Serverless Cloud Platforms

Renaud Lachaize & Thomas Ropars
M2 GI
January 2024

Acknowledgments and main references (1/2)

- S. Kounev et al. **Serverless computing: What it is, and What it is not?** Communications of the ACM. September 2023.
  - https://cacm.acm.org/magazines/2023/9/275695-serverless-computing/fulltext

- J. Schleier-Smith et al. **What serverless computing is and should become: the next phase of Cloud computing.** Communications of the ACM. May 2021.
  - https://cacm.acm.org/magazines/2021/5/252179-what-serverless-computing-is-and-should-become/fulltext
  - **Note:** This is a simplified version of the next reference.

- E. Jonas et al. **Cloud programming simplified: A Berkeley view on Serverless Computing**. Technical report. February 2019.
  - https://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.pdf

- P.Castro et al. **The Rise of serverless computing.** Communications of the ACM. December 2019.
  - https://cacm.acm.org/magazines/2019/12/241054-the-rise-of-serverless-computing/abstract

Acknowledgments and main references (2/2)

- J. Hellerstein et al. **Serverless computing: One step forward and two steps back.** In Proceedings of CIDR 2019. January 2019.
  - https://www.cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf

- Cloud Native Computing Foundation (CNCF) Working Group (WG) **Serverless Whitepaper** v1.0. 2018.
  - https://github.com/cncf/wg-serverless/blob/master/whitepapers/serverless-overview/cncf_serverless_whitepaper_v1.0.pdf

- L. Wang et al. **Peeking behind the curtains of serverless platforms**. In Proceedings of USENIX ATC 2018. July 2018.
  - https://www.usenix.org/conference/atc18/presentation/wang-liang

Outline

- An overview of the serverless paradigm
- FaaS: Functions as a Service

Preamble: The typical pain points of low-level cloud services

1. Redundancy for availability
2. Geographic distribution of redundant copies (e.g., for disaster recovery)
3. Load balancing and request routing for efficient resource utilization
4. Autoscaling (up or down) in response to changes in input workload
5. Health monitoring
6. Logging events/messages (for debugging or performance tuning)
7. System upgrades (e.g., security patches)
8. Migration to new instances as they become available

Some of the main issues to be addressed when setting up an environment for cloud users.

(Source: E. Jonas et al. Cloud programming simplified. A Berkeley view on serverless computing. 2019.)

Introduction: What is "serverless computing"?

- **A recent and growing trend in the design/usage model of cloud services.**
- The "serverless" keyword refers to the fact that **the management of the underlying resources** (which is often complex, tedious, and time-consuming) **is offloaded to the cloud provider**.
- *Obviously, internally, a serverless platform still very much relies on physical servers!*

- Main goals:
  - **Simplicity** of usage, good **productivity** for application developers
  - Support for **"glue code" between various services** hosted by a cloud provider
  - **Very fine-grained pay-per-use billing model**, especially useful for short / sporadic / unpredictable tasks

"Serverless computing" – A tentative definition (1/5)

- Unfortunately, there is no standardized definition to date. However, several elements are commonly accepted.
- **Key characteristics**
  - **Management of resource allocation is completely offloaded** from the cloud user/tenant
    - ... including the management of **auto-scaling**.
  - **Billing is based on the actual resource usage, instead of the quantity of reserved resources.**
  - **Computation and storage are decoupled**
    - (at least for general-purpose compute tasks)
    - Distinct services, scaled independently
    - Computations are usually stateless

"Serverless computing" – A tentative definition (2/5)

- **Key abstractions:**
  - **General-purpose, user-programmable compute building blocks:**
    - A set of models sitting in-between IaaS/CaaS and PaaS
    - FaaS: "Functions as a Service" (also known as "Cloud functions")
    - Also, some forms/variants of CaaS (Containers as a Service)
  - **Scalable, provider-managed storage**
    - Possibly at various levels of abstraction (e.g., object store, NewSQL database …)
    - Billing based on number of requests + volume of data
  - **Trigger rules for computations**, associated with many different types of events/services, including:
    - Communication/network services
    - Storage services
    - High-level application workflows

"Serverless computing" – A tentative definition (3/5)

- **Also includes specific services and frameworks**:
  - Encompassing processing and/or storage
  - **Example 1: Mobile Backend as a Service (MBaaS)**
    - For the common services used by mobile/web applications: notifications, cloud storage, integration with social networks, analytics, authentication, …
    - Billing based on number of service invocations + storage
    - E.g., Google Firebase/Firestore, AWS Amplify
  - **Example 2: Messaging (also known as "publish/subscribe message bus")**
    - Middleware layer providing a decoupling between data/event producers (a.k.a. "publishers") and consumers (a.k.a. "consumers")
    - Supports various communication schemes: one to many, many to one, many to many
    - Useful for many different purposes, e.g., integration of various services, task dispatch, notification, data/stream ingestion …
    - Billing based on the volume of ingested data
    - E.g., Google Cloud Pub/Sub, AWS MSK

"Serverless computing" – A tentative definition (4/5)

