functions from functions table and performs function-based matching.

Publisher pass *message* along with a list of *subscriptionTopic* to the function. Listing 5.8 shows sample input for publish content function.

```
{
    "message": "Publication text.",
    "subscriptionTopics": [
        "vision",
        "handwriting"
    ]
}
```

**Listing 5.8:** Sample Subscribe Topics Input

The function performs the following steps during execution.

- Extracts list of subscription topics from function input
- For each subscription topic retrieves subscribers information from functions table
- For each subscriber executes a custom function provided by the subscriber with publication message
- If the custom function returns true, send publication message to the subscriber

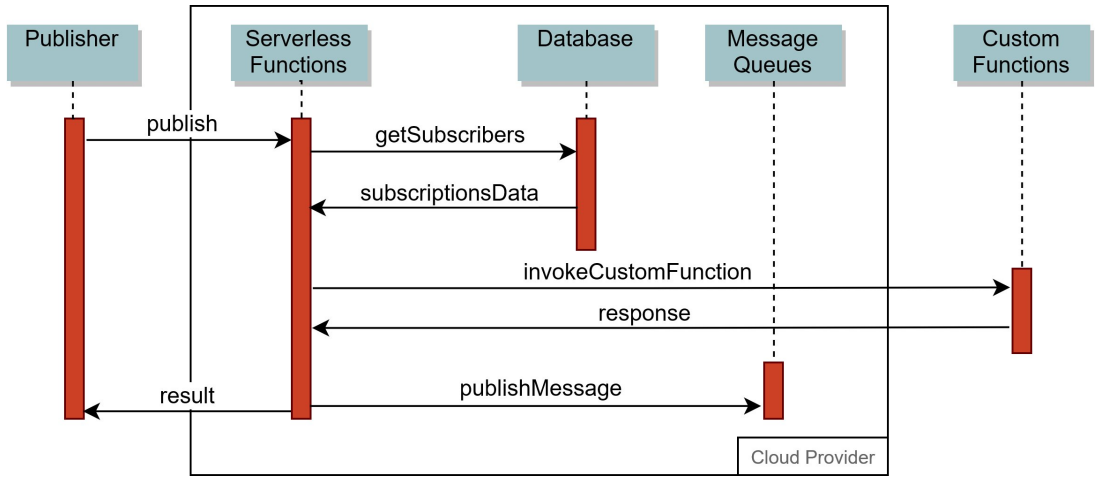



**Figure 5.6:** Function-based Matching Sequence Diagram

Every subscriber can have their own custom function and can be hosted on different locations and can require different kinds of inputs. The broker should provide some templates to publishers and subscribers. Subscribers can use these templates and design their own custom functions, and publishers can make sure their publication message can be passed to custom functions.

## 5.2 Custom Functions

For function-based matching, the subscriber needs to send URL of their own custom functions. These custom functions can vary from having very simple to complex logic. We built some custom functions for subscribers and made these available to the broker. Every function requires some input from the publisher and returns a success or failure response. Depending on the input, these functions can be divided into categories.

1. Vision Functions
2. Text Functions

### 5.2.1 Vision Functions

These functions require an image as input. They perform some computation on the input image and return success or failure depending on the result of the computation. These functions use Microsoft Vision API [33].

Table 5.4 gives list of vision functions available for subscribers.

| **Function Name** | **Description** |
|---|---|
| Analyze Image | Analyzes image and returns text description of image |
| Detect Faces | Detects whether image has faces or not |
| Detect Printed Text | Detects text from a printed document (OCR) |
| Detect Handwriting Text | Detects text from handwriting image |

**Table 5.4:** Custom Vision Functions

### 5.2.2 Text Functions

These functions require a text as input. They perform text analytics on input and returns results to the broker. These functions use Microsoft Text Analytics API [34].

Table 5.5 gives list of text functions available for subscribers.

| **Function Name** | **Description** |
|---|---|
| Detect Languages | Detects language of text |
| Detect Key Phrases | Extracts key phrases from the text |
| Detect Sentiments | Analyzes sentiments of text |

**Table 5.5:** Custom Vision Functions

## 5.3 Evaluation Application

The evaluation application is used to validate the correctness of the system. The application provides a graphical user interface to create publishers and subscribers.

Furthermore, the application can invoke serverless functions for publishers and subscribers.

Evaluation applications contains three views.

- Simulator View
- Subscriber View
- Publisher View

### 5.3.1 Simulator View

Simulator view is used to create multiple publishers and subscribers. It automatically registers subscribers to the broker. The user needs to select provider type and provide base address for the serverless broker. Figure 5.10 shows user interface for subscriber.

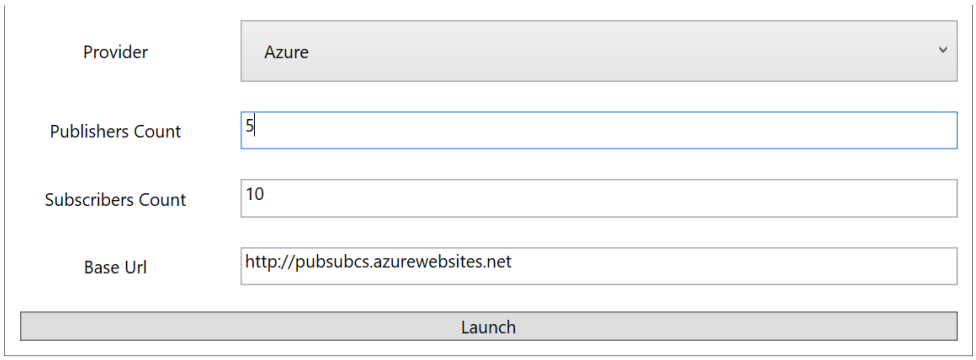



**Figure 5.7:** Simulator Interface

User can also provide a configuration file with the information and application will simulate the environment. For the configuration file in figure 5.8, simulator will create *10 subscribers* on *Azure* serverless broker hosted at *http://localhost:7071/api*. *5* subscribers will subscribe to topics *topic1 and topic2* while *2* subscribers will subscribe to function *http://localhost:7071/api/DetectFaces*.

### 5.3.2 Publisher View

Publisher view allows publisher to publish publications to the broker. The publisher can select one of its function and can execute it.

```
{
  "baseUrl": "http://localhost:7071/api",
  "providerType": "Azure",
  "applicationMode": "Subscriber",
  "nodesCount": 10,
  "topicsMetadata": {
    "nodesCount": 5,
    "topics": [
      "topic1",
      "topic2"
    ]
  },
  "functionsMetadata": {
    "nodesCount": 2,
    "functions": [
      {
        "functionType": "url",
        "subscriptionTopic": "function1",
        "matchingFunction": "http://localhost:7071/api/DetectFaces",
        "matchingInputs": "index"
      }
    ]
  }
}
```


**Figure 5.8:** Sample Configuration File

To publish a publication, some kind of input is required. Application provides sample input for all kind of publications. Publisher can override sample input as well. Figure 5.9 shows interface available for publisher.

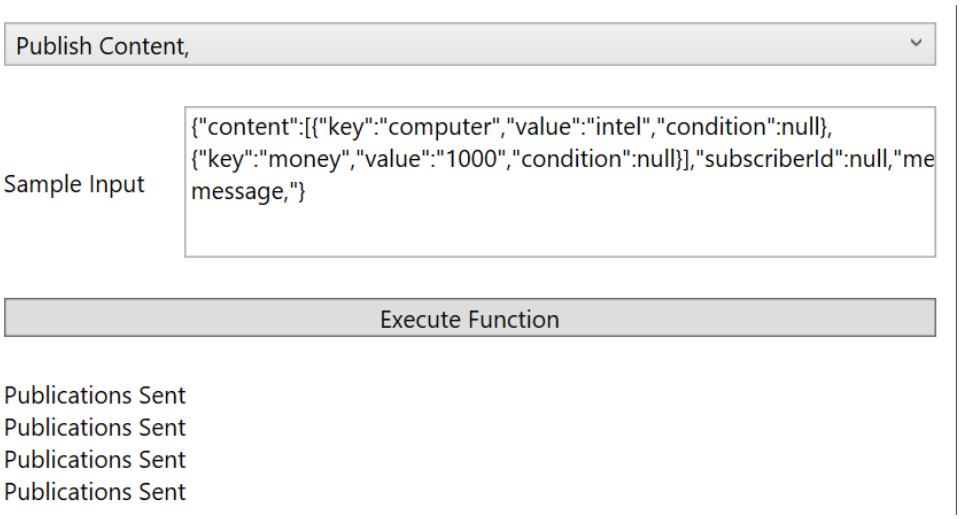



**Figure 5.9:** Publisher Interface

### 5.3.3 Subscriber View

Subscriber view allows subscriber to execute their serverless functions. Subscriber can select a function from the list of available functions and execute it. Each function require some input. Sample input is provided by the application. But subscriber can override the sample input with their own input.

