Apache Openwhisk

Distributed Software Systems, 2024/25

Davide De Rosa

1 Introduction to Serverless Computing

Scalability and elasticity are essential features of cloud computing, yet even after a decade, they remain inaccessible for many cloud users. Although cloud computing has freed developers from managing physical infrastructure, it still requires oversight of virtual resources for software deployments. These limitations, combined with the shift toward containers and microservices in enterprise application architectures, have given rise to a new deployment paradigm known as **Serverless Cloud Computing**[1].

Over the past decade, cloud platform hosting has evolved significantly[2]:

- Initially, organizations purchased or rented physical servers to run applications, incurring costs for both the hardware and its maintenance.
- This shifted with the adoption of virtualization, which allowed a single physical server to function as multiple software-defined Virtual Machines (VMs), enhancing flexibility and resource utilization. Containerization emerged as a further refinement, combining aspects of virtualization with configuration management.
- The introduction of **Platform-as-a-Service** (**PaaS**) took abstraction further, freeing users from managing servers and deployment processes.
- The latest advancement, serverless computing, builds on *PaaS* by enabling deployment of small code fragments that can autonomously scale, supporting the creation of self-scaling applications.

Serverless computing, also called **Function as a Service** (**FaaS**), completely separates backend infrastructure management from application development, hiding server maintenance responsibilities from users. In serverless computing, the cloud provider handles *server management*, *function execution*, *capacity planning*, *resource allocation*, *task scheduling*, *scalability*, *deployment*, *operational monitoring*, and *security updates*. This model implements **event-driven programming**, where applications utilize **small**, **stateless functions** (or handlers) that are **triggered by events**.
Users simply upload code, trigger stateless functions through events, and **pay only for the actual runtime of their code**.

Delegating server management to cloud providers presents both benefits and challenges for *providers* and *users*.

For **users**, serverless computing eliminates the need for server management while offering

a simplified programming model that abstracts many operational concerns. Features like autoscaling and scaling to zero enable genuine pay-as-you-go billing.
However, challenges remain, including limited support for different programming languages and libraries, state management, monitoring, debugging, and execution time constraints.

For **cloud providers**, serverless computing offers new opportunities by allowing full resource control and operational cost reduction through optimized resource management.
At the same time, providers face significant challenges such as cold starts (the delay in starting a new function instance), scheduling policies, scaling, performance prediction, dynamic resource provisioning, I/O bottlenecks, communication delays due to slow storage, pricing models, and issues around security and privacy.[1]

2 Serverless Architecture

Serverless computing has become a key model for developing and deploying cloud applications, offering a zero-administration approach by fully decoupling backend infrastructure management from application development. In this model, applications are composed of small, stateless functions or handlers that operate independently and execute in response to specific events. Cloud providers run these handlers in isolated sandboxes, such as **containers**, allowing for both **secure isolation** and **efficient autoscaling**, while enabling a genuine **pay-as-you-go** billing model.

The architecture of a serverless platform is illustrated in Fig. 1. In this setup, the serverless platform receives an event sent via HTTP or another cloud-based source. Internally, the platform includes an event queue, a dispatcher, and worker nodes. Upon receiving a request, the system identifies the appropriate function(s) to handle the event. The dispatcher then routes the request to a worker node, retrieving the necessary function code from a database.

The worker node executes the function(s), launching one or more instances depending on the request volume and available resources. If resources are sufficient, function instances (typically run in containers) are launched with restricted resources to carry out the task. If resources are limited, the function request is queued until resources become available.