- **Also includes specific services and frameworks (continued)**:
  - **Example 3: Big Data analysis**
    - **Big Data query**
      - SQL-like queries on very large data sets
      - Billing based on volume of processed data (+ storage)
      - E.g., AWS Athena, Google Cloud BigQuery
    - **Big Data transform**
      - Batch and/or stream processing
      - Billing based on processing time and/or data volume
      - E.g., AWS Glue (based on Apache Spark), Google Cloud Dataflow (based on Apache Beam)

"Serverless computing" – A tentative definition (5/5)

- **Warnings - Vocabulary**: Some people/documents:
  - Use "serverless" and "FaaS" interchangeably **[should be avoided]**
  - Use "Backend as a Service" (BaaS) as a general notion encompassing:
    - General-purpose storage systems (e.g., object storage, databases)
    - Specialized services of various kinds such as MBaaS and Big Data facilities
  - … or, in contrast, use BaaS and MBaaS interchangeably
  - Define "Serverless computing" as: FaaS + BaaS

“Serverless computing” – Another definition

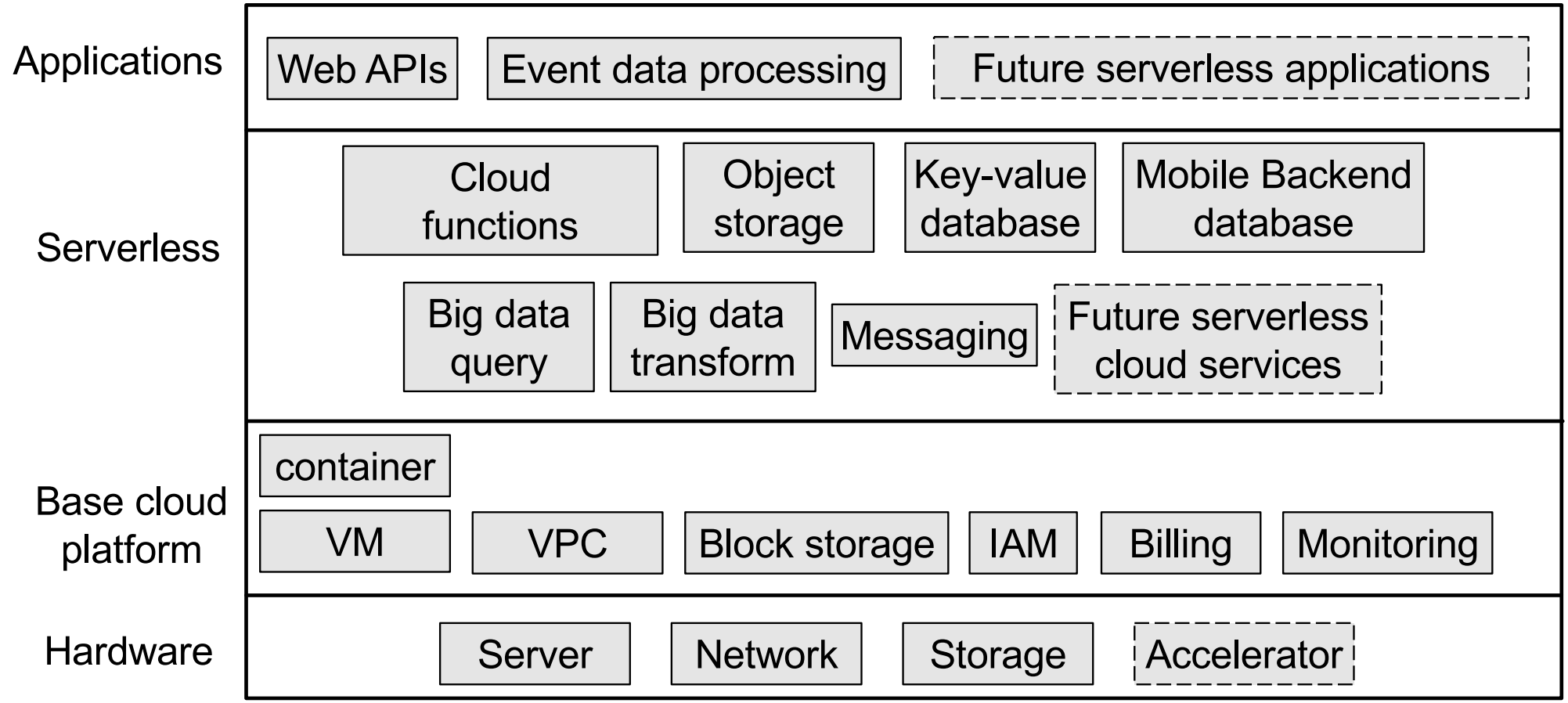
“Serverless computing is a cloud computing paradigm encompassing a class of cloud computing platforms that allow one to develop, deploy, and run applications (or components thereof) in the cloud **without allocating and managing virtualized servers and resources or being concerned about other operational aspects**.

The responsibility for operational aspects, such as fault tolerance or the elastic scaling of computing, storage, and communication resources to match varying application demands, is **offloaded to the cloud provider**.