Subscriber also receives publications from the broker. These publications can be shown in bottom part of application. Figure 5.10 shows user interface for subscriber.

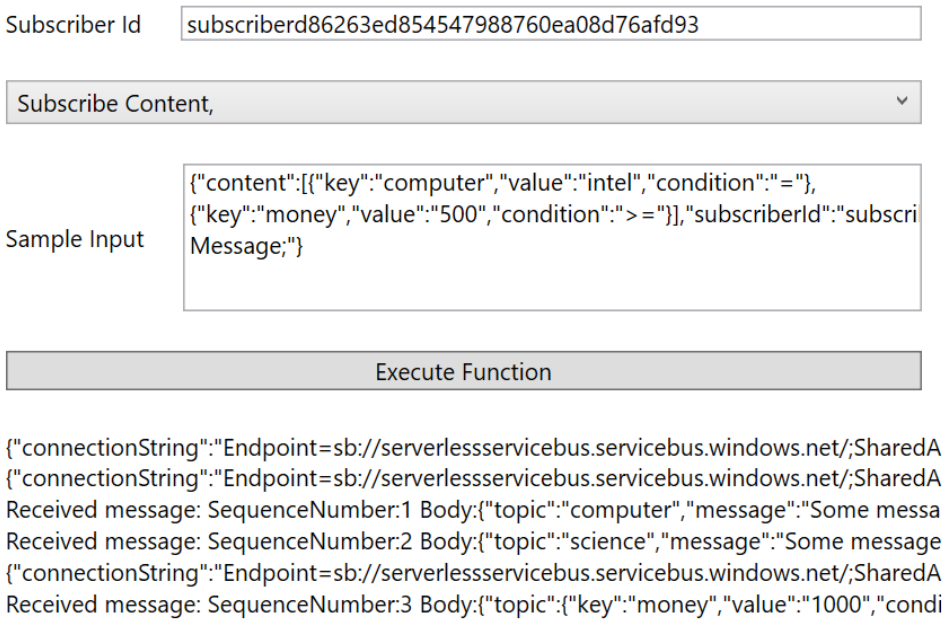



**Figure 5.10:** Subscriber Interface

# Chapter 6

# Evaluation

## 6.1 Evaluation Setup

We designed a distributed environment to measure the performance of the serverless broker. We ran different experiments with different configurations to evaluate different aspects of our system.

For each experiment, there are $p$ number of publishers who publish $p$ number of publications per second. And there are $s$ number of subscribers. Every subscriber is subscribed to one topic, one content and one custom function. And every publisher only sends one message per subscription type. At the end of each experiment, every subscriber will receive exactly $p$ number of publications.

For topic-based matching, publishers sent publications with format $(message, [topics])$ where size of message was 1KB. Every publication contains only one topic. For content-based matching, publication format was $(message, [key_1 : value_1, key_2 : value_2])$ and size of message was also 1KB. Subscriber format was $[key_1 < value_1, key_2 >= value_2]$. For function-based matching, publication format was like $(message, funtionType)$. For function-based matching, we used *detect languages* custom function.

We created five identical Virtual Machines (VMs). Each virtual machine used four virtual CPU cores, 9.77 GB of RAM, and 20 GB of SSD disk running an Ubuntu 16.04. One virtual machine is dedicated to publishers for publishing a different number of publications. On the subscriber side, an accurate number of subscribers are evenly distributed among four virtual machines. Evaluation application is deployed on all virtual machines and is used to generate different scenarios for every experiment. Each experiment is done in a clean

environment. All existing resources were removed before the experiment. We repeated each experiment three times, and calculated arithmetic mean of recorded results.

Our evaluation is divided into three main parts.

1. Correctness of System
2. Performance of Individual Components
3. End to End Evaluation

*Correctness of System* and *Performance of Individual components* are done for both Amazon Web Services (AWS) and Azure cloud providers. But *End to End Evaluation* is done only for AWS due to limited resources. The correctness of system verifies whether serverless broker works as it is intended to or not. After verifying the correctness of the system, we evaluated the performance and the latency of the system. To evaluate the performance of serverless broker we performed several types of experiments. These experiments are divided into two categories. First, we evaluated the performance of individual components used by the broker. And second, we evaluated the performance of the whole workflow.

For individual components, we ran experiments for both cold start and warm start for both cloud providers. But for an end to end evaluation we ran experiments only for the warm start.

- **Cold Start:** Resources are allocated on invocation
- **Warm Start:** Resources are already deployed

Figure 6.1 shows the different components involved in the whole workflow. Only *Subscription Matching* is different for publication matching, while the rest of the components remain same.

For AWS, we used *AWS Educate Starter Account [35]* and for Azure we used *Azure For Students [36]* resources to execute experiments for the evaluation. AWS resources are allocated by *QwikLabs [37]* and it doesn't allow us to select pricing tier for cloud resources. Azure allows us to select different pricing tiers when creating resources. It also allows selecting environment type for cloud resources, i.e dev/test, production, and isolated. We used free tier on dev/test environment. It gives us shared infrastructure, 1GB of memory, and 60 minutes/day compute power. AWS resources were deployed on *EU Frankfurt* while Azure resources are deployed on *Europe West* regions.

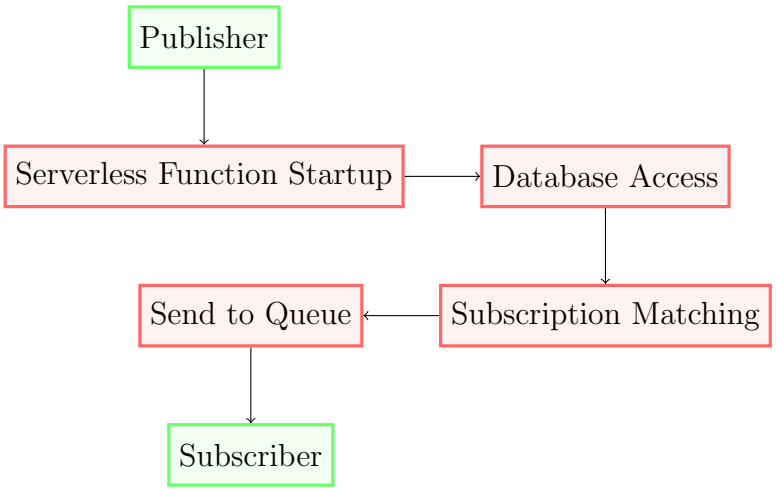



**Figure 6.1:** Components of System

## 6.2 Correctness of System

To verify the correctness of system, we made some scenarios with a fixed number of publishers and subscribers. Subscribers subscribed to different kind of subscriptions and publishers published publications. And then we verified the results by checking the number of publications received by each subscriber.

### 6.2.1 Topic-based Matching

Table 6.1 shows a scenario used for the validation of topic-based matching. In this scenario, there are four subscribers who are subscribed to different topics. Also, there are four publishers. Each publisher publishes one publication along with topics associated with it.

| **Subscriber Id** | **Subscribed Topics** |
|---|---|
| Subscriber1 | topic1, topic2 |
| Subscriber2 | topic2 |
| Subscriber3 | topic2, topic3 |
| Subscriber4 | - |

| **Publisher Id** | **Publication Topic** |
|---|---|
| Publisher1 | topic1 |
| Publisher2 | topic2, topic3 |
| Publisher3 | topic3 |
| Publisher4 | topic4 |

**Table 6.1:** Sample data for Topic-Based evaluation

Once all publishers published their subscriptions, we calculated the number of

subscriptions received by each subscriber. Table 6.2 shows result of the scenario. Results validate topic-based matching done by the broker.

| Subscriber Id | Publications Received |
|---|---|
| Subscriber1 | 2 |
| Subscriber2 | 1 |
| Subscriber3 | 3 |
| Subscriber4 | 0 |

**Table 6.2:** Result for Topic-Based matching

### 6.2.2 Content-based Matching

Like topic-based matching, we set up a similar scenario for content-based matching. There are four publishers and four subscribers. Table 6.4 shows sample data used for validation of content-based matching.

| Subscriber Id | Publications Received |
|---|---|
| Subscriber1 | 1 |
| Subscriber2 | 0 |
| Subscriber3 | 0 |
| Subscriber4 | 1 |

**Table 6.3:** Result for Content-Based matching

Table 6.3 shows number of subscriptions received by subscribers after the end of scenario. Results validate the correctness of content-based matching by the broker.