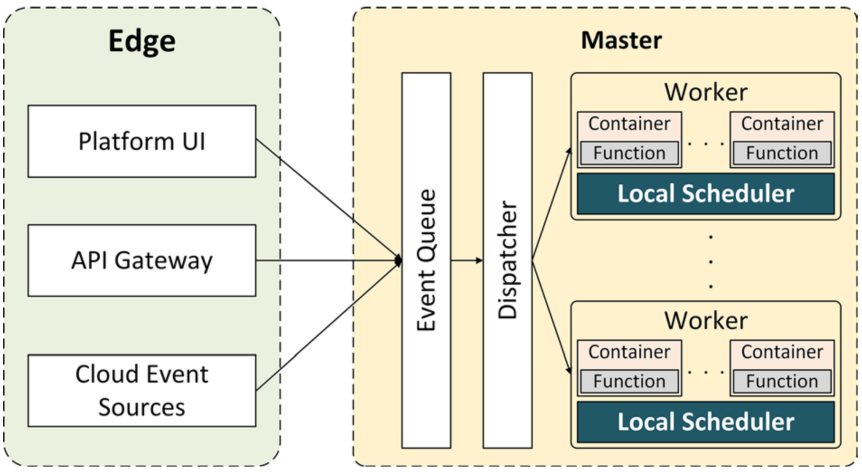

Figure 1: Blueprint of serverless platform architecture

A smart load balancer helps prevent workers from becoming overloaded. However, under high invocation rates or sudden bursts, a long execution queue can lead to significant wait times. Additionally, when resources are optimized to use the fewest nodes, some functions may experience delays, affecting **Quality of Service** (**QoS**). Consequently, the local scheduling approach plays a critical role in optimizing average function wait times, throughput, and resource utilization at worker nodes.

Starting a new instance requires initializing the container with necessary libraries, which can introduce startup latency – commonly known as a **cold start**. To reduce cold start delays, cloud providers often keep instances paused after use so they can be quickly reused for future invocations. After a function completes or reaches its maximum execution time, the worker node collects execution logs and makes the results available to the user. Scale-to-zero functionality allows users to pay only for the exact time and resources used during active function execution.[1]

3 Apache Openwhisk

Apache Openwhisk is an **open-source serverless** platform created by **IBM** that executes functions in response to events, scaling automatically and managing all necessary infrastructure and services. OpenWhisk competes with large platforms like *Nginx*, *Kafka*, *Docker*, and *CouchDB*; together, these tools help create a comprehensive serverless cloud service.

It's commonly used for applications that need **flexibility**, **scalability**, and **agility**, especially where real-time processing or dynamic scaling is important. Some pratical scenarios where Openwhisk shines are *Real-Time Data Processing*, *IoT Data Processing*, *Automated DevOps Tasks*, *Serverless API Backend* and *Disaster Recovery and Monitoring*.

Additionally, OpenWhisk provides a *Command Line Interface* (*CLI*), known as *wsk*, which allows developers to easily create, execute, and manage OpenWhisk entities across any operating system, making platform interactions straightforward for developers.

Apache Openwhisk supports flexible deployment options across various platforms due to its containerized components, which enable deployment both locally and within cloud infrastructures. It can be deployed on platforms such as *Kubernetes*, *Mesos*, and *OpenShift*.

The programming model of Apache Openwhisk is built around three core components: *Actions*, *Triggers*, and *Rules*.

Actions are stateless functions that execute arbitrary code; **Triggers** represent a class of events originating from various sources, and **Rules** link a Trigger to an Action. OpenWhisk also allows chaining Actions into sequences.

The model supports multiple programming languages, including *Java*, *Python*, and *JavaScript*, and uses an event-driven architecture where most **Actions execute in response to events**.[3]

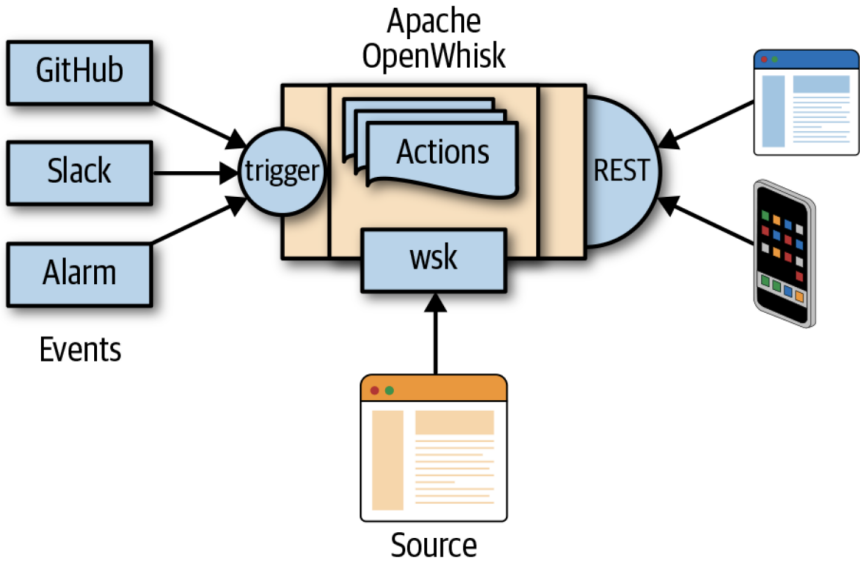

Figure 2: How Apache Openwhisk works

In summary, this platform's key features include[4]:

- **Deploys Anywhere**: with its container-based architecture, Apache Openwhisk offers versatile deployment options, supporting both local setups and various cloud infrastructures.
- **Supports Any Programming Language**: OpenWhisk is compatible with a wide range of programming languages, including NodeJS, Java, Scala, PHP, Python, and others. For unsupported platforms or languages, users can easily create and customize executables using Docker SDK to run on Docker.
- **Integration Support**: OpenWhisk enables easy integration of developed Actions with popular services through pre-built packages, either from independent projects or the default catalog. These packages offer integrations with services such as Kafka message queues, databases, mobile applications, messaging services, and RSS feeds.

- **Rich Function Composition**: functions written in multiple programming languages can be packaged with Docker for flexible invocation options, including synchronous, asynchronous, or scheduled execution. Parameter binding is recommended to prevent hardcoding service credentials directly into code.
- **Scalability and Resource Optimization**: OpenWhisk allows Actions to scale instantly, handling thousands of executions in seconds or running only as needed, such as on a weekly schedule. Resources scale automatically to match demand, pausing when idle, so users only pay for actual usage with no costs for unused resources.

4 Apache Openwhisk Main Architectural Drivers

Apache Openwhisk's architecture is shaped by several key non-functional requirements that are essential for delivering a *scalable*, *flexible*, and *reliable* Function-as-a-Service (FaaS) platform. These architectural drivers address the *performance*, *scalability*, *flexibility*, *reliability*, *security*, and *ease of use* demanded in serverless environments.
Below is an overview of the main architectural drivers and their impact on Openwhisk's design.

4.1 Scalability and Elasticity

One of the primary drivers for any FaaS platform is the ability to **scale on-demand** in response to **workload fluctuations**. In serverless environments, workloads are often highly dynamic, requiring rapid scaling up during peak demand and scaling down to save resources during idle times.

To support elasticity, Openwhisk's architecture includes components like an **Activation Dispatcher**, which manages and routes incoming function requests, and an **Invocation Controller**, which schedules these functions onto available resources.
Openwhisk supports stateless function execution, which allows multiple instances to scale in parallel across distributed nodes without dependencies. The use of container-based isolation (e.g., *Docker* or *Podman*) also enables fast provisioning of resources, which is essential for maintaining high scalability.

4.2 Low Latency and Fast Cold Start Times

In serverless architectures, the platform needs to handle sporadic or unpredictable workloads, which often leads to *cold starts* – instances where containers are launched from scratch due to inactivity. Low latency in handling requests, both during cold starts and for warm containers, is critical to ensuring a responsive user experience.

Openwhisk uses a **pre-warming strategy** where a pool of containers remains active to handle requests immediately, reducing latency from cold starts. Furthermore, the system uses a stateless design for function execution, allowing resources to be quickly reused across

requests. This architectural choice **minimizes the overhead** of setting up new containers, thus achieving **faster response times even under high concurrency**.

4.3 Fault Tolerance and Reliability

Fault tolerance is essential to ensure that the platform can **handle failures** without impacting the overall service availability. In a distributed, event-driven environment, functions must be resilient to network, hardware, and software failures.

Openwhisk's architecture is designed to distribute functions across **multiple nodes**, allowing *redundancy* and *failover capabilities*. The platform's messaging backbone, typically implemented with *Apache Kafka*, supports high-availability messaging and enables reliable event-driven processing. Additionally, Openwhisk includes **retry mechanisms** for function invocations and provides error-handling configurations that enable applications to recover gracefully from failures.

4.4 Multi-Tenancy and Isolation

Multi-tenancy is an essential feature for serverless platforms used by **multiple users or teams**, requiring isolation to prevent cross-tenant interference or security risks. Proper isolation is also critical to ensure that a high workload from one user does not negatively impact others.

Openwhisk uses **container-based isolation** to execute functions in separate containers, ensuring that the memory, processing, and storage of one function do not affect others. Additionally, Openwhisk includes resource limits and quotas per user or namespace, which helps enforce isolation and control resource usage across tenants. This architecture not only protects user functions but also maintains predictable performance across multiple concurrent workloads.