Providers apply **utilization-based billing**: they charge cloud users with fine granularity, in proportion to the resources that applications actually consume from the cloud infrastructure, such as computing time, memory, and storage space.”

(Source: S. Kounev et al. ***Serverless computing: What it is, and What it is not?*** Communications of the ACM. September 2023.)

Architecture of the serverless cloud



(Figure adapted from the following source, with modifications: Jonas et al. “Cloud programming simplified: A Berkeley view on Serverless Computing.”)

Serverless computing vs. PaaS

- Arguably, there are **some similarities** between serverless computing and traditional PaaS offerings (or even older incarnations such as hosting of dynamic web sites):
  - Stateless programming model
  - Elasticity
  - Simplified programming and operational model for application developers

- However, there are also **some key differences** (in favor of serverless):
  - **Better autoscaling**: more generic, more reactive, greater scale, down to zero …
  - **Stronger isolation** (for security and performance)
  - **Billing model** with much finer granularity
  - Provider has more incentives to reduce overheads
  - **More generic platforms**: developers have more control/flexibility over the libraries that they can use
  - **More integration with other cloud services** (via triggers)

Serverless computing – Use cases

- Many use cases for applications running **within data centers**:
  - Web and API serving
  - Data processing, (e.g., ETL)
  - Integration with 3rd-party services
  - Internal tooling (e.g., testing, continuous integration/delivery)

- Also, other use cases **closer to the end users ("edge computing"):**
  - Content Delivery Networks (CDNs)
    - E.g., putting (parts of) a Web application closer to the end users
  - IoT devices and gateways
    - E.g., for faster/real-time processing, intermittent/slow connectivity to the cloud

- Note: Initially, serverless computing has been mostly used for batch workloads. Now, it is also increasingly used for latency-sensitive operations (e.g., Web application backend).

The FaaS ETL pattern – ("Extract, Transform, Load")

- "ETL": Histrorically, an expression initially used in the context of databases (for data transfer & synchronization between different types of databases).
- In the context of FaaS, "ETL" is an analogy to describe the design pattern of most functions:
  - **Extract**: obtain input data (either inlined in parameters or fetched from a remote storage)
  - **Transform**: perform some computation on the data and produce local result
  - **Load**: send the produced results, either as a result of the invocation or by writing it in a remote storage
- As of mid-2019, the six typical use cases depicted on the home page of AWS Lambda follow the FaaS ETL pattern:
  - **Data processing**: Real-time file processing, real-time stream processing, database ETL
  - **Backends**: IoT backend, mobile backend, Web applications.

(Sources: H. Fingler et al. USETL: Unikernels for Serverless Extract Transform and Load. Why should you settle for less? In Proceedings of APSys 2019. And also AWS Lambda home page: https://aws.amazon.com/lambda/)

Serverless computing – Potential benefits

- **For cloud tenants:**
  - Simplicity
  - Productivity
  - Increased portability (w.r.t. the low-level details of IaaS/CaaS)
  - Cost-efficiency (**warning**: not always)
  - Substrate for integration of various/arbitrary services

- **For cloud providers:**
  - Fine grained & stateless tasks simplify resource allocation and packing
  - Overall better resource usage and amortization
  - Leverage for “lock-in” of existing customers, given that:
    - There are currently no standardized interfaces/features for serverless programming
    - Serverless computing facilitates the integration of various services within a cloud provider’s portfolio and thus fosters tighter coupling between them

FaaS – "Functions as a Service"

- **The main building block of serverless computing.**
- Paradigm pioneered by AWS Lambda in 2014-2015

- **Now supported by all major cloud providers.** For example:
  - Google Cloud Functions
  - Microsoft Azure Functions
  - IBM Cloud Functions
- **Some open-source platforms**
  - Apache OpenWhisk
  - OpenFaaS
  - Various projects based on Kubernetes (Knative, Fission …)

- For a detailed list of tools and platforms, see: https://landscape.cncf.io/serverless

FaaS – Main principles

- **Programming model:**
  - Functions are intended for **short-lived tasks.**
  - **Functions must be stateless** (no state must be kept between invocations).
  - If necessary, an external storage system can be used to store/retrieve state.
- In the **setup phase**, the application developer:
  - writes one or several functions, and uploads them to the cloud
  - configures the **trigger rule(s)** associated with each function
- **Execution phase:**
  - A new function invocation request is dispatched when a trigger rule fires.
  - **The request is routed to a "sandbox" (e.g., a container or a virtual machine) hosting the target function.** The target sandbox can be created on the fly or reused from a previous execution.
  - The management of sandboxes (creation, destruction, up/down scaling of function instances) and the routing of requests are fully and automatically handled by the FaaS infrastructure.

FaaS – Trigger rules

- The trigger rules for a function invocation request can be based on various events: various types and origins (internal or external to the cloud platform).
- Multiple triggers can be registered for the same function.