### 6.2.3 Function-based Matching

We had two subscribers and four publications for function-based matching. Each subscriber provided their own custom functions for publication matching. Table 6.5 shows sample data used for validation of function-based matching.

| Subscriber Id | Subscribed Content |
|---|---|
| Subscriber1 | content: content1<br>condition: <<br>value: 1000 |
| Subscriber2 | content: content2<br>condition: =<br>value: 1000 |
| Subscriber3 | content: content1<br>condition: ><br>value: 200 |
| Subscriber4 | content: content3<br>condition: =<br>value: text |

| Publisher Id | Publication Topic |
|---|---|
| Publisher1 | message: 200<br>content: content1 |
| Publisher2 | message: 500<br>content: content2 |
| Publisher3 | message: text<br>content: content3 |
| Publisher4 | message: mobile<br>content: content3 |

**Table 6.4:** Sample data for Content-Based evaluation

| Subscriber Id | Function Description |
|---|---|
| Subscriber1 | True if text is in English |
| Subscriber2 | True if image contains faces |

| PublisherId | Publication |
|---|---|
| Publisher1 | Text in Spanish |
| Publisher2 | Text in English |
| Publisher3 | Picture with faces |
| Publisher4 | Picture without faces |

**Table 6.5:** Sample data for Function-Based evaluation

After the execution of the scenario, both subscribers received one publication which is correct behavior. Table 6.6 shows number of subscriptions received by subscribers after the end of scenario.

| Subscriber Id | Publications Received |
|---|---|
| Subscriber1 | 1 |
| Subscriber2 | 1 |

**Table 6.6:** Result for Function-Based matching

### 6.2.4 Mixed Matching

After testing individual topic/content/function based matching for subscribers. We made scenarios where subscribers were subscribed to a different kind of matching. Table 6.7 shows subscribers and their subscriptions for this scenario.

| SubscriberId | Topics | Content | Function |
|---|---|---|---|
| Subscriber1 | topic1 | content1 | |
| Subscriber2 | | content1 | English Text |
| Subscriber3 | topic1 | | English Text |
| Subscriber4 | topic1 | content1 | English Text |

**Table 6.7:** Sample data for Mixed matching

We sent publications for *topic1*, *content1* and *English text*. We verified results after scenario completion which are shown in Table 6.8.

| Subscriber Id | Publications Received |
|---|---|
| Subscriber1 | 2 |
| Subscriber2 | 2 |
| Subscriber3 | 2 |
| Subscriber4 | 3 |

**Table 6.8:** Result for Mixed matching

### 6.2.5 Offline Subscribers

In this scenario, some subscribers went offline after subscribing to topics, contents or functions. We used the same scenario which we used for mixed matching (Table 6.7). After some time, offline subscribers came online, and start listening to their message queues. Results were same as the result of mixed matching (Table 6.8).

After performing different scenarios in different settings, we validated the correctness of the system.

## 6.3 Performance of Individual Components

### 6.3.1 AWS Lambdas vs Azure Functions

For this evaluation, we calculated how much time it takes a serverless function to start execution. As serverless providers deploy resources on the fly and these resources are taken back if not used for a certain amount of time. So we run experiments for both cold start and warm start.

AWS Lambdas has a restriction on the number of concurrent executions and allows 1000 concurrent instances. Azure Functions has no such limit as it depends on available resources.

#### Cold Start

For a cold scenario, we invoked a serverless function and time taken for the cloud provider to allocate resources and start execution of the instance of a serverless function. In our experiments, we invoked the same function different number of times and calculated the average time to allocate resources to the function instance.

Figure 6.2 shows average time for function to start execution for AWS Lambdas and Azure Functions. We observed that the number of instances increases the time for the allocation of resources. AWS Lambdas performs slightly better than Azure Functions. AWS Lambdas has an average time of 2456ms, and Azure Functions has an average time of 2588ms to allocate resources and start execution.

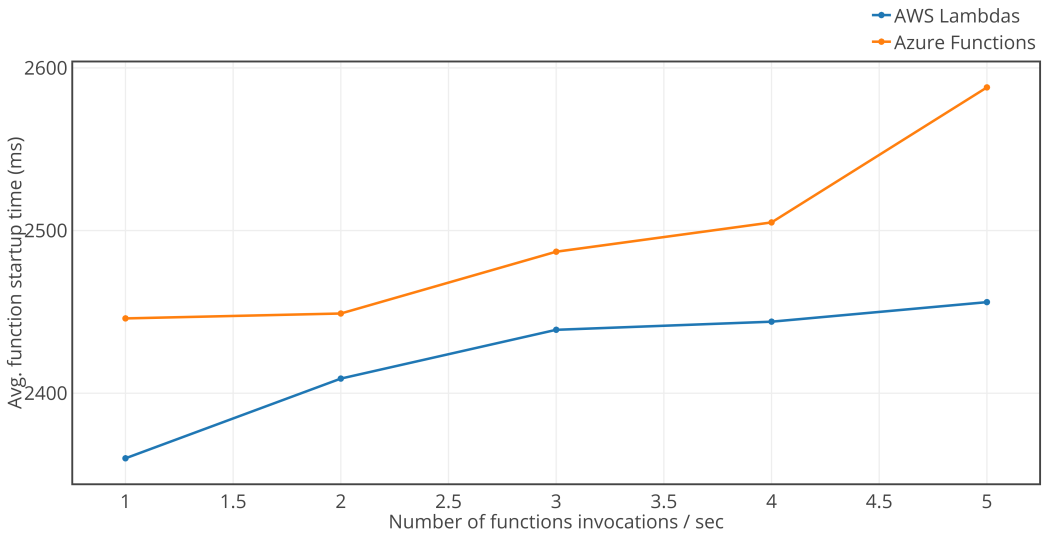



**Figure 6.2:** Startup time for multiple serverless functions execution (Cold Start)

### Warm Start

For the warm start, the scenario is the same as the cold start. But for this evaluation, we ensured resources are already deployed. We increased the number of instances and measured the time taken to start the execution of serverless functions.

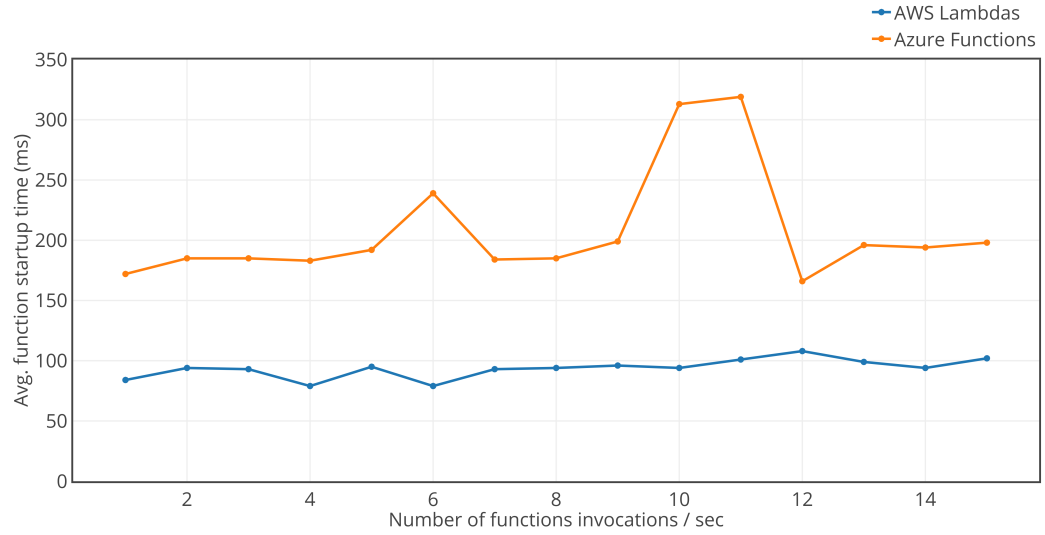



**Figure 6.3:** Startup time for multiple serverless functions execution (Warm Start)

Figure 6.3 shows average time taken for multiple instances of function to start execution. Average function time started to increase with more number of invocations. AWS Lambdas performs better than Azure Functions with the average time of 96ms and 201ms

respectively.

### 6.3.2 DynamoDB vs Azure Tables

Publisher functions access database to get information about subscribers and their subscriptions. For Azure Tables we used *Standard Performance* which uses *magnetic drives*. And for DynamoDB, we used free tier which allows fifty concurrent reads and one concurrent write operations.

