Institute of Software Technology

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit Nr. 3102-004

Aspects of Event-Driven Cloud-Native Application Development

Tareq Ahmed Ali Al-Maamari

Course of Study:	INFOTECH
Examiner:	Prof. Dr. rer. nat. Stefan Wagner
Supervisors:	Asim Abdulkhaleq, M.Sc. Dr. Andreas Nauerz (IBM)
Commenced:	April 1, 2016
Completed:	September 30, 2016
CR-Classification:	C.0 ; C.2.4 ; D.2

Abstract

Managing and configuring servers have been challenging burdens. Therefore, serverless computing has been introduced to overcome the complexities of managing servers, get rid of operations and handle different terms such as scalability and high availability. Codes or binaries in serverless computing are executed upon direct invocation or as a response to events in a highly scalable manner, which makes it most appropriate for building event-driven applications.An example of such serverless computing services provider, is OpenWhisk the open-sourced project by *IBM*. To receive events and react to them by invoking or running OpenWhisk actions (codes or Docker containers), OpenWhisk provides an ecosystem of packages of services to enable, facilitate, ease the usage of the services and subscribe to their events.

While OpenWhisk provides different powerful means and tools to interact with events, it lacks a number of important services (event sources) packages within the OpenWhisk ecosystem which are necessary to ease and facilitate subscribing to receive their events and using the different functionalities of the services. This thesis enriches the OpenWhisk ecosystem by integrating and enabling more services. A use-case-based approach is chosen to select services to be integrated and enabled. The proposed use-case is an *Early Warning System (*EWS*)* used to warn the public for disasters, possible incidents and helping rescue survivors. In this approach, diversity of the integrated services in terms of domains and vendors are guaranteed to avoid vendor lock-in and provide flexibility in the available services. The integrated services were then categorized based on taxonomic categories using the domain of services to ease out finding and organizing packages.

Acknowledgments

To start with, I do owe many thanks and gratitude to the *Deutscher Akademischer Austauschdienst (*DAAD*)* especially my contact person Mrs. Andrea Gerecke, for sponsoring my master studies at University of Stuttgart. They have been of a great support for whenever I needed help and encountered several challenges.

I do express my deepest gratitude to Prof. Stefan Wagner and Mr. Asim Abdulkhaleq for supervising my thesis and for their support and guidance. I am also thankful to my supervisors in IBM: Dr. Andreas Nauerz and Mr. Simon Moser for giving me the opportunity to do my master thesis under their supervision and for their guidance, patience and support.

I cannot forget the help from the proofreaders, their great feedback have made this thesis as it is now. My warm thanks go especially to Dr. Murad Saeed, Dr. Mustafa Al-Hajjaji, Mr. Jeremias Wagner and Dr. Manaf Gharaibeh.

Contents

List of Figures

List of Tables

List of source codes

1 Introduction

1.1 Motivation

Due to the technological changes in recent years, massive amounts of data are getting generated that require large scale of computing power to process and store. Therefore, the concept of cloud computing has developed. Cloud computing provides highly virtualized, distributed, scalable, on-demand and shared pool of resources which provide the required computing power to deal with such amounts of data. These characteristics and the benefits such as driving down the costs and allowing organization to focus more on business logic that make cloud computing involve in one way or another in every discipline and change the structure and nature of IT world.

In cloud computing, different levels of services are provided through different service models which range from the underlying hardware or virtual machines (*Infrastructure as a Service (*IAAS*)*) up to offering software as a service (*Software as a Service (*SAAS*)*). Regardless of the service model where an application is deployed, servers are still needed to run such applications either on a physical machine, *virtual machine*, *Software Container* or even in an application-centric runtime environment. Unfortunately, the time and effort spent in creating and configuring servers and taking care of the different cloud-related complexities all present a challenge. Furthermore, as servers are lifelong running, the overall utilization might be low due to the long idle periods.

The need for having a solution is highly conducive in order to evade the mentioned problems. A possible solution that 1) abstract all complexities which come with servers, 2) run a cloud-native application only when it is needed either upon a direct invocation or as a response to events, and 3) can highly scale to cope with the different types of workloads. All these characteristics lead to serverless computing, the new computing model where an application is split into small parts (i.e. snippets of code, containers or even binaries), each is executed only upon a direct request or as a response to events.

Obviously, serverless computing is event-driven in nature, which can be the best approach that fits event-driven applications well. In serverless computing, applications are executed and scaled rapidly in response to unpredictable and large demand workloads which result from emitting events. There are many use cases where cloud-native applications should react to events, therefore, it should be designed on top of *Event Driven Architecture (*EDA*)* to enable such applications to efficiently react and respond to these events.

1.2 Problem Statement

Internet of Things is one popular use-case for such EDA applications where non-computing devices like washing machines and dish-washers are emitting events to the Internet. Thus, large number of events need to be processed and stored in real-time manner. In order to receive the emitted events to react and respond to it, event consumers are required to subscribe to receive the events.

As mentioned above, serverless computing can be the best choice when it comes to interacting with events. Therefore, there is a big need for serverless computing providers to provide efficient and convenient mechanisms to allow event consumers to subscribe and receive events in a way that abstracts the underlying complexities such as taking care of communications, different implementations and the nature serverless applications that differs from server-based applications. Nowadays, there are many cloud providers who provide serverless computing services such as *International Business Machines (*IBM*)* Bluemix OpenWhisk, *Amazon Web Services (*AWS*)* Lambda, Microsoft Azure Functions, and Google Cloud Functions, each has different limitations and characteristics.

1.3 Research Objectives

Within The above highlighted issues, there is a need to carry out analysis and comparison of the various serverless computing services providers. So as the first focus of this work, the comparison will be based on aspects such as the different system limitations as well as how powerful the services ecosystem of a given provider is. The service ecosystem defines how far a given service provider is integrated with third-party services in order to minimize the effort to create a serverless function and therefore maximize the value-added. Obviously, the more services exist inside the ecosystem, the more powerful the service provider is. Therefore, the second focus of this master thesis is to enrich the services ecosystem of OpenWhisk the serverless event-driven programming services provider in order to enable event consumers to register to receive events and react to it. In addition, the enabled and integrated services should be categorized base on the characteristics or domains of the services. Specifically, the thesis aims to achieve the following research objectives:

- To compare among various serverless cloud providers.
- To investigate and propose a list of services.
- To propose a categorization system for categorizing OpenWhisk packages.
- To integrate some of the proposed services into the serverless computing provider IBM Bluemix OpenWhisk.
- To provide useful scenarios where the proposed and integrated services can be well exploited.

1.4 Thesis Structure

The rest of this thesis is organized and structured as follows:

Chapter 2 – Background: It starts with some fundamentals of the work and provides the reader with a clear background of the overall research topic, including basics in cloud computing, microservices architecture, serverless computing , OpenWhisk as well as introducing and comparing the main different serverless computing providers.

Chapter 3 – Related Work: In this chapter, the related work to this thesis is described of how far different serverless computing providers integrate, enable and enrich their services ecosystem.

Chapter 4 – Services Categorization, Integration and Enablement: In order to provide more useful scenarios, more services need to be integrated and enabled into OpenWhisk to enrich the OpenWhisk ecosystem and categorizing them. Therefor, in this chapter, we discuss the concepts behind them.

Chapter 5 – Design and Implementation: Here, we design and implement the concepts behind integration, and enable the proposed services into OpenWhisk.

Chapter 6 – Results and Evaluation: This chapter shows the results and evaluation of what have been achieved along this thesis.

Chapter 7 – Conclusions and Future Work: This chapter contains a summary of the thesis achievements and possible improvements and some recommendations for future work.

2 Background

In this chapter, different topics are described for an overall understanding of the background of the core work of this thesis.

2.1 Cloud Computing

Since the last decade, sources of data have been exponentially increasing, and they have generated enormous amounts of data. To process and store these data, a huge computing power is needed which can be expensive and might be hard to access. Therefore, different terms have been introduced to solve this issue, such as grid and cloud computing. Cloud computing which is, in fact, a highly virtualized, distributed, scalable, on-demand and shared pool of resources provides computing power as a utility where resources are provisioned and de-provisioned on-demand. In this thesis, we refer to the provider of cloud computing offerings as a cloud provider, and for the consumer of cloud computing offerings as a cloud consumer.

Due to the benefits offered by cloud computing, it is involved in many areas and a wide number of applications have been built on top of it, such as distributed storages, data analytics, IoT, communication between devices have been built on based on cloud computing. Figure 2.1 shows a mind map of this section's content.

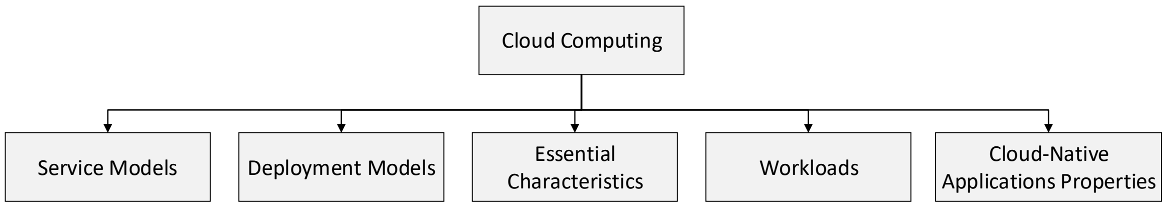
Cloud Computing
Service Models
Deployment Models
Essential Characteristics
Workloads
Cloud-Native Applications Properties

Figure 2.1: Mind map of this section

2.1.1 Cloud Service Models

In cloud computing, IT resources can be delivered to cloud consumers through different service models. Each service model has different levels of abstraction ranging from the very bottom of fundamentals resources such as processing power up to ready *Software as a Service*. According to *The NIST Definition of Cloud Computing* [MG11], there are three cloud service models which are presented as follows:

- **Infrastructure as a Service (IAAS)**: Cloud providers provide fundamental computing resources to cloud consumers such as processing and storage.
- **Platform as a Service (PAAS)**: Cloud providers provide a cloud platform where cloud consumers can deploy their applications and use ready-to-use services. The underlying cloud infrastructure is abstracted. Thus, cloud consumers have no control and knowledge about it.
- **Software as a Service (SAAS)**: Cloud providers offer to users applications which are running in the cloud with abstraction to the underling infrastructure with the possibility of limiting application specification to individual users. A good example of an SAAS service is the on-line document processing application *Office 365*.

2.1.2 Cloud Deployment Models

Regardless of the followed cloud service models, IT resources are deployed and available based on the targeted group of consumers. There are four cloud deployment models [MG11] which are presented as follows:

- **Public Cloud**: In this deployment model, cloud offerings are open to the general public use. The public term here does not necessarily mean that the namespace of a consumer is publically visible, but it is isolated and can be accessed and controlled via different means of access control. In this model, cloud resources may be owned and can be managed by different types of organizations such as academic institutions or business organizations.
- **Private Cloud**: This model has almost the same benefits and services that offered in public clouds, except that the cloud infrastructure in this model is exclusively used by a single institution and may be managed by the organization itself or third-party institutions.
- **Community Cloud**: Cloud infrastructure is shared among a specific community of different organizations that have shared missions or common restrictions such as

specific security configurations. It may be owned and managed by one or multiple organizations.

- **Hybrid Cloud**: This model is a result of combining two or more different cloud infrastructures (private, public or community). In this model, cloud resources of different cloud providers with different service deployment models can federate their resources and deliver them as a combined cloud provider [AAA+10].

2.1.3 Essential Cloud Computing Characteristics

As mentioned earlier, cloud computing resources range from fundamental resources such as servers to a ready-to-use software. Thus, cloud offerings for such resources should be characterized. According to *The NIST Definition of Cloud Computing* [MG11], there are five essential characteristics that each cloud computing provider should provide:

- **On-demand self-service**: Computing resources are provisioned and de-provisioned whenever they are required. [MG11].
- **Broad network access**: Computing resources are available and accessible over a network in a way that allows a wide range of heterogeneous clients/devices to connect to cloud resources from a wide range of locations. [MG11].
- **Measured service (pay-per-use)**: Consumption of different resources provided by the cloud provider such as processing, storage or networking resources should be measured by providing a metering system. It is responsible for leveraging a suitable measurement unit for each cloud offering resource in each cloud model such as storage used or bandwidth consumed. The metering system should also abstract and show the cloud offering as a black box to the cloud consumer. [FLR+14; MG11].
- **Resource pooling**: Computing resources are pooled and shared using multi-tenant model to serve multiple consumers in an automatic way by provisioning and de-provisioning resources automatically on-demand. As the resources are shared, each tenant's resource must be isolated from others. [MG11]
- **Rapid elasticity**: Computing resources are provisioned and de-provisioned on-demand to allow cloud offering consumers to have the exact needed amount of resources on time.

2.1.4 Application Workloads

Workloads in cloud computing are defined as the utilization of cloud resources within a specific period of time. Based on the type of cloud resources, workloads measurements change. For example, in storage cloud offerings, such measurements might be the amount of data stored. Understanding workloads will help provisioning and de-provisioning resources elastically [FLR+14]. The following types of workloads that cloud applications may experience:

- **Static Workload**: Has fixed utilization over a certain period of time. In this case, usually, there is no need for provision or even de-provision of cloud resources to the existing resources. Normally at the provision time, the needed resources can be provisioned as well as a certain amount of over-provisioning to cope with the minimal overhead.
- **Periodic Workload**: Periodic tasks or events such as transferring money in the banks by the beginning of a month will lead to periodic utilization peaks. So, during the peaking, more cloud resources are needed and vice versa. In order to scale; in case of static scaling, resources are statically provisioned as in case of static workload. However, another way is to elastically scale, and in this case, the consumed resources are exactly as needed with no over and under provisioning.
- **One-in-a-Life Workload**: One time peaking in application's life can result into different situations such as having an offer for a specific product in case of e-commerce applications. In this case, there is a big gap between the needed resources during the peak and otherwise. One way to solve this or fill up the gab is to have a static scaling which is inefficient as the needed resources may under-provisioned. Another alternative way is to have an automated elastic scaling where only the needed resources will be consumed.
- **Unpredictable Workload**: Random and unknowable utilization peaks over time will lead to unpredictable workload. In this case, it is hard to cope with it by using static scaling as the time of the unpredictable peak is unknown. Therefore, elastic scaling can be used to provision and de-provision of resources without predicting the workload.
- **Continuously Changing Workload**: Continuous increasing and decreasing on utilization will lead to perpetual changing workload. The rate of changing is consistent. As a solution, the scaling can be done statically as well as elastically. In case of static scaling, provisioning and de-provisioning can be performed in a stepwise way as some resources can be provisioned and de-provisioned in bulks such as provision additional 100 GB of storage once needed and de-provision 100

GB otherwise. In contrast, elastic scaling will be provisioned in a consistent way so the provisioned resources are almost the same as the consumed ones.

2.1.5 Cloud-Native Application Properties

Obviously, the different workloads have different scaling requirements, and therefore, they have different requirements related to the application design. One popular approach is to split an application into several small parts, so that they can be scaled easily and independently. To take an advantage from cloud environments, it is not enough to split an application into several small parts, but it has to be designed and built in a way that ensures some properties derived from cloud computing characteristics. [FH15; FLR+14]

- **Distribution**: Cloud resources are scattered over a big distributed pool of resources. Therefore, cloud-native application should be built in a way that can be distributed among the distributed resources.
- **Elasticity**: According to Shouten [Sho12], elasticity is a synonym to scalability which is the ability to cope with the increased traffic or size with a proper cost [Bon00]. Scalability should be automated in response to the different workloads. As a result, the occupied resources match the demand. Thus, cloud-native applications should be designed to be scaled. In other words, they should be able to run on independent cloud resources. Scalability can be either horizontal where more cloud resources nodes are added or vertical where more capabilities are added to the existing cloud resource node. [FLR+14]
- **Isolated State**: Comes from the requirements of being stateless. As elasticity will allow an application to be scaled out and in, which means more instances of the application should are created, thus, the application should be stateless, where each instance of the application needs to be operated independently of the other instances, and whenever a state is required, it must not be in the application (i.e. stored somewhere else).
- **Automated Management**: As a result of elasticity, resources are provisioned and de-provisioned, added and removed automatically, and consequentially, the load should be monitored to ensure provisioning the needed resources and de-provisioning the unneeded resources on time.
- **Loose Coupled**: Distributed applications consist of distributed components that may depend on each other. Thus, dependencies should be as less as possible in order to ease scaling and minimizing the impact of failing applications components.[FLR+14]

- **The Twelve-Factors**: Sometimes, building applications that work as cloud-native applications and SAAS services sometimes is tricky. This is due to the dynamic continuous expansion of applications, the dynamic collaboration of the teams involved and the cost of slow deterioration of software. The Twelve-Factors is a methodology for building SAAS applications which are 1) a declarative-based format is used for automation to improve the time and cost, 2) a high portability between environments, 3) compatible and proper for deploying on different modern cloud platforms, 4) have divergence between development and production, which enables a continuous deployment and 5) highly scalable with no or slight effect on the architecture and tools [Her].

  1. Codebase: Different versions of source code are tracked through a version control system such as *Git*. A codebase is a repository that is shared among different deployments (Production, staging, etc.)
  2. Dependencies: Twelve-factor applications should use a dependency manager such as *NodeJS Package Manager (*NPM*)* for *NodeJS* and should explicitly declare dependencies through dependency declaration manifest, thus, simplifying the setting up of the application.
  3. Config: Application's config is the configurations that differs based on deploys (production, staging, etc.). Application's config includes the resources handled to the database or any backing service, credentials of external services or specific or even pre-deploy values for different deploys. Config should not be stored within the application itself and it should not be applied to internal configuration of applications. However, it should be stored in a way that can be easily changed between deploys such as environment variables.
  4. Backing services: Backing service is any service that is used by the application to achieve its normal operations such as datastores services (CloudantDB, dashDB, etc.). A twelve-factor application should not differentiate between local and external services.
  5. Build, release, run: The codebase can be moved into a non-development deploy through different stages:
     a) **Build stage**: The code is transfered into executables called builds.
     b) **Release stage**: The build produced in the build stage is combined with the deploy's specific config. The resulting is the release that contains the build as well as the config.
     c) **Run stage**: Runs the app's release in the execution environment.

6. Processes: The app is executed in one or more processes, which imply that the app is stateless and share-nothing among the different processes.
7. Port binding: The twelve-factor application self-contained does not rely on runtime injection of webservers into the execution environment to create a web-facing service. Instead, The application should export *Hypertext Transfer Protocol (*HTTP*)* as a service by binding to a port.
8. Concurrency: The workload is handled by running different processes where each process presents a type, for each type of work, it is assigned to the corresponding process to handle it.
9. Disposability: Processes in the twelve-factor applications are disposal, which means that they are started and stopped at moment's notice.
10. Dev/prod parity: The different deployment (i.e. development, staging, and production) should have as small gab as possible between them.
11. Logs: Logs should be treated as events stream.
12. Admin processes: Any administrative and management related work should be done on an one-off admin processes in an identical environment.

2.2 Microservices Architecture

As technology emergence increases, several factors arise and require software architectures to evolve in order to cope with these factors. Interactive, dynamic and rich applications demand scalability and availability in the first place. In other words, this needs a software architecture that can scale and resist failures such as microservices architecture [Ric].

Microservices architecture is an architectural style that intends to split an application into small services. Such services comprise cloud-native application properties as discussed in the previous section. Microservices gets rid of the problems of monolithic architecture when the whole application logic is implemented in a single unit. For small applications, monolithic can fit well and can scale by using a simple load balancer, but as soon as it gets growing, and becomes bigger and bigger, different problems appears are likely to emerge, including the hardness to scale, failures to take the whole big singular unit down rather than only small part of it, challenge to adapt new technologies as well as understanding the code [BHJ15].

In microservices architecture, an application is divided into small services where each can be implemented, deployed, scaled, tested and monitored independently. To have deeper knowledge about microservices, we will discuss its basic and common characteristics in the following section:

2.2.1 Characteristics

There are common characteristics of microservices architecture, and it is not necessarily that a microservice should meet all of them, but it can be a minimum [DVE+16]:

- **Business-oriented**: Applications are splitted into small parts which serve as services. The splitting boundaries should be carefully analyzed to avoid the risks of resulting in lack of integration which will lead to a fragile system. Thus, microservices should be designed based on business objectives, but this requires more efforts and different cross-teams involvement within organization boundaries.
- **Design for failures**: Isolating failures are inescapable, as application is divided into small services. Moreover, when designing microservices, designer should ensure that a failed service does not affect the overall system/application. Automated testing and real-time monitoring mechanisms necessary to detect failures as quickly as possible. As a result, a quick restore can be done. [LF14]
- **Decentralized Data Management**: Distributed transactions over different services. In case of microservices (*Basically Available, Soft state, Eventual consistency (*BASE*)*) it is preferred over (*Atomicity, Consistency, Isolation, Durability (*ACID*)*).

  As the principle behind microservices is to split applications into small services, hence, each microservice should ideally have its own database. This will introduce polyglot persistence by using different databases such as (DB2 vs Oracle, CloudantDB vs MongoDB) and different data stores types (SQL, NoSql (document based, graphs, etc.)). However, in most of the cases, transactions need to be accessible through multiple microservices, and a centralized database which can be accessed by multiple microservices. This should be avoided as much as possible to ensure the loosely-coupled nature of microservices.
- **Discoverability**: Underlying infrastructure may fail anytime, and therefore, constructing microservices should guarantee reconfiguring services to update the location of other services that they need to connect to. Usually, service discovery can be achieved by using a service registry where all services explicitly register with.
- **Inter-service communication design**: Regardless of the way microservices are deployed (application-centric runtime environment, *Software Container*, *virtual machine*, or even as a function/action in serverless platform), a question arises: How

do services communicate with each other?. In case of one-way communication, message queues are considered a good way. In contrast, for two-way communication, this solution of message queues does not fit well. One way of handling two-way communication work is by making *Representational State Transfer (*REST*)* calls where each service calls others through a lightweight HTTP requests. This guarantees loose coupling as well as abstracting implementation details of each microservice, but there are cases where the caller service (client) requires a chatty messaging or requires multiple calls to achieve its goals (get, add, update, delete resources). Another way is to use an additional layer that sits between the clients and microservices which act as a controller to hide complexities and heterogeneity of communication.

- **Evolutionary Design**: It is common that applications requirements are frequently changed. As a result, new features and capabilities are introduced, and the good point is that only the changed microservice rather than the whole system needs to be taken care of.

2.2.2 Challenges and Complexities

Microservices also introduce a few drawbacks and complexities presented as follows [DVE+16]:

- **Testing**: As any distributed and complex applications, testing of microservices-based systems requires additional different testing components, a set of unit and integration tests, load tests as well as overall system behavior tests.
- **Monitoring**: Microservices are independently built and deployed. Therefore, it is hard to monitor them using the traditional monitoring approaches. However, to cope with this, services should be designed and built in a way that provide health check information which can be used by the monitoring system in tracking the overall behavior of microservices-based system.
- **Dependencies**: Normally, dependencies are frequently updated, which in many cases, requires changing on the code where it is used. In monolithic applications, the normal procedure of updating dependency is by updating a single component where other parts and applications use it as an interface of that dependency. In contrast, in case of microservices-based applications, different services may depend on the same dependency, and in such cases, each single rather than a single component needs to be updated.

- ***Development and Operations***: Regardless of the advanced monitoring system needed in microservices-based systems, keeping each microservice up and running is a challenge which requires more a high-quality DevOps skills to deal with it.