- **Some examples of trigger events:**
  - HTTP request to a given URL
    - For example, an external request to “API gateway” (i.e., the external entry point/façade of a Web application/service)
  - Arrival of a message in a message queue
  - Modification in an object store (e.g., creation of a new object within a bucket)
  - Modification in a database table
  - Completion of a previous function invocation (workflow / state machine)

FaaS versus IaaS – A detailed comparison (1/2)
For programmers

Characteristics	AWS Lambda (FaaS)	AWS EC2 (IaaS, on-demand instances)
When the program is run	On event selected by user (programmer)	Continuously until explicitly stopped
Programming language	Any but mostly high-level (e.g., Python)	Any
Libraries managed by	Provider (often) or user/programmer	User (programmer)
Program state	Kept in storage (stateless)	Anywhere (stateful or stateless)
Max. memory size	0.125 to 3 GiB **(2022: up to 10 GiB)**	0.5 to 1952 GiB **(2022: up to 24 TiB)**
Max. local storage	0.5 GiB **(2022: up to 10 GiB)**	0 to 3600 GiB **(2022: up to 60 TiB)**
Max. running time	15 minutes	None
Minimum accounting unit	0.1 second **(2022: 1 millisecond)**	60 seconds **(2022: 60s then 1s)**
Price per accounting unit	$0.0000002 (assuming 0.125 GiB)	$0.0000867 to $0.4080000

(Table adapted from the following source: Jonas et al. “Cloud programming simplified: A Berkeley view on Serverless Computing.” Note that the default AWS numbers are circa January 2019).

FaaS versus IaaS – A detailed comparison (2/2)
For system administrators

Characteristics	AWS Lambda (FaaS)	AWS EC2 (IaaS, on-demand instances)
Server instance type chosen by	Provider	User (sysadmin)
Autoscaling managed by	Provider	User (sysadmin)
Deployment management by	Provider	User (sysadmin)
Fault tolerance managed by	Provider	User (sysadmin)
Monitoring managed by	Provider	User (sysadmin)
Logging managed by	Provider	User (sysadmin)

(Table adapted from the following source: Jonas et al. “Cloud programming simplified: A Berkeley view on Serverless Computing.”)

FaaS – Code packaging

- The packaging/build of functions and their dependencies is typically automated, if the application programmers use one of the standard environment supported by the provider (e.g., common Python or Javascript environments).

- Programmers can also build and ship custom runtimes and/or (black-box) binary code.

- As of mid-2019, a vast majority of functions are written in Javascript (Node.js) or Python.

FaaS – Invocation model

- A function invocation can be either synchronous or asynchronous.
- **Synchronous invocation:**
  - Client waits for completion of the invocation
  - In case of failure/timeout: typically, **no automatic retry** by the FaaS platform
- **Asynchronous invocation:**
  - Client does not wait for the completion of the invocation, and can query status/result later
  - In case of failure/timeout: typically, **automatic retry** of the invocation by the FaaS platform

- Some types of trigger events have restrictions
  - Invocations triggered by message queues or storage events are asynchronous
  - HTTP-based invocations can be synchronous or asynchronous

FaaS – Execution model

- **In most designs, a given sandbox:**
  - **Only hosts a single function**
  - **Only processes a single request at a time**

- The FaaS infrastructure decides when to scale the number of sandboxes for a given function, up & down
  - **The number of sandboxes for a function may scale down to zero / up from zero**
  - Upscaling is limited by a “concurrency limit” threshold (user defined)
  - The user (cloud tenant) is only billed for the time spent by sandboxes handling function invocations, not the idle time of the sandboxes

- The above characteristics may impact the latency of the FaaS platforms for handling a function invocation request.
  - “Cold starts” versus “Warm starts”
  - Various strategies exist for reducing the performance impact of cold starts (or avoiding them).

FaaS – Workflows

- Many FaaS platforms provide support for functions workflows, allowing **to create more elaborate tasks/applications by creating sequential and/or parallel chains of functions.**
- FaaS workflows are typically:
  - Built as a higher-lever feature on top of a standard FaaS platform
  - Based on a state automata orchestrating the transitions between steps

- **Example scenario 1:** Registration of a new user in a Web application (account set up, sending of a confirmation email, etc.)
- **Example scenario 2:** Fork-join parallelism

- E.g., AWS step functions, Azure Durable functions, IBM Cloud Functions Composer

FaaS – Current limitations (1/4)

- We have already mentioned several use cases that are well adapted to
  - Web APIs
  - Simple “FaaS ETL” event-driven workloads bridging
  - Enterprise workflows
  - Coarse-grained embarrassingly parallel computations

- However, current FaaS offerings suffer from several important limitations that make them suboptimal, disappointing or even impractical for many workloads and use cases.

- In the next cases, we will discuss some of these limitations.

FaaS – Current limitations (2/4)

- **Slow storage**
  - The stateless nature of FaaS forces it to heavily rely on external storage services (e.g., object storage like AWS S3), which have poor latencies (≥ 10 ms for small objects) and possible high costs for high-throughput configurations.
  - Throughput may also be a problem (poorer than a single local SSD) and worsen if many network-intensive functions are co-located.