For this scenario, we calculated how much time is taken to retrieve data from a database table. We ran multiple serverless functions and retrieved data of hundred subscribers from the table.

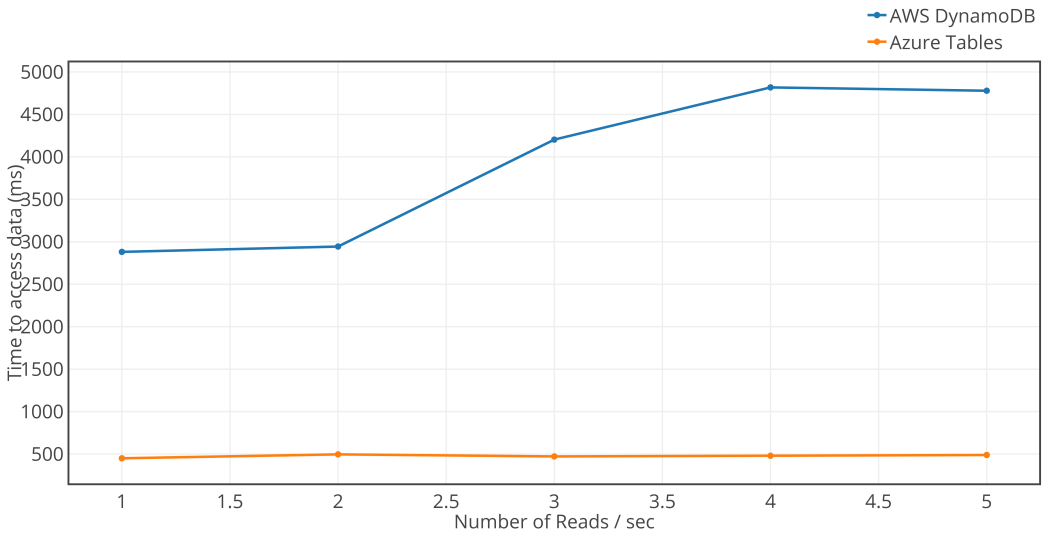



**Figure 6.4:** Average latency to read 100 subscribers information from the database (Cold Start)

As shown in figure 6.4, Azure Tables clearly outperforms DynamoDB in accessing database in cold condition. Azure Tables average time is 490ms while DynamoDB average time is 4200ms.

In warm settings, DynamoDB performance is extremely good. DynamoDB average access time is around 3ms while Azure Tables access time is 9.8ms.

### 6.3.3 SQS vs Service Bus Namespace

For this scenario, we measured the latency for subscribers to receive messages from the message queue sent from the serverless functions. For these experiments, we don't include

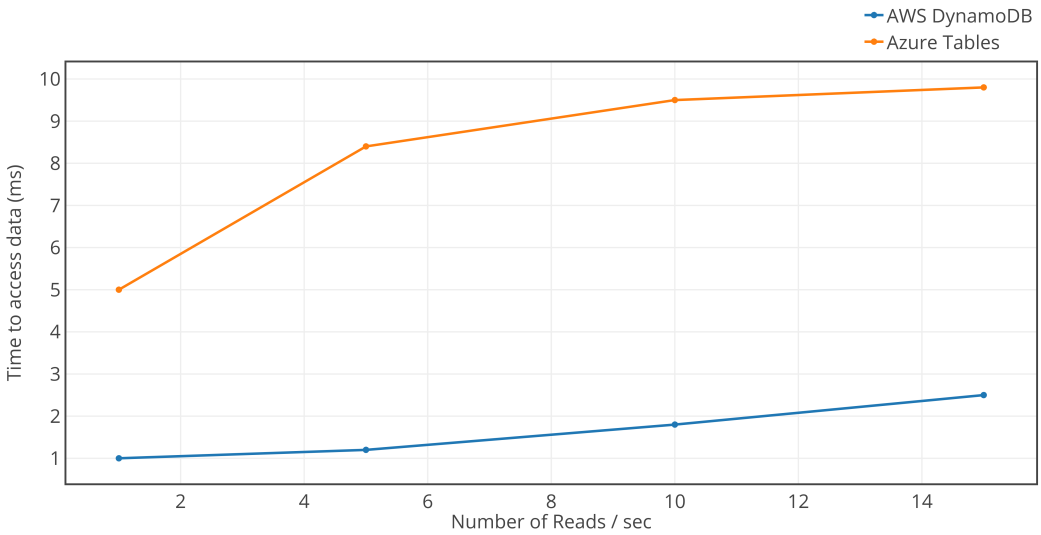



**Figure 6.5:** Average latency to read 100 subscribers information from the database (Warm Start)

time taken to perform subscription matching or getting data of queues.

For Service Bus, we used free tier which uses shared capacity and message payload size is limited to 256KB. For SQS we used free tier and standard queues. First-In-First-Out (FIFO) queues are also available in free tier, but they are not available in Europe West region. SQS queues also have a limit of 256KB on message payload size. But 64KB of the payload is billed as one request which means a message of 256KB will be billed as four requests.

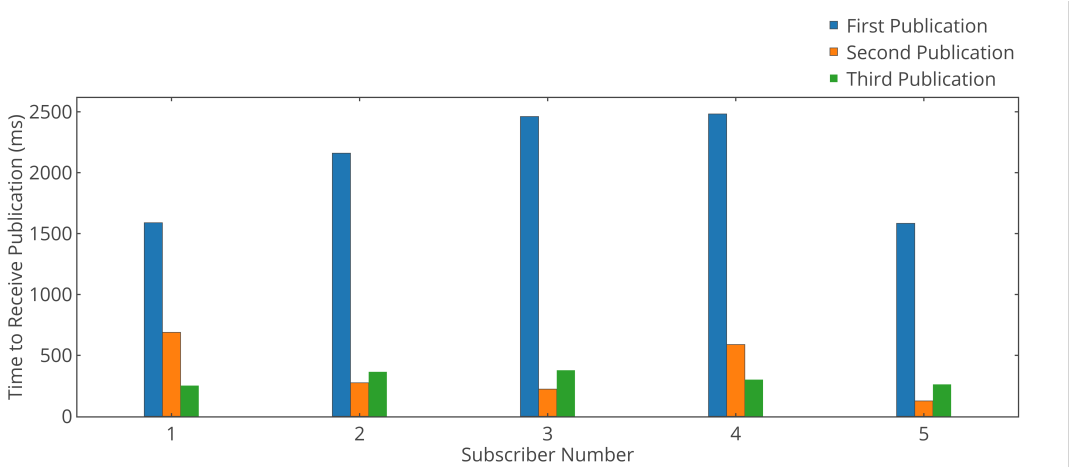



**Figure 6.6:** Average latency to publish multiple publications by subscribers (Azure)

For this experiment, we created five subscribers and sent three publications. Publications were sent with an interval of five seconds. Figure 6.6 shows results of service bus namespace

while figure 6.7 shows results of SQS. The very first publication published to subscribers queues faces cold start issue.

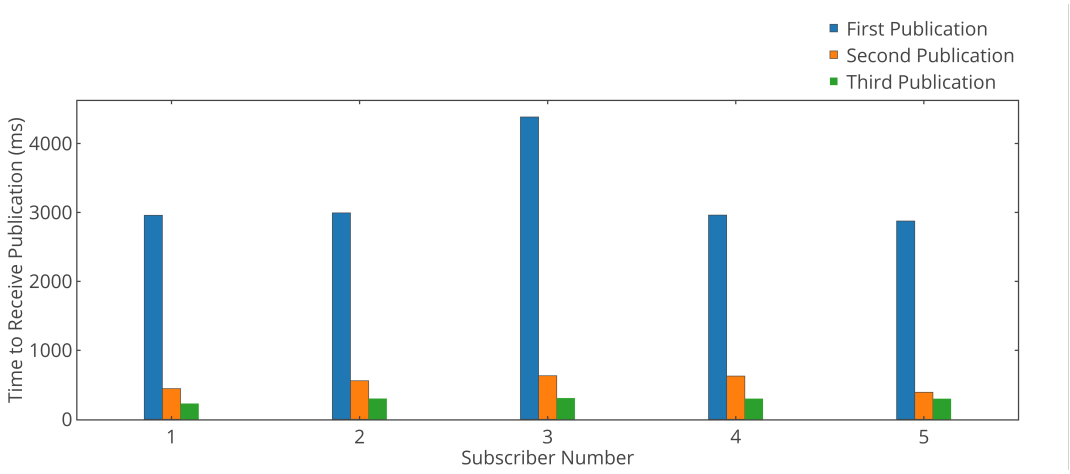



**Figure 6.7:** Average latency to publish multiple publications by subscribers (AWS)

Results show improvements in the performance of sending publications to message queues in the warm start. Average time to send a message for SQS and Service Bus Namespace in the cold start is 2055.2ms and 3435.2ms respectively. For the warm start, it is 287.4ms and 311.6ms respectively.