4.5 Extensibility and Customizability

Open-source platforms like Openwhisk aim to be flexible enough to **support various deployment scenarios**, **languages**, and **runtimes**. Extensibility allows organizations to tailor the platform to their specific needs and integrate with different cloud providers or on-premises environments.

Openwhisk's open-source, modular architecture allows developers to add custom runtimes, modify the scheduling logic, or extend the platform's capabilities with additional services. Openwhisk **supports various languages out-of-the-box** (such as JavaScript, Python, and Swift), and its plug-and-play nature lets users develop **custom runtimes** as needed. This extensibility is a key driver for Openwhisk's popularity in hybrid or private cloud deployments, as it enables full control and customizability to meet organizational needs.

4.6 Security and Compliance

Security is a primary concern in any multi-tenant environment, where data *confidentiality*, *access control*, and *isolation* are critical. Compliance with data protection standards (e.g., GDPR, HIPAA) is also often required, especially for enterprises handling sensitive data.

Openwhisk includes robust security features, such as **API authentication**, **role-based access control**, and **encrypted communications** between services. By running functions in isolated containers, Openwhisk minimizes the risk of unauthorized access between functions. Additionally, as an open-source project, Openwhisk allows users to implement and audit additional security measures in compliance with organizational and industry standards.

4.7 Observability and Monitoring

Observability is essential for maintaining high availability and performance in serverless platforms, especially given the ephemeral nature of FaaS functions. Monitoring tools are needed to gain insights into function execution, detect failures, and optimize performance.

Openwhisk includes **logging and monitoring capabilities**, enabling users to track function execution, resource usage, and latency. By integrating with external monitoring tools like *Grafana* and *Prometheus*, Openwhisk supports comprehensive observability. Furthermore, the platform's event-driven nature allows it to capture detailed traces of function invocations, which helps in debugging and optimizing application performance.

These architectural drivers form the foundation of Apache Openwhisk's design and differentiate it in the competitive landscape. By prioritizing *scalability*, *low latency*, *fault tolerance*, *multi-tenancy*, *extensibility*, *security*, and *observability*, Openwhisk provides a robust, customizable, and reliable serverless platform that can adapt to a wide variety of deployment scenarios and workload requirements.

These drivers are central to Openwhisk's appeal as an open-source, vendor-agnostic FaaS solution suitable for both cloud-native and hybrid environments.

5 Apache Openwhisk Architecture

Fig. 3 shows the architecture of *Apache Openwhisk*, which consists of two primary components: the **Controller** and the **Invoker**, built on **Nginx**, **Kafka**, **Docker**, and **CouchDB**. Together, these components enable OpenWhisk to function as a *serverless event-driven programming service*. OpenWhisk offers a *RESTful API* that allows users to submit functions and retrieve execution results.

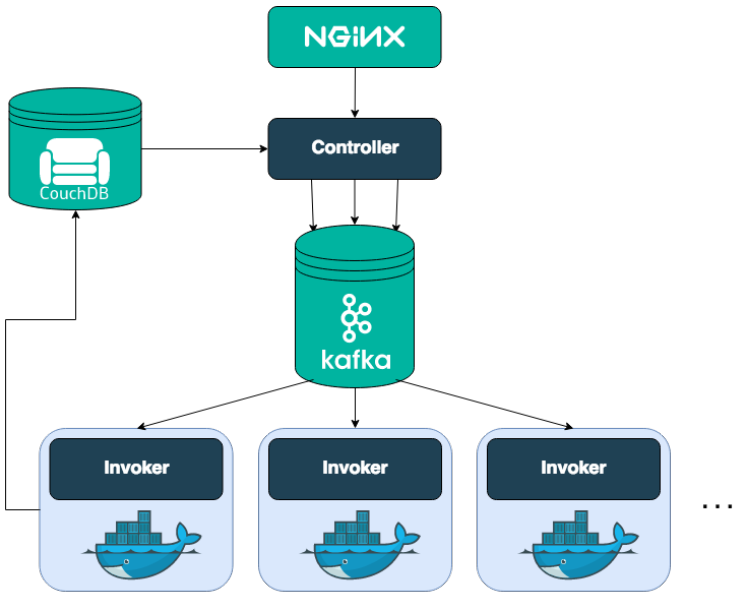

Figure 3: Apache Openwhisk architecture

Nginx routes incoming requests to the *Controller*, which handles authentication, retrieves the requested functions from the **CouchDB** database, and directs them to the *Invokers* acting as a *Load Balancer*.

Kafka, a high-performance message distribution system, facilitates communication between the *Controller* and the *Invokers*.

The *Invokers*, distributed across multiple machines and responsible for hosting serverless function containers, execute function calls by allocating resources within **Docker** containers and assigning a container to each function invocation. Essentially, *Invokers* serve as the *worker nodes* in Openwhisk (as represented in Fig. 1).

Each *Invoker* has an in-memory queue to manage function requests when resources are temporarily unavailable. Once resources are freed, functions are dequeued and executed in a **First Come First Serve** (**FCFS**) order. All *Invokers* use the same instructions embedded in the *Invoker component's source code* (written in *Scala*), ensuring uniform operation across all *Invokers*.

Users can register on the platform to upload their functions, specifying only the memory required for each function's execution. [1]

Fig. 4 shows how Openwhisk processes an action into more details. [5]

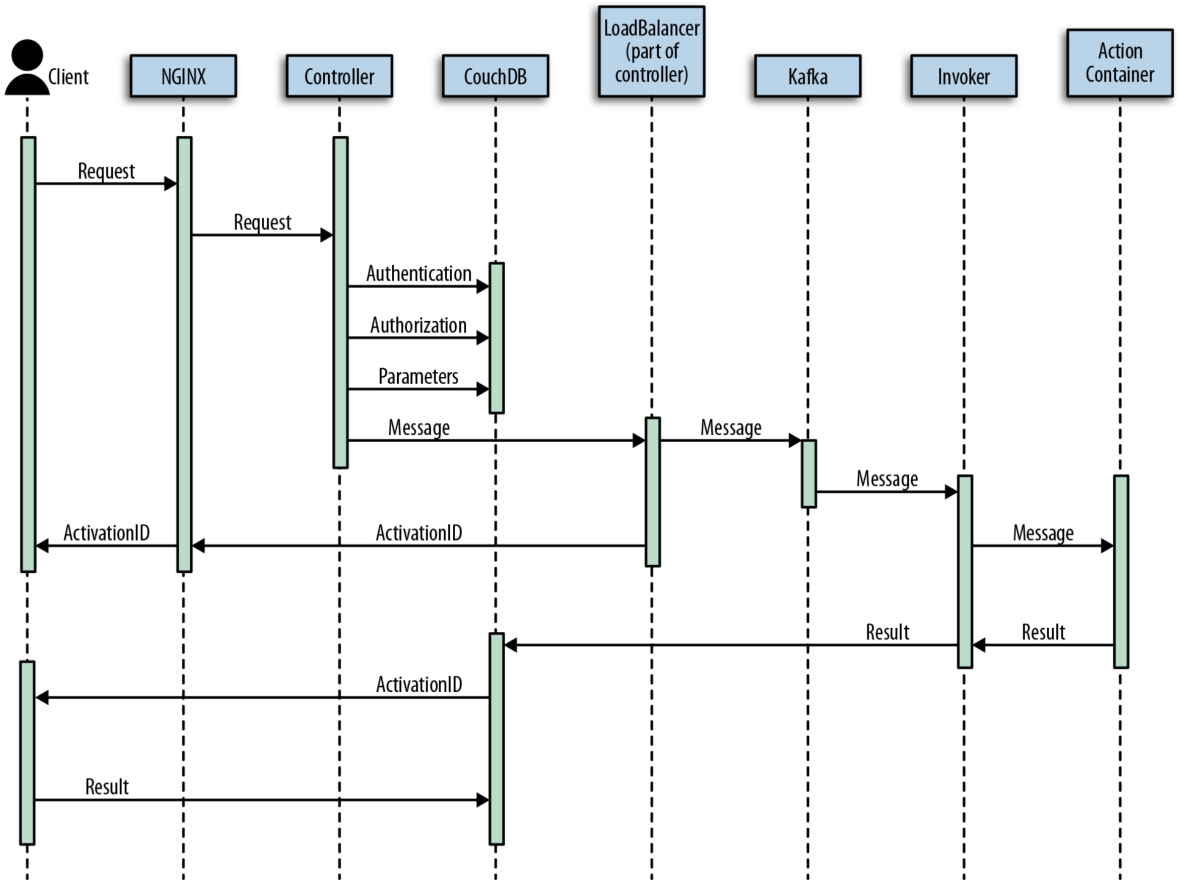

Figure 4: How OpenWhisk processes an action

Nginx

The process begins when an action is invoked, which can happen in several ways:

- Via the web, if the action is set up as a web-accessible action.
- Through the API, when one action calls another.
- By activating a trigger with a rule linked to the action.
- From the CLI (Command Line Interface).

The "client" refers to the entity initiating the action. As OpenWhisk is a RESTful system, each invocation is translated into an HTTPS request that first reaches the "edge" node. This edge node is the web server and reverse proxy, Nginx. Nginx's primary function here is to handle HTTPS, deploying the necessary certificates for secure processing.
Once secure, Nginx forwards the request to the system's core service component, the controller.

Controller
Before executing an action, the controller verifies that the action can be properly executed and initialized:

- **Authentication**: the controller must first authenticate the request to confirm it can execute the action.
- **Authorization**: after confirming the request's origin, the controller checks that the client has the necessary permissions to proceed.
- **Parameter Enrichment**: the request is then supplemented with additional parameters that are part of the action's configuration.

To carry out these steps, the controller consults CouchDB, the database used by OpenWhisk. Once the request is validated and enriched, the action is ready for execution and is sent to the load balancer, the next step in the processing flow.

Load Balancer
The load balancer in OpenWhisk is responsible for distributing the processing load across various executors, known as invokers.
Since OpenWhisk runs actions within designated runtimes, the load balancer tracks available runtime instances. When possible, it reuses existing instances to execute actions; if not, it creates new instances as needed.

When the system is ready to invoke an action, the load balancer can't send requests directly to an invoker if it is already occupied, has crashed, or if the system is undergoing a restart. Operating in a highly parallel, scalable environment means the system may not always have immediate resources available, necessitating a buffer for pending invocations. OpenWhisk uses Kafka for this buffering.

Kafka, a high-performance "publish and subscribe" messaging system, stores requests until they are ready for execution. Each action invocation is converted into an HTTPS request to Nginx, which is then transformed into a Kafka message directed to the chosen invoker.
Each message assigned to an invoker includes a unique identifier, the activation ID. Once queued in Kafka, the system offers two types of invocation responses:

- **Nonblocking Invocation**: the activation ID is immediately returned to the client as the response, completing the request. The client can later use this ID to retrieve the invocation result.
- **Blocking Invocation**: the connection remains open as the controller waits for the action to complete. Once the result is ready, it is sent directly to the client.

Invoker
In OpenWhisk, the invoker is responsible for executing actions within isolated environments provided by Docker containers. Docker containers simulate a complete operating system, offering a self-contained environment with all dependencies needed to run an action.

From the action's viewpoint, the Docker container resembles a full computer, similar to a virtual machine (VM), but container execution is significantly more efficient than VMs, making containers the preferred choice.
Docker uses images to create containers for action execution. Each runtime corresponds to a Docker image, and the invoker launches a new image for the specified runtime, then loads the action's code into it. OpenWhisk offers a variety of pre-built Docker images that support different programming languages, including JavaScript, Python, Go, and Java. These images come with initialization logic specific to each runtime.
Once the runtime environment is initialized, the invoker processes action requests, handling and storing logs to aid debugging.

After the action execution is complete, OpenWhisk stores the result in CouchDB (also used for configuration data storage). Each action result is tagged with the activation ID, which was previously provided to the client. This allows the client to retrieve the execution result later by querying CouchDB using the activation ID.

Client
The processing flow in OpenWhisk is primarily asynchronous. In this mode, the client initiates a request and receives an activation ID, which acts as a reference for the invocation. This activation ID allows OpenWhisk to store the result of the action in the database upon completion. To retrieve the final result, the client must send another request later, using the activation ID as a parameter. Once the action finishes, the result, logs, and additional information are stored in the database, making them accessible for the client.

OpenWhisk also supports synchronous processing, which operates similarly but with one key difference: the client's request remains open, blocking until the action completes, allowing the client to immediately receive the result.[5]

6 Competing solutions

Apache Openwhisk is one of many solutions in the rapidly evolving serverless and Function-as-a-Service (FaaS) ecosystem. It competes with several other platforms, each offering distinct features, operational models, and deployment options. Notable competitors include *Amazon Web Services* (*AWS*) *Lambda*, *Google Cloud Functions* and *Microsoft Azure Functions* Below is an overview of how these solutions compare with Apache Openwhisk in terms of functionality, scalability, ecosystem integration, and operational control.

6.1 AWS Lambda

AWS Lambda, launched in 2014 by **Amazon Web Services**, was the first mainstream FaaS platform and remains a leader in the market. AWS Lambda enables users to run code in response to events without managing servers.

Strengths: Lambda offers seamless integration with AWS's extensive suite of services, which

makes it highly appealing for users already in the AWS ecosystem. Its developer tooling and monitoring support are advanced, with options for logging, tracing, and error handling through services like Amazon CloudWatch and X-Ray.

Weaknesses: Lambda is closely tied to the AWS ecosystem, and migrating workloads to or from AWS can involve considerable re-architecture. Additionally, Lambda's default runtime has a cold start latency, which can affect performance for latency-sensitive applications.

Comparison with Openwhisk: while AWS Lambda excels in native AWS integration and scalability, Openwhisk offers a more flexible deployment model, allowing for cloud, hybrid, or on-premises installations, which is ideal for users with strict data locality requirements or those operating in multi-cloud environments.

6.2 Google Cloud Functions

Google Cloud Functions (**GCF**) provides an event-driven serverless compute service on **Google Cloud Platform**. It supports multiple languages, including Node.js, Python, and Go.

Strengths: GCF is tightly integrated with GCP services, including Google Cloud Pub/Sub, Google Firestore, and Firebase. It is known for rapid auto-scaling and has low latency due to Google's extensive global network infrastructure.

Weaknesses: like Lambda, GCF's integration primarily focuses on GCP, making it less flexible for users with multi-cloud requirements. Cold start times, although improved, remain a concern in certain use cases.

Comparison with Openwhisk: GCF is highly performant within the Google ecosystem but lacks the deployment flexibility that Openwhisk provides. Openwhisk's open-source model allows for higher customization and control, while GCF is a managed solution with limited customization.

6.3 Microsoft Azure Functions

Microsoft Azure Functions is a FaaS solution on the **Azure cloud platform**, supporting a range of programming languages and tight integration with Microsoft's services like Azure DevOps, Azure Cosmos DB, and Azure Event Grid.

Strengths: Azure Functions are well-suited for .NET developers, given Microsoft's strong support for C# and Visual Studio integration. The platform is known for its extensive developer tooling, ease of use, and support for both consumption-based and dedicated pricing plans.

Weaknesses: Azure Functions, like other proprietary platforms, has limitations around multi-cloud or on-premises deployment. There are also some concerns around cold start

latency, though Microsoft offers a premium plan to mitigate this.

Comparison with Openwhisk: Azure Functions offers more extensive support for .NET and is tightly integrated with the Azure ecosystem, while Openwhisk is more language-agnostic and portable. Openwhisk's flexibility for hybrid or on-premises deployment may be more appealing for organizations that require complete control over their serverless environments.

Each of these competing solutions has strengths that make them ideal for specific use cases, with choices largely influenced by an organization's cloud strategy, operational requirements, and existing skill sets. While AWS Lambda, Google Cloud Functions, and Azure Functions offer powerful, managed FaaS options, they may be less flexible for organizations seeking to avoid vendor lock-in or requiring more control over their infrastructure.

Apache Openwhisk, as an open-source, language-agnostic, and flexible platform, stands out for users seeking an adaptable, hybrid FaaS solution that can span both cloud and on-premises environments, without the proprietary constraints of fully managed platforms.

7 Implementation

Before talking about the implementation, I want to define the main objective for this simple implementation. I want to expose an HTTP GET endpoint, which return a simple *Hello World* string. Unfortunately, that's where the simple stuff ends.

First of all, I tried to read **Apache Openwhisk documentation**[1] to understand more the Openwhisk environment's setup process.
After different attempts to make the *Docker* setup work, I even tried the *Java Standalone* version. Nothing was working the way it was supposed to. I even tried deploying everything on different virtual machines, other than my own machine.

I then decided to talk with *Prof. Ciancarini* to get a different point of view of the problem. He suggested to write to *Michele Sciabarrà*, the author of the book **Learning Apache OpenWhisk**[2], which runs a company that uses Apache Openwhisk as the backbone of their service. The company is called **Nuvolaris**[3], and it offers a new serverless approach to Kubernetes.
While talking with Sciabarrà, he suggested to use their *spin-off* of Openwhisk, called **Apache OpenServerless**[4], which has a simpler way of deploying the Openwhisk core functionalities.

Unfortunately, this process will not be any simpler. I had issues related to this process of deployment too. Reading their documentation, the process was similar to the other one,

[1] `https://openwhisk.apache.org/documentation.html`
[2] `https://www.oreilly.com/library/view/learning-apache-openwhisk/9781492046158/`
[3] `https://www.nuvolaris.io/`
[4] `https://openserverless.apache.org/`

with less dependencies needed.
After some days, during which I tried every possible way of making it work, I tried talking directly with the creators of the Apache OpenServerless software. With their help, I managed to get the system to work, and to finally deploy my *"simple"* implementation.

Heartfelt thanks go to *Michele Sciabarrà* and his team, for real-time support while debugging the various problems encountered.
Also, a big thank you to the **ADM team**[5] – especially *Emanuele Grasso*. Thanks to his support, I was provided with a virtual machine that met the requirements of OpenServerless and allowed me to perform all the necessary tests. Their work goes beyond this, providing a very useful service for all students in the Computer Science department and beyond.

Let's get back to the implementation.

Using the *OpenServerless CLI* (**ops**), I managed to get everything up and running on the virtual machine. I then wrote a really simple **Python** script, which return an *Hello World* message:

```
def main(args):
    return {"body": "hello world\n"}
```

Using the *ops package create test_dss* command, I created a test package where *actions* can be stored.

ops action create test_dss/greet script.py –web true is going to create the action related to the Python script. Using the *–web* flag with a value of *true* or *yes* allows an action to be accessible via **REST** interface without the need for credentials.

To get the URL related to the action, the *ops action get test_dss/greet –url* command is used. This returns the action HTTP endpoint, ready to be called and tested.

An example of this whole execution of commands can be:

```
debian@vm-DavideDeRosa:~/test$ ops package create test_dss
ok: created package test_dss
debian@vm-DavideDeRosa:~/test$ ops action create test_dss/greet script.py --web true
ok: created action test_dss/greet
debian@vm-DavideDeRosa:~/test$ ops action get test_dss/greet --url
ok: got action greet
http://localhost:80/api/v1/web/nuvolaris/test_dss/greet
debian@vm-DavideDeRosa:~/test$ curl http://localhost:80/api/v1/web/nuvolaris/test_dss/greet
hello world
```

If you need to update the script, *ops action update test_dss/greet script.py –web true* is all you need to do.

This test shows just the core functionalities of Openwhisk (and OpenServerless). There is

[5] https://students.cs.unibo.it/

a lot more that can be done with this tool. The documentation shows a lot of possible implementations to do, all related to serverless functions.
In a possible future study, it would be possible to test Openwhisk performance compared to other Cloud based solution – like *AWS Lambda* or *Google Cloud Functions*.

References

[1] A. Banaei and M. Sharifi, "Etas: predictive scheduling of functions on worker nodes of apache openwhisk platform," *The Journal of Supercomputing*, pp. 1–36, 2022.

[2] K. Djemame, M. Parker, and D. Datsev, "Open-source serverless architectures: an evaluation of apache openwhisk," in *2020 ieee/acm 13th international conference on utility and cloud computing (ucc)*, pp. 329–335, IEEE, 2020.

[3] S. Quevedo, F. Merchán, R. Rivadeneira, and F. X. Dominguez, "Evaluating apache openwhisk-faas," in *2019 IEEE fourth ecuador technical chapters meeting (ETCM)*, pp. 1–5, IEEE, 2019.

[4] L. P. Huy, S. Saifullah, M. Sahillioglu, and C. Baun, "Crypto currencies prices tracking microservices using apache openwhisk," *Frankfurt University*, 2021.

[5] M. Sciabarrà, *LEARNING APACHE OPENWHISK: developing open serverless solutions*. O'Reilly Media, 2019.