- **Lack of fine-grained coordination between functions**
  - Current FaaS platforms provide no means for fast/simple/serverless notifications between tasks (which may be running on different machines).
  - At best, application designer must provide their own solutions, typically deployed on long-running virtual machines.
  - More generally, functions are not network addressable. They can initiate outbound network connections but cannot receive inbound connection requests or messages.
  - This exacerbates the lack of locality (no client stickiness) and the negative performance impact of slow storage.

FaaS – Current limitations (3/4)

- **Poor performance for popular communication patterns**
  - Typical communication patterns used in parallel/distributed applications (e.g., broadcast, aggregation, shuffle) have much lower performance than with virtual machines.
  - Indeed, FaaS applications cannot control the placement of tasks and therefore cannot leverage traditional optimizations based on hierarchical communications. Consequently, FaaS based applications generate more network messages.

- **Lack of performance predictability**
  - Warm vs. cold starts
  - Hardware and workload heterogeneity

FaaS – Current limitations (4/4)

- **No (or very limited) support for diverse resource requirements**
  - Users have very limited means to express specific hardware/resource requirements. The only configuration options are generally the number of CPUs and the RAM capacity (with limited options, and often in a coupled way).
  - No support for hardware accelerators
- **Limited lifetime**
  - A given task can run at most for 15 minutes.
  - There is no guaranteed way to persist state locally.
- **Risks of vendor/provider lock-in**
  - There are currently no standardized APIs and features.
  - Many functions as acting as glue between services of a given provider.
  - Possibility for users to deploy their own FaaS infrastructure (on top of IaaS/HaaS) but may be complex and costly.

Storage characteristics for FaaS: current vs. ideal

	Block storage	Object storage	File system	Elastic database	Memory store	"Ideal" storage service for FaaS
Cloud function access	No	Yes	Yes	Yes	Yes	Yes
Transparent provisioning	No	Yes	Capacity only (not IOPS)	Yes	No	Yes
Availability and persistence guarantees	Local & persistent	Distributed & persistent	Distributed & persistent	Distributed & persistent	Local & ephemeral	Various
Latency (mean)	< 1 ms	10 – 20 ms	4 – 10 ms	8 – 15 ms	< 1 ms	< 1 ms
Storage capacity (1 GB / month)	$0.10	$0.023	$0.30	$0.18 – $0.25	$1.87	~$0.10
Throughput (1 MB/s for 1 month)	$0.03	$0.0071	$6.00	$3.15 – $255.1	$0.96	~$0.03
IOPS (1/s for 1 month)	$0.03	$7.1	$0.23	$1 – $3.15	$0.037	~$0.03

Cost

(Table adapted from the following source: Jonas et al. "Cloud programming simplified: A Berkeley view on Serverless Computing." Note that the AWS numbers are circa January 2019).

FaaS platform case studies

We will study two examples of open-source FaaS platforms:

- Apache OpenWhisk
- CNCF Knative

Case study: Apache OpenWhisk (1/2)

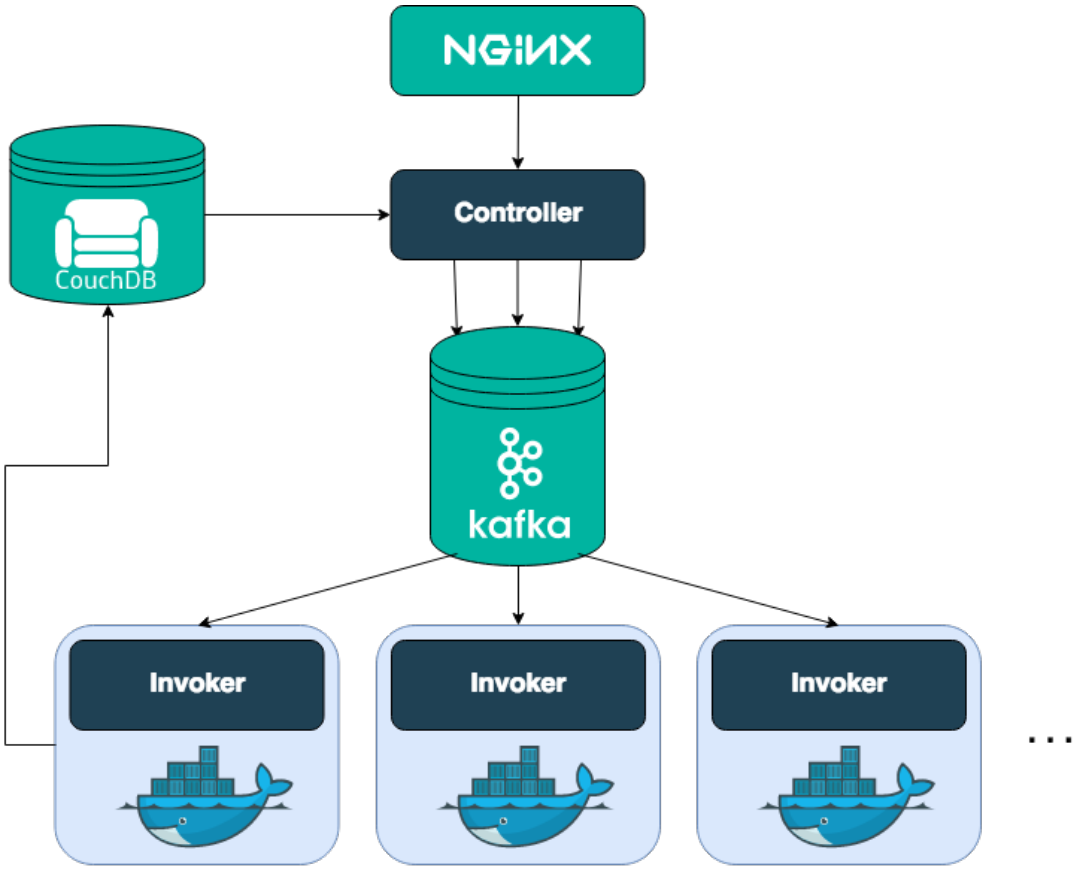
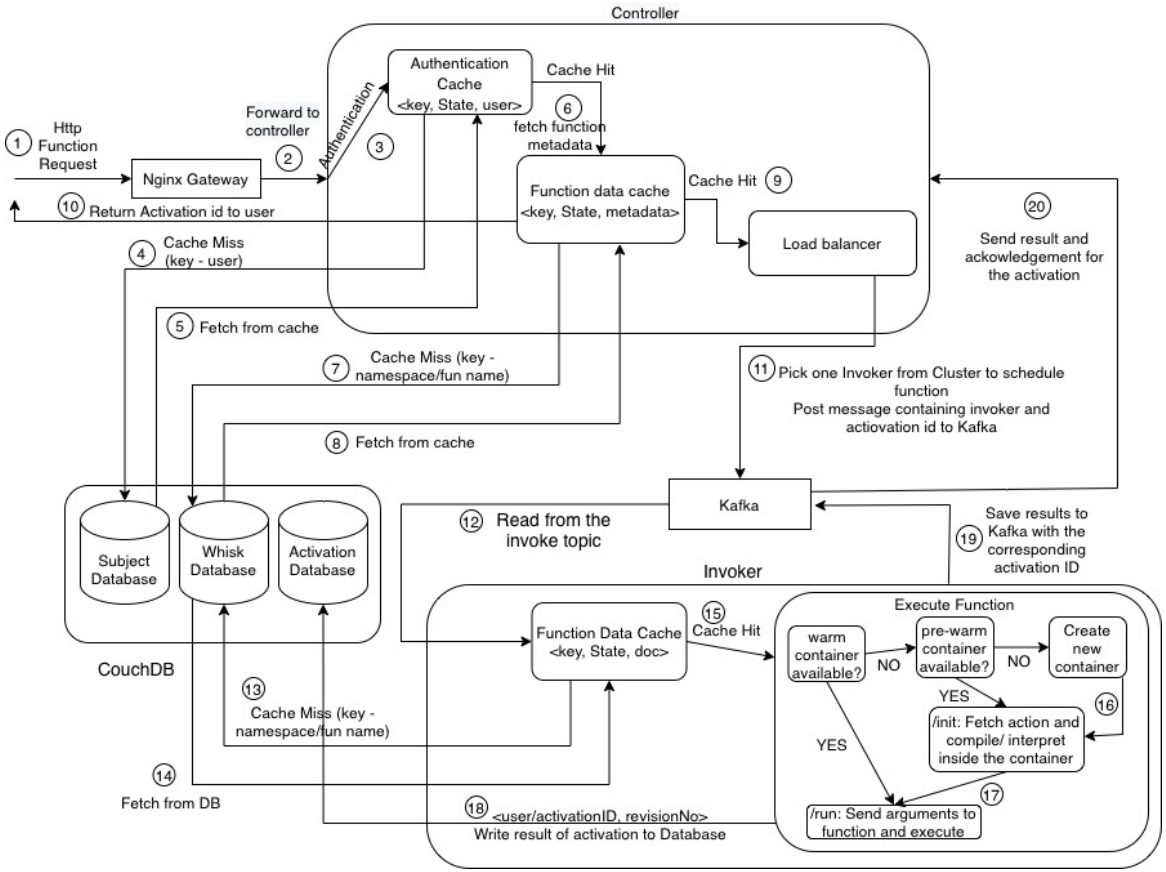


Image source & more details about the different steps:
https://github.com/apache/openwhisk/blob/master/docs/about.md#how-openWhisk-works)

Case study: Apache OpenWhisk (2/2)



(source: https://github.com/apache/openwhisk)

Case study: Knative (1/9)

- An open-source project initially created by Google and now managed by the CNCF (Cloud Native Computing Foundation).
- **Based on Kubernetes, with additional components and customization.**
- Knative is a framework aimed at simplifying the design and hosting of serverless/FaaS applications.
- Knative includes two main parts:
  - **Knative Serving**: aimed at deploying/upgrading/scaling functions, and routing function invocations.
  - **Knative Eventing**: aimed at interconnecting heterogeneous systems through an event-driven architecture (routing events between events producers and consumers, using brokers and triggers).
- Here, we focus on Knative Serving.

Case study: Knative (2/9)

Knative Serving – Main concepts:

- **"Configuration"**:
  - A Configuration describes the desired state of a deployed system (including code + setup information): chosen container image, environment variables, etc.
  - Knative automatically converts this high-level statement into low-level Kubernetes concepts such as "deployments".

- **"Revision"**: an immutable snapshot of a Configuration.
  - Modifying a Configuration creates a new Revision.
  - Multiple revisions of the same application can coexist.

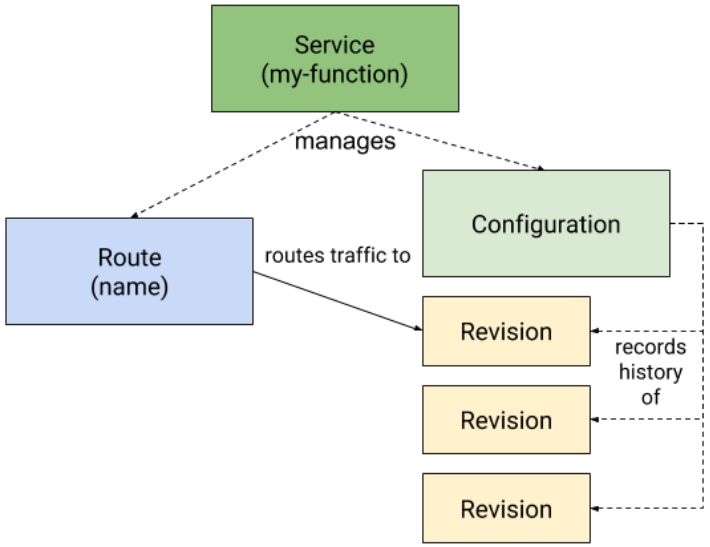
- **"Route"**: maps a network endpoint to one or more Revisions.

- **"Service"**: encapsulates a Route and a Configuration.

Case study: Knative (3/9)

Knative Serving – Main concepts – Wrap up:

- A **Route** provides a named endpoint and a mechanism for routing traffic to …
- **Revisions**, which are immutable snapshots of code + config, created by a …
- **Configuration**, which acts as a stream of environments for Revisions.
- A **Service** acts as a top-level container for managing a Route and Configuration which implement a network service.



Source for text and figure: Knative documentation
https://github.com/knative/specs/blob/main/specs/serving/overview.md

Case study: Knative (4/9)

Knative Serving – Main components:

- **Serving controller**
  - Encapsulates several sub-components named **“reconcilers”**
  - Each reconciler is a feedback controller dedicated to a specific aspect: services, routes, configurations, revisions, labels, …

- **Networking controllers**
  - Configure TLS/SSL certificates
  - Configure HTTP Ingress routing

Case study: Knative (5/9)

Knative Serving – Main components (continued) :

- **Webhook**
  - Intercepts, validates and enriches the records (services, routes, configurations) submitted by the user.

  - Among other things, this component is in charge of:
    - Checking the validity of the submitted config parameters
    - Setting up default values for implicit parameters
    - Injecting routing and networking configuration into Kubernetes

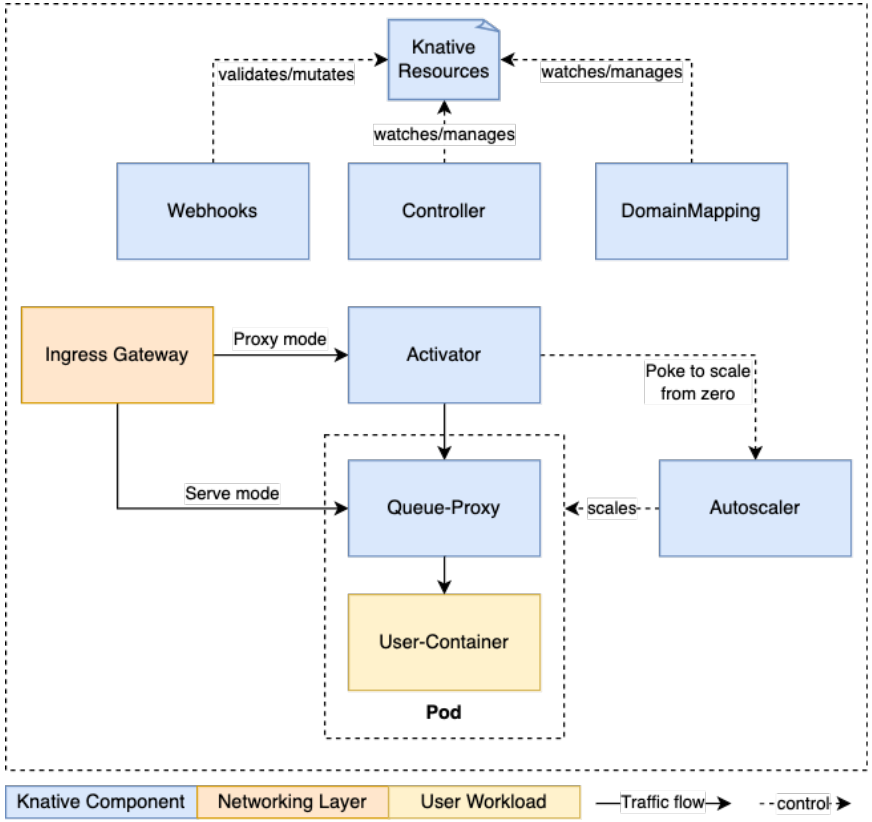
Case study: Knative (6/9)