2.3 Serverless Computing

Traditionally, applications ran on physical machines. A large number of machines are expensive, hard to configure, manage, maintain and are frequently underutilized. Moreover, scalability cannot be achieved easily due to different reasons such as the size and complexity of modern operating systems [BDGR97]. Under those circumstances, virtualization (Figure 2.2) provides abstraction on the physical hardware level and consolidates the hardware, thus, utilizing resources more efficiently, decreasing the cost and improving maintenance. Above the abstraction level, multiple virtual machines run simultaneously, and each can be a separate server running on different operating systems. Hence, multiple servers can be run simultaneously on those virtual machines.

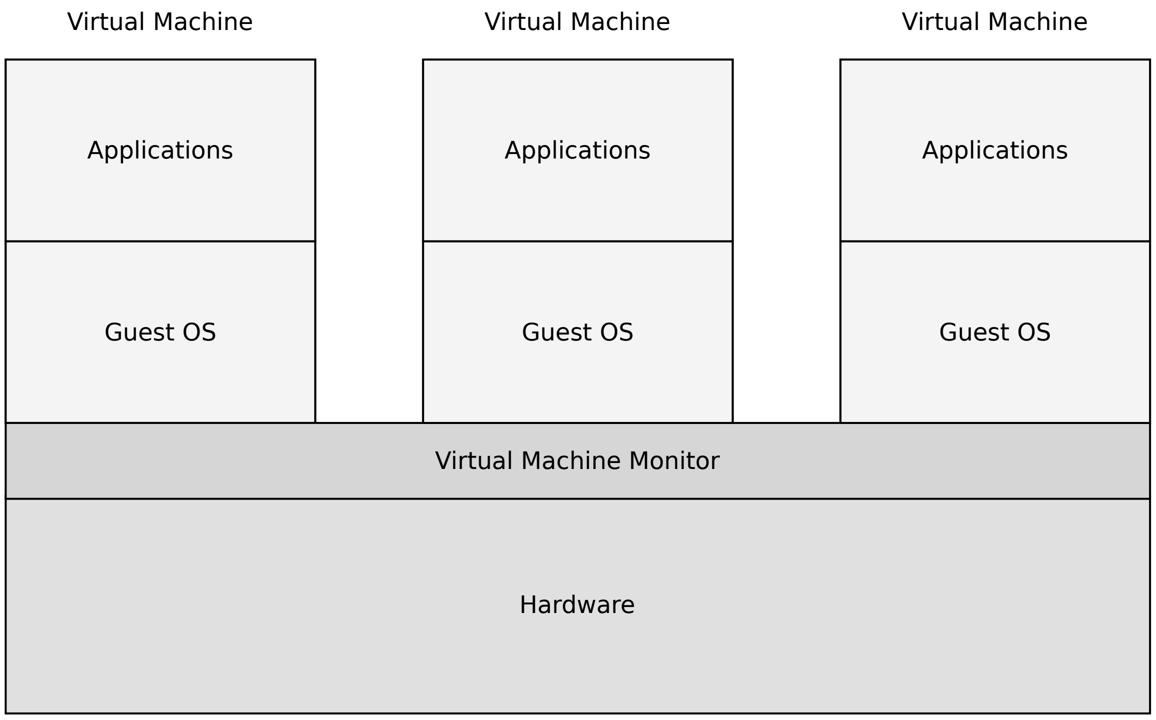

Figure 2.2: Hardware level virtualization

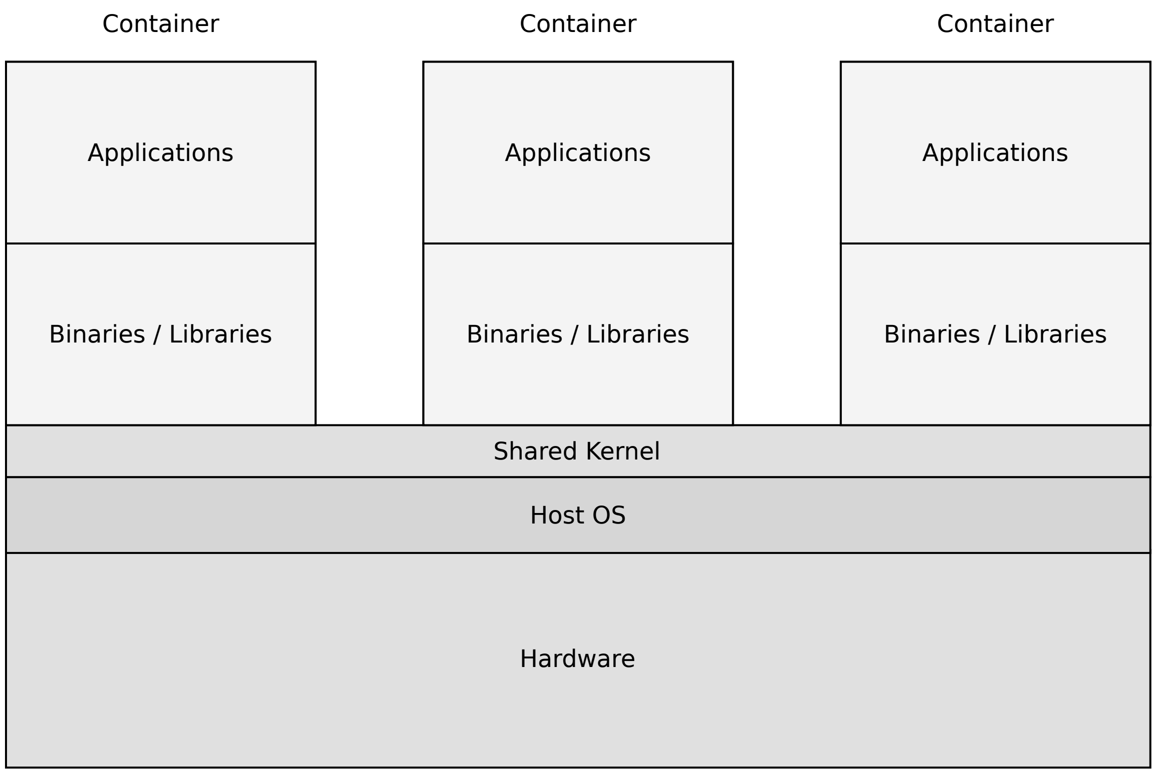

Figure 2.3: Operating system level virtualization (containers)

Despite all the pretty good advantages of virtualization, it is not ideal due to the overhead of having a completely installed operating system on each *virtual machine* which makes it hard to scale efficiently. On the other hand, virtualization on the operating system level such as containers is lighter-weight and gets rid of hardware-based virtualization problems. In containers, the operating system kernel is shared among different containers, each of which is a separated package of Unix-style processes [Mer14a]. With these containers, servers can be configured and managed easier but managing many tasks, such as concurrency, memory management and database access still represents a burdens. Moreover, servers configuration in general is hard, complex and an error-prone process [XLR+04]. In addition, applications are limited by the servers' startup time to scale efficiently. Consequentially, serverless computing the new computing model hides the complexities of managing servers, get rid of operations and handling different terms such as scalability and high availability. Serverless computing is a new computing model that exploits containers to execute users' code rather than a whole application.

In serverless computing, the deployable unit is a snippet of code or even a whole container which is executed only by an explicit request or as a response to an event. It changes the way of architecting and implementing cloud-native applications in a way similar to microservices architecture except that in servlerless computing a microservice is executed only upon a request.

2.3.1 Basic Principle

The basic principle behind serverless computing is similar to traditional servers such as *Apache Tomcat* [Fou16]. In *Apache Tomcat*, there is a pool of threads (The executor). Whenever there is a new incoming request, an available thread within the pool is used to handle that request. Similarly, serverless computing providers such as OpenWhisk has a pool of containers. Whenever there is a new incoming request to execute a specific code or container that has been provided by the user, an available container executes that code and returns the result to the caller (requester). As these containers and threads whenever they finish their work or are interrupted by any signals such as timeout, they are returned back to the pool and are marked ready to handle new requests. The key here is that these containers are shared among different users (multi-tenancy) and are managed by the serverless computing provider. Thus, all complexities of servers are hidden in such a way that users do not need to involve with any servers. Figure 2.4 shows an example of an abstracted overview of the basic principle of serverless computing.

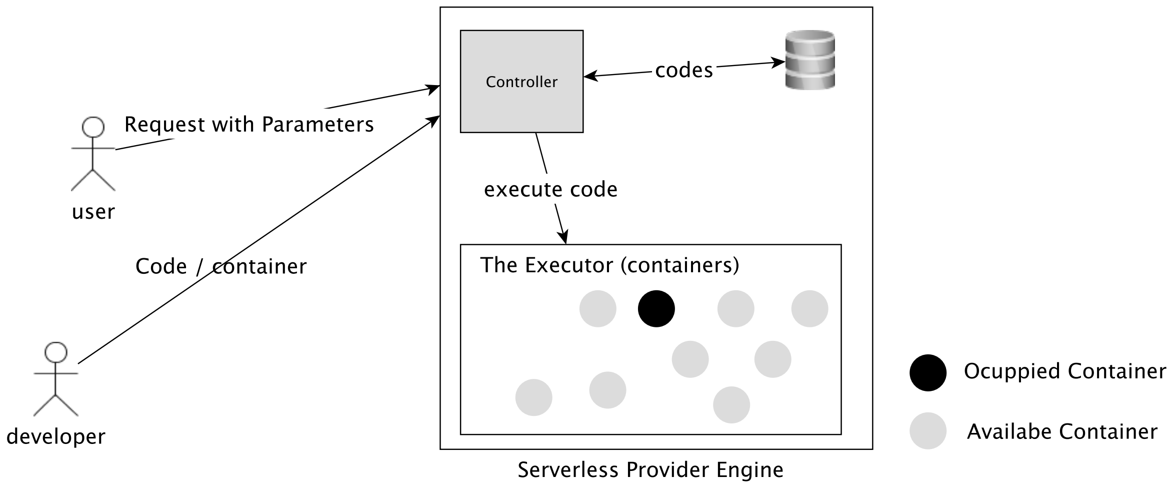

Figure 2.4: Abstracted overview of serverless computing

2.3.2 Advantages

Highly and rapidly scalable applications can be created on top of serverless computing, where the more requests come in, the more containers are created to handle these requests. They can be reused for new incoming requests in a very fast manner. All of these operations of handling and configuring the underlying infrastructure is abstracted and handled by the provider. Thus, operations such as managing scalability, availability, maintenance, monitoring and other operations can be discarded, which allows organizations to focus more in application logic. Furthermore, since containers are reused and users' code is only executed whenever it is invoked. Resources are also highly utilized and terms like over or under provisioning are hidden.

2.3.3 Disadvantages

Although serverless computing has several advantages, unfortunately, many aspects are abstracted and hidden. Its providers take care of setting up the infrastructure of fundamental resources such as the type of *Central Processing Unit (*CPU*)* that are needed to execute users' requests. Thus, users with specific infrastructure requirements cannot change the underlying resources to meet their requirements. Moreover, containers' pools are shared based on the multi-tenancy principle where a container can be reused to execute a code of different tenants. Therefore, many concerns are rising and security is one of them such as the concerns of accessing other tenants' data within a container that may has been used by other tenants.

2.4 IBM Bluemix OpenWhisk

The Serverless market has been incredibly increasing due to the benefits it brings. Currently, there are many serverless computing providers who offer different serverless computing services. This section provides a bird's-eye view of one of the new serverless computing services provider (IBM OpenWhisk). OpenWhisk is an open-sources cloud-based, serverless, distributed and event-driven engine that provides event-driven programming services which allow you to build powerful and convenient event-driven applications.

2.4.1 High Level Architecture

Since OpenWhisk is event-driven, it allows organizations to build efficient and powerful event-driven systems by providing all tools needed to interact with the events. Event producers emit events to OpenWhisk through different means, e.g. direct REST *Application Programming Interface (*API*)* requests. In order to receive such events in OpenWhisk, event producers need to be configured probably. As result of event emitting, an action or a sequence of actions are executed with the event payload as parameters. Actions can be written in JavaScript(Node.js), Swift, Java, Python or even can be a Docker container and it can be user-defined or even a ready-to-use action within OpenWhisk ecosystem such as “send an SMS” action in Twilio package or actions that are shared by another user. Figure 2.5 shows a high level architecture of OpenWhisk [Ope16].

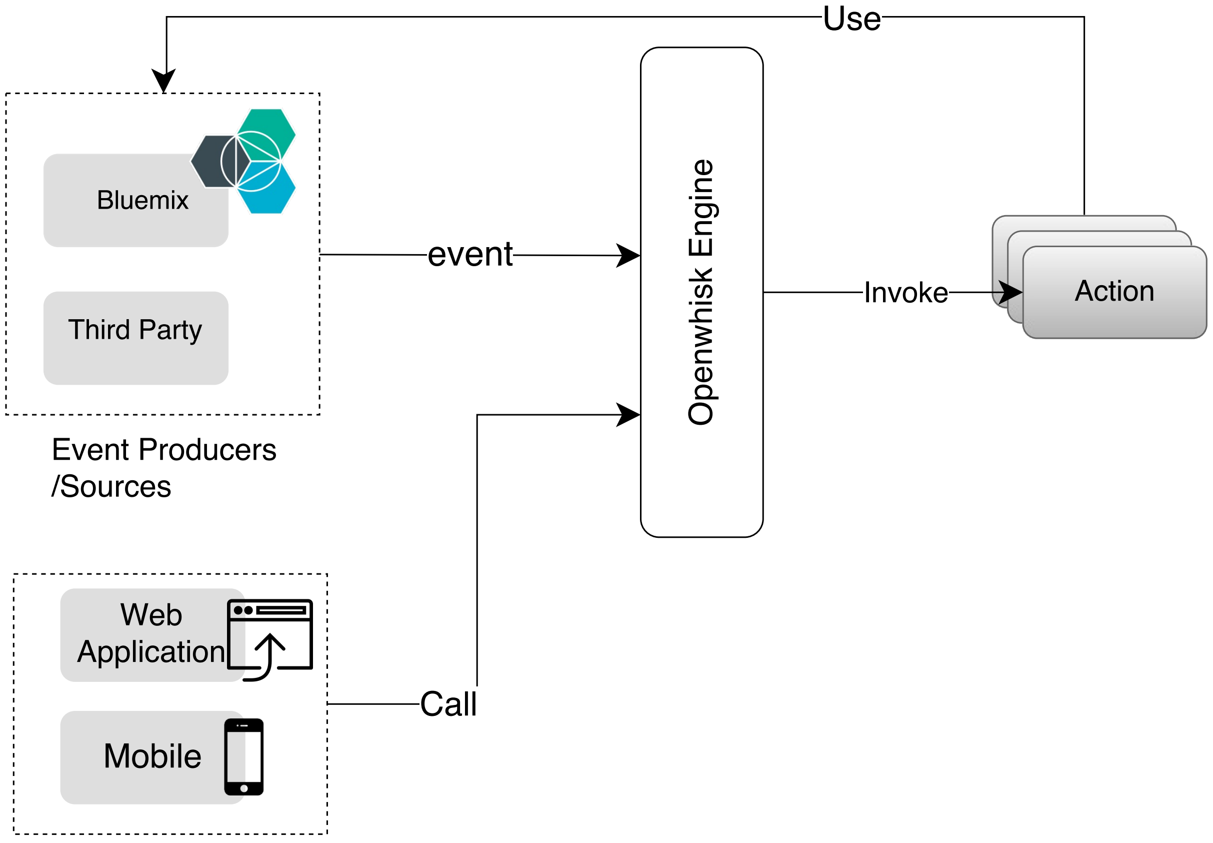

Figure 2.5: High level architecture

2.4.2 OpenWhisk Entities

OpenWhisk provides different tools and means to interact and respond to events. The following is a list of OpenWhisk entities:

- **Action**: A snippet of code or a container that represents the application logic and it is executed only upon a direct request or as a result of rule activation. In Listing 1, an example of JavaScript action is to calculate the speed of a moving objects as well as the invocation command and the result as *JavaScript Object Notation (*JSON*)* object.

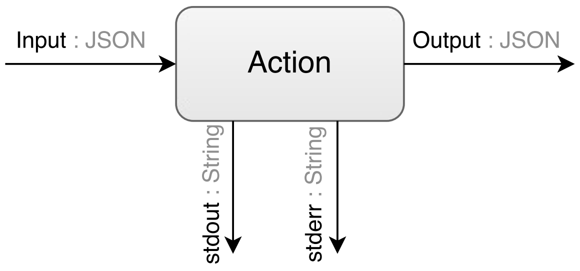

Figure 2.6: OpenWhisk Action

- **Actions Sequence**: Pipeline of actions which are chained together by their output, so that, the output of an action is passed as input to the next action in the chain. Figure 2.7 shows a high level architecture of actions sequence.

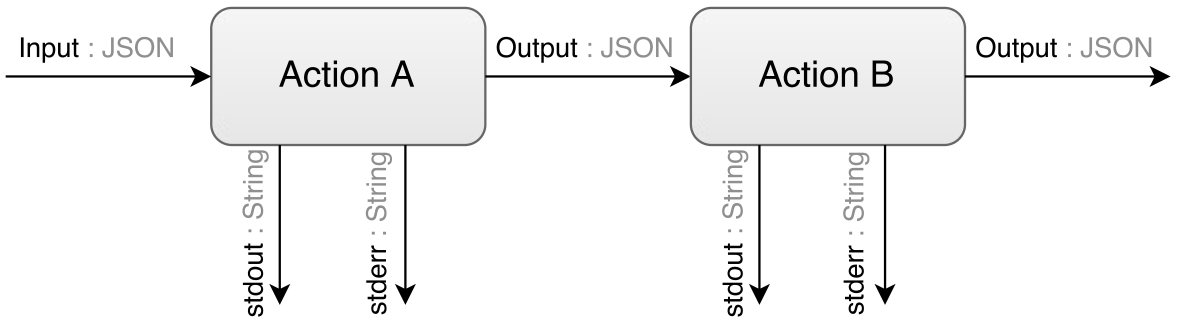

Figure 2.7: OpenWhisk Actions Sequence

- **Trigger**: A class of event types where each trigger represents a unique type of events such as a trigger for incoming IMAP emails which is fired whenever there is an incoming email.
- **Rule**: Association between triggers and actions, and it is kind of bindings. Each rule is a *1 to 1* relationship between a trigger and an action or sequence of actions. A rule is activated as a result of trigger firing. Consequentially, the associated action or sequence of actions are executed with the payload of the trigger.

```javascript
/**
 * An action to calculate speed of a moving object, it recieves time in seconds and
 ↪  distance in meters and return speed in meters per second
 */
function main(parameters) {
    console.log("Incoming request with parameters: ",parameters)
    return new Promise(function(resolve, reject) {
        if (!parameters.hasOwnProperty('time') ||
 ↪  !parameters.hasOwnProperty('distance')) {
            console.error("time, distance or both parameters are missing")
            reject("missing required parameters")
        }
        if (parameters.time <= 0) {
            console.error("time should be larger than zero")
            reject("Time value is invalid")
        }
        resolve({"speed": (parameters.distance/parameters.time)})
    })
}
```

```
#Invoke example action through terminal
$ wsk action invoke example -p time 120 -p distance 50
```

```
//Output of invocation (JSON)
{
  "speed": 0.16666666666666666
}
```

Listing 1: Example of JavaScript action

- **Feed Action**: A special action that manages feed of events from a specific event producer (i.e. {create, pause, unpause and delete). Normally, feed actions are used by triggers to control subscription through the trigger itself (i.e. 'wsk trigger create newComment –feed /whisk.system/github/webhook -p' and 'wsk trigger delete newComment' to delete the subscription). It has different life cycle events such as delete, create, pause and unpause.
- **Feed**: A separate application or service that produces events, manages subscriptions or does additional procedures to get events from another service in special cases. In other words, it is stream of events that feed OpenWhisk to fire OpenWhisk triggers and are subscribed by feed actions.
- **Activation**: A record that contains information about action invocation, triggers firing or rules activation. The information includes a unique id, time of start and end of execution, version number, as well as the logs and the result returned. Listing 2 is an example of an activation record. Such information provides important

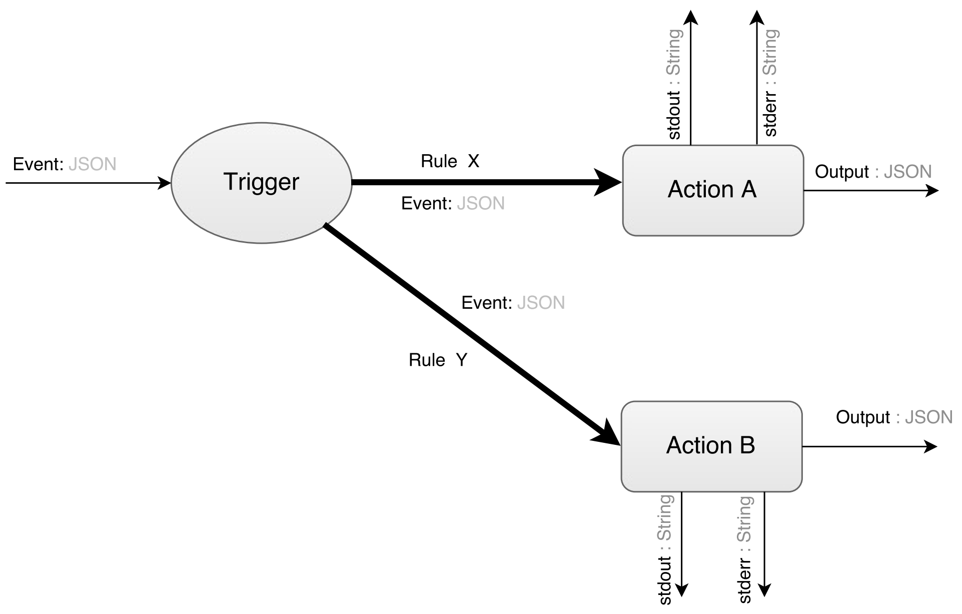

Figure 2.8: Relationship between triggers, rules and actions

information which can help in debugging and analyzing systems built on top of OpenWhisk.

- **Package**: A group for organizing related actions and feed actions. Packages in OpenWhisk can have global parameters, so that, whenever an action or a feed action is invoked, the global parameters of the package are passed as well. Figure 2.9 shows an example of Watson IoT platform package, which contains a feed action *webhook* to register receiving events which are sent to the platform, as well as other actions to interact with the Platform.
- **Namespace**: Class of all entities; packages, actions, feed actions, triggers, rules and activations which belong to a namespace.
- **Packages and Namespaces**: Each namespace N may contain a set of packages, actions, feed actions, triggers, rules and activations. Each package may contain actions, feed actions, triggers and rules.