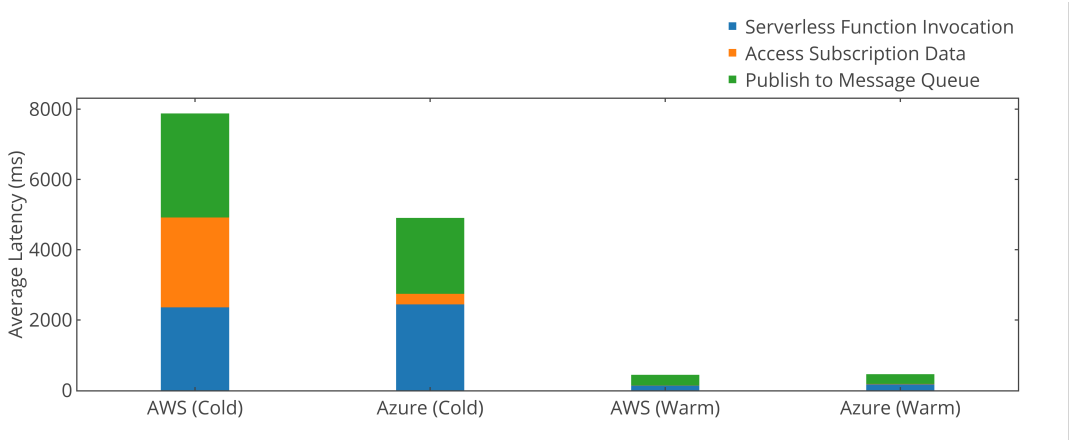



**Figure 6.8:** Comparison between different services of AWS and Azure for publication matching

### 6.3.4 Subscription Matching

Both providers share the same code for matching different type of subscriptions. And hence performance depends only on CPU power. For this experiment, we created fifty subscribers

and twenty publishers. Each subscriber is subscribed to one topic, one content, and one custom function. All subscribers are subscribed to same subscriptions. And publishers publish only those publications which pass the matching criteria. So for each type of subscription matching, every subscriber will receive twenty publications. Figure 6.9 shows a comparison graph between AWS and Azure.

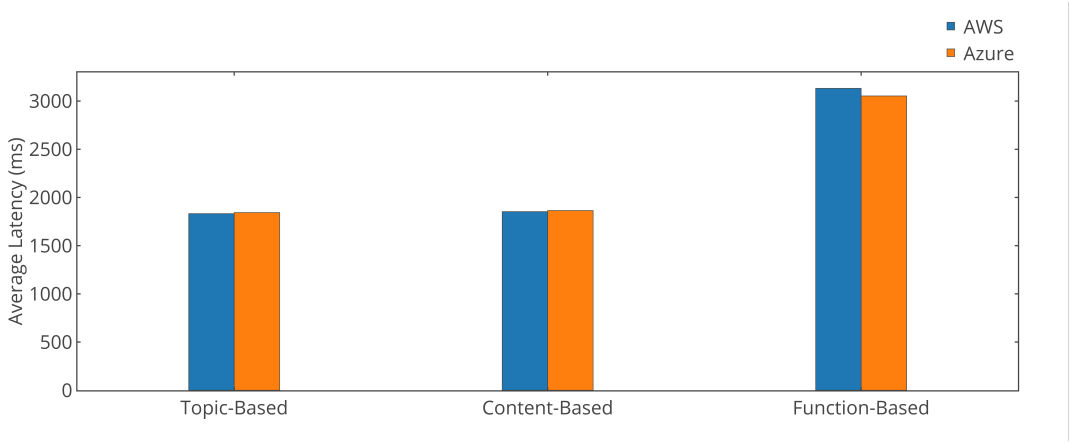



**Figure 6.9:** Average latency comparision of AWS and Azure serverless broker for delivering 20 publications per second to 50 subscribers

The only difference occurs in case of function-based matching. Custom functions provided by subscribers can be hosted anywhere. In our case, we hosted these custom functions on Azure, so for function-based matching Azure Functions perform a little better than AWS Lambdas as shown in figure 6.9.

### 6.3.5 Payload Size

Message queues have a limit on payload size. The maximum size of one message is 256 KB. Additional resources are required if the message size is greater than the limit. In this experiment, we sent publications of different payload sizes to fifty subscribers.

Figure 6.10 shows with the increase in payload size, overall latency of delivering publications increases. From the experiments, we figured out invocation time of serverless function also increases with the increase of message payload.

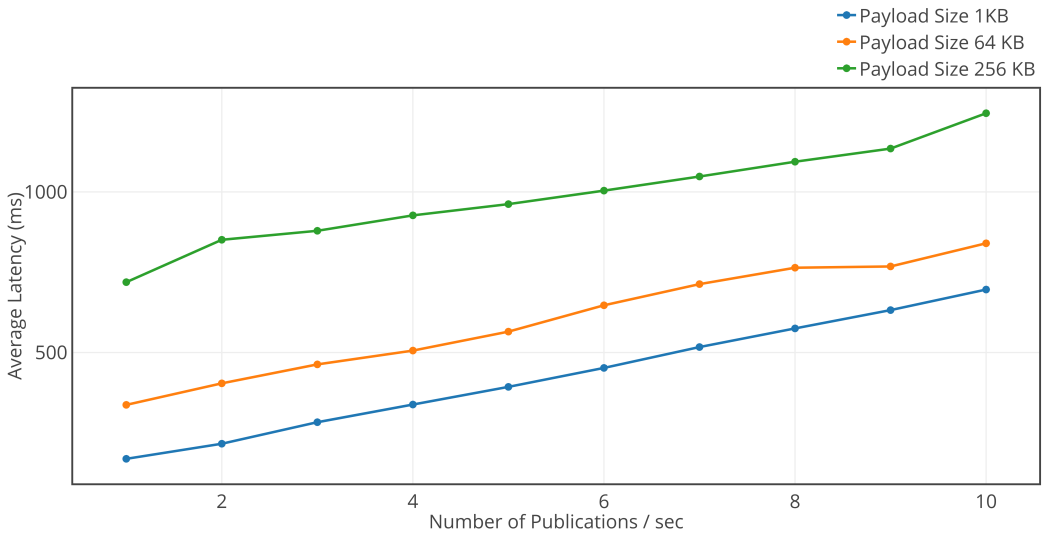



**Figure 6.10:** Average latency of delivering multiple publications to 50 subscribers with different payload sizes

## 6.4 End to End Evaluation

Publishers and subscribers are two main parts of the system. In this section, we performed experiments to measure the latency of end to end subscription matching in response to a different number of publishers and subscribers. We ran two kinds of experiments. In the first experiment, we fixed the number of subscribers and increased number of publications per second. In the second experiment, we fixed the number of publications per second and increased number of subscribers. At the end of each experiment, we measured the time taken to send publications to all subscribers message queues.

### 6.4.1 Fixed Number of Subscribers

In this experiment, we fixed number of subscribers to 100 and sent different number of publications per second to these subscribers. We ran experiments with 5, 10, 25, 50, 100, 250, and 500 publications per second. As shown in the Figure 6.11, with the increase in publications per second the total time for receiving publications also increases. We observed that all serverless functions took same time to start execution and time taken to get subscribers information from database was same as well. The only different occurs when multiple serverless functions start to send publication to one message queue at the same time. Sending multiple publications to message queue at same time decreases the performance of the system.

