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**.

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.

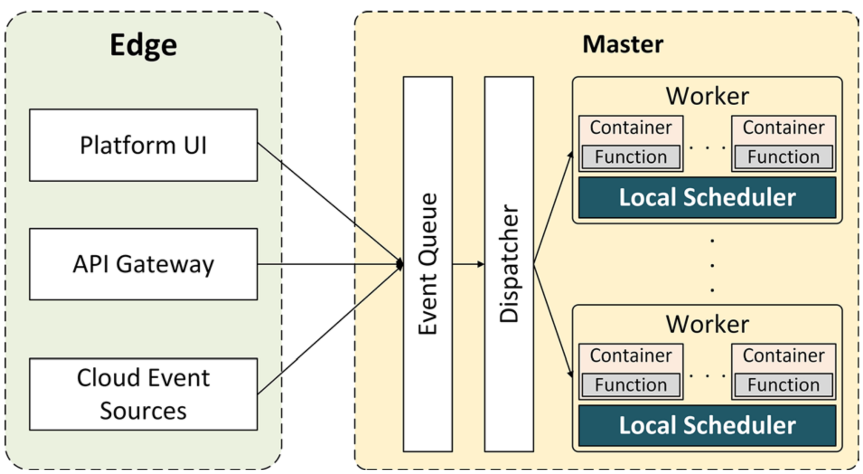

Figure 1: Blueprint of serverless platform architecture

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 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.

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**.[2]

In summary, this platform's key features include[3]:

- **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 Architecture

Fig. 2 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.

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]

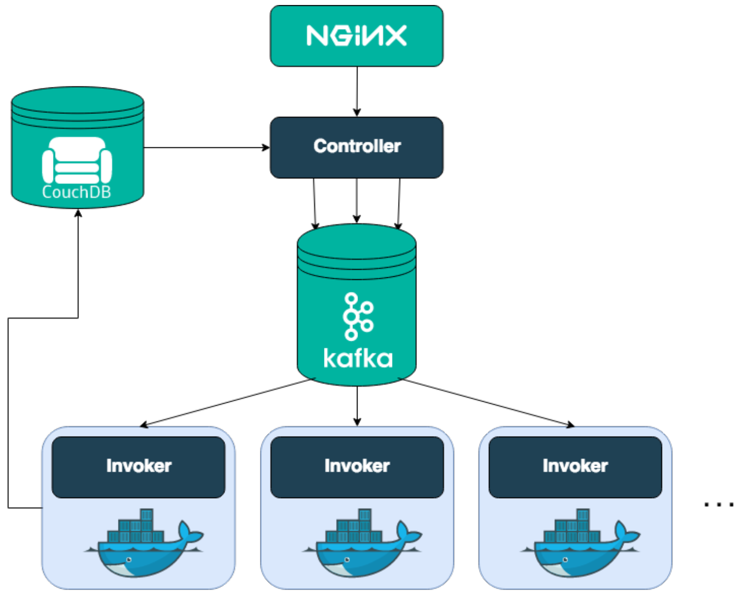

Figure 2: Apache OpenWhisk architecture

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] 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.

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