

# Apache OpenWhisk

Distributed Software Systems, 2024/25

---

Davide De Rosa

# Introduction to Serverless Computing

---

# Introduction to Serverless Computing

**Serverless Computing** has transformed how we develop and deploy cloud applications. Traditional cloud computing required developers to manage virtual machines or containers, while **Serverless** eliminates this burden.

This model operates through **small, stateless functions** triggered by **events**, with the **cloud provider** managing backend infrastructure. Key benefits include:

- **Autoscaling** to handle varying loads.
- A **pay-as-you-go** model, where costs align with actual function usage.

However, Serverless isn't without its challenges. Cold starts, limited runtime languages, and debugging complexities still hinder its adoption.

Despite this, serverless computing has become integral to modern architectures, thanks to its simplicity and cost efficiency.

# Introduction to Serverless Computing

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

- 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 Architecture

---

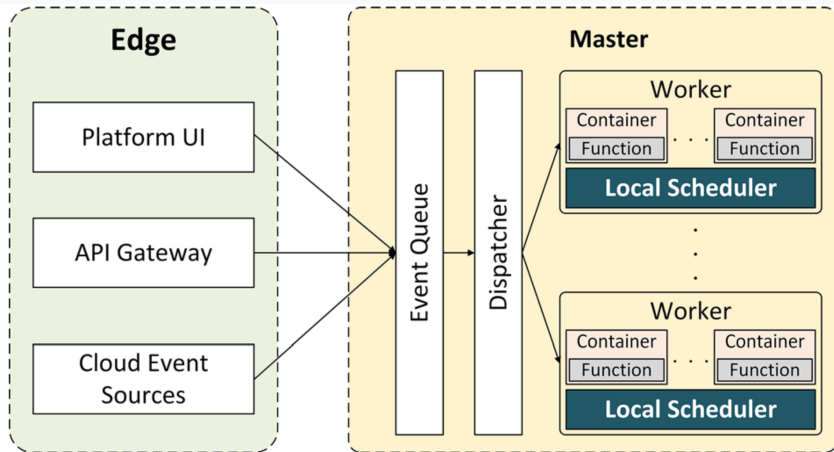
# Serverless Architecture

At its core, Serverless Computing revolves around simplicity and efficiency. Functions run in isolated containers, with each triggered by specific events like HTTP requests or database updates. The process begins when an event triggers the platform:

- An **event queue** receives the trigger and informs a dispatcher.
- The **dispatcher** assigns the function to a **worker node**.
- If a **worker** is free, it executes the function. Otherwise, the request is queued.

A load balancer ensures smooth operations, though challenges like resource contention and cold starts remain. To address this, providers pre-warm containers, minimizing latency during new invocations.

# Serverless Architecture



# Apache Openwhisk

---



# Apache Openwhisk

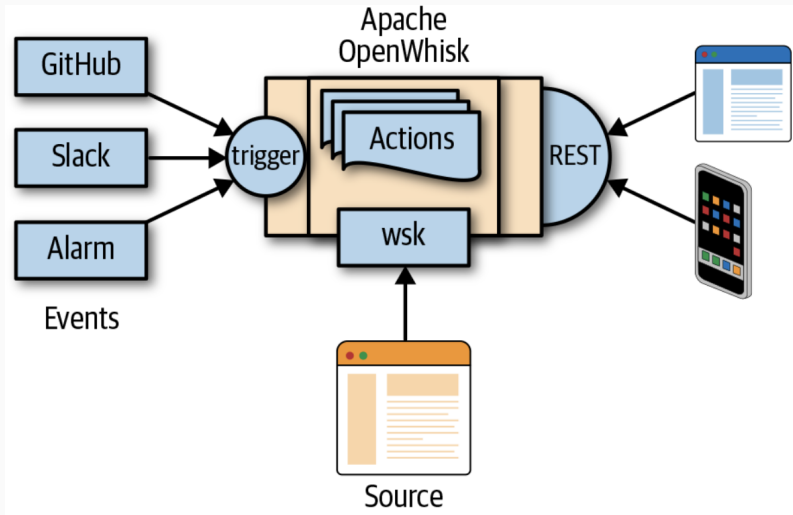
**Apache Openwhisk** is **IBM's open-source** serverless platform. It competes with giants like *AWS Lambda* and *Google Cloud Functions* by offering flexibility and openness. Developers use Openwhisk to build applications requiring *real-time processing*, *IoT data handling*, or *serverless APIs*.

Its design is **event-driven**, and its programming model is built around three elements:

- **Actions:** Stateless functions that process events.
- **Triggers:** Events that activate functions.
- **Rules:** Bindings connecting triggers to actions.