2.4.3 System Design / Architecture

As shown in Figure 2.10 (inspired by [Sut16]), OpenWhisk is a microservice-based system where each component is a microservice that is implemented, tested, integrated and deployed independently. OpenWhisk consists of 1) Edge, 2) Controller and 3) Invokers. The edge is the component that manages the access to the system, which contains a

```
{
    "activationId": "7dbe1c66d54943d9a963a268518aa103",
    "annotations": [],
    "end": 1470755469209,
    "logs": [
        "2016-08-09T15:11:09.202059151Z stdout: Incoming request with parameters:  {
    ↪  distance: 20, time: 120 }"
    ],
    "name": "actionExample",
    "namespace": "talmaam@de.ibm.com",
    "publish": false,
    "response": {
        "result": {
            "speed": 0.16666666666666666
        },
        "status": "success",
        "success": true
    },
    "start": 1470755468651,
    "subject": "talmaam@de.ibm.com",
    "version": "0.0.12"
}
```

Listing 2: Activation example

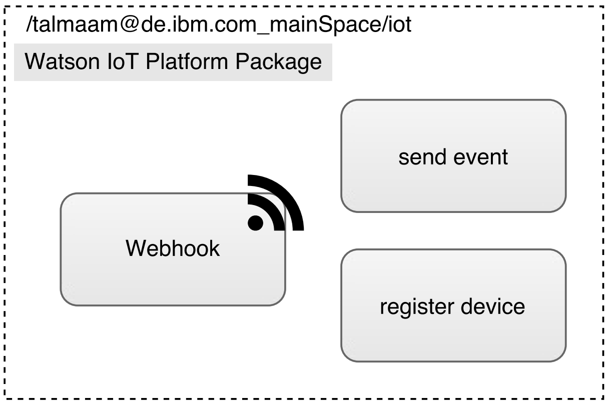
/talmaam@de.ibm.com_mainSpace/iot
Watson IoT Platform Package
send event
Webhook
register device

Figure 2.9: Package Example: Watson IoT Platform Package

proxy where OpenWhisk CLI and the different OpenWhisk *Software Development Kit (*SDK*)*s can be downloaded from. In addition, it forwards the different requests to the controller to handle it. The controller receives the request such as package creation. Before processing the request, the controller checks whether the request is authorized, and if the request is not an action invocation, the controler then serves it and responds directly with the result. If the request is an action invocation, the controller assigns it to an available Invoker to execute the action. Each Invoker has a pool of containers (executors) of different runtimes, an Invoker receives the request and assigns it to an available executor or creates a new executor to execute it and store the result or the invocation state into an activation record. All entities and entitlements are stored persistently in a document-based database (Cloudant/CouchDB). To ensure availability and maintain reliability, all components are monitored by explicit registration and repeatedly update their states through a monitor agent to the monitoring dashboard.

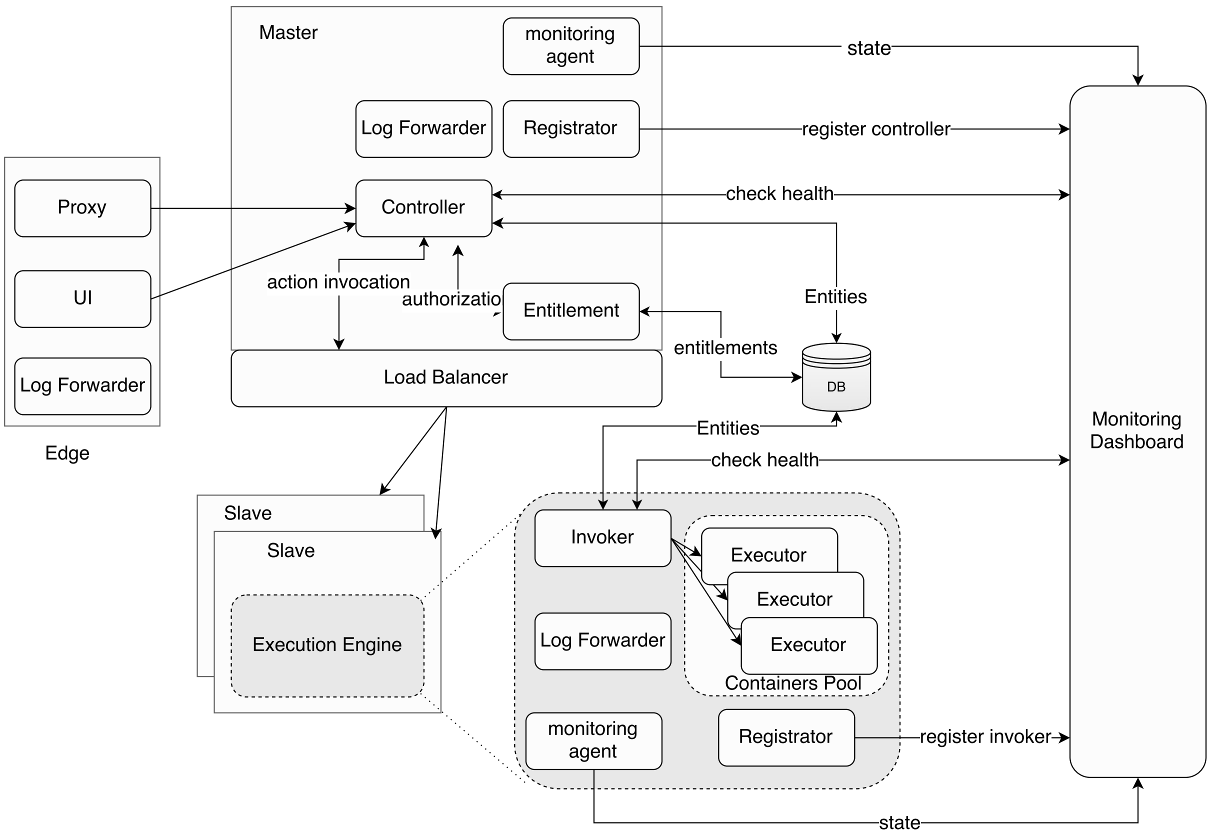

Figure 2.10: System Architecture

2.4.4 OpenWhisk Services Ecosystem

Powerful applications outsource some of their secondary work, tasks or features to external services such as storage and database capabilities to get rid of many complexities like taking care of eventual consistency in databases. They also allow applications to focus more on business logic and improve the expenses. OpenWhisk provides an ecosystem of a collection of packages for external services which help applications to use such services efficiently. This master thesis focuses on enriching this ecosystem. Chapter 4 – Services Categorization, Integration and Enablement provides more information.

2.4.5 Specifications and Characteristics

To have a better understanding of OpenWhisk, different characteristics and specifications of OpenWhisk are described as follows:

- **Openess**: OpenWhisk is open-sourced, and therefore, its code is exposed and publically accessible, and developers and specialists can see what is under hood, contribute and get involved. Typically, being open-sourced makes hard-to-measure technical aspects such as reliability and security more visible. Furthermore, openness provides many business benefits as well, such as the flexibility by licenses and escaping from vendor lock-in [MF07].
- **Extensibility** One of the special cases of design for change is extensibility which is the ability to add new features and capabilities with slight or no effects on the existing design of a system [Par79]. OpenWhisk allows to add more action runtimes (action containers) with no changes on the current design of OpenWhisk. Adding more actions runtimes can be done through action containers API except light changes on the controller and invoker to add some properties of the new runtime such as the requirements to execute an action within the new action container (action runtime) (e.g. “code” for JavaScript action and “image name” for containers) [Sut16].
- **Throughput**: OpenWhisk caches the executed actions (actions containers), which leads to higher throughput of actions execution and improvement of the overall throughput of systems that are built on top of OpenWhisk.
- **Quality**: Generally, open-sourced software are known to be of a higher level of quality due to the open-source community contributions, peer reviews and suggestions for imporvements.
- **Immutability and Versioning**: Actions are immutable, and to make changes on actions, changes are done locally and then they are committed to create a new version. Versions can be accessed through the different interaction means such as CLI, REST API and the UI.
- **Asynchronous and Synchronous Invocation**: Variation of invocation paradigms provides more options to support different needs and requirements of systems. Thus, more powerful systems can be built by using both paradigms. OpenWhisk provides the choice to invoke actions synchronously (blocking) as well as Asynchronously (non-blocking). In the case of blocking invocation, the process of invoking an action is blocked till the action returns an output, throws an error or triggers an interrupt signal (e.g. timeout). In case of non-blocking, an activation

id is returned immediately which may include the output of the execution in case of success invocation.

- **Real-Time Pipeline Processing and Content Enriching**: Actions sequence provide a mean of realtime pipeline processing, where an output of an action is passed as input of the next action in the chain. Actions sequence can act as a content-enriching as it enriches the input from the previous action in the chain and passes it as input to the next action.
- **Sharing**: Entities such as packages, actions and feed actions can be publically shared which allows other users to get benefit from it.
- **Statelessness**: Statelessness is a design paradigm where the state data of an action is separated and hosted somewhere else. Actions are stateless where statuses are not available across invocations. Thus, state data should be externalized, and this can be achieved by using other services.
- **Request-Response Pattern**: OpenWhisk is accessible through REST where a requester sends a request such as a request to invoke an action to OpenWhisk. Based on the execution paradigm specified in the request, OpenWhisk responds by either an activation Id in the case of non-blocking execution or by the activation record itself (including the execution output) in the case of success blocking execution.
- **Ordering**: Order of Actions invocation or firing triggers is not guaranteed. OpenWhisk is accessible through REST API which is HTTP/*Transmission Control Protocol (*TCP*)* based protocol where requests are not ordered. Hence, ordering is not guaranteed [Ope16].
- **Idempotent**: Idempotent operation is an operation that can be applied multiple times with no change on the result. This can be expressed mathematically as $f(x) = f(f(x))$. OpenWhisk actions are idempotent, even if OpenWhisk does not enforce this property [Ope16].
- **At-most-once Semantic**: When requesting for action invocation, OpenWhisk records the incoming request and dispatches a new activation for it. OpenWhisk replies with the activation Id in the case of non-blocking invocation to confirm receiving the request. OpenWhisk will attempt to to invoke the action once with one of the following output [Ope16]:
  - **Success**: Invocation is completed with successful action execution.
  - **Application error**: invocation is succeeded, but an error within the action itself is returned on purpose.

- **Action developer error**: Action invoked successfully with abnormal and uncompleted execution due to some problems with the action implementation itself such as existing of syntax errors.
- **Whisk internal error**: OpenWhisk is unable to invoke the action, and the result is recorded in the *status* object within the activation record.

2.5 Serverless Computing Providers

Serverless computing is a hot topic in cloud computing due to its significant impact on software architecture. This section goes through different serverless computing providers and explores them to provide a clear overview about of the differences between them. Thus, users can decide which provider to use based on their requirements. Here, we will briefly analyze the most popular serverless providers from different aspects such as the different system limitations.

2.5.1 Amazon Web Services Lambda

Besides the most popular service by AWS *Elastic Compute Cloud (*EC2*)*, AWS Lambda is also getting more lighter. AWS Lambda is the first serverless computing services provider (refer to 2.3 for more information about serverless computing). Lambda allows you to upload your code with its dependencies (called Lambda function) and execute such functions with AWS infrastructure. Hence, all complexities of the underlying infrastructure are hidden. Functions can be written in NodeJS(JavaScript) *v0.10.36* and *v4.3.2* but not *v6.0*, Python *2.7* or Java *8*. In addition, functions can be executables as they can be run using spawning child processes [Wag15]. Unfortunately, there is no support for containers which is a good option, especially when there is a need to execute binaries and other non-supported runtimes without spawning child processes. Lambda functions can be invoked through the *User Interface (*UI*)*, AWS CLI or through the REST API using AWS API Gateway. Unfortunately, there is no support for direct invocation through HTTP requests which can be done by separating API manager (AWS API Gateway). Lambda also supports versioning (Immutable versions) through Lambda Aliases, which is a pointer for a specific Lambda functions, each version of the same function has different Alias. As any other serverless computing provider, AWS Lambda limits the underlying resources as shown in table 2.1 [Ser16]:

Scalability	automatic & transparent
Max of code entity	250 MB (code/dependencies)
Concurrent execution	100
Max execution time	300 seconds
Max memory	1536 MB
Programming languages	JavaScript(NodeJS), Python and Java
Dependencies management	Packaging code and dependencies (ZIP or *JAR*)
Deployments	code/zip (Lambda, *S3*)
Versioning	Aliases
***HTTP* support**	Only through *API* Gateway
Logging	CloudWatch
Authentication	IAM
Openness	closed-source and proprietary system
Pipelining	No
Sharing	No
Scheduling service	Yes

Table 2.1: AWS Lambda limits & properties

Besides the above resources limitations, quota plans can be used to limit a specific consumer with predefined plans, so that, a single consumer is restricted to a specific number of function executions. Lambda Functions executions are logged using the AWS *CloudWatch* service. The logs contain different records such as the memory used, the maximum memory that is available for the function, execution duration time and the code size.

2.5.2 Google Cloud Functions

Google Cloud Functions at the moment of writing this thesis is still an alpha. Like any other serverless provider, Google Cloud Functions provide services for executing codes in serverless manner in response to events or as a direct invocation through different means. Until now, JavaScript (NodeJS) is the only supported programming language. Google Cloud Functions supports NPM to manage NodeJS dependencies. Invocation and management of functions can be done through Google Cloud CLI as well as a built-in support of HTTP requests. Google Cloud Functions have two flavored functions, HTTP and *Background* functions. HTTP *functions* are these functions which can be accessible through HTTP requests, such functions are fully HTTP controlled which can handle all HTTP operations. The other functions called Background Functions that are executed on response to events. The only difference between the two mentioned flavors is the signature of the function. Any updates on functions are versioned based on the GIT versioning concept. In addition, logging in Google Cloud Functions is managed by the Google Cloud Logging service. In addition. logs can be also logged to other services such as StackDriver using specific SDK which provides more features to store, search, analyze, monitor, and alert on log data and events from Google Cloud Platform in general.

Scalability	automatic & transparent
Max of code entity	no limits
Concurrent execution	no limits
Max execution time	no limits
Max memory	1024 MB
Programming languages	JavaScript(NodeJS)
Dependencies management	*NPM*
Deployments	ZIP (Cloud Storage
Cloud Source Code Repositories)	
Versioning	Cloud Repositories Service (branches and tags)
***HTTP* support**	full support
Logging	Stackdriver Logging
Authentication	*OAuth* 2.0
Openness	closed-source and proprietary system
Pipelining	No
Sharing	No
Scheduling service	Yes

Table 2.2: Google Cloud Functions limits & properties

2.5.3 Microsoft Azure

Another main serverless computing services provider is Microsoft Azure Functions. Azure Functions allows users to execute code or executables as an explicit request or as a response to events. Currently, the supported runtimes are C, NodeJS, Python, F#, PHP, Bash, CMD, Java and also executables. Regarding the dependencies, Azure Functions supports NuGet and NPM. Moreover, they have a full support of HTTP almost the same as Google Cloud Functions. A specific application folder structure as well as an additional configuration file are required to deploy a functions application. Hence, more complexities are introduced when it is compared to other serverless providers. Unlike the other providers, maximum concurrent functions invocations in Azure Functions are limited by the size of memory assigned and scalability is nontransparent where users are able to see how many instances of computing power are used.

Scalability	automatic & nontransparent
Max of code entity	no limits
Concurrent execution	based on the memory assigned
Max execution time	no limits
Max memory	1536MB
Programming languages	C#, Node.js, Python, F#, PHP, Batch, Bash, Java, or executable
Dependencies management	NuGet and NPM
Deployments	*SCM* services, *FTP* and Web deploys
Versioning	-
***HTTP* support**	full support
Logging	embedded
HTTP support	full support
Logging	Stackdriver Logging
Authentication	*OAuth* 2.0
Openness	closed-source and proprietary system
Pipelining	No
Sharing	No
Scheduling service	Yes

Table 2.3: Microsoft Azure Functions limits

2.5.4 Auth0 Webtasks

Webtask is another serverless computing services provider. The main idea behind Auth0 Webtask is to execute NodeJs runtime based code upon a direct invocation. In Webtask, the only supported programming language is JavaScript (NodeJS runtime), and other runtimes can be supported through by using Transpilers (Transpilation). It is a source-to-source compilation where a source code written in programming language is translated into another programming language. In case of Webtasks, it is translated into NodeJS using specific NodeJS modules to achieve this. Dependencies in Webtasks are limited to the most popular 600 modules in NodeJS. The code units in Auth0 are called Webtasks, and a Webtask is a snippet of code written in JavaScript or another programming language (that can be traspiled as mentioned before). Webtasks have a full HTTP support, and it comes with three different function signatures (callback only, context and callback as well as a signature for the HTTP support function (request, response and callback)). Webtasks can be invoked and managed through two means, using REST API and Webtasks CLI. Authentication in Webtasks is token-based, and tokens are are used to restrict a specific user or an application from using specific Webtask. Hence, Webtask can be restricted by different logics, such as restricting a specific Webtask to execute code from a specific *Uniform Resource Locater (*URL*)*, or restricting the execution of a Webtask by a point of time (e.g. executing before or not after a specific point of time). Moreover, Auth0 Webtasks provides a mean of persistency during Webtasks execution, so that, states can be stored and accessed during different execution of a Webtask. Above that, standard output and errors are logged and can be accessed through Webtasks REST API and Webtasks CLI in a real-time manner.

Scalability	automatic & transparent
Max of code entity	no limits
Concurrent execution	based on the memory assigned
Max execution time	no limits
Programming languages	JavaScript(Node.JS)
Dependencies management	most 600 famous modules in *NPM*
Deployments	code files
Versioning	-
***HTTP* support**	full support
Logging	embedded
Authentication	Auth0
Openness	closed-source and proprietary system
Pipelining	No
Sharing	No
Scheduling service	Yes

Table 2.4: Auth0 webtasks limits

2.5.5 Overall Comparison

To have an overview of all serverless providers, table 2.5 is a summary of what have been discussed in this section, the table summarizes the differences between the observed providers in various aspects.

Provider	**IBM OpenWhisk**	**AWS Lambda**	**Azure Functions**	**Google Functions**	**Auth0 Webtasks**
Scalability	transparent	transparent	nontransparent	transparent	transparent
Max of code entity	48 MB	250 MB	no limits	no limits	no limits
Concurrent execution	100 per namespace	100 per region	based on the memory	no limits	no limits
Max execution time	300 s	300 s	no limits	no limits	no limits
Max memory	512 MB	1536 MB	1536 MB	1024 MB	-
Dependencies	few *NPM* modules	ZIP or *JAR*	NuGet and NPM	*NPM*	600 *NPM* modules
Deployments	code files/JARs	code/zip (Lambda, *S3*)	*SCM* services, *FTP* and Web deploys	ZIP (Cloud Storage & Cloud Source Code Repositories)	code files
Versioning	embedded	Aliases	embedded	Cloud Repositories Service (branches and tags)	-
***HTTP* support**	embedded and limited	API Gateway	full support	full support	full support
Logging	embedded	CloudWatch	embedded	Stackdriver Logging	embedded
Authentication	Basic Authentication	*IAM*	*OAuth* 2.0	*OAuth* 2.0	Auth0
Openness	Yes	No	No	No	No
Pipelining	Sequence Action	No	No	No	No
Sharing	Yes	No	No	No	No

Table 2.5: Serverless providers comparison

3 Related Work

This chapter intends to provide an overview of the related work of the core work of this thesis, in this chapter we explore how the main serverless computing services providers integrate and enable event sources as well as what are the different means to interact with events.

3.1 Amazon Web Services Lambda

AWS Lambda provides only what is called lambda functions, but there are no additional tools to support real-time pipelining or rule management. Event sources in AWS Lambda are limited to a few of AWS services which are *API Gateway*, AWS IoT, *Alexa Skills Kit*, *Alexa Smart Home*, *CloudWatch Events - Schedule*, *CloudWatch Logs*, *Cognito Sync Trigger*, *DynamoDB*, *Kinesis*, *S3* and *SNS*. As Lambda is a closed-source and propriety system, it is hard to have a deeper look at the current used mechanisms to subscribe and receive events from events sources. According to Lambda, both models (poll and push) are used to get events from the services mentioned above. Moreover, the absence of additional tools such as a tool to provide the ability to easily enable and disable single event sources from executing specific function introduces another deficiency of Lambda. In addition, event sources are not categorized, which provides a lower quality user experience [Ser16].

3.2 Google Cloud Functions

Event sources in Google Cloud Functions are limited to only two services *Google Cloud Pub/Sub* and *Google Cloud Storage*. Further different event sources might emit their events through *Google Cloud Pub/Sub*. *Google Cloud Pub/Sub* supports both models to receive events, push using what is called Webhooks and poll models. Google Cloud Functions is also closed-source. Therefore, it is hard to have a deeper look at the mechanisms used to integrate with the event sources. Like AWS Lambda Functions, event sources are not categorized. Anyway, Google Cloud Functions is still in Alpha, and therefore, it is early to consider it in the list as a mature player in the market [Goo16].

3.3 Microsoft Azure

Azure functions provide triggers and bindings, and they are, tools to interact to events. Unlike OpenWhisk, azure does not provide a mean to managing rules. Some of the Azure services as well as third-party services are integrated and can emit events to execute Azure Functions in response to it. They are, Azure DocumentDB, Azure Event Hubs, Azure Mobile Apps (tables), Azure Notification Hubs, Azure Service Bus (queues and topics), Azure Storage (blob, queues, and tables) and GitHub (webhooks). As Azure Functions is based on Microsoft WebJobs SDK, the open source SDK [Azu16a] provides tools to ease coding background-processing Functions. The registration for receiving events from event sources is done through WebJobs Bindings. The previous mentioned integrated services with Azure Functions are baked into the core WebJobs SDK, while other event sources and services can be integrated with Azure Functions by building WebJobs extensions Bindings [Azu]. Furthermore, a binding is an entity that is used to configure the source of an event to execute an Azure function as response (Trigger Bindings) and Non-Trigger Binding to allow using other services from Azure Functions such as sending a SendGrid email from Azure Functions. Trigger Bindings in the SDK provides Listener interfaces that can be used by the host (service producer) that will be used whenever there is a new event (either by poll or push models) to trigger an Azure Function to be executed. Moreover, there is no categorization for event sources, which also lowers the quality of users' experience [Azu16b].

3.4 Auth0 Webtasks

Unfortunately, there is no event-source or event-driven paradigm support, and executions of Webtasks are done upon an explicit request through the two Webtasks interaction means (REST API and Webtasks CLI) [Inc16].

4 Services Categorization, Integration and Enablement

This chapter describes our designed categorization system for OpenWhisk packages. In addition, various means which can be used to integrate and enable services and event sources to interact with OpenWhisk and execute codes or run containers in response to events. The chapter also explores the different criteria and approaches that are going to be used to propose the services to be integrated. Figure 4.1 shows the flow of work in this chapter.

Design Categorization System
Propose Services
Integrate Services

Figure 4.1: High level workflow of this chapter

4.1 OpenWhisk Ecosystem

Event processing requires a huge amount of computing power in a realtime manner which needs highly scalable computing resources to achieve the goal of EDA. Not only processing events is important, but also reacting to it by consuming or producing data and resources as response to such events is important. As OpenWhisk is a serverless, event-driven and distributed programming services provider, event-sources are one of the main ingredients in OpenWhisk. Without event-sources, OpenWhisk is a poor system. Thus, OpenWhisk provides an ecosystem of packages of different services that allow OpenWhisk users to easily subscribe for receiving events and react to them. Interacting with events in OpenWhisk is done by executing actions in response to it either by using users provided actions or using actions within the ecosystem such as using the *store* action within *CloudantDB* package.

This chapter intends to propose, implement, test and integrate the proposed services into OpenWhisk to enrich the ecosystem. The selection of the services and the concepts behind the integration as well as the challenges that might be faced will be discussed.

4.1.1 OpenWhisk Packages

Currently, the OpenWhisk ecosystem has different ready-to-use Packages. As an example, the following are some of them:

- **Github**: Contains a feed action (webhook), and it used by triggers to register for receiving different events from Github such as new commits or pull requests. Moreover, whenever a new event occurs, the trigger associated and registered by the webhook is fired. Consequently, actions or sequence of actions may be invoked.
- **Slack**: Contains an action to post a message to a slack channel.
- **Weather**: Contains an action to get hourly weather forecast using Weather API service.
- **Watson**: Contains different actions to interact with OpenWhisk such as converting speech to text, text to speech and translating texts.
- **Utils**: Contains different actions that may be used as utilities such as an action to split a string into array of strings.
- **Samples**: Contains various examples in different supported programming languages such as hello world actions.