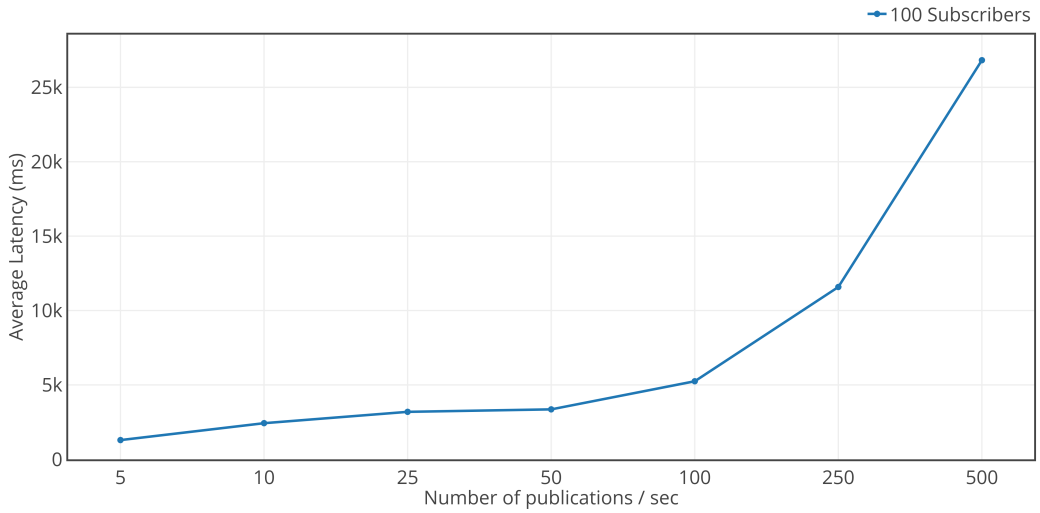



**Figure 6.11:** Average latency of delivering different number of topic-based publications to 100 subscribers

We also tried an experiment with 1000 publications per second, but the output was inconsistent for multiple experiments. We noticed an increase in time for serverless function startup and database access. The main reason behind this was the limitations of concurrent requests. Also, all publications were not delivered to all subscribers due to timeout issues.

Table 6.9 shows time taken by one of the subscriber to receive publications. It can be observed that how overall time increases with adding more publications.

| **Publication Number** | **Publication Received (ms)** |
|---|---|
| 1 | 220 |
| 2 | 294 |
| 24 | 1746 |
| 25 | 1827 |
| 49 | 2521 |
| 50 | 2593 |

**Table 6.9:** Time taken by a subscriber to receive a certain publication

### 6.4.2 Variable Number of Subscribers

In the next experiment, we fixed the number of publications per second but changed the number of subscribers. We ran experiments with 50, 100, 150, 200, 300 and 500 subscribers. We ran multiple experiments with a different number of publications per second.

Each subscriber has their own dedicated message queue. Since adding new subscribers doesn't affect other subscribers message queues, changing number of subscribers doesn't have a drastic effect on overall latency as compared to increasing number of publications. However, we got timeout errors while sending publications to subscribers queues when the number of subscribers increased. Results (figure 6.12) shows results of latency of delivering several publications for a different number of subscribers. Experiments show that broker was able to send 25 and 40 publications per second to 300 subscribers, but it failed to send 50 publications per second. Also, the broker was not able to send 25 and 40 publications to 400 subscribers. It is observed that in case of failure average time was quite similar due to timeout issues.

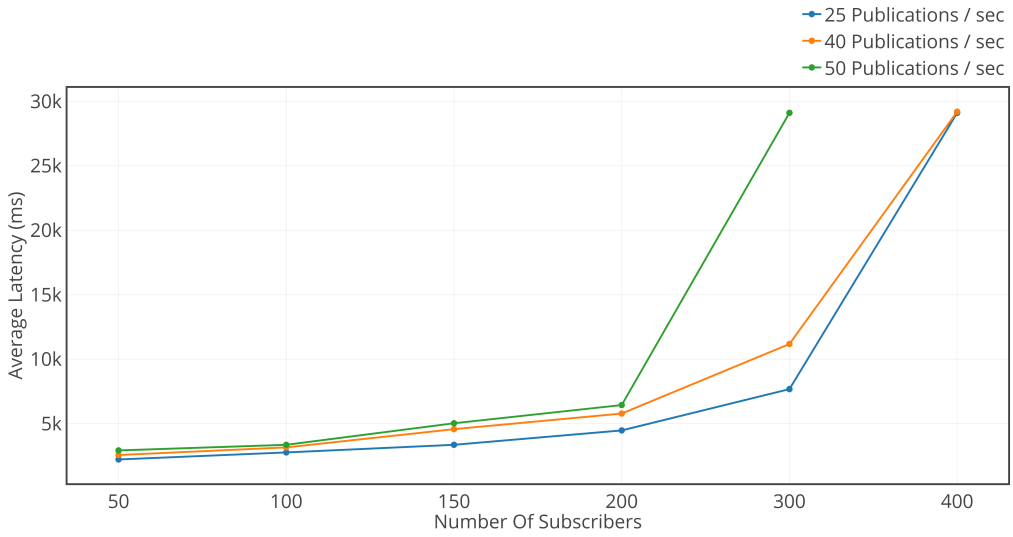



**Figure 6.12:** Average latency of delivering different topic-based publications per second to different numbers of subscribers

Theoretically, there is no limit for the number of queues and for the number of messages in a queue, which means we can have an infinite number of subscribers. But there can be a problem when saving subscriptions information in the database. As the number of concurrent write requests is limited in the database, so it can result in an error state.

Furthermore, we also performed content-based matching and function-based matching with same configurations. Figure 6.13 summarizes the measured latency for different

matching with a different number of subscribers. Topic-based matching and content-based matching performance are quite similar. While for function-based matching, additional time is required to execute custom functions provided by the subscribers.

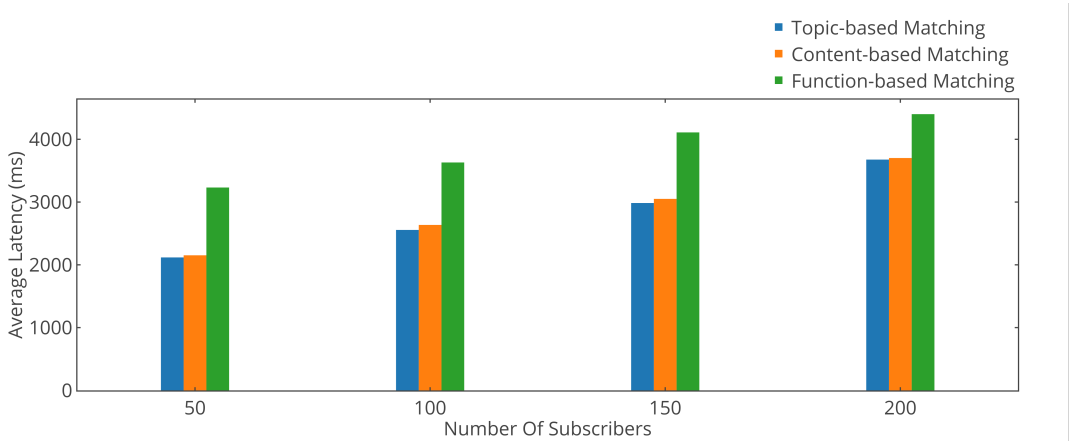



**Figure 6.13:** Average latency of delivering fifty publications to a different number of subscribers

### 6.4.3 Conclusion

We performed a detailed component based evaluation for both providers. We changed the number of publishers and subscribers in our experiments until we find a trend in performance. From the trends, we concluded that increasing the number of publishers has a bigger impact on the overall latency of the system. While adding more subscribers to the system also impacts the overall latency but has relatively a less impact. Serverless function startup time and getting subscriptions data from the database remains nearly the same and doesn't get affected by concurrent access. Message queues are using shared resources so increasing number of publishers and subscribers will affect the overall latency of the system.

For serverless function startup, AWS Lambdas are better than Azure Functions. Azure Tables clearly beats DynamoDB in cold start, but DynamoDB performs way better in the warm start. For message queues, SQS keeps getting better with more subscriptions as compared to Service Bus Namespace.

## 6.5 Results & Findings

### 6.5.1 Publishers & Subscribers

There is no limit on the number of publishers. But to submit multiple publications at the same time, it depends on how many serverless functions can be executed at once. Serverless functions require certain resources to run. These resources contain a number of cores, memory, and storage. Cloud providers provide different packages of resources which can be selected depending on use.

We executed a serverless function with a different number of concurrent users. With the increase of concurrent users, response time started to increase.

| **Serverless Provider** | **Average Response** | **Concurrent Calls** |
|---|---|---|
| AWS Lambdas | 82ms | 250 |
| Azure Functions | 205ms | 250 |

**Table 6.10:** Concurrent access of Serverless Functions

AWS Lambdas allow 1000 concurrent executions of serverless function. AWS Lambda function timeout is 900 seconds (15 minutes). Azure Function doesn't have a limit on concurrent executions. It depends on the number of available resources. Azure Function timeout is 300 seconds (5 minutes). Publishing several publications at the same time decreases the overall performance of the system due to the limitation of the number of concurrent requests to queue client.

AWS Lambdas support dynamic scalability as traffic increases, but it is dependent on account concurrent execution limit. Azure Functions also supports dynamic scalability which can be adjusted using consumption plan and app service plan.