Knative Serving – Main components (continued):

- **Autoscaler + Activator + Queue Proxy**
  - These three components interact to **react to input load/traffic changes** (i.e., significant increases or decreases of the number of client invocation requests).
  - The **(pod) Autoscaler** monitors the current level of traffic and adjusts the number of pod instances deployed for the service, possibly down to zero.
  - The **Queue Proxy** is deployed as a sidecar in every (pod) instance of the service.
    - It keeps tracks of the requests assigned to a given pod instance, and their completion.
    - It buffers and throttles the incoming requests to enforce the per-pod concurrency limit set by the application.
    - It regularly sends metrics (e.g., queue depth) to the Autoscaler.
  - The **Activator** receives and buffers the incoming traffic when there is no pod instance, until one or several instances are created by the Autoscaler.

Case study: Knative (7/9)

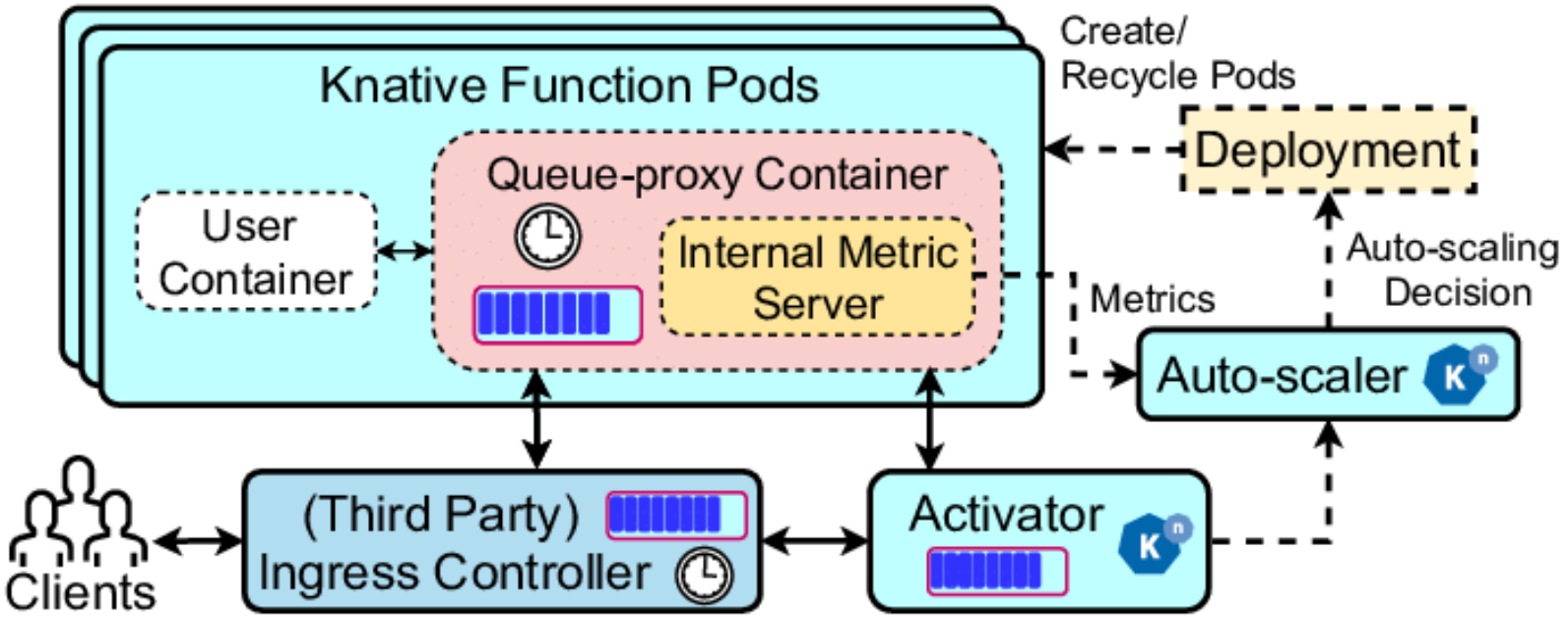
Knative Serving – Main components – Wrap up:



(Source: https://knative.dev/docs/serving/architecture/)

Case study: Knative (8/9)

Knative Serving – Autoscaling – Wrap up:



Source: J. Li et al. Analyzing Open-Source Serverless Platforms: Characteristics and Performance. 2021.

Case study: Knative (9/9)

For more details:

- Knative Serving documentation: https://knative.dev/docs/serving/

- Book: “Knative in action” by Jacques Chester

  - Free chapter: “Introducing Knative serving”: https://freecontent.manning.com/introducing-knative-serving/

  - Full ebook freely available from Vmware: https://tanzu.vmware.com/content/ebooks/knative-in-action