4.2 Services Categorization

Categorization is a fundamental problem in cognitive science, and it is significant in observation, thinking, correspondence and consideration [FGS11]. The term categorization has different meaning from the point of view of the discipline involved [CL05]. According to Cohen and Lefebvre [CL05], categorization is the process of recognizing, differentiating and understanding objects and ideas and grouping these objects and ideas based on similarities into classes called categories [FGS11]. The importance of categorization comes from the need for differentiating between objects. So, without categorization, objects are incomparable, matchless and inimitable [FGS11]. Moreover, it is impossible to predict new unknown objects and ideas.

4.2.1 Types of Categories

- **Taxonomic**: Categories are organized into hierarchies of abstract categories that share common properties or features. For instance, *Personal Computer (*PC*)* and Mac both are computers that have CPU and memory.

- **Script**: Used to categorize entities that have the same role but not complementary roles in a script. A script is a schema for a routine event. For example, water and juice both has the same role of what to drink script.
- **Thematic**: Categorizes or grouping associated entities or entities that have a complementary relationship and at the same time, they are not similar (e.g. a computer and a mouse which forms a thematic pair because a mouse is used to control computer).

In the context of OpenWhisk, categorization is the process of grouping related actions and feed actions into packages, and grouping packages into categories to illustrate the relationship between the grouped packages. Consequentially, a few benefits can be gained:

- **Clarity of Knowledge**: Clear knowledge about package content and functionalities is provided where different services share similar content and functionalities. For instance, considering categorization based on domains, a domain that contains different IoT services are grouped into one category. It is obvious that this category shows that all of these services are related to the interaction with events in the world of IoT.
- **Accessibility**: Easiness to find, reach out and use services. In the case of large number of services, it is hard to find specific services easily, but with categorization, a service can be reached out through the expected category. For example, in considering the previous mentioned example, the IoT category, to reach out IBM Watson IoT Platform service, it can be easily predicted that it is located within a category called "IoT".
- **Common Traits**: Shows packages that share specific traits as a category, regardless of the similarity it is based on, such as a specific domain, feature, property or any other similarity. It shows that all of the category members share the similarity. Again, considering the example of IoT category, all of the category members share the same domain, and therefore, have similar traits.
- **Convenience**: Categorizing new packages is more convenient as it allows defining the boundaries and characteristics of the packages to be categorized, new packages can be categorized, and new packages can be categorized by making few predictions, and therefore, categorizing new and previously unknown services is convenient.

The mentioned three kinds of categories are applicable in the context of OpenWhisk, but not all of them make sense to use. Script and Thematic categories can form unlimited pairs and scripts contents. For example, possible script of IoT applications in IBM Bluemix is the Watson IoT platform to transfer events through *Message Queuing*

*Telemetry Transport (*MQTT*)* protocol, and a document based database such as Cloudant and an IOT rule-base real-time insights service such as Watson IOT Real-Time Insights. This can be applicable to Thematic categories, which leads also to big number of pairs that may depends on a precise use-case. Thus, Taxonomic categories are the most appropriate for categorizing OpenWhisk packages as the resulted classes of categories are more general, which may lead to few drawbacks. However, it can be discarded and search feature can be added to search within the same category or even through all categories. Moreover, other categories can be structured from the habits of users of combining different services that are not similar or have a complementary relationship to extrapolate scripts and thematic pairs. This may help users with specific use-case or a problem to get benefit from it. Anyway, that is not the focus of this thesis and it can be listed in the list of future work in chapter 7.

As mentioned, Taxonomic categories are based on similarities, as the majority of services are black-box. Thus, scopes of similarities are limited to vendor, openness and domain. The next subsection shows how such scopes can be used as similarities:

4.2.2 Categorization Similarities

Similarities in general can be scoped within different classes, due to the fact that the majority of services are black-box. Therefore, the characteristics or properties are limited to only three. In this thesis, we proposed using three general characters to categorize OpenWhisk packages, and these characteristics or similarities have been proposed with consideration of the different knowledge of users.

- **Vendor**: Different services may share the same vendor, which makes it an applicable similarity to categorize such services into the same category under the same vendor. For instance, the IBM Bluemix platform has various third-party services, and a possible categories based on vendor can be, an IBM and non-IBM services.
- **Openness**: From the source openness perspective, services can be divided into open-sourced and close-source services. Such classification can also be used to have two main categories, open-sourced and close-sourced services.
- **Domain**: Different services rely on a specific domain only. So, each service is categorized in a specific domain based on its functionalities. This resembles how IBM Bluemix *Cloud Foundry (*CF*)* applications and services are categorized. The only difference is that, in OpenWhisk there are more services in other domains that don't exist in IBM Bluemix, but they be powerfully exploited by OpenWhisk.

Since categories in Taxonomy categories are organized in hierarchies, the above similarities can be combined into a categorization system. For instance, we can start with the domain hierarchy with sub-hierarchies using other similarity such as openness and vendors. Listing 3 shows an example of such multi-hierarchal categorization.

- **Cognitive Computing**
  - IBM
    - AlchemyAPI
    - Visual Recognition
  - Cognitive Scale
    - Cognitive Graph
    - Cognitive Insights
- **Internet of Things**
  - Watson IoT Platform
  - Watson IoT Real-Time Insights
  - Watson IoT Driver Behavior
  - Flow Things

Listing 3: An example of multi-hierarchal categorization

4.2.3 OpenWhisk Categorization

To categorize packages in OpenWhisk, a categorization system needs to be designed to provide the benefits we discussed earlier. Therefore, we propose a categorization system for OpenWhisk based on Taxonomic categories.

In this section, we propose the domains used in categorizing OpenWhisk packages. Different domains are also grouped into one domain since they have common features. Domain categorization contains different sub-domains based on the parent domain, which provides meaning and simplicity of the content and reaching the services. Here is a list of a proposed categories using domains:

- **Cognitive Computing**

- **Internet of Things**
- **Workflow and Integration**
- **Data and Analytics**
- **Storage**
- **Social and Media Platforms**
- **APIs**
- **DevOps**
- **Communication and Security**
- **Mobile Computing**

4.3 Services Selection

After proposing a service categorization for OpenWhisk, we will evaluate which services that should be considered for integration. Therefore, the proposing process will be based on a criteria matrix through two main approaches: usage-based and use-case-based selection.

The list of services that can be integrated and enabled within OpenWhisk is long, and it may become tricky. Therefore, it might not be useful or will not add-value to OpenWhisk, and we propose a criteria matrix to help selecting and proposing different services into OpenWhisk. Moreover, different approaches to proposing the services are discussed under the two main approaches, use-case based and usage-based selection.

4.3.1 Selection Criteria

Whenever picking up, selecting and proposing services, different questions are raised: In which domain should be located? How many users that may get benefit if a service is enabled within the OpenWhisk ecosystem? Does the enabled service lead to a significant reduction in operations and infrastructure cost?. All of these questions guide us to the needed criteria. In this thesis, we do not mainly focus on costs, as OpenWhisk is still in beta, and it does not have a pricing model till the time of writing this thesis. From the listed questions, we can group all these criteria into two main approaches: usage-based and use-case based approaches. Moreover, the proposed list should contain various services in respect to both domain and vendor to avoid vendor lock-in and enable more services to serve a broader number of target users.

4.3.2 Usage-Based Selection

In this selection approach, *usage statistics* such as the number of active service instances and number of users is not ideal to use as a criteria for selecting and proposing services to be integrated and enabled within the OpenWhisk ecosystem. This is because it may end up with a list of services with identical domains such as only database services, and 2) new services with lower statistics can be more powerful and its usage might improved when it is integrated and enabled as an event source or used as an action to respond to events.

As mentioned, this approach may lead to an inconsistent selection, and therefore, we need another approach that allows proposing a list based on the criteria above. A use-case based selection approach is another approach that can be used to propose a list with the mentioned criteria.

4.3.3 Use-Case-Based Selection

Next, we evaluate services based on a set of pre-defined use-cases rather than the usage statistics, the evaluation process was carried out in cooperation with a team of architects and developers in IBM.

A use case or scenario-based selection is an approach that can be used to propose a list of services to be integrated into OpenWhisk, the proposing process based on pre-defined scenarios that different technologies and services from different vendors are involved. To have a well-defined scenario, A criteria matrix is proposed that include the general and specific services selection criteria for use cases selection.

4.3.3.1 Use-Case Selection Criteria

Picking up the most suitable use case might be tricky, and therefore, we proposed various criteria for selecting the scenario. An ideal use case, a use case that can generate different types of workloads, especially peaking workloads. This is to show how much powerful is serverless computing especially OpenWhisk in such case. Moreover, it should contain different services which rely on different domains from different vendors, This means that more options will be available for users and to avoid vendor-lock. In addition, the proposed use case should be distributed,especially in terms of its different components and its functionalities. It should also be simple and applicable to be used in real life.

4.3.3.2 Scenarios

- **EWS**: *Early Warning System* EWS is a system for early warning in catastrophic and critical security situations. The importance of having EWS increases day by day. In this scenario, an EWS is proposed to be built on top of OpenWhisk that involves different services, including third-party services. Implementation of a very simple scenario can be using different sensors scattered all over the areas intended to be covered. All of them are connected to an IoT platform as well as a mobile application used by people to report incidents and possible terrorists attacks. Then, whenever a change occurs, a trigger is fired or an action or sequence of actions are invoked in OpenWhisk. Such actions can be firing a public alarm, report to authorities, etc.
- **Serverless *Software-Defined Networking (*SDN*)* Controller**: As SDN Controller is event-driven where a specific code is executed based on events such as a new message is sent. In the SDN world, routers send events (e.g. message arrived, message sent or node joint). In the other hand, events and actions on the controller (e.g. change routs or new node joint). The main good thing is that the controller here is an event-driven and microservices based. This means that there is more scalability of the parts having more workloads as well as a higher utilization of resources.
- **Accidents Monitoring**: In many cases car accidents happen in empty roads (either because of weather impact, animals on roads, lack in roads maintenance, etc.), which leads to late accident reporting. Therefore, late health care can cause death, as a solution, reporting system can be built on top of OpenWhisk. Whenever an accident occurs, a trigger is fired to invoke an action or sequence of actions which will report the accident, its location and the number of people inside the vehicle.
- **Production Lines in Manufacturing**: In manufacturing, there are many of situations where OpenWhisk can be exploited (e.g. scheduling of Production Line). This situation is not going to run lifetime, but it is usually based on a specific event. In car manufacturing, there are groups of robots, and in this case, scheduling their work is necessary and dynamic scheduling based on the type of product manufactured, especially in multi-line production can be done through OpenWhisk, where a specific business logic is executed on time for a specific event.
- **LEGO Mindstorms Controller**: Mindstorms robots can be connected to different sensors and actuators (e.g. motion, RFIDs, light, etc.), in combination with

OpenWhisk and different services, thus leading to puissant applications. Here, a scenario to control Mindstorms is proposed by using Mobile by accelerometer sensors, activate a specific sensor and analyze data in the cloud using Watson, etc..

- **Traffic Violations**: Breaking a traffic sign is usually detected by a camera. The car plate photo is uploaded into the cloud, thus, a lifetime running server is used. OpenWhisk can play an important role in providing event-driven serverless traffic monitoring system. Breaking the traffic sign (Event) will fire a trigger to invoke sequence of actions to report the violation to the system or execute an action to close a street, etc.
- **Serverless Bots**: Bots markets have been incredibly increasing, and therefore, there are different awesome bots in different platforms. When it comes to Bots, there are unlimited ideas where OpenWhisk can be used as a back-end in mix with different services. The followings are only examples of its use:
  - Docs Uploader: Bot to store data through OpenWhisk to a back-end storage (e.g. Object Storage Service).
  - Convert a Markdown file into HTML.
  - General Purpose Controller bot.

Based on the use-case selection criteria, we can have a long list of use cases that comply with the use case selection criteria. To pick up the most suitable scenario, a brainstorming session was accomplished with a group of developers and architects within IBM. After some discussions, based on voting (Figure 4.2), we have picked the EWS scenario. This is because it is the most applicable and useful scenario in real life. It also adds value to OpenWhisk, and it complies with the selection criteria.

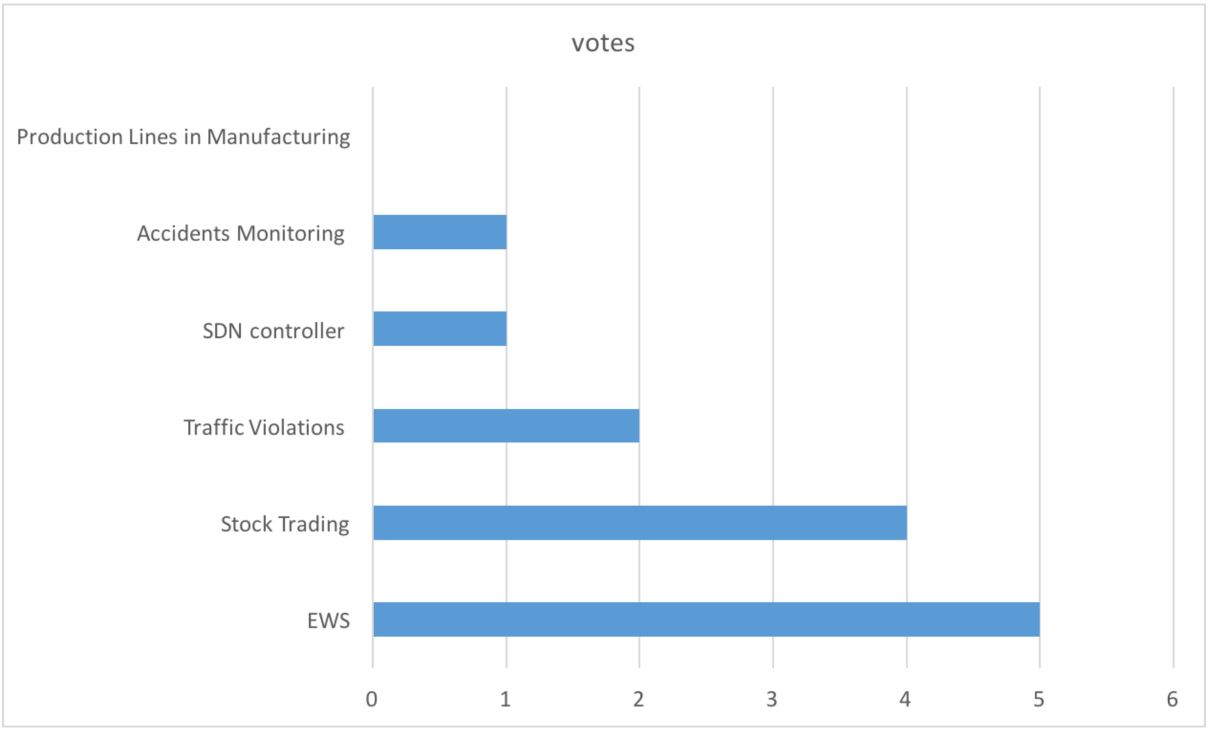

Figure 4.2: Scenario votes

4.4 Services Integration

4.4.1 Overview

Enhancing users' experience and adding value to OpenWhisk are the main goals of enriching the OpenWhisk ecosystem. Adding and enabling more services into the ecosystem usually do not represent a straightforward process in which different methods can be used. In addition, not all of them are always applicable. Not only using a service from OpenWhisk to create or update a resource or data in a service is needed, but emitting and receiving events to OpenWhisk is the main parts. However, it should be noted that a few complexities may be faced during the integration of different services into OpenWhisk. Even if the integration is accomplished in different ways, a standard template will be required.

In this section, different methods of integration services into OpenWhisk used in this study are explained. Moreover, a template which was designed to standardize the integration packages is discussed.

4.4.2 Integration Methods

Based on what was mentioned in the previous subsection, integrating more services will obviously enhance users' experience and therefore, adds value to OpenWhisk. Mainly, there are two main approaches to enabling receiving events by OpenWhisk: 1) Event producer (service) which provides means to subscribing for events such as webhooks, or 2) when an event producer doesn't support the means in **1**, then, another method should be used such as polling. In this subsection, we discuss the different methods that can be used to integrating services into OpenWhisk:

- **Polling** Polling or busy-wait polling is a mechanism for communication, usually, between *Operating System (*OS*)* and I/O devices where synchronous operations to query for the status of I/O devices are made. Polling has been also used for communication between clients and servers where a client sends a request to query for a status or a change in the server, and therefore, the client is blocked until the server replies. Polling leads to a high consumption of resources due to the number of connections opened to do the polling (Figure 4.3). To minimize the high consumption of resources, *Long-Polling* can be used when there is a longer blockage. In this case, the server replies whenever an event occurs. Otherwise it is blocked, and as a result, a small number of long connections are opened. Hence, an additional management for long-life connections is required such as supporting and managing multi-threads to provide asynchronous operations to prevent long blockages [Inc13]. In the context of OpenWhisk, polling can be done either by periodically invoking an action to do polling, or by using a feed/trigger provider to poll on behalf of the user (Figure 4.4). For the former, due to the timeout limitation, it may not be an efficient option.

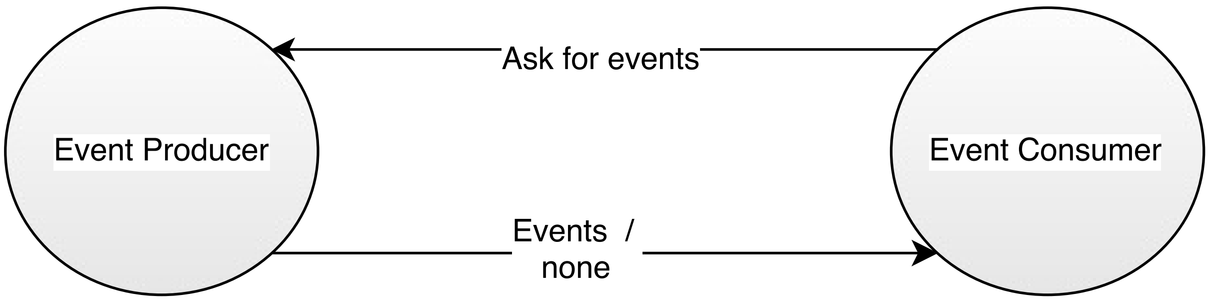

Figure 4.3: Polling

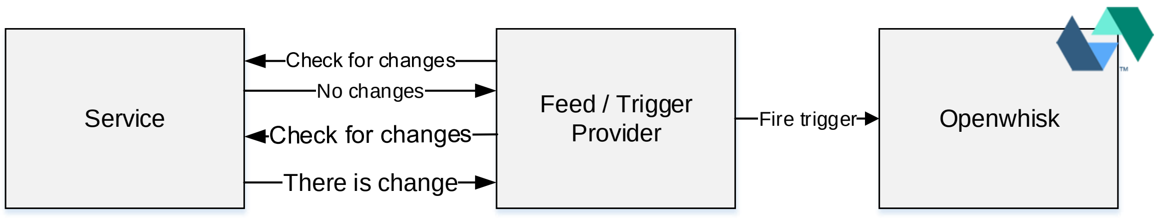

Figure 4.4: Polling using feeds

- **Webhooks** In the software development world, there are many cases where developers need to execute specific code as a response to specific events within an OS or application. Hooks are used where an application is hooked to specific components in OS or an application to execute specific code either as an extension to the component feature or as a response to an event generated by a specific component. In web world, a webhook refers to a mechanism to define callbacks over HTTP, usually a HTTP POST request to a URL provided by the user. Thus, there is almost no overhead in the client's side [Inc13]. Webhooks are used to receive notifications or events, and Github webhooks can be a good example where the user registers for specific events. Whenever an event occurs, Github calls back the HTTP with the event in payload. Webhooks are easy to implements [Lin07], because it is a request done by the service/server whenever an event occurs. In many cases, a large number of events may occur at once, and therefore, peaking workload may hit the callback host. Hence, the callback host should cope with such workloads. For OpenWhisk, a feed action is used to configure a webhook on an external service to fire an OpenWhisk trigger whenever an event occurs.

- **Websockets** Websocket is a two-way communication protocol over TCP. Websockets stream messages between a server and a client. Unlike HTTP, websocket communication starts with a handshake, and then, data are streamed between the client and the server with no additional handshakes. Websockets can be ideal for real-time communication which makes it a good options for event-streaming. In the context of OpenWhisk, communication through websockets can be done through normal actions, but unfortunately, it might not be a good option due to the limitation of maximum execution time of actions. As a solution, a feed/trigger provider can be used to open websockets connections on behalf of a user and fire a trigger whenever an event is streamed (Figure 4.5).

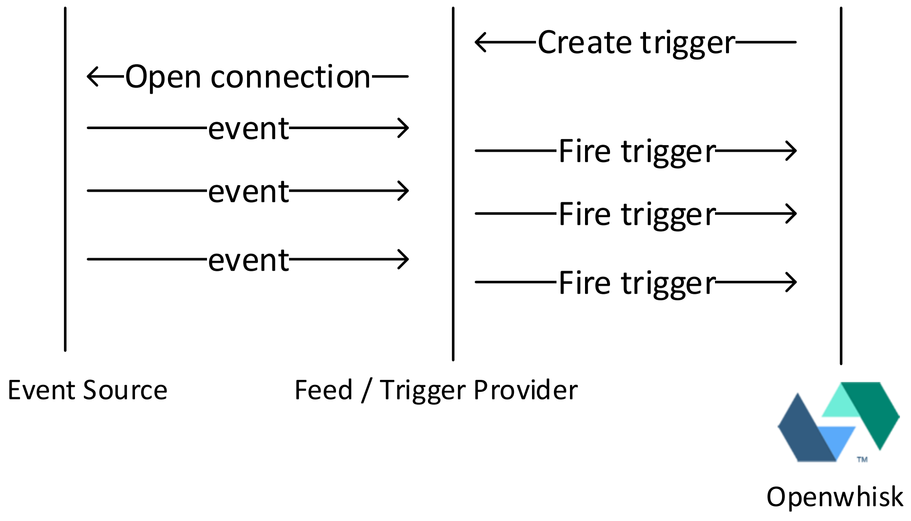

Figure 4.5: Websockets using feeds

- **Message Queues** Message queues have been used for inter-process and inter-threads communication for a while. The principle behind message queues is to provide one-way communication through queues, where a publisher publishes a message to a queue so that a subscribed consumer can pull it from the queue. In addition to inter-process and threads communication, message queues are used for asynchronous communication between applications and systems and it is used as an integration method for heterogeneous and independent components and applications. Message queues can be used as an event channel where event producers emit events to the message queue which can be pulled by OpenWhisk. IBM Message Hub is used by OpenWhisk for this purpose. Till now, this method has not been activated yet. Figure 4.6 shows an abstracted architecture of messaging queues.