The number of subscribers is dependent on message queue quota. We were able to create 500 subscribers on both AWS and Azure platform. Publication maximum payload size is 256KB for both providers.

### 6.5.2 Statelessness

To overcome the stateless limitation of serverless computing, we have used the database and message queues.

Both providers have maximum request restrictions on database access. In our experiments, we didn't face any problem in getting subscribers data from the database. In warm start, database access is really fast and reliable. But the same cannot be said about database writing. We faced multiple failure issues while writing subscribers data to the database. The overall database has a very small impact on the overall performance o system.

Message queues use shared resources. Adding multiple messages to the queue decreases the performance of the system. However, all messages are added to the queues. During our experiments, all publications were added to the queues and subscribers were able to receive these publications. Cloud providers provide configuration for handling duplicate messages which can be used as well.

### 6.5.3 Custom Functions

Custom functions provided by subscribers can be hosted anywhere. This can impact the performance of the system. Serverless functions will invoke these functions and need their output to execute further. Round trip time can vary depending on the location where these functions are hosted. Also, serverless functions have an execution time limit. These custom functions need to be completed within the time limit. Subscribers need to ensure these function complete in specific time. Also, every subscriber will have their own custom function. So it is possible that some subscribers receive early messages as their custom function gets executed earlier.

Another problem with custom functions is defining input for these functions. Publishers and subscribers need to know what input a custom function takes. For example, we hosted functions which take image and text. If the publisher publishes a publication but is unaware which custom function requires what kind of input, matching will not happen.

### 6.5.4 Cold start

In serverless computing paradigm, resources are only allocated when they are required. Cloud providers clean these resources when these resources are not used by anyone. Due to this cold start scenario is likely to happen. In the evaluation section, we have discussed in detail about the factor of the cold start in the performance of the system.

One solution to handle this problem is to invoke serverless functions after regular interval of time with dummy data. Azure has *Time Trigger* for Azure Functions while AWS has *CloudWatch Event* for AWS Lambdas. These triggers can be used to invoke broker

functions after a specific time. The idea behind this is to invoke functions, so that cloud provider doesn't release resources.

### 6.5.5 Caching

Whenever a publication is published, serverless function fetches subscriptions data from the database. Caching can be applied here to cache table data. So every time a publication is published, the function gets subscriptions data from the cache instead of making database calls. However, getting subscribers data from the database doesn't take much time and has a very small impact on the overall performance of the system.

Due to some reasons caching is not feasible in our solution. First one is the cold start. As we already discuss cloud provider take back resources when not in use. So this will clean cache as well. Caching will work perfectly fine if resources are up all the time. But once resources are deallocated, caching will be gone. The second one is data being changed. Whenever a new subscription is added in the system, the cache will be invalid. In this scenario, information needs to propagate so that the cache is cleaned and updated again. In our model, maintaining and updating cache will add overhead to the system and may create inconsistencies.

We used caching to save client connections in serverless functions. We stored *IAmazonSQS*, *AmazonLambdaClient*, and *HttpClient* in cache. Caching helped in improving performance of sending publications to subscribers message queues.

### 6.5.6 Monitoring

Serverless paradigm lacks proper support for testing and debugging in production environment. Monitoring serverless functions is quite different than traditional client/server. Cloud providers monitoring and logging tools to view usage and errors of functions. But developer has to tackle problems of their code by their own.

### 6.5.7 Cost

For both providers, we used student subscriptions. These subscriptions can be used to develop and test the system. But these subscriptions don't have access to the premium tier of resources. For example, magnetic drives are used for Azure Tables storage. Premium tier has access to Solid State Drive (SSD) which can improve the performance of database

access. Service Bus Namespace also used shared resources in free tier while premium tier has dedicated resources.

Table 6.11 compares costs of both providers. Both have the same price for execution of a serverless function, however, have slightly different costs for resource consumption.

| **Resource Consumption Billing** | **AWS** | **Azure** |
|---|---|---|
| Resources Consumption GB | 512 MB / 1024 MB | 512 MB / 1024 MB |
| Total Resources GB-s | 1.5 million GB-s | 1.5 million GB-s |
| Monthly Free Grant | 400,000 GB-s | 400,000 GB-s |
| Total Billable Consumption | 1.1 million GB-s | 1.1 million GB-s |
| Resource Consumption Price | $0.00001667/GB-s | $0.000016 /GB-s |
| Total Resource Cost | $18.34 | $17.60 |

| **Executions billing calculation** | **AWS** | **Azure** |
|---|---|---|
| Total Monthly Executions | 3 million | 3 million |
| Monthly Free Executions | 1 million | 1 million |
| Price per million Executions | $0.20 | $0.20 |
| Total Execution Cost | $0.40 | $0.40 |

**Table 6.11:** Resource and Execution comparison of AWS Lambdas and Azure Functions

For DynamoDB, first 25GB per month is free. Afterward, it charges $0.306 per GB. Azure Tables has different prices depending on the amount of data stored. For the first TB, it costs $0.07 per GB.

SQS gives 1 million free requests each month. For Standard Queue, SQS charges $0.40 for one million requests. And for FIFO, SQS charges $0.50 for one million requests. Service Bus has different tiers for pricing. In basic tier, one million requests cost $0.05.

## 6.6 Development and Testing

Development and testing of serverless functions is a challenging task. Both providers provide online portal to develop functions online. Online Integrated Development Environment (IDE) can be used to develop simple functions, but for complex functions online portal is not a viable option.

### 6.6.1 Azure

Microsoft comes with their own tools for development: Visual Studio [38] and Visual Studio Code [39]. Both tools have great integration with Azure cloud resources. These tools make the development of serverless functions quite easy. With minimal settings, developers can develop their serverless functions, and deploy these function to the Azure cloud. Also, Azure automatically adds triggers to the functions, and hence no extra effort is required.

These tools allow hosting of serverless functions on the local machine and allow the user to invoke these functions. The developer can debug these functions. This reduces a lot of testing effort from the developer.

### 6.6.2 AWS

AWS lacks their own IDE. Different code editors and plugins can be used to develop lambdas. AWS Toolkit [40] allows Visual Studio to connect to AWS resources. It allows publishing functions to the AWS server. It also allows executing serverless functions from the IDE. However, it is not possible to debug serverless function like Azure. Some open source plugins are available which allow debugging lambdas locally, but they require additional steps and dependencies.

Also, triggers are added manually to the functions. API Gateway is used to make a public API which is hooked with serverless functions.

Development and testing wise, Azure is relatively easy than AWS. Local debugging saves a lot of development time and makes developer life easy.

# Chapter 7

# Summary

In this thesis, we designed and developed a scalable and fault tolerant pub/sub system using serverless computing paradigm. Our pub/sub system performs topic-based, content-based, and function-based matching. Function-based matching is a new type of subscription matching which allows subscribers to provide their own custom implementation for subscription matching.

## 7.1 Status

We developed and deployed pub/sub model on AWS Lambdas and Azure Functions. We broke down the functionality of pub/sub broker into smaller independent microservices and deployed these microservices to the serverless platform. We designed the broker in such a way that it can be extended to support new kind of subscription matching. We have used two cloud resources to overcome the limitations of serverless computing paradigm. We used DynamoDB and Azure Tables to store information of subscribers and their subscriptions. We used Simple Queue Service (SQS) and Service Bus Namespace to deliver publications to subscribers.

We also developed an evaluation application to test several aspects of the system. Our results verify the correctness and fault tolerance of system. Our experiments verify that our system scales up in response to the increasing numbers of publishers and subscribers. We found trends which help in predicting performance of system for increasing numbers of publishers and subscribers. AWS Lambdas and Azure functions scales very well with the increase of parallel invocations. However, writing multiple messages in parallel to message queues decrease the performance of system.

## 7.2 Conclusions

In summary, the serverless computing paradigm is a good fit for backend applications which can be decomposed into small microservices. Different cloud services can be used to overcome the limitations of serverless computing. The serverless providers take care of scaling, resources management and execution of microservices.

Development and testing of serverless functions are challenging. Microsoft provides good debugging and development environment for the development of serverless functions. While AWS also provides plugins for development of serverless function but lacks tools for proper debugging and testing.

## 7.3 Future Work

During the development of the serverless pub/sub system, we encountered a lot of things which can be investigated further to extend the system.

#### Existing Solutions

Both AWS and Azure provide their own services for topic-based matching using Simple Notification Service (SNS) and Service Bus Topics. Also, there are standard tools such as Kafka, RabbitMQ, etc. We can investigate how our system performs in comparison to these solutions. We can also investigate whether it is possible to extend existing solutions and add function-based matching in these solutions.

#### Limitation of Free Tier

We used student subscriptions of AWS and Azure. Free tiers are good for development and testing but are not suitable for deployment of a real application. For example, Azure Tables is using magnetic disks instead of SSDs and message queues were using shared resources. Also, there were restrictions on the number of concurrent requests for reading and writing database. All these affect the overall performance of the system on a large scale. For real environment, premium resources are used which perform much better than free tier.

### Message Queues

Performance of the system decreases when multiple messages are sent to a message queue. Instead of writing messages one by one to a queue, batch messages can be sent to a queue. The process of writing messages to queues can be reworked to increase the performance of the system.

The overall performance of message queues decreases with the passage of time. For example, we send 25 publications to a subscriber ten times, then the overall latency to receive 25 publications will increase with every iteration. This could be due to shared resources or some unknown reason which needs to be identified.

### Cold Start

We provided the evaluation of individual components in both cold and warm start. Both AWS and Azure provide time triggers which can be used to stop deallocation of serverless resources and hence avoid cold start problem. But it is possible that multiple instances of one serverless function are running. We need to come up with a mechanism which ensures a warm start for multiple executions. Or design a system which automatically allocates and deallocates resources by predicting the usage of resources.

### Alternative Resources

We used different cloud resources to overcome the limitations of stateless nature of serverless functions. We can also look into alternative resources and can do comparision of these resources. For example, we use Azure CosmoDB instead of Azure Tables for database storage.

### Function-based Matching

Serverless functions have a limit on execution time. This restricts the logic in custom functions provided by the subscribers. We can design a system which targets this limitation of serverless functions. Also, these custom functions can be divided into a chain of smaller serverless functions and can be hosted on the same cloud provider.

# Bibliography


[1] Statista, "Market share prediction of cloud services providers." https://www.statista.com/statistics/805942/worldwide-market-share-cloud-service-providers/, 2017. Accessed: 2018-06-18.

[2] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pp. 405–410, 2017.

[3] James, "Azure functions vs aws lambda – scaling face off." https://www.azurefromthetrenches.com/azure-functions-vs-aws-lambda-scaling-face-off/, 2018. Accessed: 2018-08-13.

[4] James, "Azure functions – significant improvements in http trigger scaling." https://www.azurefromthetrenches.com/azure-functions-significant-improvements-in-http-trigger-scaling/, 2018. Accessed: 2018-08-13.

[5] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless computing: An investigation of factors influencing microservice performance," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 159–169, 2018.

[6] WoSC2018, "Fourth international workshop on serverless computing (wosc) 2018." https://www.serverlesscomputing.org/wosc4, 2018. Accessed: 2018-08-13.

[7] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, fast and slow: Low-latency video processing using thousands of tiny threads," in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, (Berkeley, CA, USA), pp. 363–376, USENIX Association, 2017.

[8] M. Yan, P. Castro, P. Cheng, and V. Ishakian, "Building a chatbot with serverless computing," in *Proceedings of the 1st International Workshop on Mashups of Things and APIs*, MOTA '16, (New York, NY, USA), pp. 5:1–5:4, ACM, 2016.

[9] I. Baldini, P. C. Castro, K. S. Chang, P. Cheng, S. J. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. M. Rabbah, A. Slominski, and P. Suter, "Serverless computing: Current trends and open problems," *CoRR*, vol. abs/1706.03178, 2017.

[10] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with openlambda," in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, (Denver, CO), USENIX Association, 2016.

[11] J. Spillner, "Snafu: Function-as-a-service (faas) runtime design and implementation," *CoRR*, vol. abs/1703.07562, 2017.

[12] M. Fowler, "Microservices." `https://martinfowler.com/articles/microservices.html/`, 2015. Accessed: 2018-04-19.

[13] A. Kharenko, "Monolithic vs. microservices architecture." `https://articles.microservices.com/monolithic-vs-microservices-architecture-5c4848858f59/`, 2015. Accessed: 2018-04-19.

[14] Y. Cui, "How to migrate existing monoliths to serverless." `https://blog.binaris.com/how-to-migrate-existing-monoliths-to-serverless/`, 2018. Accessed: 2018-04-23.

[15] Y. Cui, "How to migrate existing microservices to serverless." `https://blog.binaris.com/your-guide-to-migrating-existing-microservices-to-serverless/`, 2018. Accessed: 2018-04-23.

[16] M. Villalba, "6 things to know before migrating an existing service to serverless." `https://serverless.com/blog/6-things-to-know-before-migrating-an-existing-service-to-serverless/`, 2017. Accessed: 2018-04-23.

[17] D. Happ and A. Wolisz, "Limitations of the pub/sub pattern for cloud based iot and their implications," in *2016 Cloudification of the Internet of Things (CIoT)*, pp. 1–6, Nov 2016.

[18] P. Nasirifard, A. Slominski, V. Muthusamy, V. Ishakian, and H.-A. Jacobsen, "A serverless topic-based and content-based pub/sub broker: Demo," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference: Posters and Demos*, Middleware '17, (New York, NY, USA), pp. 23–24, ACM, 2017.

[19] Aws.amazon.com, "Amazon pub/sub messaing." `https://aws.amazon.com/pub-sub-messaging/`, 2018. Accessed: 2018-05-15.

[20] Aws.amazon.com, "Amazon simple notification service." `https://aws.amazon.com/sns/`, 2018. Accessed: 2018-05-15.

[21] Aws.amazon.com, "Aws lambda." https://aws.amazon.com/lambda/, 2018. Accessed: 2018-04-12.

[22] Aws.amazon.com, "Amazon simple queue service." https://aws.amazon.com/sqs/, 2018. Accessed: 2018-05-13.

[23] Azure.microsoft.com, "Azure app service." https://azure.microsoft.com/en-us/services/app-service/api/, 2018. Accessed: 2018-06-23.

[24] Aws.amazon.com, "Aws console." https://aws.amazon.com/console/, 2018. Accessed: 2018-04-12.

[25] Aws.amazon.com, "Aws command line interface." https://aws.amazon.com/cli/, 2018. Accessed: 2018-04-12.

[26] Aws.amazon.com, "Test serverless lambdas." https://docs.aws.amazon.com/lambda/latest/dg/test-sam-cli.html, 2018. Accessed: 2018-04-13.

[27] Aws.amazon.com, "Amazon api gateway." https://aws.amazon.com/api-gateway/, 2018. Accessed: 2018-05-15.

[28] Azure.microsoft.com, "Create function using azure portal." https://docs.microsoft.com/en-us/azure/azure-functions/functions-create-first-azure-function/, 2018. Accessed: 2018-04-13.

[29] Azure.microsoft.com, "Create function using visual studio." https://docs.microsoft.com/en-us/azure/azure-functions/functions-create-your-first-function-visual-studio, 2018. Accessed: 2018-04-13.

[30] Aws.amazon.com, "Amazon dynamodb." https://aws.amazon.com/dynamodb/, 2018. Accessed: 2018-05-13.

[31] Azure.microsoft.com, "Table storage." https://azure.microsoft.com/en-us/services/storage/tables/, 2018. Accessed: 2018-05-13.

[32] Azure.microsoft.com, "Azure api apps." https://azure.microsoft.com/en-us/services/service-bus/, 2018. Accessed: 2018-05-13.

[33] Azure.microsoft.com, "Computer vision api." https://azure.microsoft.com/en-us/services/cognitive-services/computer-vision/, 2018. Accessed: 2018-06-23.

[34] Azure.microsoft.com, "Text analytics api." https://azure.microsoft.com/en-us/services/cognitive-services/text-analytics/, 2018. Accessed: 2018-06-24.

[35] Aws.amazon.com, "Aws educate." https://awseducate.qwiklabs.com/catalog_lab/555/, 2018. Accessed: 2018-04-12.

[36] azure.microsoft.com, "Azure for students." `https://azure.microsoft.com/en-us/free/students/#free-products-section`, 2018. Accessed: 2018-04-12.

[37] Aws.amazon.com, "Qwiklabs." `https://www.qwiklabs.com/home?locale=en`, 2018. Accessed: 2018-04-12.

[38] Microsoft.com, "Visual studio ide." `https://visualstudio.microsoft.com/`, 2018. Accessed: 2018-04-13.

[39] Microsoft.com, "Visual studio code." `https://code.visualstudio.com/`, 2018. Accessed: 2018-04-13.

[40] A. W. Services, "Aws toolkit for visual studio." `https://marketplace.visualstudio.com/items?itemName=AmazonWebServices.AWSToolkitforVisualStudio2017/`, 2018. Accessed: 2018-04-13.