What sets Openwhisk apart is its adaptability. It supports deployment on platforms like *Kubernetes* and *OpenShift*, and its CLI tool makes it accessible across operating systems.

# Apache Openwhisk



## Apache Openwhisk Key Features

- **Deploys Anywhere:** with its container-based architecture, Openwhisk offers versatile deployment options, supporting local setups and cloud infrastructures.
- **Supports Any Programming Language:** Openwhisk is compatible with a wide range of programming languages. For unsupported platforms or languages, users can easily create and customize executables using Docker.
- **Integration Support:** Openwhisk enables easy integration of developed Actions with popular services through pre-built packages.
- **Rich Function Composition:** functions written in multiple programming languages can be packaged with Docker for flexible invocation options, including synchronous, asynchronous, or scheduled execution.
- **Scalability and Resource Optimization:** Openwhisk allows Actions to scale instantly. Resources scale automatically to match demand, pausing when idle, so users only pay for actual usage with no costs for unused resources.

# **Apache Openwhisk**

## **Main Architectural Drivers**

---

# Apache Openwhisk Main Architectural Drivers

The success of Apache Openwhisk lies in its architectural foundations, tailored to meet the demands of Serverless Computing:

- **Scalability and Elasticity:** stateless functions scale instantly to meet demand. Openwhisk leverages containerized isolation for rapid provisioning.
- **Low Latency:** pre-warming containers addresses the infamous cold start issue.
- **Fault Tolerance:** using *Apache Kafka* as a messaging backbone ensures reliable event handling, even in distributed setups.
- **Extensibility:** Openwhisk's modular architecture lets developers customize runtimes, add integrations, or tailor deployments for hybrid clouds.
- **Security:** containers provide isolation, while role-based access control and encrypted communications strengthen protection.

By addressing these factors, Openwhisk aligns with enterprise needs while maintaining its open-source ethos.

# Apache Openwhisk Architecture

---

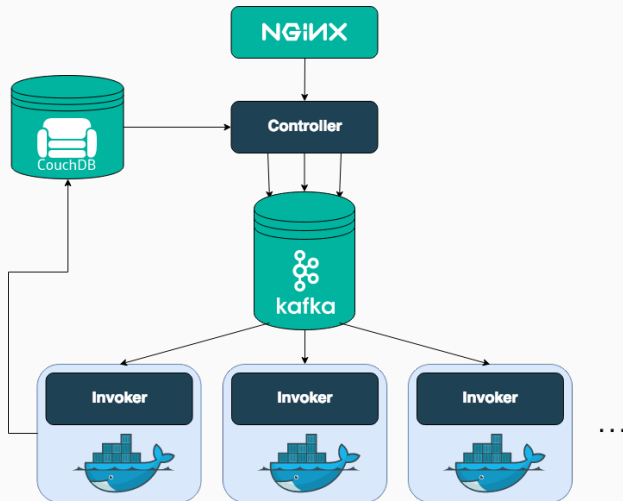
# Apache Openwhisk Architecture

Openwhisk's architecture 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.

# Apache Openwhisk Architecture





# 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.

# Apache Openwhisk Architecture

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.

## Competing solutions

---

# Competing solutions

- **AWS Lambda**

- **Strength:** deep integration with AWS services.
- **Weakness:** vendor lock-in.
- **Comparison:** Lambda excels in AWS ecosystems, while Openwhisk's open-source model appeals to those avoiding lock-in.

- **Google Cloud Functions**

- **Strength:** low latency and tight GCP integration.
- **Weakness:** limited multi-cloud deployment.
- **Comparison:** Openwhisk offers more deployment flexibility.

- **Azure Functions**


- **Strength:** strong .NET support.
- **Weakness:** constrained by Azure's ecosystem.
- **Comparison:** Openwhisk's language agnosticism and hybrid deployment edge out Azure for broader needs.

# Implementation

---

## 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 returns an *Hello World* message:



```
def main(args):  
    return {"body": "Hello World\n"}
```

## Implementation

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.

# Implementation

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
```



## 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** – 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.



## References

---

## References

- Etas: predictive scheduling of functions on worker nodes of apache openwhisk platform (Banaei, Ali and Sharifi, Mohsen), 2022
- Evaluating apache openwhisk-faas, Quevedo (Sebastián and Merchán, Freddy and Rivadeneira, Rafael and Dominguez, Federico X), 2019
- Crypto currencies prices tracking microservices using apache OpenWhisk (Huy, Lam Phuoc and Saifullah, Saifullah and Sahillioglu, Marcel and Baun, Christian), 2021
- Open-source serverless architectures: an evaluation of apache openwhisk (Djemame, Karim and Parker, Matthew and Datsev, Daniel), 2020
- LEARNING APACHE OPENWHISK: developing open serverless solutions (Sciabarrà, Michele), 2019