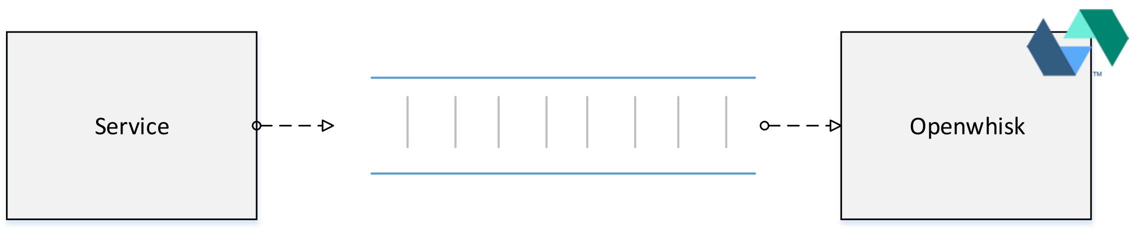

Figure 4.6: Message queues

Message queues introduce a layer between an event producer (a service) and a consumer (OpenWhisk actions). So that, both the event producer and consumers are decoupled. Thus, functionalities are separated and self-contained [Cha12; Joh], which also allows a resilience recovery in failures where the consumer is still able to get messages after the recovery. Moreover, both the producer and consumer can be scaled with no effect to each other. Queues persist messages (events) till the consumer successfully completes processing. Thus, events and message are kept safe and messages are guaranteed to be delivered and they will be processed after all. Queues also provide a mean of ordering the events in a way that they are processed in *First In, First Out (*FIFO*)* manner. It also provides asynchronous communication between event consumers and producers, and therefore, consumers (OpenWhisk actions) can process the events asynchronously with a different rate than the production rate. However, polling is still needed by the subscriber to get the messages from the queue.

4.4.3 Package Structure

As mentioned earlier, OpenWhisk packages contain different actions and feed actions. When building a package for services that need to have additional treatments and procedures such as doing polling on behalf of the users, a separate provider service is needed to achieve that. Packages may get large, and therefore, hard to deploy, test and integrate into OpenWhisk. Thus, we designed a package template that is assumed to provide an ideal package structure to organize and standardize the structure of packages and facilitate the process of integration from OpenWhisk side. Figure 4 shows the proposed tree structure of packages.

As shown in Listing 4 and Figure 4.7, there are four directories for actions, feed actions, tests and tools as well as scripts for installation and uninstallation, if necessary, a directory for a trigger provider. Following is a brief description the different components and parts that are proposed to include within the package template:

- **Actions** Actions directory contains all actions in the package which might be implemented in different programming languages that are supported by OpenWhisk. Naming conventions of actions should comply with camelCase naming pattern and obey the naming restrictions by OpenWhisk. In addition, unit tests for each action functionality should be provided and placed in the package test class.
- **Feeds** Feeds directory contains all feed actions. Naming conventions should be as the Actions using camelCase as well as complying with OpenWhisk naming restrictions. feed actions should support the different life cycles of feeds:

```
openwhisk-package-template/
 CONTRIBUTING.md
 LICENSE.txt
 README.md
 actions
    hello_world.js
 feeds
    feed.js
 tests
    credentials.json
    credentials.json.enc
    src
       TemplateTests.scala
    template_credentials.json
 tools
    travis
        build.sh
 TriggerProvider
    src/
  Dockerfile
    ...
 install.sh
 uninstall.sh
```

Listing 4: OpenWhisk package template

- *Create*: Create the subscription and register the trigger to be fired whenever there is an event.
- *update*: Update the subscription, which can be achieved by providing the parameters to be updated.
- *Pause*: Suspend receiving the events (the binded trigger will not be fired).
- *Unpause*: Is the opposite to pause, which will reinstate the subscription, resume receiving the events and fire the binded trigger whenever there is an event.

Moreover, feed actions should provide a mean of event filtering. The subscription should specify what are the events to subscribe for rather than subscribing for all events.

• **Tests** Unit tests should be provided to test and verify functionality of each individual action and feed action. Tests may need credentials, and therefore, it can be placed within a pre-configured credential file to be used by OpenWhisk during the test phase. In the case of continuous integration (i.e. TravisCI), credentials should be encrypted first to avoid secrets leaking. Moreover, sequential execution of tests

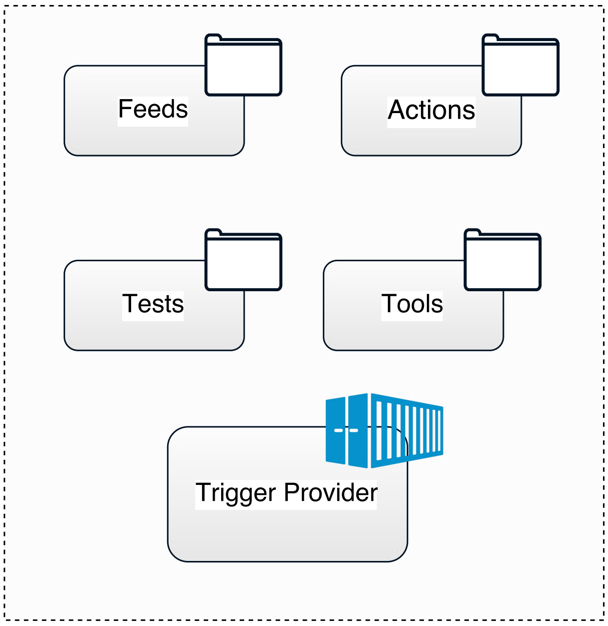

Figure 4.7: Package template content

units should be maintained in case of dependent test units. Otherwise, parallel unit tests are more efficient.

- **Continuous Integration** This Package structure template assumes that Travis is used for continuous integration. Hence, a configuration file for Travis is provided to automate the the processes of configuring the environment, installing and testing the package. As mentioned in the previous sub-section, credentials should be encrypted, which is achieved by either Travis CLI or using any encryption tool such as OpenSSL.
- **Feed** Trigger Provider or feed is an application that stands between OpenWhisk and an event producer to poll or coordinate the process of subscription and receiving events on behalf of OpenWhisk and fire the registered trigger. As OpenWhisk is a microservice based system where each microservice is a container, it is a good practice to containerize the trigger provider. Thus, the processes of integration, deployment and testing process will be easier and more efficient. Furthermore, Trigger provider should provide a mean to monitoring. In the context of OpenWhisk, different components have a monitoring agent (as mentioned in 2.4.3), and therefore, Trigger Provider should provide a monitoring agent. Sometimes, we refer to the feed as a trigger provider since it is a source for firing triggers.
- **Installation** Installing a package is, in fact, a two-steps process that involves creating a new OpenWhisk package and then creating actions and feed actions inside the package. While creating the package, a well-written description and the

bindings parameters should be defined. Actions in their turn, should be described as well. Parameters are defined, sample input and output are provided. For feed actions, same as normal actions except that there is no output. Installation script should support installing the package to OpenWhisk regardless the environment used (e.g. local or Bluemix deployment). However, uninstallation script functions the opposite as it will delete the actions and feed actions and then the package itself. Moreover, if necessary, installation and unstallation should include a support for the feed so that it can deploy and undeploy the feed whenever the package is installed and uninstalled, respectively.

4.5 Challenges

Enabling services and integrating them into OpenWhisk is not a straightforward process. There are few challenges and issues we faced during the implementing, testing and integration phases. These challenges are presented as follows:

- **Communication**: As we mentioned, websockets are proffered over other protocols in real time streaming. Unfortunately, not all services support websockets. In addition, webhooks are more suitable for serverless since it does not require any life-long connections, also unfortunately, not all services support it. Thus, In many cases such as IMAP package implementation, it requires long-life connections (IMAP IDLE) to do polling. Therefore, an external web application (feed) is introduced between OpenWhisk and IMAP service which is not efficient.
- ***Denial of Service (*DoS*)* Attacks**: In case of feeds/trigger providers, a feed is shared among different users in multi-tenancy-based, and therefore, in case of polling as an example, the feed does polling to the same service for different users, such polling requests may be large (all comes from the same feed which has the same address), hence, the service that the feed polling it, may consider this as a DoS attack and block it.
- **Multi-tenancy**: In multi-tenancy architecture, resources are shared, in case of feeds, feeds are shared also. Therefore, it introduces different complexities such as managing multi-threading and handling events to fire a trigger in OpenWhisk. In addition, multi-threading makes the recovery process longer on failures.
- **Heterogeneity of Data Types**: OpenWhisk supports JSON representation of messages through HTTP REST. Unfortunately, not all services or event sources support JSON, and therefore, another layer is introduced between OpenWhisk and the event producer to map the data-type to JSON. For this purpose, an API manager such as IBM API Connect can be used.

5 Design and Implementation

The previous chapter presented the two main approaches are proposed in this study to use them in selecting services to be integrated into OpenWhisk. Moreover, a categorization system for OpenWhisk packages was designed. This chapter applies what has been discussed so far by 1) introducing the use-case that will be used to propose services to implement and integrate into OpenWhisk, 2) implementing and integrating the proposed services by *1*, and 3) categorizing the integrated services in *2* based on the proposed categorization system.

5.1 Early Warning System Scenario

In the previous chapter, two approaches were discussed to propose a list of services to be integrated, and because the first approach (usage statistics based) is not efficient, the second approach (scenario-based) was proposed to overcome the issues of the first approach. The second approach is based on a pre-defined scenario that complies with the criteria matrix discussed in the previous chapter. A list of scenarios was proposed and the EWS scenario was selected based on voting.

The use of warning systems can help in reducing damage and increasing the number of survivors from disasters and accidents such as volcanoes, floods and possible terrorist attacks. We used the EWS use case in proposing different services to integrate and enable in OpenWhisk. Figure 5.1 provides an overall overview of EWS scenario.

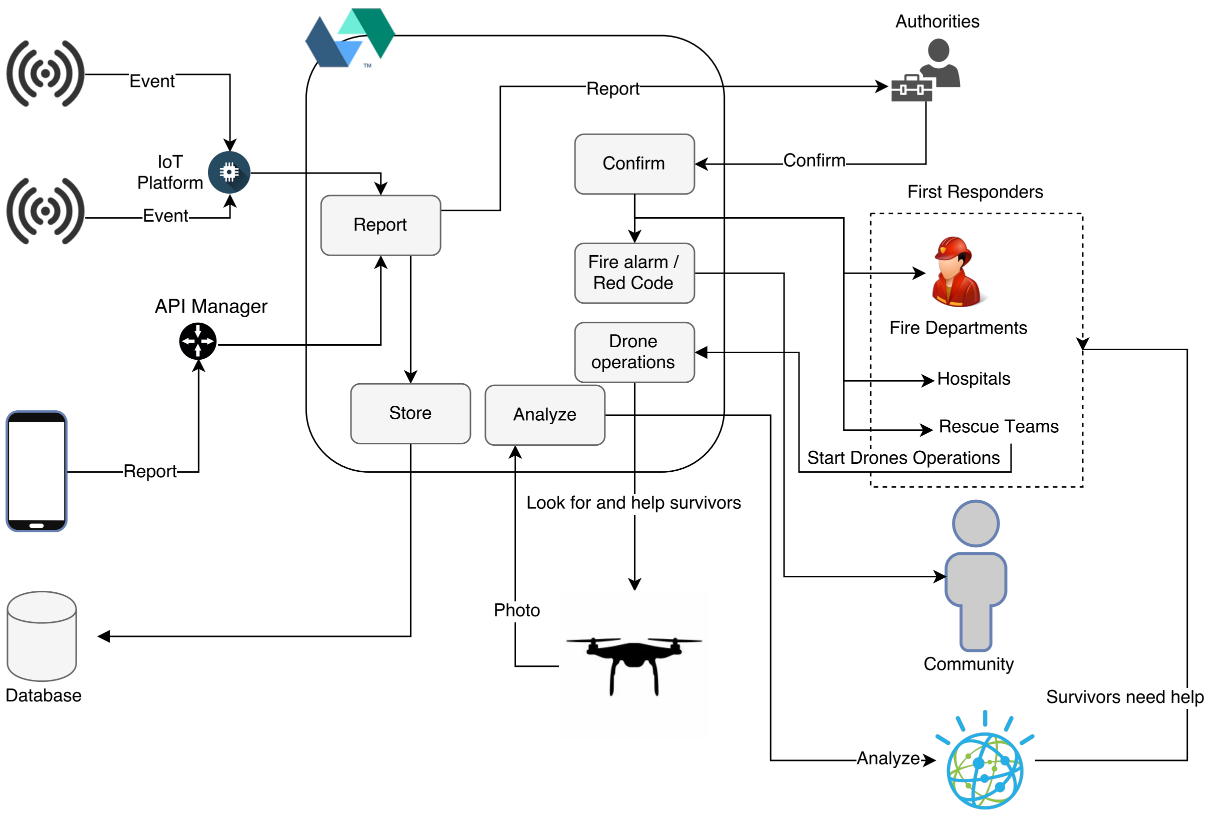

Figure 5.1: High level architecture of the EWS scenario

As shown in figure 5.1, it consists of different components, different technologies and services from different domains and vendors and it is applicable in real life. Since the scenario consists of relatively large components, we will discuss each component separately:

5.1.1 Sensors and Human Reporting

Disasters such as earthquakes, volcanic eruptions and tsunamis almost occur because of Earth movements. They are impossible to stop and predict, but, it is possible to limit the damage and increase the number of survivors [Eva11]. Usually, such disasters start with light vibrations under the ground, and therefore, it might be possible to predict the pattern within seconds or minutes before the disaster hits. Currently, there are many institutions scatter specialized sensors all over the areas that are sensible to Earth movements. Then, measurements were analyzed and actions were taken based on the results. In this scenario, the EWS was built on the top of OpenWhisk. A group of sensors were also scattered and connected to an IoT platform though MQTT protocol, for that, IBM Watson IoT platform was used, which provided a flexible environment and tools to manage and monitor devices and events. In combination with IBM Watson IoT Real-Time insights, an OpenWhisk trigger can be fired through webhooks callback whenever an event is filtered through an Watson IoT Real-Time insights Rule. Reports in EWS can be either from a person or a sensor as follows:

- Humans can report incidents and possible terrorist attacks through a mobile application built for this particular purpose. Users can attach files, write descriptions and provide the exact location of the incident. The report then will fire an OpenWhisk trigger through IBM API Connect to secure access to OpenWhisk, decouple OpenWhisk API and provide analytics capabilities. As soon as the trigger is fired, the report analyzer action starts analyzing the report.
- There are events from sensors and different kinds of sensors such as temperature and humidity were involved. When a change occurs, it sends an event through MQTT to IBM Watson IoT Platform which is connected to IBM Watson IoT Real-Time insights to send the events to OpenWhisk using *Webhooks*. The *Webhook* will fire a trigger as a callback, and then the trigger then will invoke the report analyzer action.

5.1.2 Report Analyzer

Not every report is critical, and every report should go through a filter to filter out spam reports by humans or reports with invalid sensor values or values that rely within the predefined limits (safe threshold). Such predefined values are defined by professionals. The valid reports then go through authority confirmation process. Figure 5.2 shows a data flow diagram for the report analyzer process.

5.1.3 Authority Confirmation

A web or mobile application is used by specialized authorities in each area to check reports and confirm it. Whenever a report is confirmed, an authorizer specifies the target of the alarm and code red, inform the hospitals and other first responders in the area and defines the targeted area. Moreover, if the report is by a human rather than sensors, a reporter may attach media files, which is retrieved from a back end. The *Box* storage service is used for this purpose. A data flow diagram for the process is shown in figure 5.3.

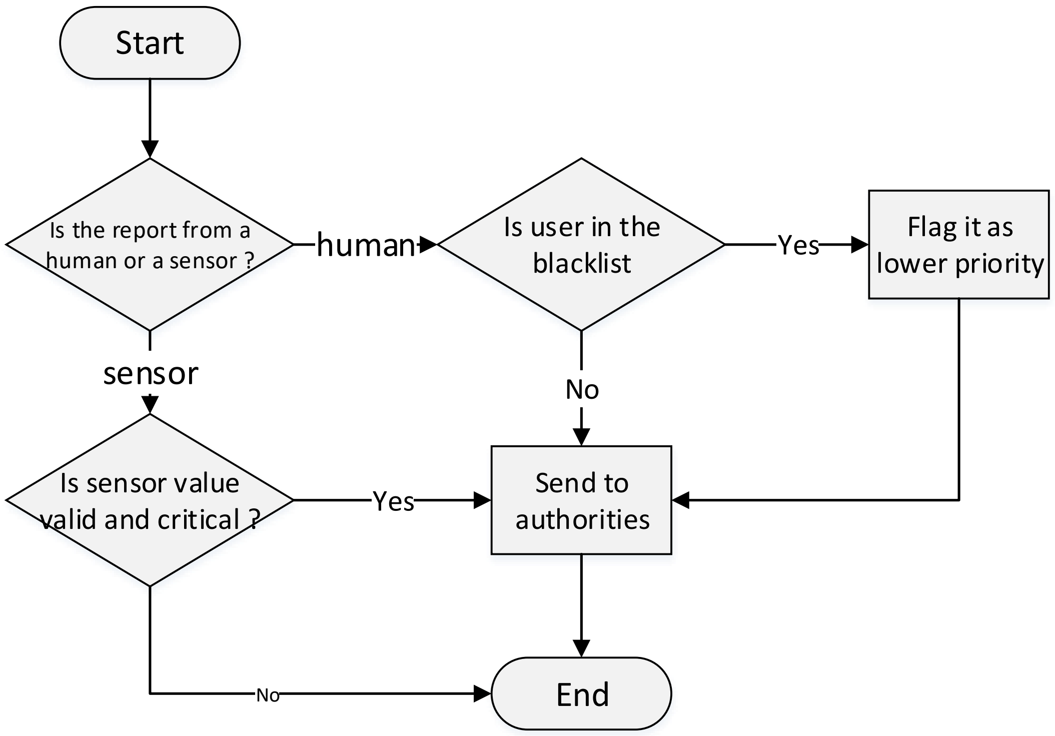
Start
Is the report from a human or a sensor ?
human
Is user in the blacklist
Yes
Flag it as lower priority
No
sensor
Is sensor value valid and critical ?
Yes
Send to authorities
No
End

Figure 5.2: Data flow diagram of the report analyzer process

5.1.4 Alarm and Red Code Firing

In confirming any reports, authorities decide the actions to be done as a consequence to the report. Such actions may be firing a public or red code alarms, and this includes, sending *Short Message Service (*SMS*)* to the citizens in the targeted area to inform and guide them.

5.1.5 Rescue Operations

Besides the rescue teams, drones are involved to help them identify survivors' locations and provide urgent medical supplementaries Drones are distributed over the targeted area after dividing it into small squares, and each square is covered by a drone or group of drones. Drones help locating survivors by taking photos of the covered areas and send it through OpenWhisk to analyze the taken photos by IBM Watson Cognitive services.

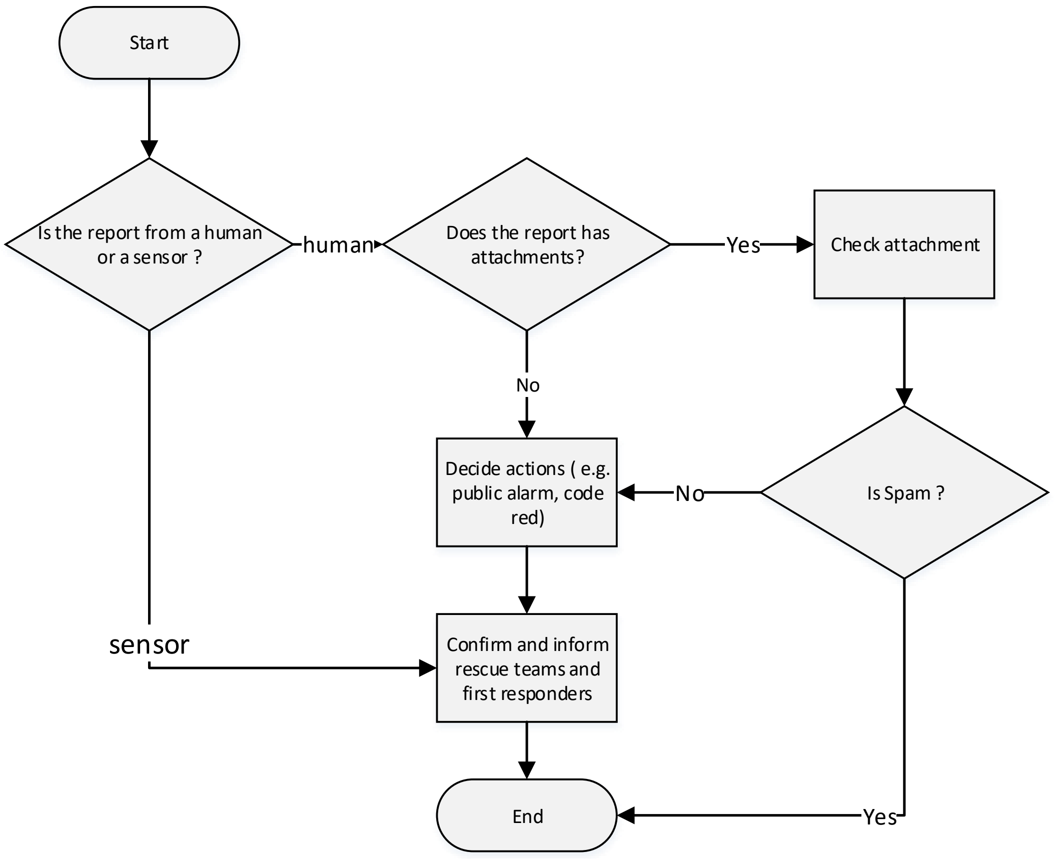
Start
Is the report from a human or a sensor ?
human
Does the report has attachments?
Yes
Check attachment
No
Decide actions (e.g. public alarm, code red)
No
Is Spam ?
sensor
Confirm and inform rescue teams and first responders
End
Yes

Figure 5.3: Authorities confirmation process data flow diagram

Not every drone can be used in rescue operations, it should be picked up based on some specifications, such as being able to flight for a long time, being programmable, being able to fly autonomously and able to carry essential medical supplementaries. The flow diagram of rescue operations using drones is illustrated in figure 5.4.

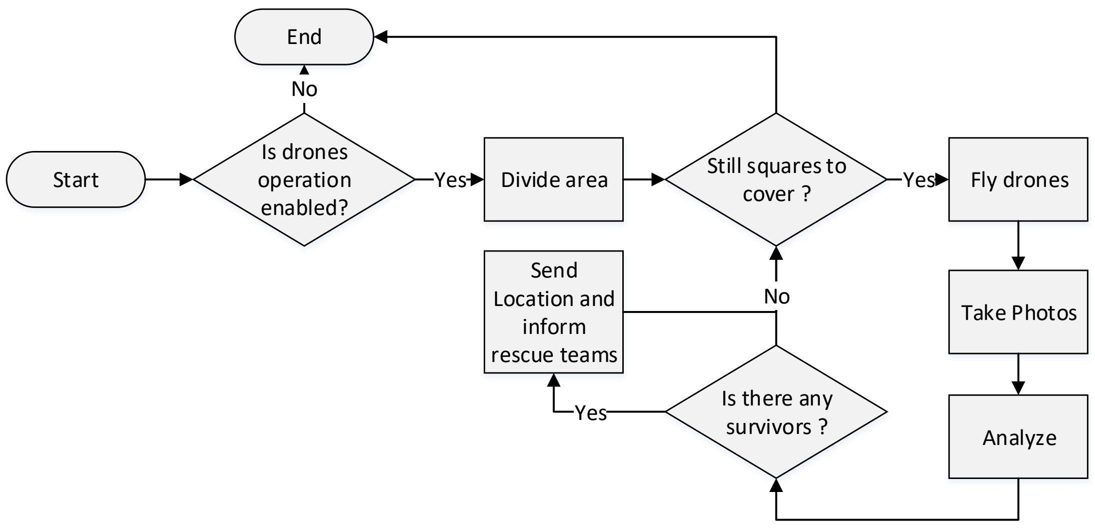

Figure 5.4: Rescue operations flow diagram

In the previous chapter, different integration methods were discussed, including polling that requires life-long applications that can do polling on behalf of OpenWhisk users. Such application is called *Feed* or *Trigger Provider*. An example of such services, is a service for incoming emails through IMAP protocol, since IMAP is a protocol rather than a service by itself. Obviously, it is not possible to use webhooks in this case, and instead, polling is the best solution. But, polling requires life-long connections to continue poll for new emails. OpenWhisk actions are executed only on-demand with limited timeout, and therefore, actions cannot run all the time. As a possible solution, a workaround can be done by scheduling an action to do polling within small period of time (e.g. 2 seconds) but this leads to an overhead in the context of managing the schedule. Instead, a separate service that stands between OpenWhisk and IMAP server to do polling on behalf of the user is required. Such service is called a feed or trigger provider, which is discussed in details in the .

5.2 Feed / Trigger Provider

A feed or trigger provider is an application that stands between OpenWhisk and a service or event source to provide persistent connections such as polling for new events. Feeds are shared among users in a multi-tenancy manner. Figure 5.5 shows an abstracted overview of feeds, and the use case diagram of feed is in figure 5.6.

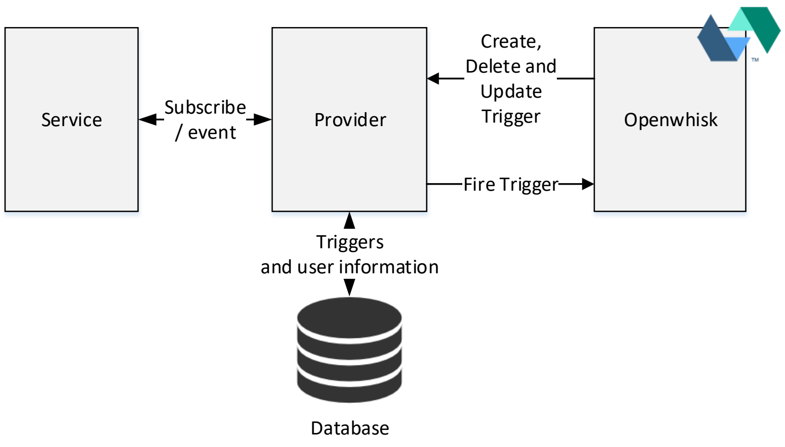
Service
Subscribe / event
Provider
Create, Delete and Update Trigger
Openwhisk
Fire Trigger
Triggers and user information
Database

Figure 5.5: High level architecture of feeds

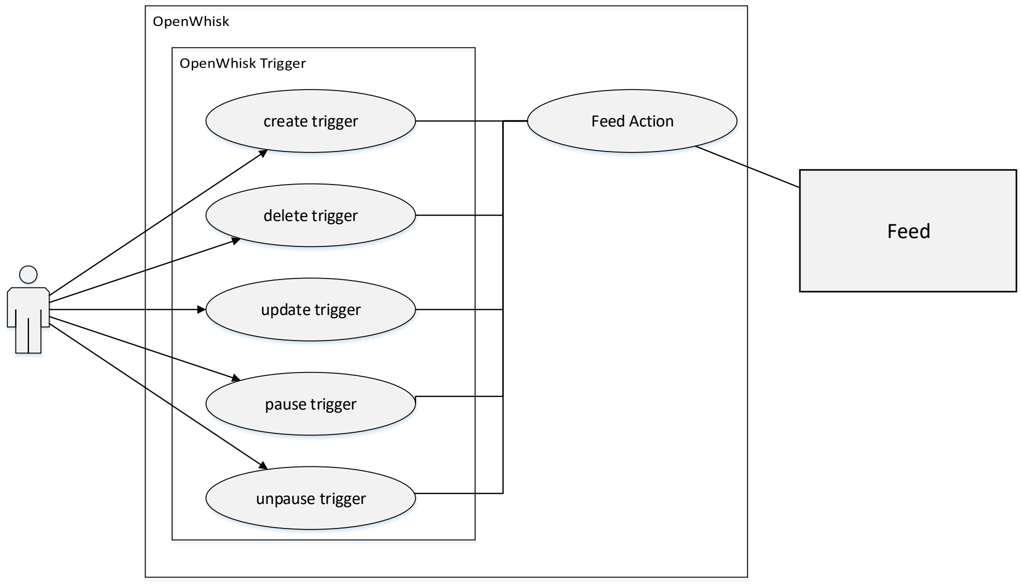

Figure 5.6: Feed use case diagram

Feed Features and Characteristics: There are common characteristics and features that we think every feed should have to provide an efficient role in registering, receiving events and firing triggers:

- **Scalable**: Providers should be highly scalable to handle different types of workloads that are generated by the events producer, the feed is built as a multi-tenant service which is shared among different OpenWhisk users. Moreover, states within the feed should be kept externally such as an external database to ease scaling.

- **Persistence**: Applications may fall at any time due to any failures that may arise, such as due to unavailability of the target service. Feed should be built on a way that can be recovered after a failure. Main substance of recovery, is persistence, where all important information such as users and triggers information are stored in an external database.
- **Logging**: In the software world, everything may fail. In huge systems and applications, tracing the cause of a failure is not an easy thing, thus, logging helps to trace the failure's root to fix it. Feed should log everything for every user, so that, all the generated transactions by the feed are logged.
- **Security**: Provider should resist and protect the vulnerabilities assets such as users credentials. A powerful encryption mechanism with a random salt can be used to protect the secrets. Furthermore, such encrypted data should be stored in a safe place.
- **HTTP REST Endpoints**: Besides the above features and characteristics, feeds communicate with OpenWhisk through HTTP REST, and therefore, feeds should provide different endpoints to create, pause, unpause and delete triggers. Moreover, the *content-type* should be *application/*JSON.

5.3 IBM Watson IoT Platform Package

From the proposed scenario, the first proposed service to integrate is Watson IoT platform. The platform is used to manage devices and transfer the events through MQTT protocol. IBM Watson IoT platform provides a toolkit to build scalable IoT applications, the toolkit includes gateway devices and devices management, and also it allows users to process events in realtime. [Tea16a]

Events are sent to the platform through the MQTT messaging protocol, to receive such events in OpenWhisk, there are two ways, either by using an MQTT Feed [Tho16] or through another service *Watson* IoT *Real-Time Insights* which is connectable to the platform, and therefore, this package does not contain any feed actions to manage event subscription. Figure 5.7 shows an overview of the relationship between OpenWhisk and Watson IoT platform.

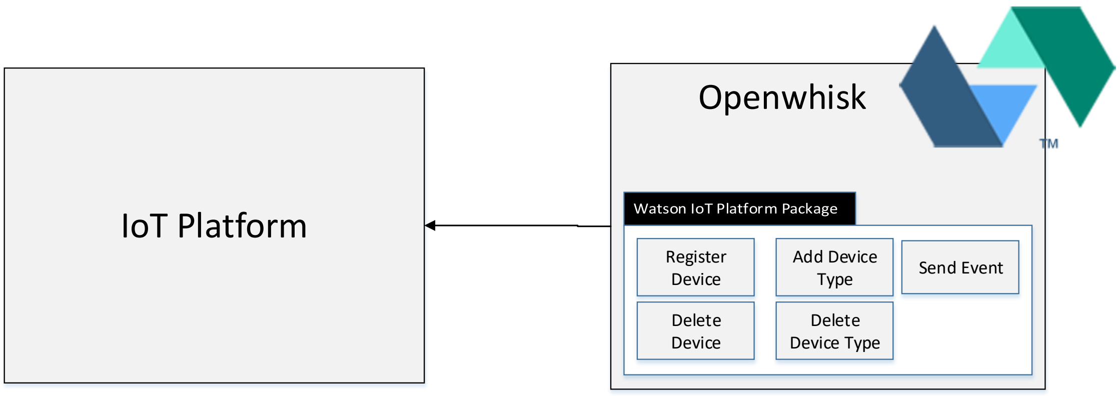

Figure 5.7: Watson IoT platform and OpenWhisk

5.3.1 Actions

- **Create Device Type**: Create a new device type, a device type is a group of devices that share common characteristics such as name and description. All parameters are supported and can be specified to provide highly configurable action and a better user experience (table 5.1). In the context of the scenario, the devices with the same type form a device type.

  As any other entity within OpenWhisk, this action can be used through OpenWhisk CLI or through OpenWhisk REST API. Listing 5 shows an example of creating a new device type using */whisk.system/iot/createDeviceType* action.

```
        #Invoke create device type action through terminal
        $ wsk action invoke /whisk.system/iot/createDeviceType -p orgId 'xxxxx'
↪ -p apiKey 'yyyyyy' -p apiToken 'zzzzzzzz' -p typeId 'RaspberryPi' --blocking
```

```
        //Simple Output
        {
          "classId": "Device",
          "createdDateTime": "2016-06-16T10:27:43+00:00",
          "deviceInfo": {

          },
          "id": "RaspberryPi",
          "updatedDateTime": "2016-06-16T10:27:43+00:00"
        }
```

Listing 5: Usage example of device type action

Parameter	Type	Required	Description	Options	Default	Example
apiKey	string	yes	Watson IoT platform API key	-	-	"XXXXX"
apiToken	string	yes	Watson IoT platform API authentication token	-	-	"YYYYYYYYY"
orgId	string	yes	Watson IoT platform organization ID	-	-	"xvfrw1"
typeId	string	yes	Device Type ID	-	-	"sampleType"
serialNumber	string	no	The serial number of the device	-	-	"10211002XYZ"
manufacturer	string	no	The manufacturer of the device	-	-	"Texas Instruments"
model	string	no	The model of the device	-	-	"HGI500"
deviceClass	string	no	The class of the device	*false,true*	*false*	false
description	string	no	The descriptive name of the device	-	-	-
fwVersion	string	no	The firmware version currently known to be on the device	-	-	"1.0"
hwVersion	string	no	The hardware version of the device	-	-	"1.0"
descriptiveLocation	string	no	A descriptive location, such as a room or building number, or a geographical region	-	-	"Office 220, building 16"
metadata	object	no	Metadata of the device	-	-	""customField1": "customValue1","customField2": "customValue2""

Table 5.1: Create device type parameters

- **Add Device**: In order to add a device (sensors in the scenario) into the platform (Watson IoT platform), it should be associated to a device type. A device in the platform should be connected to the internet and able to send data to the cloud. Same as the previous action, all parameters are supported to enhance user experience (table 5.2).

  Usage of this action is not different from others, it can be done through OpenWhisk CLI as well as OpenWhisk REST API. Listing 6 shows an example of adding a device into the platform.

- **Delete Device Type**: Whenever a device type is no longer needed, it can be deleted, for such purpose another action is implemented within this package to delete a device type. Table 5.3 shows the parameters of the action and in listing 7 an example of using it.

Parameter	Type	Required	Description	Options	Default	Example
apiKey	string	yes	IoT platform API key	-	-	"XXXXX"
apiToken	string	yes	IoT platform API auth token	-	-	"YYYYYYYYY"
orgId	string	yes	IoT platform org ID	-	-	"xvfrw1"
typeId	string	yes	Device Type ID	-	-	"newDevice"
deviceAuthToken	string	no	Device auth token	-	-	"anUnhackableToken"
deviceId	string	yes	Device ID	-	-	"sampleType"
serialNumber	string	no	Device serial number	-	-	"10211002XYZ"
manufacturer	string	no	Device manufacturer	-	-	"Texas Instruments"
model	string	no	Device model	-	-	"HGI500"
deviceClass	string	no	Device class	*false,true*	*false*	false
description	string	no	Device descriptive name	-	-	-
fwVersion	string	no	Device firmware version	-	-	"1.0"
hwVersion	string	no	Device hardware version	-	-	"1.0"
descriptiveLocation	string	no	A descriptive location	-	-	"Office 220, building 16"
long	decimal	no	Longitude in decimal degrees	-	-	9.038550
lat	decimal	no	Latitude in decimal degrees	-	-	48.665390
elev	decimal	no	Elevation in meters	-	-	507
accuracy	decimal	no	Position accuracy in meters	-	-	5
measuredDateTime	string	no	Location measurement date and time	-	-	"2016-05-19T11:36:42.825Z"
metadata	object	no	Metadata of the device	-	-	-

Table 5.2: Add device parameters

Parameter	Type	Required	Description	Options	Default	Example
apiKey	string	yes	IOT platform API key	-	-	"XXXXX"
apiToken	string	yes	IoT platform API auth token	-	-	"YYYYY"
orgId	string	yes	IoT platform org ID	-	-	"xvfrw1"
typeId	string	yes	Device Type ID	-	-	"newDevice"

Table 5.3: Delete device type parameters

```
            #Invoke delete device type action through terminal
            $ wsk action invoke /whisk.system/iot/deleteDeviceType -p orgId 'xxxxx'
↪ -p apiKey 'yyyyyy' -p apiToken 'zzzzzzzz' -p typeId 'RaspberryPi' --blocking
```

```
            //Simple Output
            {
              "success": "device type deleted",
            }
```

Listing 7: Usage example of delete device type action

```
            #Invoke add device action through terminal
            $ wsk action invoke /whisk.system/iot/registerDevice -p orgId "xxxxx"
↪ -p apiKey "yyyyyy" -p apiToken "zzzzzzzz" -p typeId "RaspberryPi" -p deviceId
↪ "deviceId" --blocking
```

```
//SimpleOutput{
  "apiToken": "xxxxxxxxx",
  "clientId": "d:orgId:RaspberryPi:deviceId",
  "deviceId": "deviceId",
  "deviceInfo": {

  },
  "refs": {
    "diag": {
      "errorCodes":
↪ "/api/v0002/device/types/RaspberryPi/devices/deviceId/diag/errorCodes/",
      "logs": "/api/v0002/device/types/RaspberryPi/devices/deviceId/diag/logs/"
    },
```

Listing 6: Usage example of add device action

- **Delete Device**: A device can be unregistered or deleted from the platform, */whisk.system/iot/deleteDevice* action allow users to achieve that. Table 5.4 shows the parameters and listing 8 provides a usage example.

Parameter	Type	Required	Description	Options	Default	Example
apiKey	string	yes	IoT platform API key	-	-	"XXXXX"
apiToken	string	yes	IoT platform API auth token	-	-	"YYYYY"
orgId	string	yes	IoT platform org ID	-	-	"xvfrw1"
typeId	string	yes	Device Type ID	-	-	"newDevice"
deviceId	string	yes	Device ID	-	-	"sampleType"

Table 5.4: Delete device parameters

- **Send Device Event**: There are many cases where an event need to be sent on behalf of a device such as for testing purposes. Watson IoT platform provides an HTTP REST endpoint to send an event to the platform on behalf of a device. In other words, an event from an application. Send event action is another action within IoT package, it takes different parameters as shown in table 5.5 and can be used as shown in listing 9.

```
            #Invoke delete device action through terminal
            $ wsk action invoke /whisk.system/iot/deleteDevice -p orgId 'xxxxxxxx'
↪ -p apiKey 'yyyyyyyy' -p apiToken 'zzzzzz' -p typeId 'RaspberryPi' -p deviceId
↪ "deviceId" --blocking
```

```
            //Simple Output
            {
               "success": "device deleted"
            }
```

Listing 8: Usage example of delete device action through OpenWhisk CLI

Parameter	Type	Required	Description	Options	Default	Example
apiKey	string	yes	IoT platform API key	-	-	"XXXXX"
apiToken	string	yes	IoT platform API auth token	-	-	"YYYYYY"
orgId	string	yes	IoT platform org ID	-	-	"xvfrw1"
typeId	string	yes	Device Type ID	-	-	"newDevice"
deviceId	string	yes	Device ID	-	-	"sampleType"
eventName	string	yes	Event name	-	-	"temperature"
eventBody	object	yes	Event Data	-	-	"'temp':'42'"

Table 5.5: Send event parameters

```
            #Invoke send device event action through terminal
            $ wsk action invoke /whisk.system/iot/deleteDevice -p orgId 'xxxxx' -p
↪  apiKey 'yyyyyyyyy' -p apiToken 'zzzzzzz' -p typeId 'sampleiot' -p deviceId
↪  "TareqDevice44" --blocking
```

```
            //Simple Output
            {
              "success": "event is sent"
            }
```

Listing 9: Usage example of send event action

5.4 IBM Watson IoT Real-Time Insights Package

IBM Watson IoT Real-Time Insights is another service by Watson IoT. To analyze events and data that sent to the platform, Watson IoT Real-Time Insights is used. It provides different powerful tools to filter and process data and events. As response to incoming events, different real-time insights actions can be done such as request a webhook to fire an OpenWhisk trigger as a callback or send an email. The filtering is achieved by what is called real time insights rules, the filtering is based on the events attributes such as a value within event payload or context such as its priority. Different real-time insights actions can be associated to a rule. In other words, the cardinality relationship between real-time insights rules and real-time insights actions is (one-to-many). Watson IoT Real-Time Insights package in OpenWhisk contains different actions and a feed action. Figure 5.8 shows an overall architecture of real-time insights and OpenWhisk [Tea16b].

In the context of the scenario, IBM Watson IoT Real-Time Insights is used to create a webhook on the events come from the platform, any event by the sensors are transferred through MQTT to the platform, IBM Watson IoT Real-Time Insights service then fire an OpenWhisk trigger to check the values reported by the sensors.

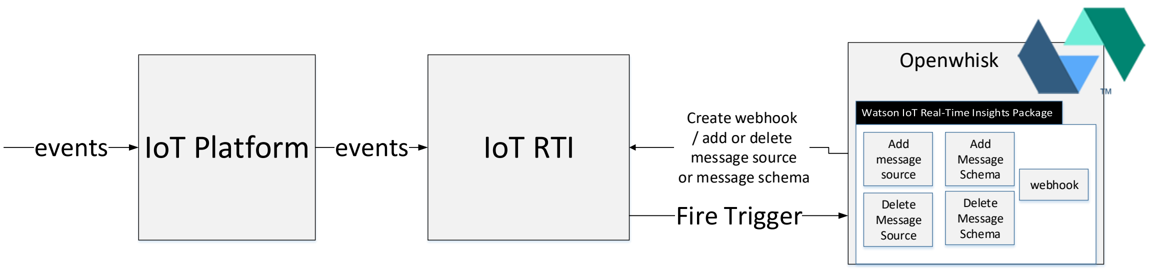

Figure 5.8: Watson IoT RTI and OpenWhisk

5.4.1 Actions

- **Add Message Source**: Message source in Watson IoT Real-Time Insights is nothing but Watson IoT Platform. Currently, the cardinality relationship between the platform and real-time insights services is one-to-one, where Real-Time Insights service have only one message source. Add Message Source is an action to facilitate adding a message source to the real-time insights service. Table 5.6 shows the needed parameters and in listing 10 an example of using the action through OpenWhisk CLI.

Parameter	Type	Required	Description	Options	Default	Example
iotapiKey	string	yes	IoT platform API key	-	-	XXXXX
iotapiToken	string	yes	IoT platform API auth token	-	-	YYYYY
apiKey	string	yes	IoT RTI API key	-	-	XXXXX
authToken	string	yes	IoT platform RTI auth token	-	-	YYYYY
orgId	string	yes	IoT platform org Id	-	-	xvfrw1
name	string	no	name of the message source	-	msgSource +orgId	msgSource htpsa
disabled	boolean	no	disable or enable the message source	-	-	sampleType

Table 5.6: Add message source action parameters

```
            #Invoke add message source action through terminal
            $ wsk action invoke /whisk.system/iot-rti/addmessagesource -p orgId
↪ "xxxxx" -p apiKey "yyyyyy" -p authToken "zzzzzzzz" -p typeId "sampleiot" -p
↪ deviceId "deviceId" --blocking
```

```
            //Simple Output
            {
    "apiKey": "XXXXX",
  "authToken": "YYYYY",
  "created": "27 Jun 2016 17:20:35 GMT",
  "disabled": false,
  "id": "grNwDDKD",
  "name": "source1",
  "orgId": "zxdo1w",
  "updated": "27 Jun 2016 17:20:35 GMT"
}
```

Listing 10: Usage example of add message source action

- **Delete Message Source**: In case of the need to delete a message source, this action can do that. The required parameters as well as a usage example are in table 5.7 and listing 11 respectively.

Parameter	Type	Required	Description	Options	Default	Example
apiKey	string	yes	RTI API key	-	-	"XXXXX"
apiToken	string	yes	RTI service authentication token	-	-	"YYYY"
name	string	no	name of the message source	-	-	"mesgSourceName"

Table 5.7: Delete message source action parameters

```
        #Invoke delete message source action through terminal
        $ wsk action invoke /whisk.system/iot-rti/deletemessagesource -p apiKey
↪  'yyyyyy' -p authToken 'zzzzzzzz' -p name 'source1' --blocking
```

```
        //Simple Output
        {"success": "message source deleted"}
```

Listing 11: Usage example of delete message source action

- **Add Message Schema**; A message schema in Real-Time insights describes events attributes and it used to parse the incoming events. The required parameters, sample request and usage are in table 5.8, listing 12 and listing 15 respectively.

Parameter	Type	Required	Description	Options	Default	Example
apiKey	string	yes	RTI API key	-	-	"XXXXX"
authToken	string	yes	RTI service authentication token	-	-	"YYYYYYYYY"
name	string	yes	message schema name (must be unique)	-	-	"message schema"
items	object	yes	JSON object that describe the schema	-	-	"["name": "value", "description": "value", "type": "int", "subItems": []]"

Table 5.8: Add message schema action parameters

```
        #Invoke add message schema action through terminal
        $ wsk action invoke /whisk.system/iot-rti/addmessageschema -p name
↪  'messageSchemaName' -p items "$(cat items.json)" -p apiKey 'XXXXXX' -p
↪  authToken 'YYYYYY' --blocking
```

```
        //Simple Output
        [{ "name": "value", "description": "value of event", "type": "int",
↪  "subItems": [] }]
```

Listing 13: Usage example of add message schema action

- **Delete Message Schema**: Another action within the RTI package is to delete an existing message schema, the parameters and sample are in table 5.9 and listing 15 respectively.

```
  //Simple Output
  {
  "created": "27 Jun 2016 14:47:03 GMT",
  "deviceType": null,
  "format": "JSON",
  "id": "YPtEVgFY",
  "items": [
    {
      "composite": false,
      "description": "value",
      "event": null,
      "formula": null,
      "id": 1,
      "keyIndex": false,
      "length": null,
      "metaui": null,
      "name": "value2",
      "subItems": [

      ],
      "subType": null,
      "timestamp": false,
      "type": "int"
    }
  ],
  "name": "messageSchemaName",
  "updated": "27 Jun 2016 14:47:03 GMT"
}
```

Listing 12: Message schema request example

Parameter	Type	Required	Description	Options	Default	Example
apiKey	string	yes	RTI API key	-	-	"XXXXX"
authToken	string	yes	RTI service authentication token	-	-	"YYYY"
name	string	yes	message schema name (must be unique)	-	-	"mesgSchema"

Table 5.9: Delete message schema action parameters

```
            #Invoke delete message schema action through terminal
            $ wsk action invoke /whisk.system/iot-rti/addmessageschema -p name
 ↪ 'messageSchemaName' -p apiKey 'XXXXXX' -p authToken 'YYYYYY' --blocking
```

```
            //Simple Output
            {
             "success": "message schema deleted"
            }
```

Listing 14: Usage example of delete message schema action

5.4.2 Feed Action

To fire an OpenWhisk trigger as response to events come from the sensors, this feed action can be used to subscribe for events from Watson IOT Real-Time Insights service which is connected to Watson IoT Platform.

As we discussed in section 4.4, Webhooks are preferred over other means, as it has zero-overhead on the client side (i.e. OpenWhisk action), and in most cases, no need for feeds. IBM Watson IOT RTI service provides a mean to create a webhook to call an HTTP endpoint as callback. To create and setup the webhook, a feed action is implemented. Table 5.10 shows the required parameters and in listing 5.10 a sample usage.

Parameter	Type	Required	Description	Options	Default	Example
apiKey	string	yes	glsrti API key	-	-	"XXXXX"
authToken	string	yes	RTI service authentication token	-	-	"YYYYYYYYY"
schemaName	string	yes	Messages Schema name	-	-	"schema"
condition	string	yes	is a predicate or some conditions joined with binary logical operators	-	-	"schema.value>1"
callbackBody	string	no	message body of the triggered event	-	" "rule" : "ruleName" , "condition" : "ruleCondition" , "message" : "message" "	" "rule" : "ruleName" , "condition" : "ruleCondition" , "message" : "message" "
description	string	no	rule description	-	"A rule created by OpenWhisk Feed @ current date and time"	"A rule created by Openwshisk feed"
severity	integer	no	severity of the rule, higher number means lower priority	1,2,3,4	4	4

Table 5.10: Webhook parameters

```
        # Create OpenWhisk Trigger with webhook feed action
        $ wsk trigger create rtiFeed --feed /whisk.system/iot-rti/webhook -p
↪  apiKey 'XXXXXXXX' -p authToken 'YYYYYYYY' -p schemaName 'schema'  -p condition
↪  'schema.value>1'
```

Listing 15: Usage example of webhook feed action

The webhook creates an RTI rule and an RTI webhook action to fire an OpenWhisk trigger and associate the created action to the rule, it also supports the different trigger life cycle event such as create, delete and update. Since pause and unpause are still not activated within OpenWhisk, we haven't supported that.

5.5 IMAP Package

Incidents as well can be reported through emails, IMAP is a protocol for incoming emails, since it is a protocol rather than a service, thus, the integration methods are limited to polling, and therefore, a feed is needed to stands between OpenWhisk and IMAP server and do polling to fire an OpenWhisk trigger whenever an email is arrived. This package contains a feed action to manage the subscription through the feed. Figure 5.9 shows an overall architecture of OpenWhisk IMAP Feed.

5.5.1 Feed Actions

As shown in figure 5.11, the feed action is used by a trigger to subscribe for incoming emails, the subscription is done through the feed by providing different endpoints to create, update and delete triggers. Table 5.11 shows the needed parameters for IMAP feed action.

Parameter	Type	Required	Description	Options	Default	Example
host	string	yes	IMAP server endpoint	-	-	"imap.gmail.com"
username	string	yes	IMAP username	-	-	"YYYYYYY"
password	string	yes	IMAP password	-	-	"XXXXXXX"
mailbox	string	yes	IMAP mailbox	-	-	"INBOX"

Table 5.11: IMAP feed action parameters

```
            # Create a trigger with IMAP feed action
            $ wsk trigger create imapTrigger -p user 'almaamaritest@gmail.com' -p
  ↪  pass 'XXXX' -p host 'imap.gmail.com' -p mailbox 'INBOX' --feed
  ↪  /whisk.system/imap/imapFeed
```

Listing 16: Usage example of creating a trigger with IMAP feed

5.5.2 Feed

As discussed IMAP feed is required to do polling on behalf the user, manage and secure credentials. Data is encrypted to a avoid sercrets leaking. We have implelented this feed as a NodeJS web application, and it is deployed to Bluemix [Tea15]. Since polling is not efficent, IMAP IDLE command allow to have long connection opened to receive new email, for that purpose, a NodeJS module implementation for IMAP is used to provide a convinent way to use IMAP IDLE.

The feed contains different endpoints to create, update and delete triggers. It encrypts users data with a random salt to avoid secrets leaking. Moreover, the feed can recover and continue working, and therefore, there is no loss in term of functionalities and data.

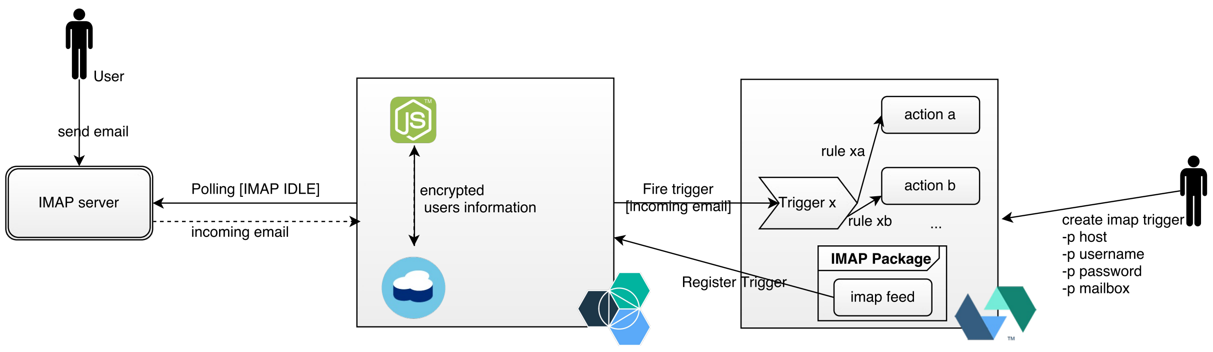

Figure 5.9: IMAP package

5.6 IFTTT Package

IFTTT is a business-centric event-driven services provider, it allows users to perform different IFTTT actions in response to changes on a service. It is different from Open-Whisk, as IFTTT is business-centric rather than application-centric. In IFTTT, users are not able to execute code as a response to changes on services that are integrated to IFTTT.

In the context of the proposed use-case (EWS), IFTTT can play various roles such as alerting users through their mobiles or posting alarms to the different social networks. Currently, there are around 353 channels (services) that are integrated and can be used. Therefore, integrating IFTTT with OpenWhisk allows OpenWhisk users to use these different 353 services that are integrated with IFTTT.

The integration between OpenWhisk and IFTTT is done by implement an IFTTT *channel* for OpenWhisk. In IFTTT, there are triggers and actions, same as OpenWhisk, triggers are events that occurred and generated by a service, triggers are the *if* part, in the other hand, IFTTT actions are the *that* part such as sending an email through *Gmail* or post a tweet in *Twitter*.

The channel contains two IFTTT triggers and actions as follow:

- **[Trigger] Triggers Fired from OpenWhisk**: Triggers from OpenWhisk can be used as triggers in IFTTT, the channel allows users to list all OpenWhisk triggers within user's namespace and select the wanted trigger. Unfortunately, this trigger doesn't provide realtime processing. Instead, another trigger is proposed.
- **[Trigger] Notify Action is invoked**: To overcome with the burden in the previous trigger, we introduce an additional action to fire IFTTT trigger, the action is located within *ifttt* package. Whenever the action is invoked, different parameters can be passed, the only required parameter which is the *eventName* which is used to differentiate between the different invokactions of the same action. Three additional parameters can be spcified to provide more flexibility. The notify action can be associated to an OpenWhisk trigger through an OpenWhisk rule to fire IFTTT trigger whenever the OpenWhisk trigger is fired.
- **[Action] Invoke OpenWhisk Action**: Through the channel, list of all actions within the user's namespace can be seen, the user then is able to to spcify which action to invoke whenever IFTTT is fired. Additional parameters can be specified to pass it to OpenWhisk action.
- **[Action] Fire OpenWhisk Trigger**: Similar to the previous action, except that, instead of invoking actions, it allows users to fire OpenWhisk triggers. Also, additional parameters can be specified to pass it as OpenWhisk trigger payload.

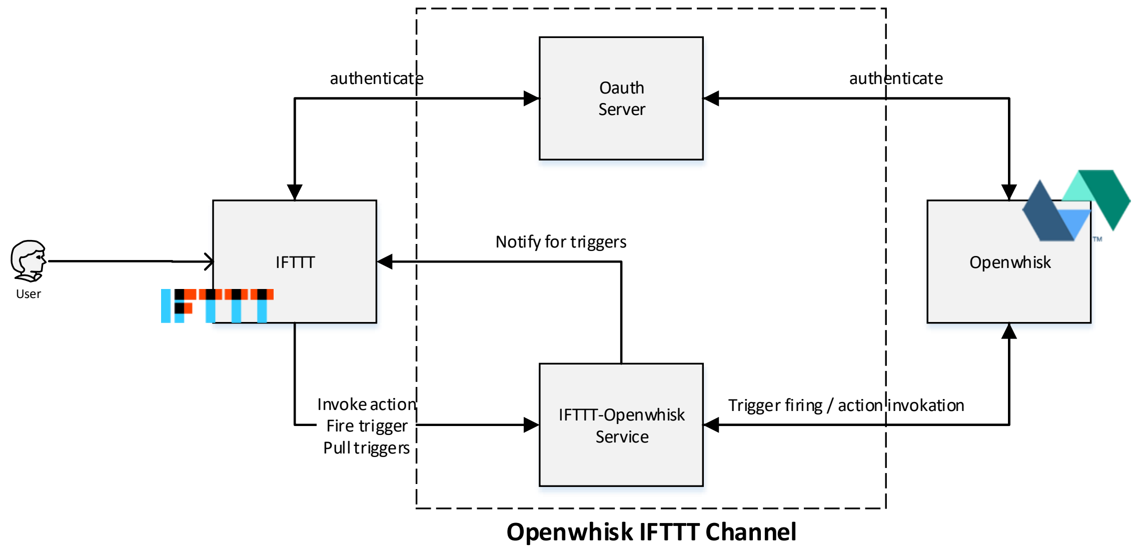

Figure 5.10: OpenWhisk channel in IFTTT

5.7 Box Package

Back again to the scenario, reports by humans may contain media files as attachments, therefore, a storage service is needed to store the attachments and inform the confirmor about the attachmants.

Box is an online storage service, it provides different services to store and share files. Box provides a powerful API to interact with the different Box entities. As we mentioned, we mainly focus on enabling more event sources. For such purpose, we will focus on this package to provide a mean for sbscribing for events and firing triggers as response to the events.

Box developers have built two versions of Box API for webhooks, so, two versions of this package has been implemented:

5.7.1 Webhooks V1

However Box doesn't provide any API to interact with Box webhooks, Box UI is the only mean that can be used to interact to Box Webhooks V1. Box webhooks v1 allow users to subscribe for different events such as file creations and uploads. Through UI, users are able to to specify the callback endpoint URL as well as the payload format. Webhooks v1 supports 2 main content types for the payload *application/x-www-form-urlencoded*, and *application/xml* but not *application/json*. The limitation on available content-types introduces an issue, as OpenWhisk doesn't accept requests *Content-Type* rather than *application/json*, and therefore, an additional layer between OpenWhisk and Box to map the content type into *application/json* is required, this can be done either by using an API manager such as IBM API Connect or a feed. We have experimented both, using IBM API Connect as well as implementing a feed.

Since using IBM API Connect as a content type mapper is not a main focus of this thesis, in this section, we implement a feed to stands between OpenWhisk and Box, the feed is responsible to subscribe for events on behalf of the user, map the incoming event callbacks to *application/json* as well as firing the associated trigger whenever an event is occurred. Figure 5.11 an architecture of box mapper feed is shown.

As mentioned, the only mean to interact with the webhooks in V1 is the UI, and therefore, users need to specify the endpoint URL of the feed explecitly, which provides a bad user experience.

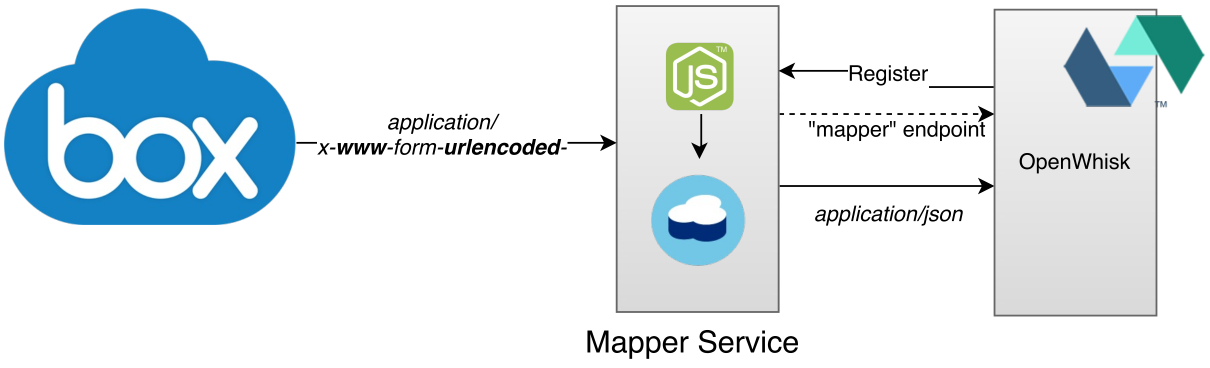

Figure 5.11: Box mapper feed

5.7.2 Box API V2

Recently, Box has annouced *Webhooks V2* within their API *V2*, where developers can interact with *Box Webhooks* through REST API. Moreover, webhooks v2 support *application/json*, and therefore, there is no need for any additional layer to map content type. A drawback of *Box Webhooks v2*, is that, developers need to specify the id of the box entities (file or folder) to subscribe for their events. Ids are hard to find and requires additional steps, as a workaround, we implemented an additional OpenWhisk action to get information about folders or files by their names, hence, users can pick up the id and use the *webhook* feed action to create a trigger to subscribe for events on that entity. *Box webhooks V2* authentication is based on *Open Authentication (*OAUTH*)* 2.0, but unfortunately, users need to update the authentication token regulary.

5.7.2.1 Actions

- **Search**: As mentioned, Box Webhooks V2 require the exact id of the Box entity to receive events on it, and therefore, since it is not possible to find the id within the UI, users can use this action to search for Box entities by name. This action requires two parameters as shown in table 5.12.

Parameter	Type	Required	Description	Options	Default	Example
query	string	yes	Query body of the search	-	-	"video.mp4"
bearer	string	yes	Box *OAuth* 2.0 bearer token	-	-	"YYY"

Table 5.12: Search action parameters

5.7.2.2 Feed Action (Webhook)

As mentioned earlier, *Box* API *V2*, provides endpoints to create and manage webhooks. *Webhooks V2* allow users to specify the events type to register for, so that, if any of such events occurred, the registered callback endpoint will be called, in this case, it is an OpenWhisk trigger. Table 5.13 contains the parameters required by the feed action.

Parameter	Type	Required	Description	Options	Default	Example
targetId	string	yes	Query body of the search	-	-	"video.mp4"
bearer	string	yes	Box OAUTH 2.0 bearer token	-	-	"YYY"

Table 5.13: Box webhooks feed action parameters

6 Results and Evaluation

This chapter presents the results and evaluates what we have achieved in this thesis. The first section (**??**) shows the results of comparison of the main serverless computing providers by listing the strengths and weaknesses of each one in different aspects. In the second section (6.2), the importance of categorizing OpenWhisk packages is highlighted. Then, the benefits gained from proposed services selection approach and how that affected the selection process and observed benefits from the proposed scenario are discussed in section (6.3). Finally, in section (6.4), we go through the integration and enabling process of the proposed services and observe the benefits and temptations of integrating them.

6.1 Serverless Computing Providers

?? The first research objective of this thesis, is to analyze and compare the different main serverless computing services providers. The comparison helps specialized users to have a look at the different features and characteristics of the main serverless providers to choose or select the most suitable provider that can meet certain requirements. Throughout chapter 3, a comparison of different aspects has been conducted. The results obtained from this comparison are presented and discussed as follows:

- **Programming Languages & Runtimes**: We found that all of the observed serverless computing providers support JavaScript programming language through NodeJS runtime, it is obvious since JavaScript is the most popular used programming language. In addition, containers are not supported as a serverless entity by almost all of the observed providers except OpneWhisk. In conclusion, the more programming languages and other execution means such as containers make the serverless provider more powerful, as it provides flexibility about the runtimes and technologies to use.
- **Scalability**: Almost all of the observed providers provide automatic and transparent scalability except Microsoft Azure Functions which provides automatic and nontransparent scalability where users are able to see the amount and instances of the resources being used.

- **Concurrent Execution**: Another aspect is the maximum concurrent entities (functions or containers) execution. The results show that OpenWhisk and AWS Lambda limit the maximum concurrent execution to a specific threshold. In contrast, maximum concurrent execution in Microsoft Azure Functions is limited by the size of the assigned memory. For Google Cloud Functions and Auth0 Webtasks, they have no limitations on the maximum concurrent execution.
- **Execution Time**: One of the most important aspects is the maximum execution time of serverless codes or containers (called actions in OpenWhisk, functions in Google Cloud Functions and Microsoft Azure Functions and Webtasks in Auth0 Webtasks). OpenWhisk and AWS Lambda limit the maximum execution time to 300 seconds, the others have no limitations.
- **Code Size**: OpenWhisk as well as AWS Lambda limit the maximum size of code entities to 48 MB and 250 MB respectively. On the other hand, Google Cloud Functions, Microsoft Azure Functions and Auth0 Webtasks have no limits.
- **Memory**: As one of the main aspects in the comparison, Table table 6.1 shows that OpenWhisk has the minimum maximum memory size within the observed serverless computing providers. However, AWS Lambda, Google Cloud Functions and Microsoft Azure Functions have a maximum size of memory around 1025 MB.
- **Dependencies Management**: Management of dependencies has been one of the main features that every provider should offer. Yet, our results indicate that, all of the providers do not provide an ultimate powerful dependencies management. OpenWhisk as an example, in JavaScript (NodeJS), supports a few famous npm modules. Similarly, Auth0 Webtasks supports the most famous 600 NPM modules. Moreover, OpenWhisk also supports packaging dependencies and codes into a single *Java ARchive (*JAR*)*. In contrast, Microsoft Azure and Google Cloud Functions manage dependencies using dependencies descriptors such as *package.json* in NPM. AWS Lambda does not support any dependencies management tools, but it allows developers to upload ZIP packaged of the code and dependencies.
- **Deployments**: There are three main approaches to upload code and dependencies that current serverless computing providers support, 1) either by direct upload (e.g. CLI), 2) to a storage service or 3) using *SCM* services. All the observed providers support *1*, AWS Lambda and Google Cloud Functions support *2*, but only Google Cloud Functions as well as Microsoft Azure Functions allow deploying functions through *Source Code Management (*SCM*)* services.
- **Versioning**: From normal version numbers to branches and tags version controls, the observed providers versioning techniques vary. Google Cloud Functions and Microsoft Azure Functions support SCM based version control (branches and

tags). However, AWS Lambda provides versioning through what is called *Aliases*. OpenWhisk supports simple version numbers.

- **Logging**: OpenWhisk, Azure Functions and Webtasks provide embedded services for logging. Unlike the mentioned providers, AWS Lambda and Google Cloud Functions use external services for logging.
- **Openness**: Obviously, all of the observed providers except OpenWhisk are closed-source proprietary systems. As mentioned in section 2.4, OpenWhisk is an open source project, which benefits from the temptations of the open source community such as providing a higher quality system due to the pair review and contributions from the community.
- **Pipelining**: OpenWhisk is the only provider that provides built-in real-time pipeline processing through sequence actions. This can help in situations where different actions need to be executed in sequence as a chain with no change on the action itself.
- **Sharing**: Sharing different entities allows users to exchange expertise, speed up development and increase overall productivity by sharing actions that have been shared by other users.

Table 6.1 shows a high level comparison over all observed serverless providers. The table presents the differences between the observed providers in various aspects.

Provider	**IBM OpenWhisk**	**AWS Lambda**	**Azure Functions**	**Google Functions**	**Auth0 Webtasks**
Scalability	transparent	transparent	nontransparent	transparent	transparent
Max of code entity	48 MB	250 MB	no limits	no limits	no limits
Concurrent execution	100 per namespace	100 per region	based on the memory	no limits	no limits
Max execution time	300 s	300 s	no limits	no limits	no limits
Max memory	512 MB	1536 MB	1536 MB	1024 MB	-
Dependencies	few *NPM* modules	ZIP or *JAR*	NuGet and NPM	*NPM*	600 *NPM* modules
Deployments	code files/JARs	code/zip (Lambda, *S3*)	*SCM* services, *FTP* and Web deploys	ZIP (Cloud Storage & Cloud Source Code Repositories)	code files
Versioning	embedded	Aliases	embedded	Cloud Repositories Service (branches and tags)	-
***HTTP* support**	embedded and limited	API Gateway	full support	full support	full support
Logging	embedded	CloudWatch	embedded	Stackdriver Logging	embedded
Authentication	Basic Authentication	*IAM*	*OAuth* 2.0	*OAuth* 2.0	Auth0
Openness	Yes	No	No	No	No
Pipelining	Sequence Action	No	No	No	No
Sharing	Yes	No	No	No	No

Table 6.1: Serverless providers comparison

6.2 OpenWhisk Categorization

Another research objective of this thesis is to design a categorization system for the existing OpenWhisk packages. Moreover, the designed and proposed OpenWhisk categorization system should be used to categorize the services integrated in chapter 5.

As mentioned in section 4.2, the designed categorization system is based on Taxonomic categories rather than Script or Thematic categories since Script and Thematic categories can form unlimited pairs based on the scripts and themes formed (Section 4.2). Categorization in general provides the following points of strength:

- **Clarity of Knowledge**: Clear knowledge about package content and functionalities is provided where different services share similar content and functionalities. For instance, consider categorization based on domains, domains containing different IoT services are grouped into one category. Obviously, this category shows that all of these services are related to the interaction with events in the world of IoT.
- **Accessibility**: Easy to find, reach out and use services. In case of a large number of services, it is hard to find specific services easily, but with categorization, a service can be reached out through the expected category. For example, consider the previous mentioned example, the IoT category, to reach out IBM Watson IoT Platform service, it can be easily predicted that it is located within a category called "IoT"
- **Common Traits**: Show packages that share specific traits, a category, regardless of the similarity it is based on. Example of this are a specific domain, feature, property or any other similarity. other similarity. This means that all members of the category members share similar features. Again, consider the example of IoT category, all of the category members share the same domain, and therefore, have similar traits.
- **Convenience**: Categorizing new packages is more convenient, and it allows defining the boundaries and characteristics of the packages to be categorized. New packages can be categorized by making few predictions, and therefore, categorizing new and previously unknown services is convenient.

The proposed categorization system is based on three main similarities that can be used for taxonomic categories: domain, vendor and openness. The limitation of these three similarities comes from the fact that most of the services are black-box and further features are useless and cannot be used properly in categorization. In addition, multi-hierarchy categorization can be formed based on these three used similarities (domain, vendor and openness) as explained in section 4.2. The following is the resulted categorization of the existing available OpenWhisk packages using the domain similarity:

- **Cognitive Computing**: Watson Cognitive Services
- **Internet of Things**: IBM Watson IoT Platform, IBM Watson IoT Real-Time Insights
- **Workflow and Integration**: IFTTT, *Rich Site Summary (*RSS*)* Feeds
- **Data and Analytics**: Weather
- **Storage**: Box
- **Social and Media Platforms**: Slack, IMAP, Twilio, SendGrid
- **APIs**
- **DevOps**: Github
- **Communication and Security**: IBM Message Hub
- **Mobile Computing**: IBM Push Notification

6.3 Services Selection

Proposing services to be integrated and enabled within OpenWhisk is one of the main goals of this thesis, in order to have a well defined list of services to be integrated. So, users can get maximum benefits from this list. Therefore, we proposed the following two selection approaches that can be used to propose services to be integrated into OpenWhisk:

6.3.1 Usage Statistics Based Selection

The first approach is based on the usage statistics of the different services within Bluemix. However, since the usage statistics of the services within Bluemix are considered as IBM confidential data, we cannot share it here, and the results of the statistics should suffice to discuss here. The resulted statistics show that the top used services are within the same domain, and, they are from the same vendor. Thus, usage based selection approach cannot be a good option in proposing services with different domains so that more users can get benefits of it and different vendors to avoid vendor lock-in. In addition, new services can be more powerful if they are integrated into OpenWhisk to provide serverless and event-driven services. Therefore, we have decided to propose another selection approach to overcome the aforementioned issues. It presented as follows:

6.3.2 Use-Case Based Selection

This approach is proposed to overcome with the issues of the previous approach. It is based on a pre-defined scenario. To select a good scenario, a criteria matrix was defined to ensure that the use-case 1) contains different services from different domains and vendors, 2) can be applicable in real life and 3) can generate different types of workloads especially peaking workloads. Based on the defined criteria matrix, we proposed a list of scenarios have been proposed in different areas as follows:

- **EWS**
- **Serverless SDN Controller**
- **Accidents Monitoring**
- **Production Lines in Manufacturing**
- **LEGO Mindstorms Controller**
- **Traffic Violations**
- **Serverless Bots**

6.3.2.1 Early Warning System

From the proposed scenarios list in section 4.3.3.2, EWS was selected based on voting (Figure 6.1 shows the voting result) among the different proposed scenarios (Section 4.3.3.2). The voting process was placed after a brainstorming session with a group of architects and developers within IBM. The scenario is discussed in details in section 5.1. The following is a list of the proposed services to be integrated:

- Watson IoT Platform Package: To receive events from sensors through MQTT
- Watson IoT Real-Time Insights Package: To fire OpenWhisk triggers whenever there are event sent to the IoT platform.
- IMAP Package: To execute actions in response to incoming emails.
- Box Storage Package: Reports may include media files which will be stored in Box, so, authorities can retrieve it whenever it is uploaded.

- IFTTT OpenWhisk Channel Package: IFTTT allows users to interact with 353 different services in event-driven manner.

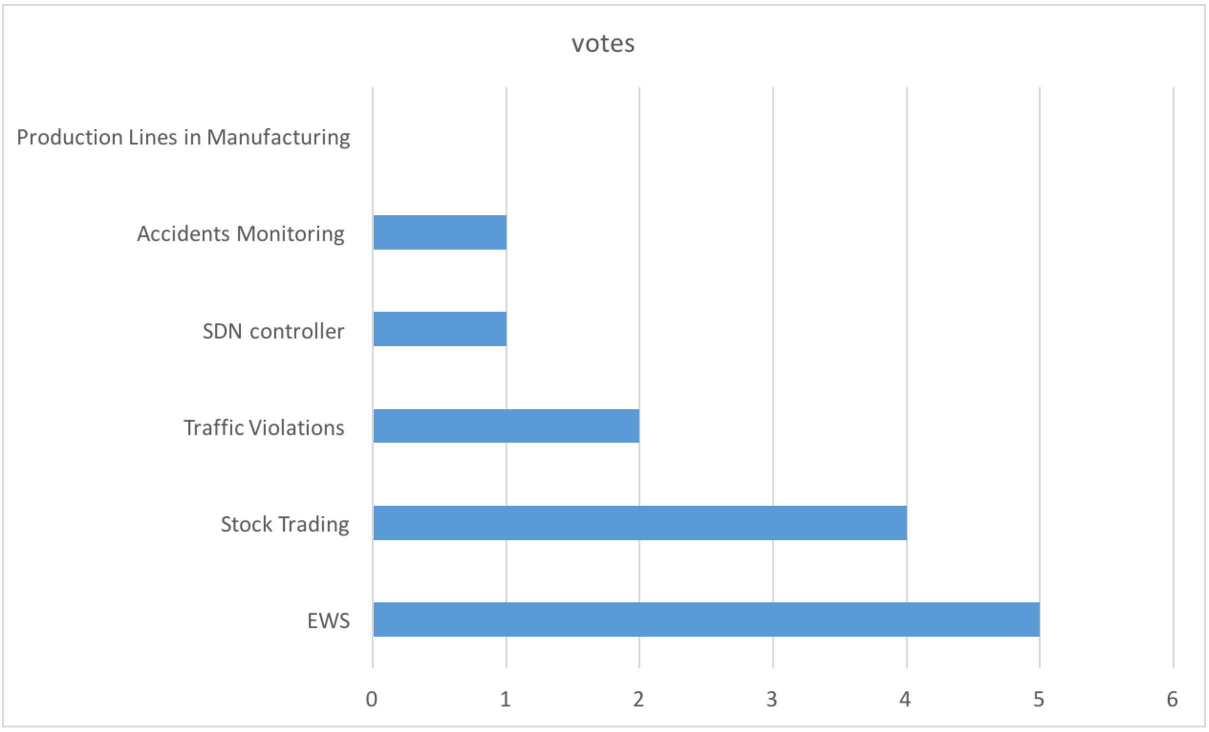

Figure 6.1: Scenario votes

The cost of running this scenario was improved by building it to the top of OpenWhisk. The scenario after all is a bunch of OpenWhisk actions (each represents a microservice) which are easy to implement, test and integrate. So, effort and time were improved as well.

The server-based implementation of this scenario can be as a bunch of microservices, where each component in the scenario is a microservice. In this case, the microservices will be deployed into Bluemix using boilerplates runtimes. With consideration of the price model of the runtimes in Bluemix, this scenario assumes that hundreds of thousands of reports from both sources (human or sensors) can be reported, and therefore, scalability should be taken care of it. As a minimum requirement, we will assume that around 5000000 request (report) are created per second with more than 2 instances (i.e. 10 instances) of 512 MB memory each, and this will cost 170 EUR for each microservice. Since there are two main components in the scenario, they are, report analyzer and report confirmation. These two components use different services to receive events, analyze and confirm reports. Thus, a database service, a storage service, a visual recognition analysis service and IoT services are involved, and prices of these

services are not considered since the prices of using such services are not changed. Since OpenWhisk does not have a pricing model till now, instead, we will use the average price from the current main serverless providers (0.173833 EUR per million executions *Gigayte per Second (*GBS*)*). Using the online Serverless Cost Calculator tool [SA]. The result of the comparison can be summarized as:
Serverless Based: Two Microservices with 10 instances of 512 MB (170 EUR * 2): 340 EUR
Server Based: 4 actions with an estimated execution time 200 milliseconds, memory 512 MB and 1 million requests): 1.28367 EUR

The results show that there is improvement up to 264,86 % in cost and the operations cost are eliminated at all.

6.4 Services Integration

The main goal of this thesis, is to enrich the services ecosystem of OpenWhisk by integrating and enabling more services in different domains and from different vendors. Thus, a list of proposed services to be integrated was made based on the aforementioned approach. Enriching the ecosystem in general has different advantages and disadvantages as well, They are summarized as follows:

- **Scalability**: Integrating services allow events to be handled in OpenWhisk in a highly scalable manner. Therefore, different factors are improved in case of converting applications into event-driven nature such as cost and resources consumption.
- **Improve Productivity / Flexibility**: It allow users to focus more on business logic instead of managing event producers to subscribe for events or taking care of different complexities of integrating. Integrated packages simplify the process of subscription for events either through a Feed or directly from the events producer. Moreover, different common actions ready to use are provided, which saves users' time and efforts to build the same common actions.
- **Effectiveness**: Efficiency depends on the integration method used. In case of integration methods that require long-life connection, feeds are needed to handle the connections and manage the different subscriptions. Therefore, more resources are consumed. But, sharing the feed makes it at the same time more efficient when comparing it to non-shared feed where each user builds and manages their own feed.

Following, specific benefits of each integrated services are discussed separately:

6.4.1 Template Package

After proposing the list of services to be integrated is proposed, integration process started. Each package represents service integration which contains different entities (actions, feed actions and if necessary a feed). The integration process may get tricky due to the different aspects, and therefore, the integrator should take care of it such as the testing and processes. To organize the integration, testing and processes, a standard package was designed to provide a mean of standardization and to organize the different phases. In addition to the mentioned advantages, the template allows integrators to focus more on the business logic of the integration as an overall rather than taking care of minor details. It will also eventually lead to a higher level of productivity.

6.4.2 Watson IoT Platform & Watson IoT Real-Time Insights Packages

Watson IoT Platform and Real-Time Insights packages allow users to interact with the both services and to execute actions in response to events sent through MQTT. Besides to the overall advantages of enriching the services ecosystem of OpenWhisk, without this package, users are not able to interact with the both services, and executing code as response to events in OpenWhisk becomes hard and error prone.

6.4.3 IMAP Package

IMAP package allows users in OpenWhisk to interact with incoming emails through the IMAP protocol by invoking actions in response to new emails. Besides the general benefits from integrating and enabling more events in OpenWhisk, IMAP package eases the process of subscribing for new emails and hides complexities of handling polling and failures.

6.4.4 Box Storage Package

With Box package, users are able to interact with Box and execute OpenWhisk actions as a response to events on Box. In this work, we implemented two versions of the package. The first version is not supported by any Box APIs. Managing Box Webhooks V1 is done through Box Developers UI, while Webhooks V2 can be managed through Box API V2. Generally, with this package, firing OpenWhisk triggers as a response to the changes in Box is doable, and complexities are hidden and parameters are simplified. Unfortunately, A mapper that stands between OpenWhisk and Box to map data content to JSON was needed as OpenWhisk only supports *application/json* and Box Webhooks V1 do not support *application/json*, and therefore, more resources are required which introduce additional overhead to manage. In addition to the mapper, Box Webhooks V1 do not support setting the callback headers or even do not support any mean of authentication to the callback handler (in this case OpenWhisk). Box Webhooks V2 (Based on Box API V2) on the other hand, do not require any additional layers between OpenWhisk and Box, which eliminate any overhead. Same as Webhooks V1, Webhooks V2 do not provide any mean to handle authentication to OpenWhisk (callback handler). Moreover, additional steps are needed to configure Webhooks V2 such as finding the id of the Box entity (file or folder) to subscribe for events produced by activities on them.

6.4.5 IFTTT OpenWhisk Channel & Package

We have built an IFTTT channel for OpenWhisk. By enabling the channel into IFTTT, the following multiple benefits could be gained:

- Triggers or actions in OpenWhisk can be fired or invoked as a response to events from the different +353 services integrated in IFTTT.
- OpenWhisk users are also able to use the +353 different services integrated with IFTTT. This eliminates the overhead of integrating these 353+ different services directly into OpenWhisk.
- Cost and resources consumption are significantly improved, as feed might be required to get events from event producers (services), which obviously eliminates more resources consumption to handle the feed, and therefore, cost is improved.

7 Conclusions and Future Work

Servers have been the base of deployment and running online applications for a while, but unfortunately, servers are not ideal in many cases due to some disadvantages such as complexities of managing servers and operations and low utilization. Therefore, serverless computing is introduced to overcome the complexities and burdens of servers. Serverless computing allows to execute code, binaries or even containers upon direct request or as response to events.

To consume and process events by events consumers, consumers should subscribe for receiving events if the events producers support that, otherwise, other means should be followed which may require lifelong connections, and therefore, more complexities are introduced and should be taken care of it such as the limitation on maximum execution time of serverless entities (i.e. actions in OpenWhisk). Therefore, a separate service (feed or trigger provider) is needed to handle the long connections to get events from the events producers. OpenWhisk as an example, has services ecosystem that contains different packages of services, a package contains actions and feed actions to ease and facilitate using the services and subscribing for receiving events to fire or invoke trigger or actions in OpenWhisk.

An original goal of this thesis is to enrich OpenWhisk ecosystem by implementing and integrating more services, so that, users can use the services and subscribe to events which produced by these services. In this thesis, we implemented and integrated various services in terms of domain and vendor. The selection of the services to be integrated was based on two selection approaches, a usage-statistics and use-case based selection, each approach has cons and pros, the use-case based approach was followed since it proposes services that are diverse in domains and vendors.

To integrate more services, an analysis of the different methods that can be used to integrate such services are done, and in order to standardize and organize the implementation, integration and testing phases, we have designed and implemented a standard template, so that, developers can save time, effort and focus more in implementing the business logic of the integration.

The integrated services then are categorized to provide a mean of grouping and facilitate reaching out services. For that purposes, a categorization system was designed, we found that a Taxonomic categories is a best-fit category approach for OpenWhisk, thus, few similarities we found that they are applicable as base of Taxonomic categories, they are, domain, vendor and openness. We have used the domain aspect as it is more clear and understandable by users.

A minor goal of this thesis, is to compare the different main serverless computing services providers, the comparison was based on different aspects such as system limits and scalability transparency.

As future work, more services to be integrated and enabled can be achieved. Moreover, to ease the implementation process of the integration itself, in the other words, implementing the feeds. we propose to design and build an integration framework.

Glossary

APACHE TOMCAT is an open source software that powers numerous large-scale, mission critical web applications across a diverse range of industries and organizations [Fou16] 30

DEVELOPMENT AND OPERATIONS is a practical term of the collaboration of both operations and development engineers in the whole product/service life cycle [Mue10] 27

SOFTWARE CONTAINER is the lightweight and nimble cousin of *virtual machine*, where a virtualization in the Operating System level rather than in the hardware level. [Mer14b] 15

VIRTUAL MACHINE a software application that runs an operating system and other applications, the virtualization here is by virtualizing the hardware. 15

Acronyms

ACID Atomicity, Consistency, Isolation, Durability 26

API Application Programming Interface 31

AWS Amazon Web Services 16

BASE Basically Available, Soft state, Eventual consistency 26

CF Cloud Foundry 52

CLI Command-Line Interface 13

CPU Central Processing Unit 31

DAAD Deutscher Akademischer Austauschdienst 4

DEVOPS *Development and Operations* 27

DOS Denial of Service 65

EC2 Elastic Compute Cloud 40

EDA Event Driven Architecture 15

EWS Early Warning System 3

FIFO First In, First Out 61

FTP File Transfer Protocol 42, 44, 94

GBS Gigayte per Second 98

HTTP Hypertext Transfer Protocol 25

IAAS Infrastructure as a Service 15

IAM Identity and Access Management 44, 94

IBM International Business Machines 3, 16

IFTTT If This Then That 9

IMAP Internet Message Access Protocol 9

IOT Internet of Things 9

JAR Java ARchive 40, 44, 92

JSON JavaScript Object Notation 33

MQTT Message Queuing Telemetry Transport 51

NPM NodeJS Package Manager 24

OAUTH Open Authentication 42, 44, 89

OS Operating System 59

PAAS Platform as a Service 20

PC Personal Computer 50

REST Representational State Transfer 26

RSS Rich Site Summary 96

RTI Real-Time Insights 9

S3 Simple Storage Service 40, 44, 94

SAAS Software as a Service 15

SCM Source Code Management 42, 44, 92

SDK Software Development Kit 35

SDN Software-Defined Networking 56

SMS Short Message Service 69

TCP Transmission Control Protocol 39

UI User Interface 40

URL Uniform Resource Locater 43

Bibliography

[AAA+10] M. Ahronovitz, D. Amrhein, P. Anderson, A. De, J. Armstrong, E. A. B, J. Bartlett, R. Bruklis, M. Carlson, R. Cohen, T. M. Crawford, V. Deolaliker, P. Downing, A. Easton, R. Flores, G. Fourcade, T. Hanan, V. Herrington, B. Hosseinzadeh, S. Hughes. *Cloud Computing Use Cases - A white paper*. 2010. URL: http://cloudusecases.org (cit. on p. 21).

[Azu] M. Azure. *Azure WebJobs SDK Extentions*. URL: https://github.com/Azure/azure-webjobs-sdk-extensions (cit. on p. 48).

[Azu16a] M. Azure. *Azure WebJobs SDK*. 2016. URL: https://github.com/Azure/azure-webjobs-sdk/ (cit. on p. 48).

[Azu16b] M. Azure. *Microsoft Azure Functions Documentation*. Microsoft Azure. 2016. URL: https://functions.azure.com (cit. on p. 48).

[BDGR97] E. Bugnion, S. Devine, K. Govil, M. Rosenblum. "Disco: Running Commodity Operating Systems on Scalable Multiprocessors". In: *ACM Trans. Comput. Syst.* 15.4 (Nov. 1997), pp. 412–447. URL: http://doi.acm.org/10.1145/265924.265930 (cit. on p. 28).

[BHJ15] A. Balalaie, A. Heydarnoori, P. Jamshidi. "Migrating to Cloud-Native Architectures Using Microservices: An Experience Report". In: *ArXiv e-prints* (July 2015). arXiv: 1507.08217 [cs.SE] (cit. on p. 25).

[Bon00] A. B. Bondi. "Characteristics of Scalability and Their Impact on Performance". In: *Proceedings of the 2Nd International Workshop on Software and Performance*. WOSP '00. Ottawa, Ontario, Canada: ACM, 2000, pp. 195–203. URL: http://doi.acm.org/10.1145/350391.350432 (cit. on p. 23).

[Cha12] Chance. *Top 10 Uses For A Message Queue*. 2012. URL: https://www.iron.io/top-10-uses-for-message-queue/ (cit. on p. 62).

[CL05] H. Cohen, C. Lefebvre. *Handbook of Categorization in Cognitive Science*. Elsevier Science, 2005. URL: https://books.google.co.uk/books?id=5WDfl14RgKMC (cit. on p. 50).

[DVE+16] S. Daya, N. Van Duy, K. Eati, C. Ferreira, D. Glozic, V. Gucer, M. Gupta, S. Joshi, V. Lampkin, M. Martins, et al. *Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach*. IBM Redbooks, 2016 (cit. on pp. 26, 27).

[Eva11] M. Evans. *Natural Disasters*. 2011. URL: http://www.earthtimes.org/encyclopaedia/environmental-issues/natural-disasters/ (cit. on p. 68).

[FGS11] T. Frey, M. Gelhausen, G. Saake. “Categorization of concerns: a categorical program comprehension model”. In: *Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools*. ACM. 2011, pp. 73–82 (cit. on p. 50).

[FH15] A. FELDMAN, C. HENRY. “Best Practices for Developing Cloud-Native Applications and Microservice Architectures”. In: *The Net Stack* (2015). URL: http://thenewstack.io/best-practices-for-developing-cloud-native-applications-and-microservice-architectures/ (cit. on p. 23).

[FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer Publishing Company, Incorporated, 2014 (cit. on pp. 21–23).

[Fou16] T. A. S. Foundation. *Apache Tomcat*. June 2016. URL: https://tomcat.apache.org (cit. on pp. 30, 105).

[Goo16] Google. *Google Cloud Functions*. Google Inc. 2016. URL: https://cloud.google.com/functions (cit. on p. 48).

[Her] Heroku. *The twelve-factor app methodology for building robust SaaS*. URL: http://12factor.net/ (cit. on p. 24).

[Inc13] Z. Inc. *Rest Hooks Docuemntation*. 2013. URL: resthooks.org/docs (cit. on pp. 59, 60).

[Inc16] A. Inc. *Auth0 Webtasks Documentation*. 2016. URL: https://webtask.io/ (cit. on p. 48).

[Joh] L. Johansson. *What is message queueing?* URL: https://www.cloudamqp.com/blog/2014-12-03-what-is-message-queuing.html (cit. on p. 62).

[LF14] J. Lewis, M. Fowler. *Microservices , a definition of this new architectural term*. 2014. URL: http://martinfowler.com/articles/microservices.html (cit. on p. 26).

[Lin07] J. Lindsay. *Web hooks to revolutionize the web*. 2007. URL: http://progrium.com/blog/2007/05/03/web-hooks-to-revolutionize-the-web/ (cit. on p. 60).

[Mer14a] D. Merkel. *Docker: Lightweight Linux Containers for Consistent Development and Deployment*. May 2014. URL: http://www.linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment (cit. on p. 29).

[Mer14b] D. Merkel. "Docker: Lightweight Linux Containers for Consistent Development and Deployment". In: *Linux J.* 2014.239 (Mar. 2014). URL: http://dl.acm.org/citation.cfm?id=2600239.2600241 (cit. on p. 105).

[MF07] L. Morgan, P. Finnegan. "Benefits and Drawbacks of Open Source Software: An Exploratory Study of Secondary Software Firms". In: *Open Source Development, Adoption and Innovation: IFIP Working Group 2.13 on Open Source Software, June 11–14, 2007, Limerick, Ireland*. Ed. by J. Feller, B. Fitzgerald, W. Scacchi, A. Sillitti. Boston, MA: Springer US, 2007, pp. 307–312. URL: http://dx.doi.org/10.1007/978-0-387-72486-7_33 (cit. on p. 38).

[MG11] P. M. Mell, T. Grance. *The NIST Definition of Cloud Computing*. Tech. rep. Gaithersburg, MD, United States, 2011 (cit. on pp. 20, 21).

[Mue10] E. Mueller. *What Is DevOps?* Aug. 2010. URL: https://theagileadmin.com/what-is-devops/ (cit. on p. 105).

[Ope16] I. OpenWhisk. *OpenWhisk*. 2016. URL: https://developer.ibm.com/openwhisk/ (cit. on pp. 32, 39).

[Par79] D. L. Parnas. "Designing Software for Ease of Extension and Contraction". In: *IEEE Trans. Softw. Eng.* 5.2 (Mar. 1979), pp. 128–138. URL: http://dx.doi.org/10.1109/TSE.1979.234169 (cit. on p. 38).

[Ric] C. Richardson. *Microservice architecture patterns and best practices*. URL: http://microservices.io (cit. on p. 25).

[SA] P. Sbarski, the A Cloud Guru Team. *Serverless Cost Calculator*. URL: http://serverlesscalc.com/ (cit. on p. 99).

[Ser16] A. W. Services. *Amazon Web Services Lambda*. Amazon Web Services, Inc. 2016 (cit. on pp. 40, 47).

[Sho12] E. Shouten. "Rapid elasticity and the cloud". In: (2012). URL: http://www.thoughtsoncloud.com/2012/09/rapid-elasticity-and-the-cloud/ (cit. on p. 23).

[Sut16] P. Suter. "OpenWhisk Deep Dive: the action container model". July 2016. URL: http://www.slideshare.net/psuter/openwhisk-deep-dive-the-action-container-model (cit. on pp. 35, 38).

[Tea15] B. Team. *Bluemix Runtimes Docuemntation*. International Business Machines. 2015. URL: https://console.ng.bluemix.net/docs/cfapps/runtimes.html (cit. on p. 86).

[Tea16a] I. W. I. P. Team. *IBM® Watson™ IoT Platform HTTP REST API*. International Business Machines. 2016. URL: https://docs.internetofthings.ibmcloud.com/swagger/v0002.html (cit. on p. 74).

[Tea16b] I. W. I. R.-T. I. Team. *IBM® Watson™ IoT Real-Time Insights HTTP REST API*. International Business Machines. 2016. URL: https://iotrti-prod.mam.ibmserviceengage.com/apidoc/ (cit. on p. 80).

[Tho16] J. Thomas. *OpenWhisk MQTT Feed*. 2016. URL: https://github.com/jthomas/openwhisk_mqtt_feed (cit. on p. 74).

[Wag15] T. Wagner. "Running Arbitrary Executables in AWS Lambda". In: *AWS Lambda blog* (2015). URL: https://aws.amazon.com/blogs/compute/running-executables-in-aws-lambda/ (cit. on p. 40).

[XLR+04] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, L. Zhang. "A Smart Hill-climbing Algorithm for Application Server Configuration". In: *Proceedings of the 13th International Conference on World Wide Web*. WWW '04. New York, NY, USA: ACM, 2004, pp. 287–296. URL: http://doi.acm.org/10.1145/988672.988711 (cit. on p. 29).

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature