上海交通大学
SHANGHAI JIAO TONG UNIVERSITY

SJTU

Lecture 4 Introduction to Serverless Computing

马汝辉

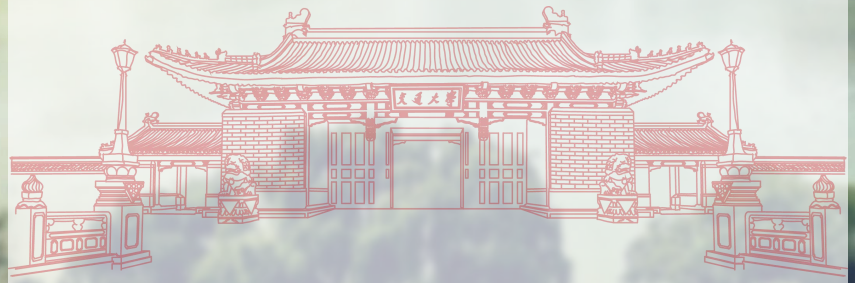
上海交通大学

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

目录

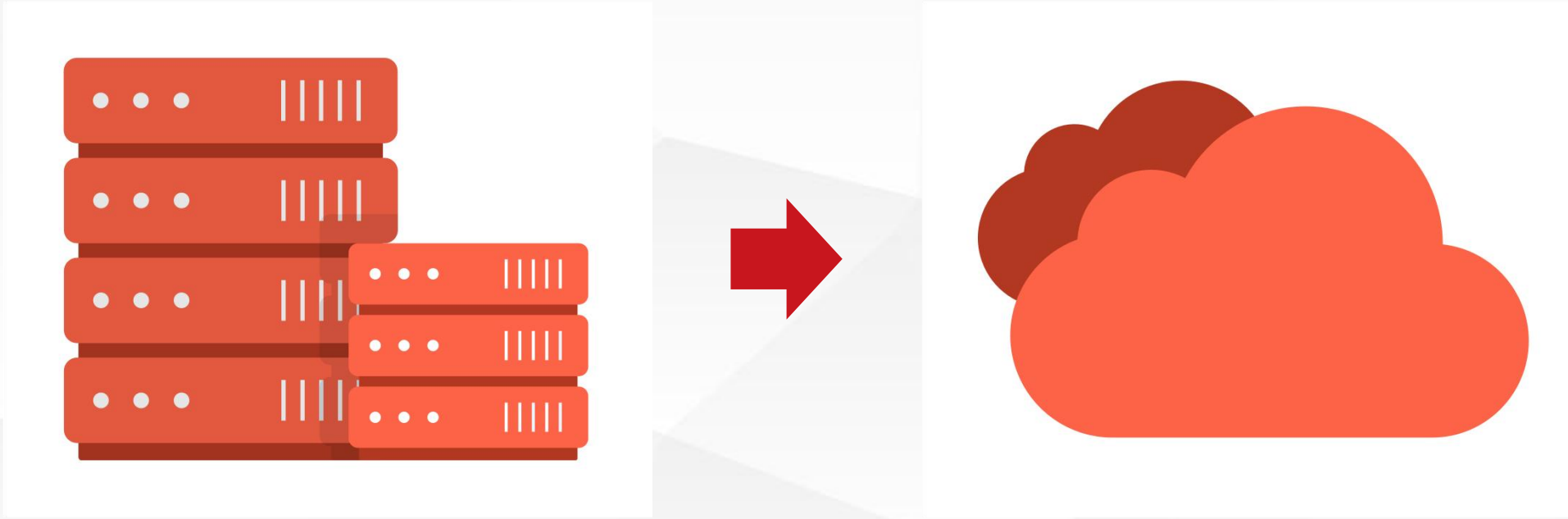
01

The Evolution of Cloud



The Tradition: On-Premise

- On-premise entails an IT infrastructure where all hardware, software, and data storage are managed **onsite**. It ensures complete control and is ideal for businesses to prioritize security and direct oversight of their systems.
- Cloud computing enables on-demand access to IT services on the **Internet**. Public clouds are third-party-owned, delivering resources online.



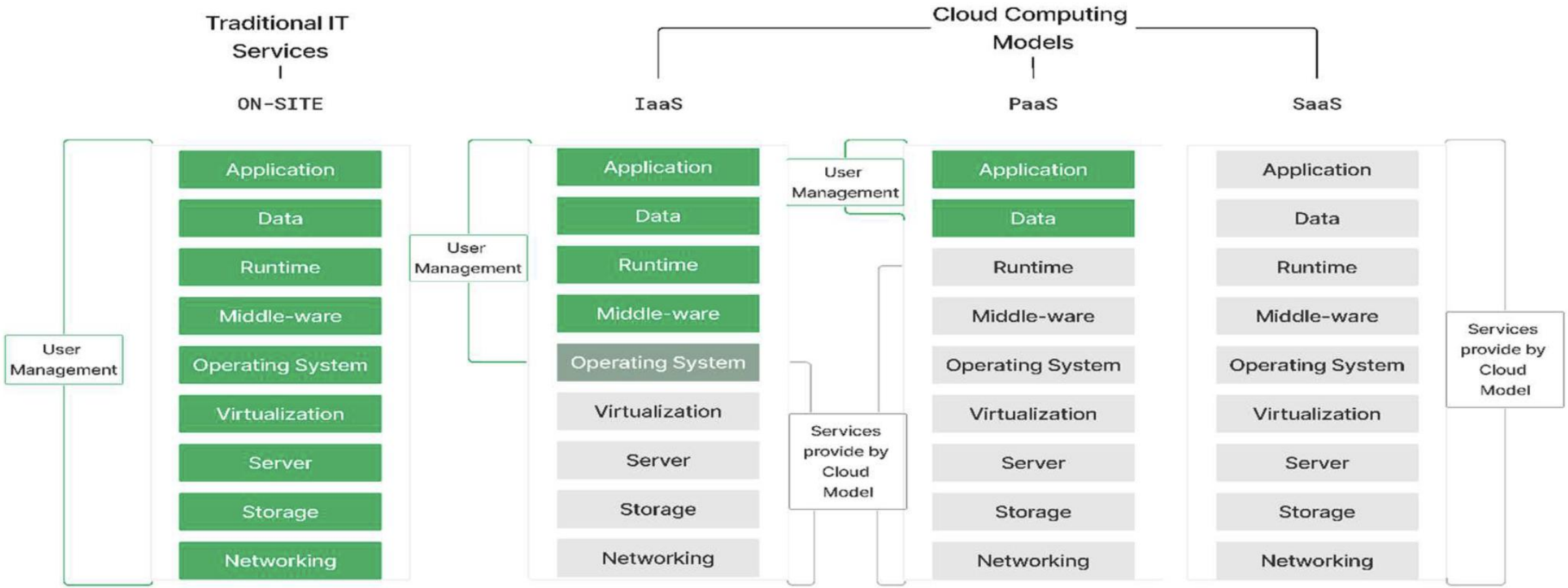
https://www.cloudpanel.io/blog/on-premises-vs-cloud-computing/

The Tradition: On-Premise

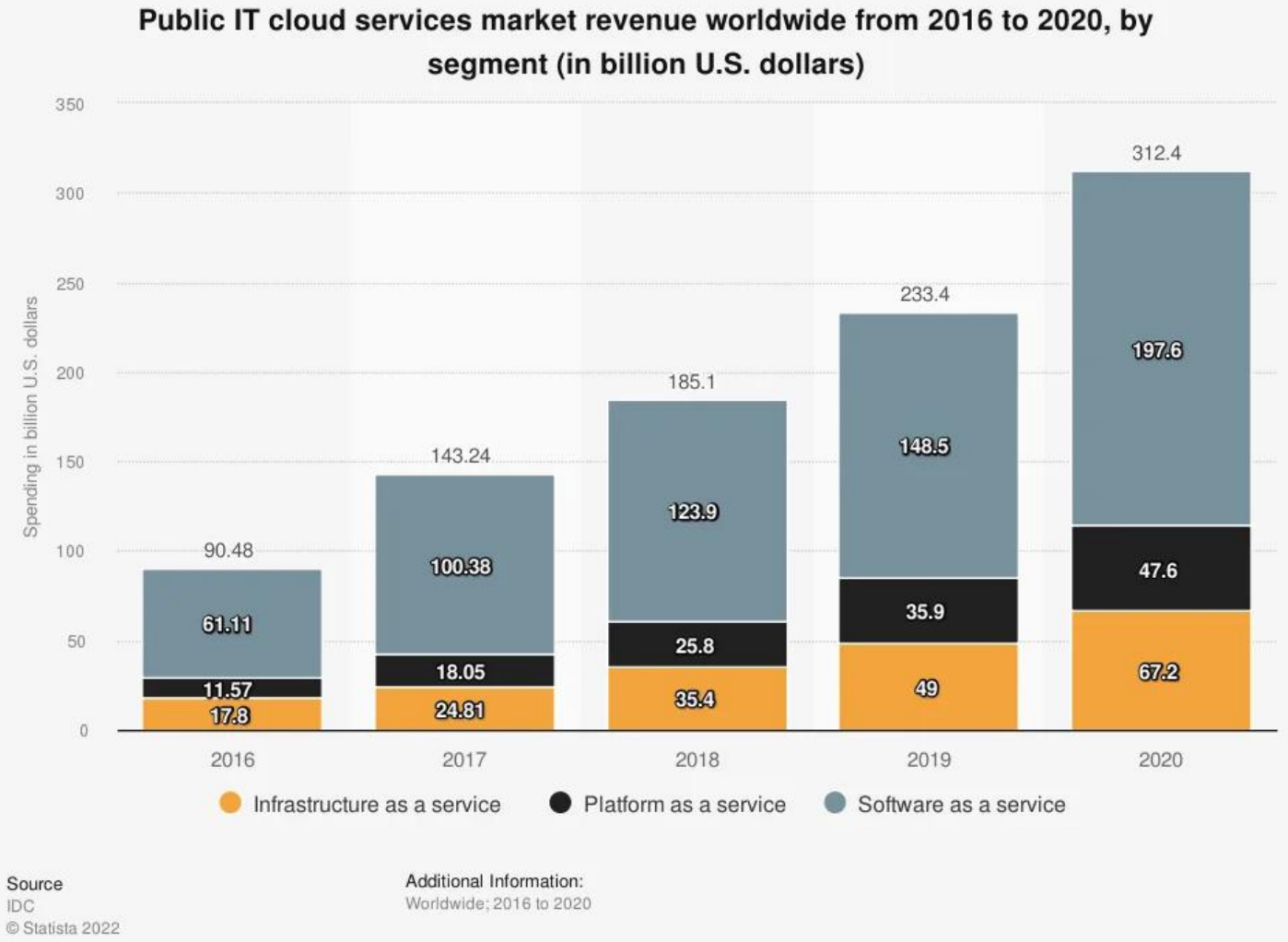
- Drawbacks of On-Premises Computing:
  - High initial **costs** and capital **expenses**: Setting up a system on-premises requires a lot of money upfront. Organizations must spend money on tools, networking equipment, and other systems parts.
  - Limited **scale** and the possibility of **overprovisioning**: On-premises setups may have trouble scaling significantly when resource demand increases quickly. Organizations must correctly predict what they will need in the future.
  - IT **management and maintenance** complexities: On-premises equipment must be managed and maintained constantly. This means keeping the hardware in excellent condition, updating the software, adding security patches, and fixing problems. Organizations need skilled IT staff to handle these jobs, which can add to business costs and resource use.

https://www.hpe.com/emea_europe/en/what-is/on-premises-vs-cloud.html

On-Premise → Cloud Computing



IaaS, PaaS, and SaaS



https://www.bigcommerce.com/articles/ecommerce/saas-vs-paas-vs-iaas/

Infrastructure as a Service (IaaS)

- IaaS, or infrastructure as a service, is on-demand access to cloud-hosted physical and virtual servers, storage and networking—the **backend IT infrastructure** for running applications and workloads in the cloud.
  - The difference is that the cloud service provider **hosts, manages and maintains** the hardware and computing resources in its own data centers.
  - IaaS customers use the hardware via an **internet connection**, and pay for that use on a **subscription** basis.
  - Typically IaaS customers can choose between **virtual machines (VMs)** hosted on shared physical hardware (the cloud service provider manages virtualization) or **bare metal servers** on dedicated (unshared) physical hardware.
  - Customers can **provision, configure and operate** the servers and infrastructure resources.

https://www.ibm.com/think/topics/iaas-paas-saas

Infrastructure as a Service (IaaS)

Benefits of IaaS include:

- **Higher availability**: With IaaS a company can create redundant servers easily, and even create them in other geographies to ensure availability during local power outages or physical disasters.
- **Lower latency, improved performance**: Because IaaS providers typically operate data centers in multiple geographies, IaaS customers can locate apps and services closer to users to minimize latency and maximize performance.
- **Improved responsiveness**: Customers can provision resources in a matter of minutes, test new ideas quickly and quickly roll out new ideas to more users.
- **Faster access to best-of-breed technology**: Cloud providers compete with each other by providing the latest technologies to their users, IaaS customers can take advantage of these technologies much earlier (and at far less cost) than they can implement them on premises.

https://www.ibm.com/think/topics/iaas-paas-saas

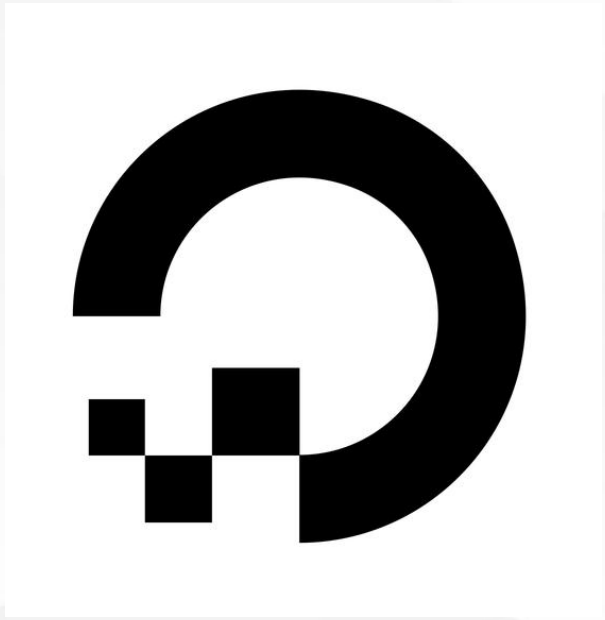
Infrastructure as a Service (IaaS)

- Popular examples of IaaS:

AWS EC2

rackspace®

Google Compute Engine (GCE)



Digital Ocean

Platform as a Service (PaaS)

- PaaS, or platform as a service, is on-demand access to a complete, ready-to-use, cloud-hosted **platform** for developing, running, maintaining and managing applications.
  - The cloud services provider hosts, manages, and maintains **all the hardware and software** included in the platform—servers (for development, testing and deployment), operating system (OS) software, storage, networking, databases, middleware, runtimes, frameworks, development tools—as well as related services for security, operating system and software upgrades, backups and more.
  - Users access the PaaS through a graphical user interface (GUI), where development or DevOps teams can collaborate on all their work across the entire application lifecycle including coding, integration, testing, delivery, deployment and feedback.

https://www.ibm.com/think/topics/iaas-paas-saas

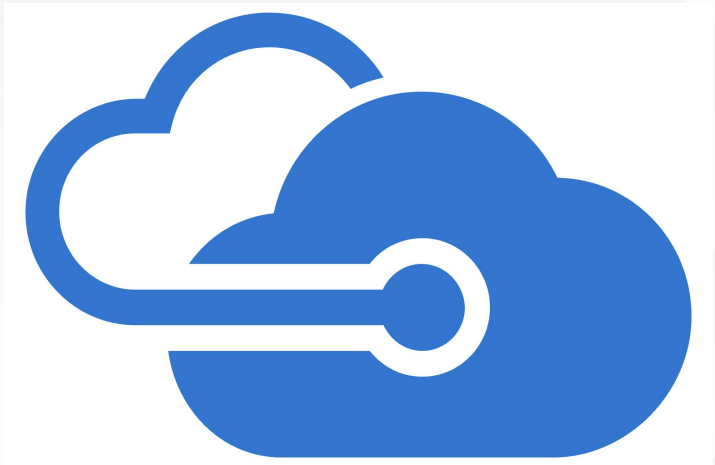
Platform as a Service (PaaS)

- The primary benefit of PaaS is that it allows customers to build, test, deploy run, update and scale applications more quickly and cost-effectively than they might if they had to build out and manage their own on-premises platform. Other benefits include:
  - **Faster time to market**: PaaS enables development teams to spin-up development, testing and production environments in minutes, rather than weeks or months.
  - **Low- to no-risk testing and adoption of new technologies**: PaaS platforms typically include access to a wide range of the latest resources up and down the application stack.
  - **Simplified collaboration**: As a cloud-based service, PaaS provides a shared software development environment, giving development and operations teams access to all the tools they need, from anywhere with an Internet connection.
  - **A more scalable approach**: With PaaS, organizations can purchase extra capacity for building, testing, staging and running applications whenever they need it.

https://www.ibm.com/think/topics/iaas-paas-saas

Platform as a Service (PaaS)

- Popular examples of PaaS:



Windows Azure

AWS Elastic Beanstalk

Google APP Engine

Software as a Service (SaaS)

- SaaS, or software as a service, is on-demand access to ready-to-use, cloud-hosted application software.
  - Users pay a **monthly or annual fee** to use a complete application from within a web browser, desktop client or mobile app. The application and all of the infrastructure required to deliver it—servers, storage, networking, middleware, application software, data storage—are hosted and managed by the SaaS vendor.
  - Typically, the vendor ensures a level of availability, performance and security as part of a **service level agreement (SLA)**. Customers can add more users and data storage on demand at additional cost.

https://www.ibm.com/think/topics/iaas-paas-saas

Software as a Service (SaaS)

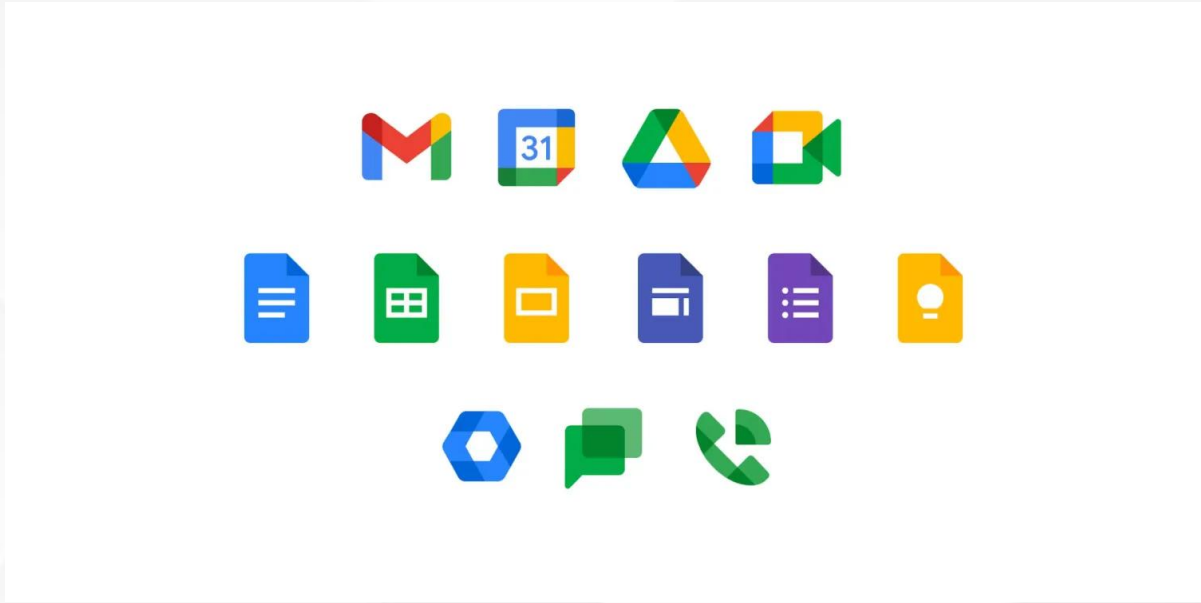
- The main benefit of SaaS is that it offloads all infrastructure and application management to the SaaS vendor.
- All the user has to do is create an account, pay the fee and start using the application. The vendor handles everything else, from maintaining the server hardware and software to managing user access and security, storing and managing data, implementing upgrades and patches and more.

https://www.ibm.com/think/topics/iaas-paas-saas

Serverless Computing

- Popular examples of SaaS:

BigCommerce



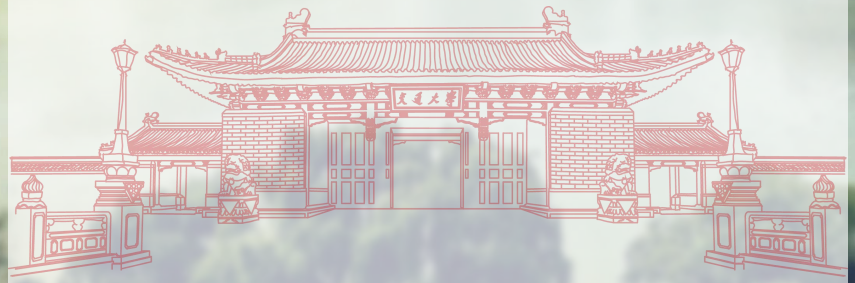
Google Workspace

Serverless Computing

- Serverless takes the abstraction offered by PaaS to the next level, particularly with Function as a Service (FaaS).
  - While PaaS provides a platform for you to deploy your entire application (or microservices), you often still think in terms of constantly running servers or instances that you **configure for scaling**. With PaaS, you might define how many instances of your application to run, and you still pay for that allocated capacity, even if it‘s idle.
  - Serverless (specifically FaaS) pushes this further by abstracting away the **runtime environment** itself and focusing purely on the execution of individual functions triggered by events.

02

What is Serverless?



What is Serverless?

- Serverless **does not mean** "no servers." The name notwithstanding, servers in serverless computing are managed by a cloud service provider (CSP).
- Developers can focus on writing the best **front-end application code and business logic** with serverless computing.
- The cloud provider **handles the rest**—provisioning the cloud infrastructure required to run the code and scaling the infrastructure up and down on demand as needed.
- Moreover, developers never pay for **idle capacity** with serverless. The cloud provider spins up and provisions the required computing resources on demand when the code executes and spins them back down again—called **"scaling to zero"**—when execution stops.

https://www.ibm.com/think/topics/serverless

The Origins of Serverless

- Serverless originated in 2008 when Google released **Google App Engine (GAE)**, a platform for developing and hosting web applications in Google-managed data centers.
- In 2014, Amazon introduced **AWS Lambda**, the first serverless platform. Named after functions from lambda calculus and programming, AWS Lambda, a FaaS model, helped the serverless computing framework gain mass-market appeal and rapid adoption among software developers.
- In 2016, **Microsoft Azure Functions** and **Google Cloud Functions** launched their serverless platforms.

https://www.ibm.com/think/topics/serverless

The Serverless Ecosystem

Serverless and FaaS

- Serverless is more than function as a service (FaaS)—the cloud computing service that enables developers to run code or containers **in response to specific events or requests** without specifying or managing the infrastructure required to run the code.
- FaaS is the compute model central to serverless, and the two terms are often used interchangeably. Compared to FaaS, serverless is an **entire stack of services** that can respond to specific events or requests and scale to zero when no longer in use—and for which provisioning, management and billing are handled by the cloud provider and invisible to developers.
- In addition to FaaS, these services include **databases and storage**, **Application programming interface (API) gateways** and **event-driven architecture**.

https://www.ibm.com/think/topics/serverless

The Serverless Ecosystem

Serverless databases and storage

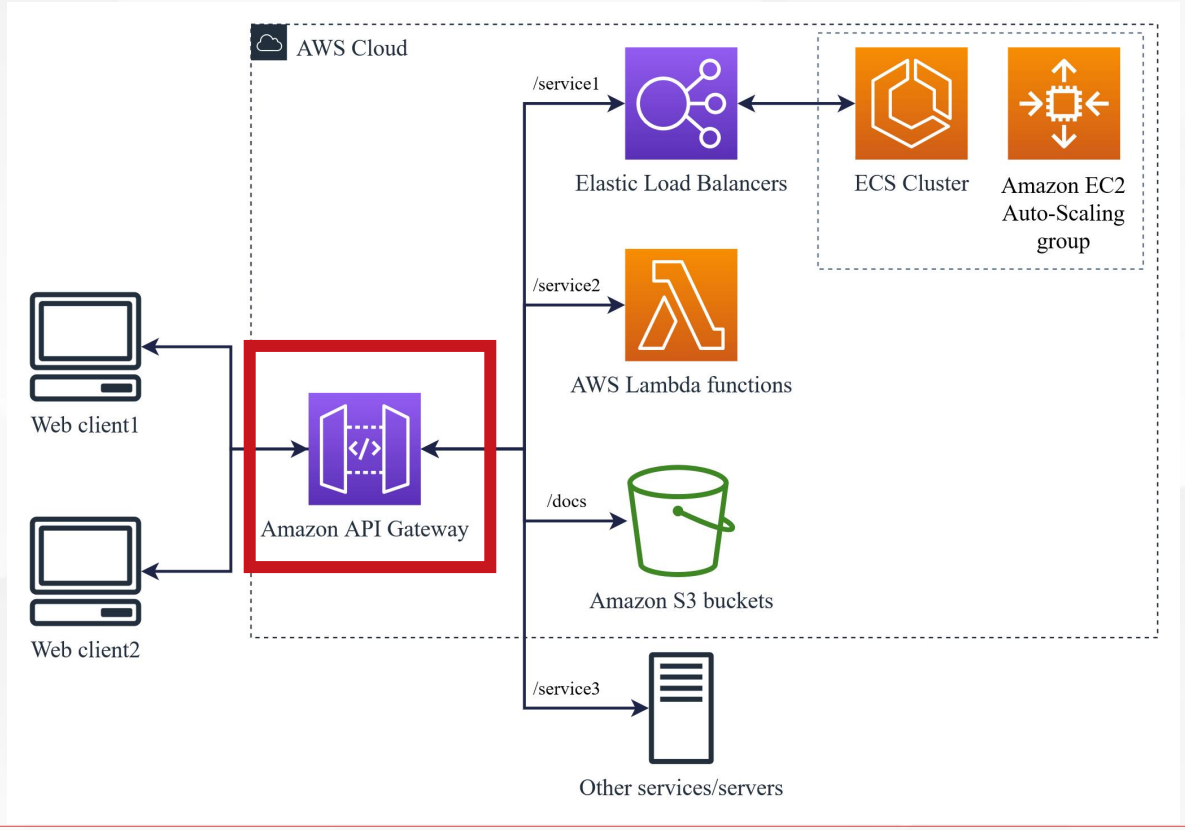
- Databases (SQL and NoSQL) and storage (particularly object storage) are the foundation of the **data layer**.
- A serverless approach to these technologies involves transitioning away from provisioning “instances” with defined capacity, connection and query limits and moving toward models that **scale linearly with demand** in both infrastructure and pricing.

https://www.ibm.com/think/topics/serverless

The Serverless Ecosystem

API gateways

- API gateways act as **proxies** to web application actions and provide HTTP method routing, client ID and secrets, rate limits, CORS, viewing API usage, viewing response logs and API sharing policies.



https://www.ibm.com/think/topics/serverless

The Serverless Ecosystem

Event-driven architecture (EDA)

- Serverless architectures work well for event-driven and stream-processing workloads, most notably the open-source **Apache Kafka** event streaming platform.
- Automated serverless functions are stateless and designed to handle individual events. These functions have become an essential part of event-driven architecture (EDA)—a software design model built around the publication, capture, processing and storage of events.
- In an EDA framework, **event producers** (for example, microservices, APIs, IoT devices) send real-time event notifications to **event consumers**, activating specific processing routines.

https://www.ibm.com/think/topics/serverless

The Serverless Ecosystem

Containers, Kubernetes and Knative

- Serverless applications are often deployed in **containers**.
- Kubernetes is an open-source **container orchestration platform** that automates container **deployment, management and scaling**. This automation dramatically simplifies the development of containerized applications.
- Knative is an open-source extension to Kubernetes that provides a **serverless framework**. Knative works by abstracting away the code and handling the network routing, event triggers and autoscaling for serverless execution.

https://www.ibm.com/think/topics/serverless

Pros and Cons of Serverless

Pros

- Improved developer productivity
- Pay for execution only
- Develop in any language
- Streamlined development or DevOps cycles
- Cost-effective performance
- Reduce latency
- Usage visibility

Cons

- Less control
- Vendor lock-in
- Slow startup (aka, cloud start)
- Complex testing and debugging
- Higher cost for running long applications

https://www.ibm.com/think/topics/serverless

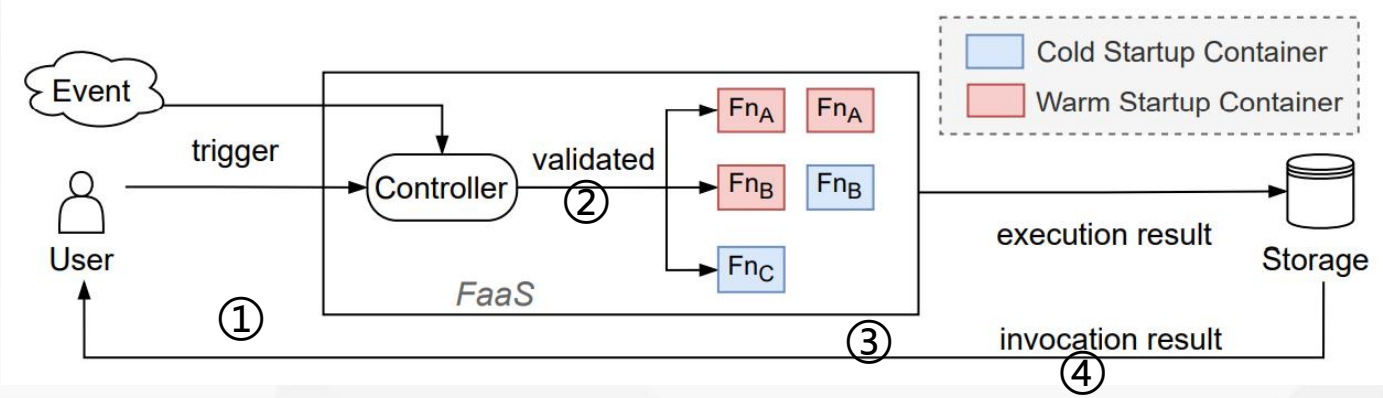
Serverless Applications

①The serverless system receives triggered API queries from the users.

②The controller validates them, and invokes the functions by creating new sandboxes (aka the *cold startup*) or reusing running warm ones (aka the *warm startup*).

③Each function invocation runs in an individual container or a virtual machine. The serverless system can scale them horizontally according to the actual application workload.

④ Each execution worker accesses a backend database to save execution results.



A simple serverless application model

By further configuring triggers and bridging interactions, users can customize the execution for complex applications:

- web applications
- real-time data processing
- AI reasoning
- video transcoding
- ...

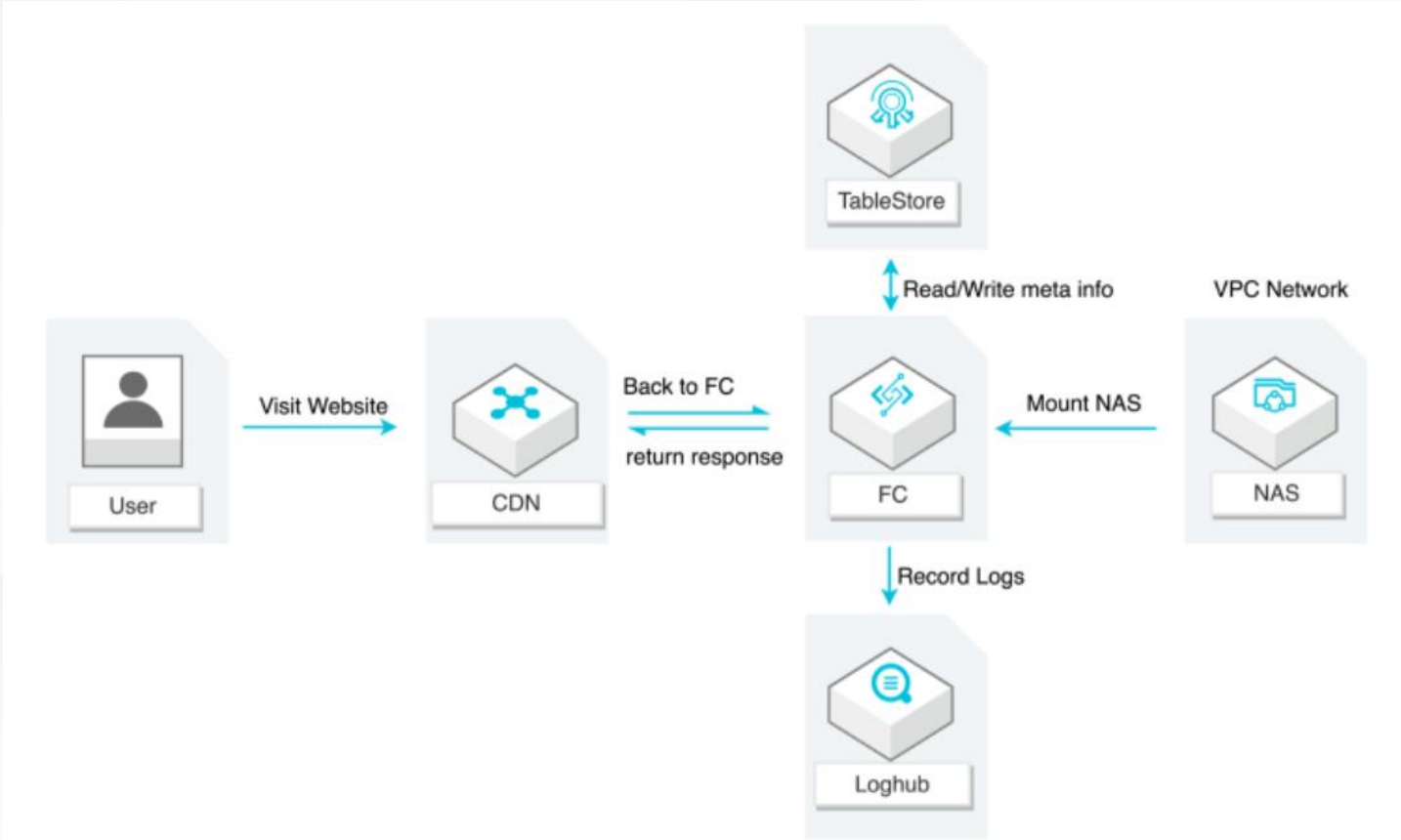
Serverless Applications

Case 1: Web Applications

O&M-free Function Compute allows frontend engineers to build cloud-native web applications by writing business code, effectively improving the publication and iteration efficiency and reduces O&M costs.

Benifits:

- Free from O&M operations and build applications more efficiently
- Elastically handle load peaks and valleys with high availability features
- Provide cost-effective and high-performance services
- Smoothly migrate traditional applications to function compute



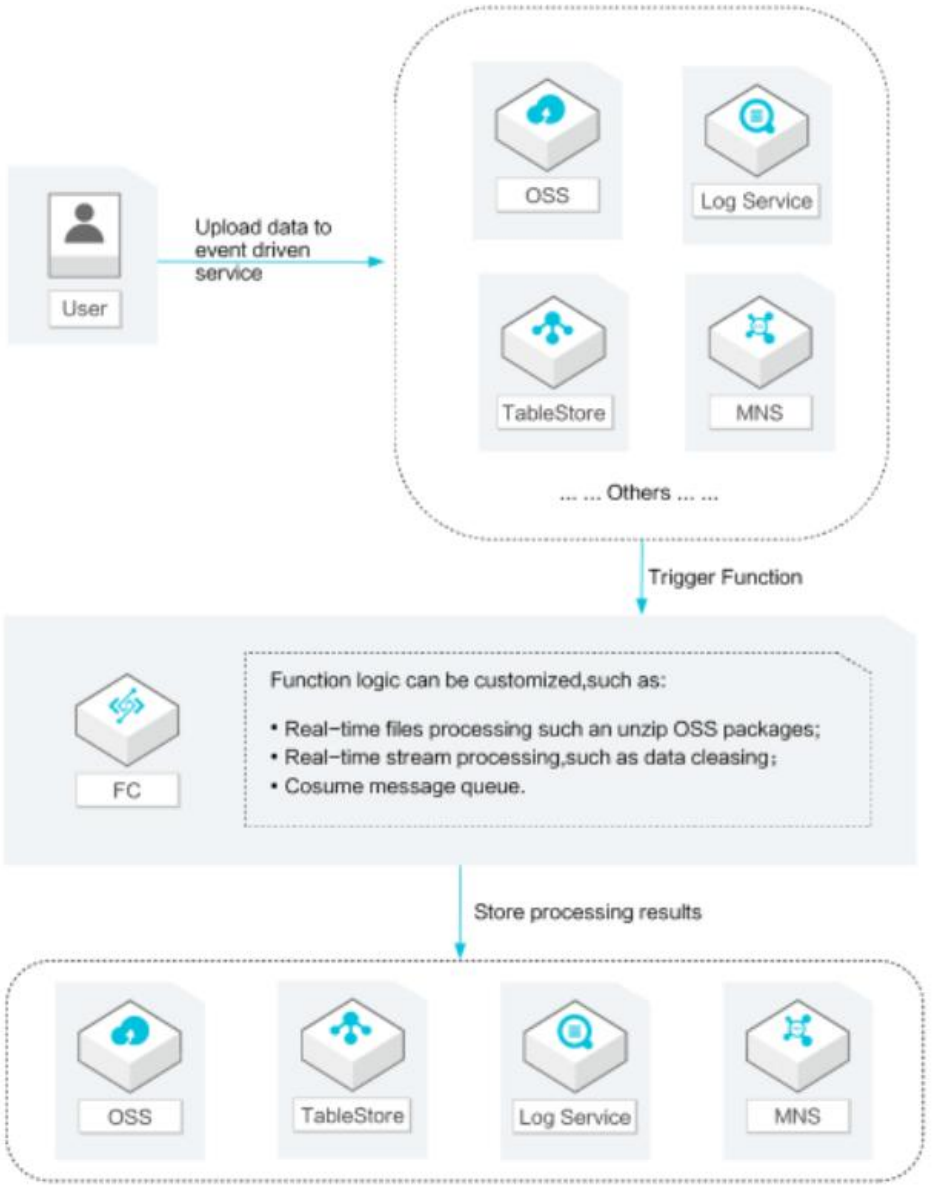
Serverless Applications

Case 2: Real-time Data Processing

Function Compute provides multiple event sources. The event triggering mechanism can process data in real time with just a few lines of code and simple configurations. For example, the mechanism can decompress OSS packages, cleanse logs generated by Log Service or Tablestore data, and customize consumption of MNS messages.

Benifits:

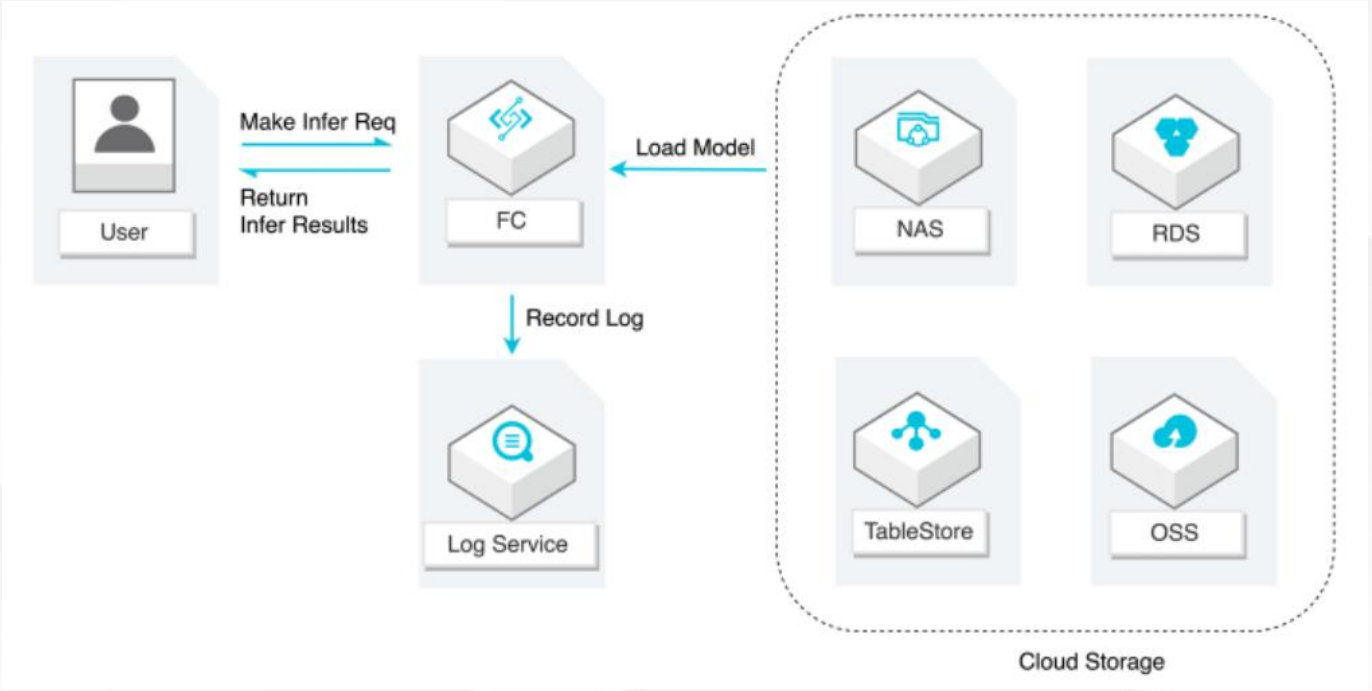
- Integrate multiple easy-to-configure event sources
- Flexibly customize processing logic



Serverless Applications

Case 3: AI Reasoning

O&M-free and elastically scalable Function Compute allows algorithm engineers to convert trained models into elastic and highly available reasoning services.



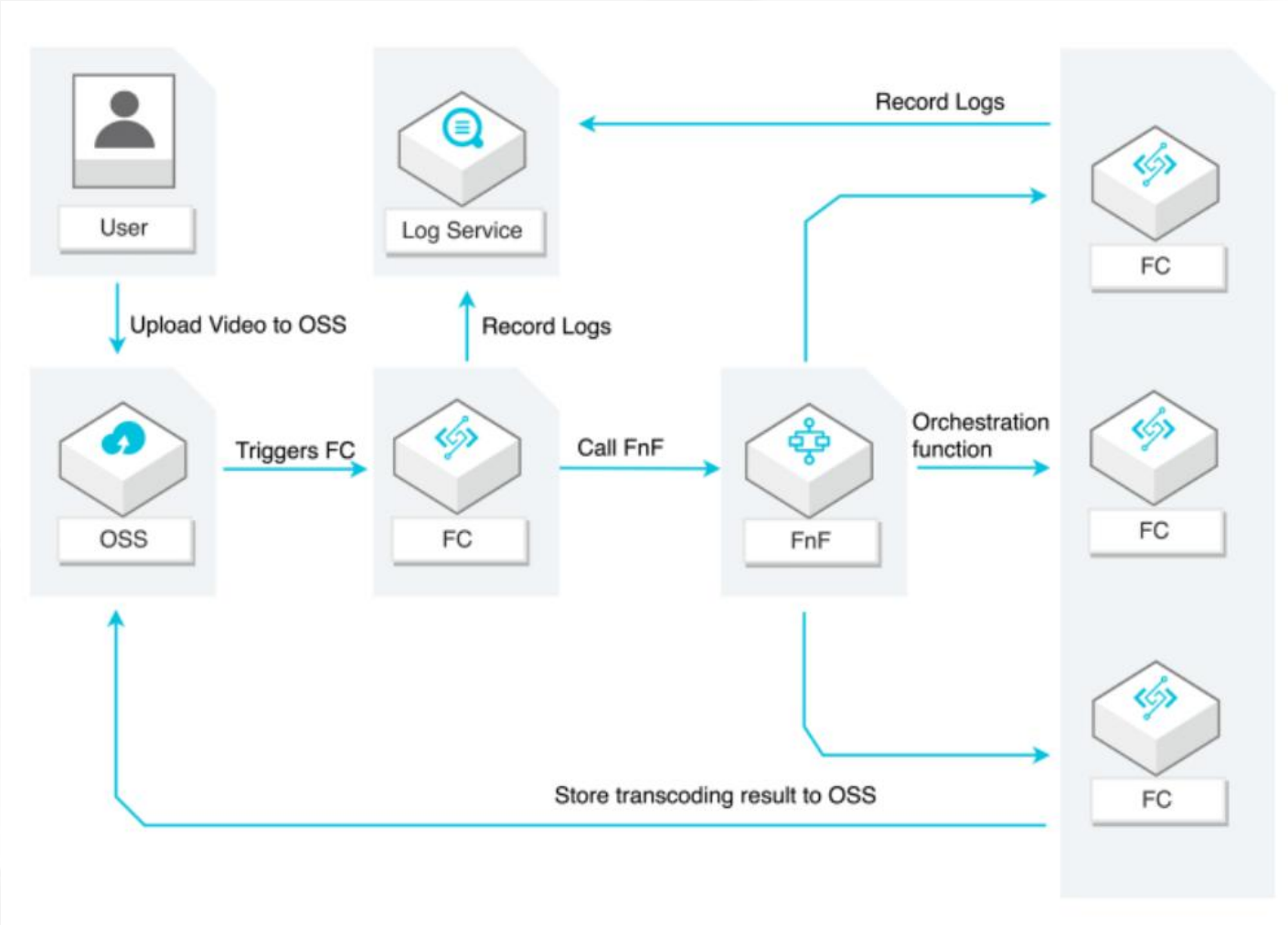
Benifits:

- Enable AI engineers to focus more on algorithms and avoid complex O&M operations
- Mobilize tens of thousands of computing resources to eliminate the computing power bottleneck
- Provide multiple versions for A/B testing to reduce model-launching risks
- Install third-party libraries by one click to smoothly debug in local environments

Serverless Applications

Case 4: Video Transcoding

Function Compute and Function Flow can be used together to create elastic and highly available Serverless video processing systems that have enhanced performance and efficiency as well as lower costs.

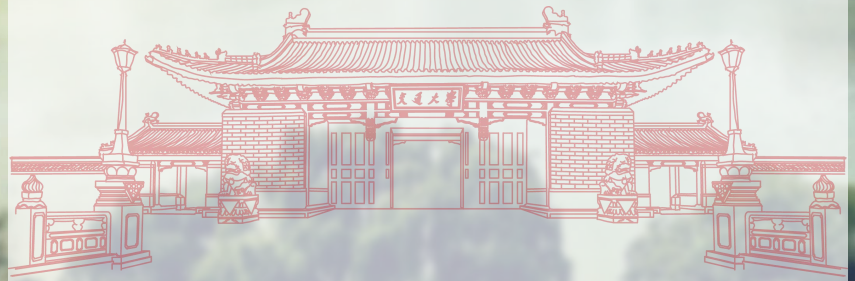


Benifits:

- Flexible transcoding: support custom transcoding processing logic
- Low cost: provide costs reductions of over 75%
- Parallel transcoding: automatically scale based on the number of video files
- Fast migration: lower migration costs and simplified operations

03

How Serverless Works



Serverless Platforms

Commercial Serverless Platforms

- AWS Lambda
- Google Cloud Functions
- Microsoft Azure Functions
- IBM Cloud Functions
- Alibaba Cloud Function Compute
- Tencent Cloud's Serverless Cloud Function (SCF)
- ...

Open-source Serverless Platforms

- OpenWhisk
- OpenFaaS
- Kubeless
- Knative
- Fission
- Nuclio
- ...

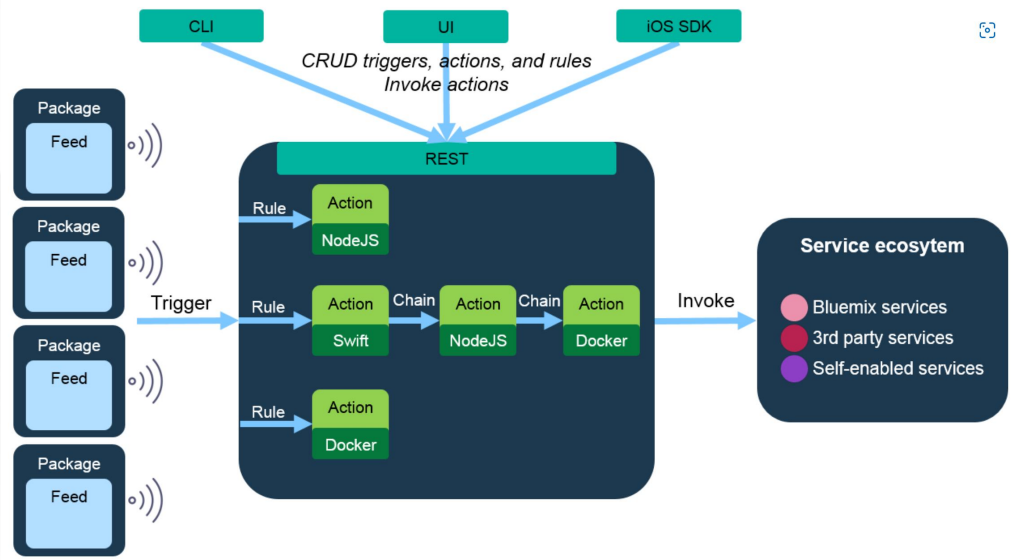
OpenWhisk

OpenWhisk is an event-driven compute platform also referred to as Serverless computing or as Function as a Service (FaaS) that runs code in response to events or direct invocations.

OpenWhisk offers a rich **programming model** for
- creating serverless APIs from functions
- composing functions into serverless workflows
- connecting events to functions using rules and triggers

APACHE OpenWhisk™

Characteristics
- deploys anywhere
- write functions in any language
- integrate easily with many popular services
- combine your functions into rich compositions
- scaling per-request
- optimal utilization



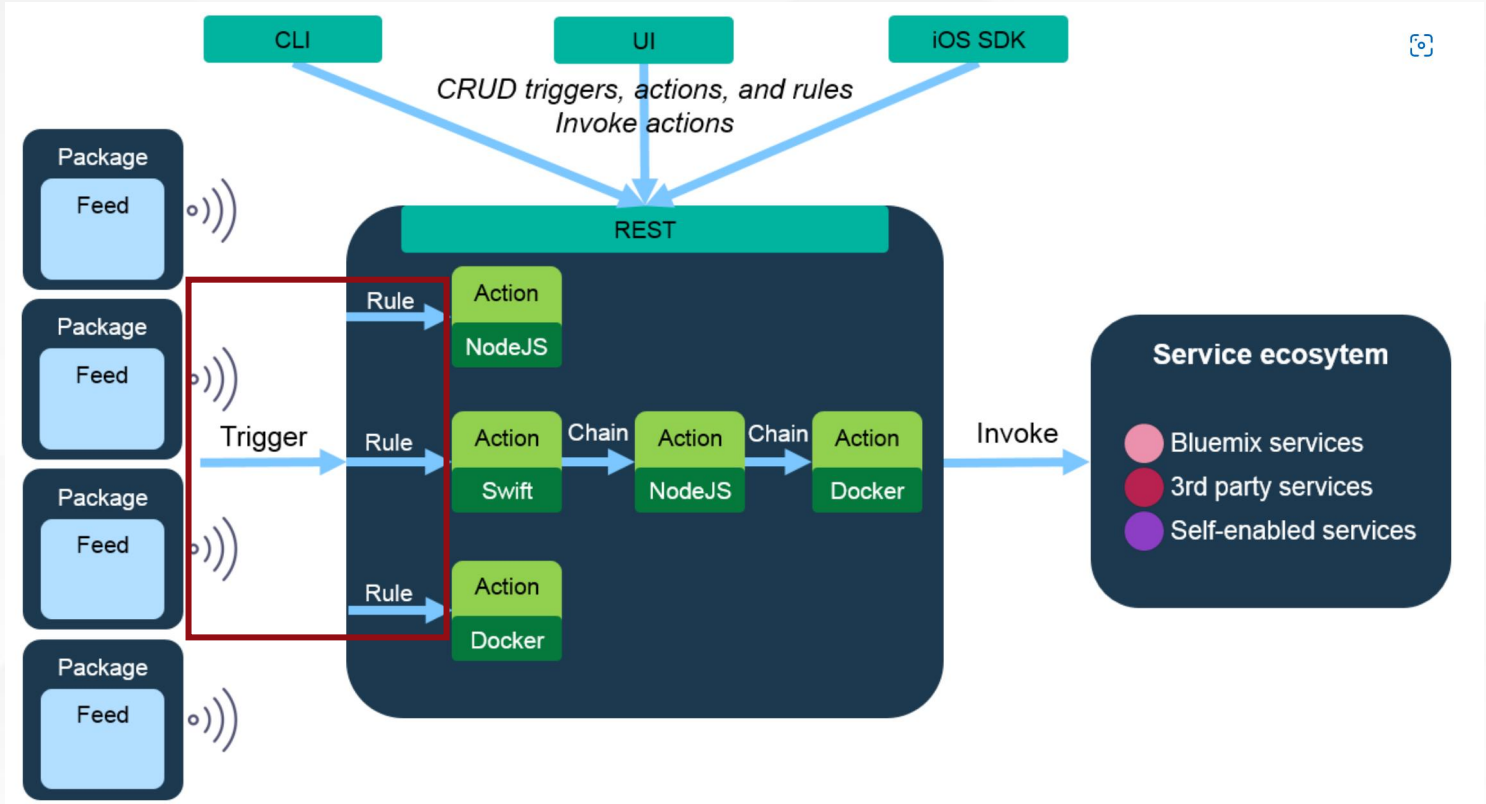
the high-level OpenWhisk architecture

OpenWhisk

Examples of **Events** include

- changes to database records
- IoT sensor readings that exceed a certain temperature
- new code commits to a GitHub repository
- HTTP requests from web or mobile apps
- ...

Events from external and internal event sources are channeled through a **trigger**, and **rules** allow actions to react to these events.



the high-level OpenWhisk architecture

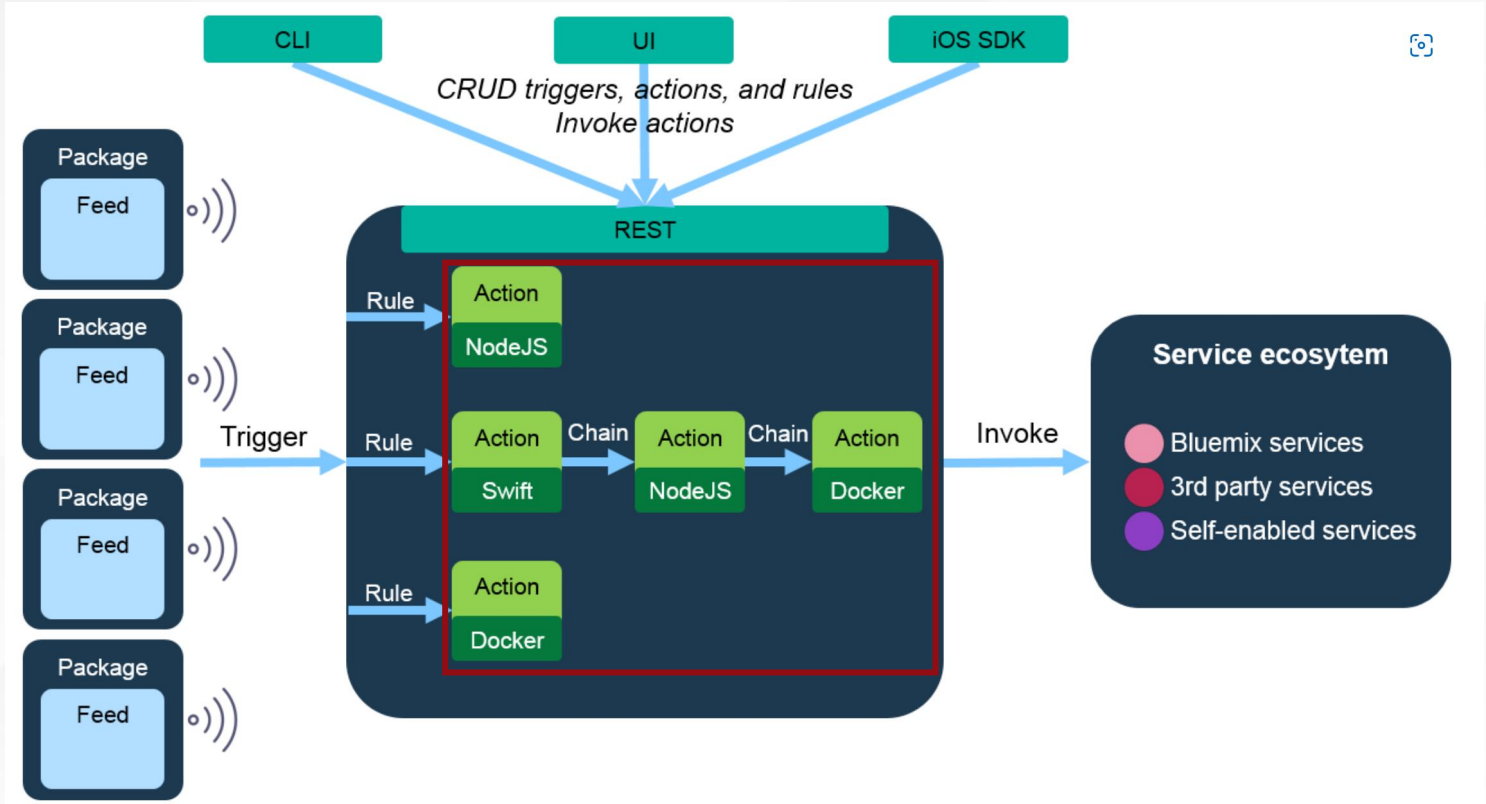
OpenWhisk

Actions can be

- small snippets of code (JavaScript, Swift and many other languages are supported)
- custom binary code embedded in a Docker container

Actions in OpenWhisk are instantly deployed and executed whenever a trigger fires.

The more triggers fire, the more actions get invoked. If no trigger fires, no action code is running, so there is no cost.



the high-level OpenWhisk architecture

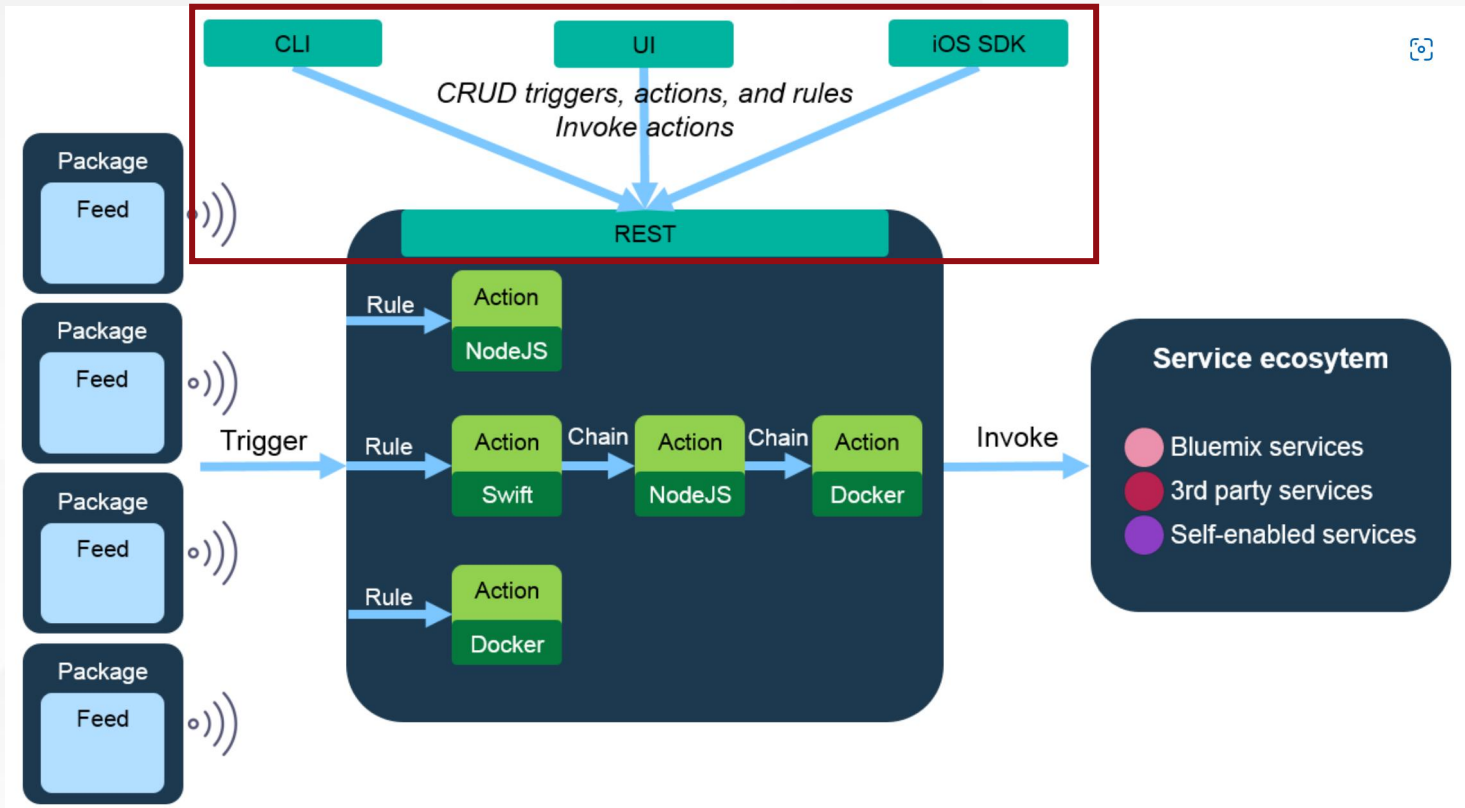
OpenWhisk

In addition to associating actions with triggers, it is possible to **directly invoke** an action by using

- the OpenWhisk API
- CLI
- iOS SDK

A set of actions can also be chained without having to write any code.

Each action in the **chain** is invoked in sequence with the output of one action passed as input to the next in the sequence.

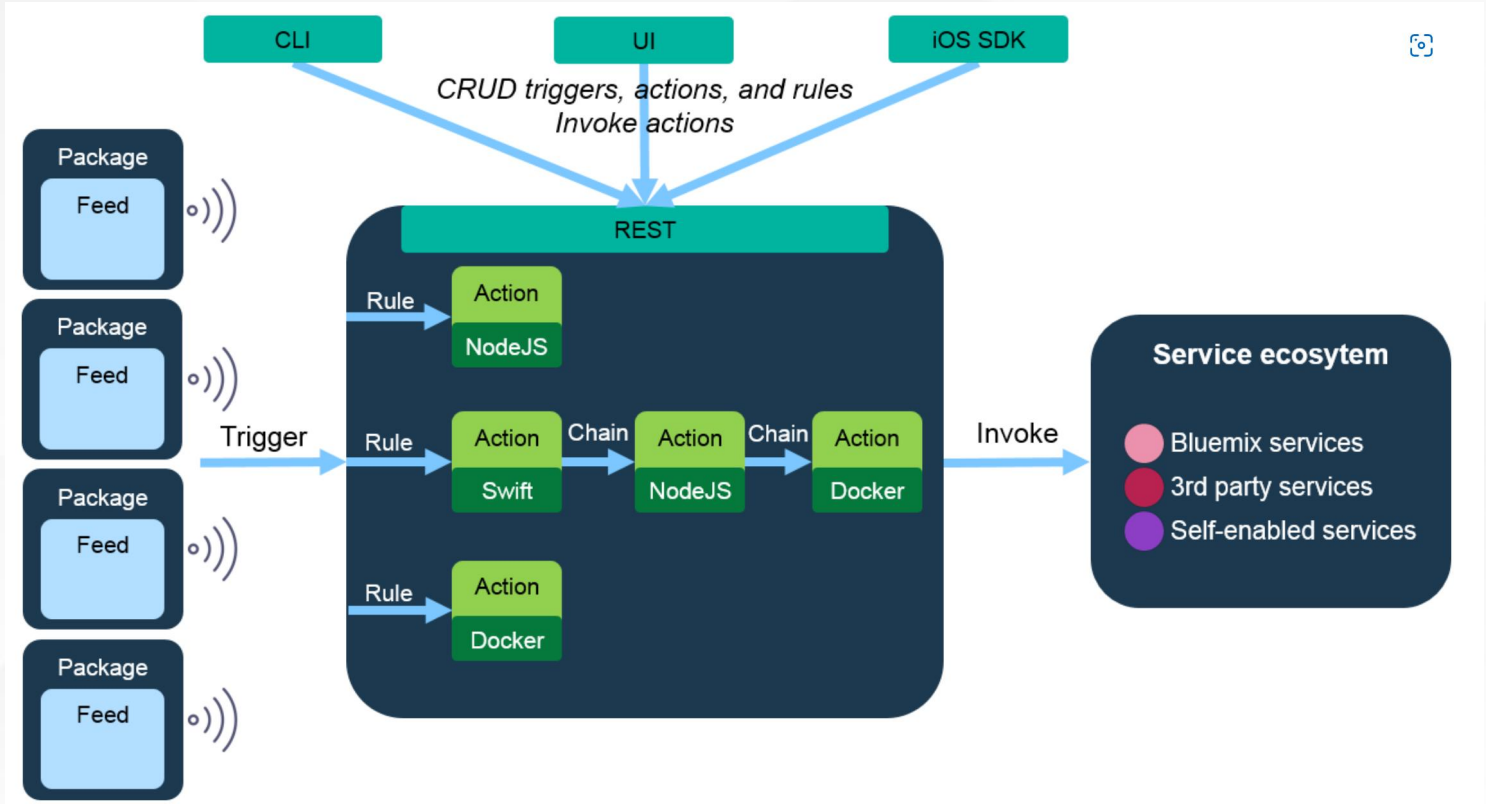


the high-level OpenWhisk architecture

OpenWhisk

With traditional long-running virtual machines or containers, it is common practice to deploy multiple VMs or containers to be resilient against outages of a single instance.

However, OpenWhisk offers an alternative model with **no resiliency-related cost overhead**. The **on-demand execution** of actions provides inherent scalability and optimal utilization, as the number of running actions always matches the trigger rate.



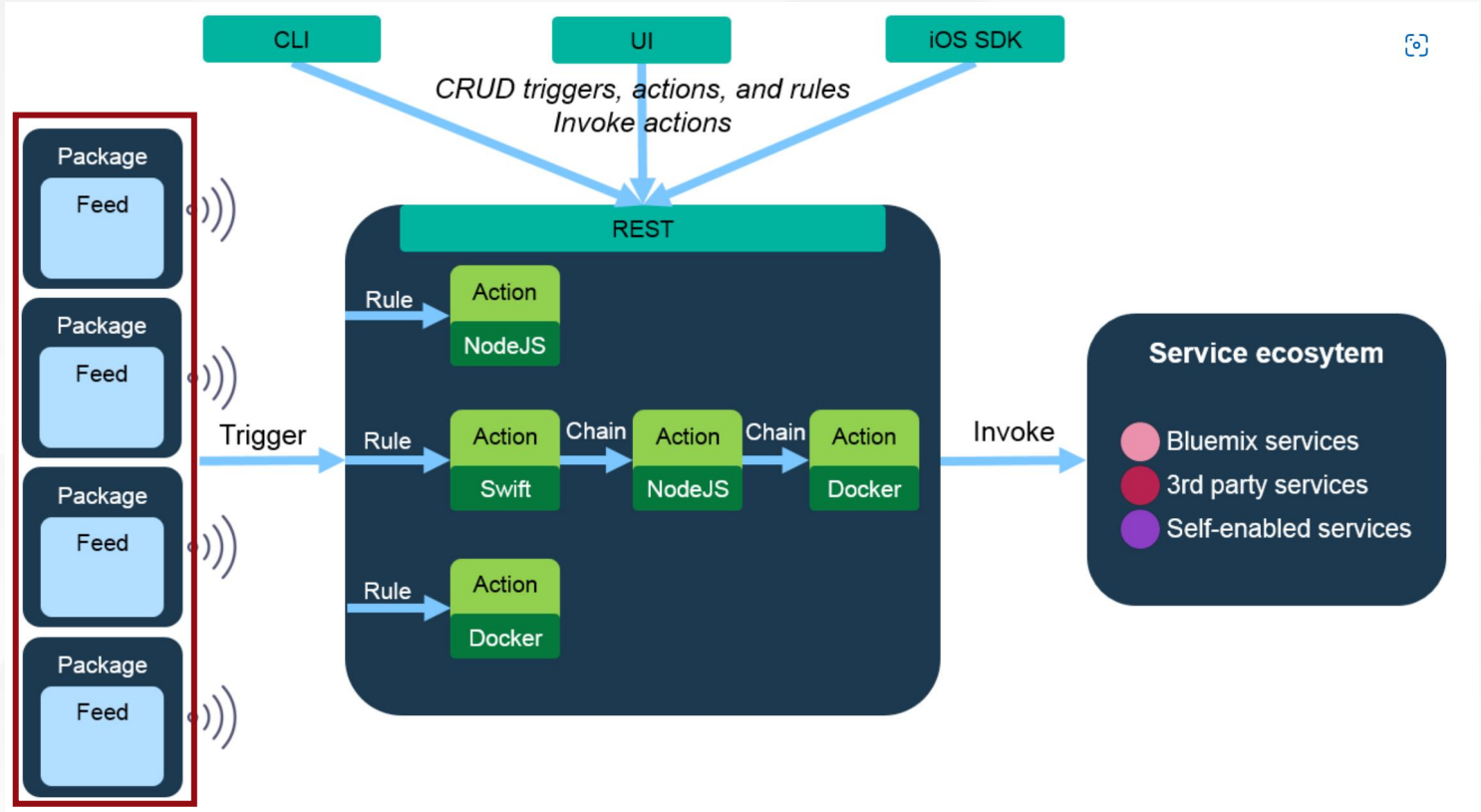
the high-level OpenWhisk architecture

OpenWhisk

Integrations with additional services and event providers can be added with **packages**. A package is a bundle of feeds and actions.

A **feed** is a piece of code that configures an external event source to fire trigger events.

Actions in packages represent reusable logic that a service provider can make available so that developers not only can use the service as an event source, but also can invoke APIs of that service.



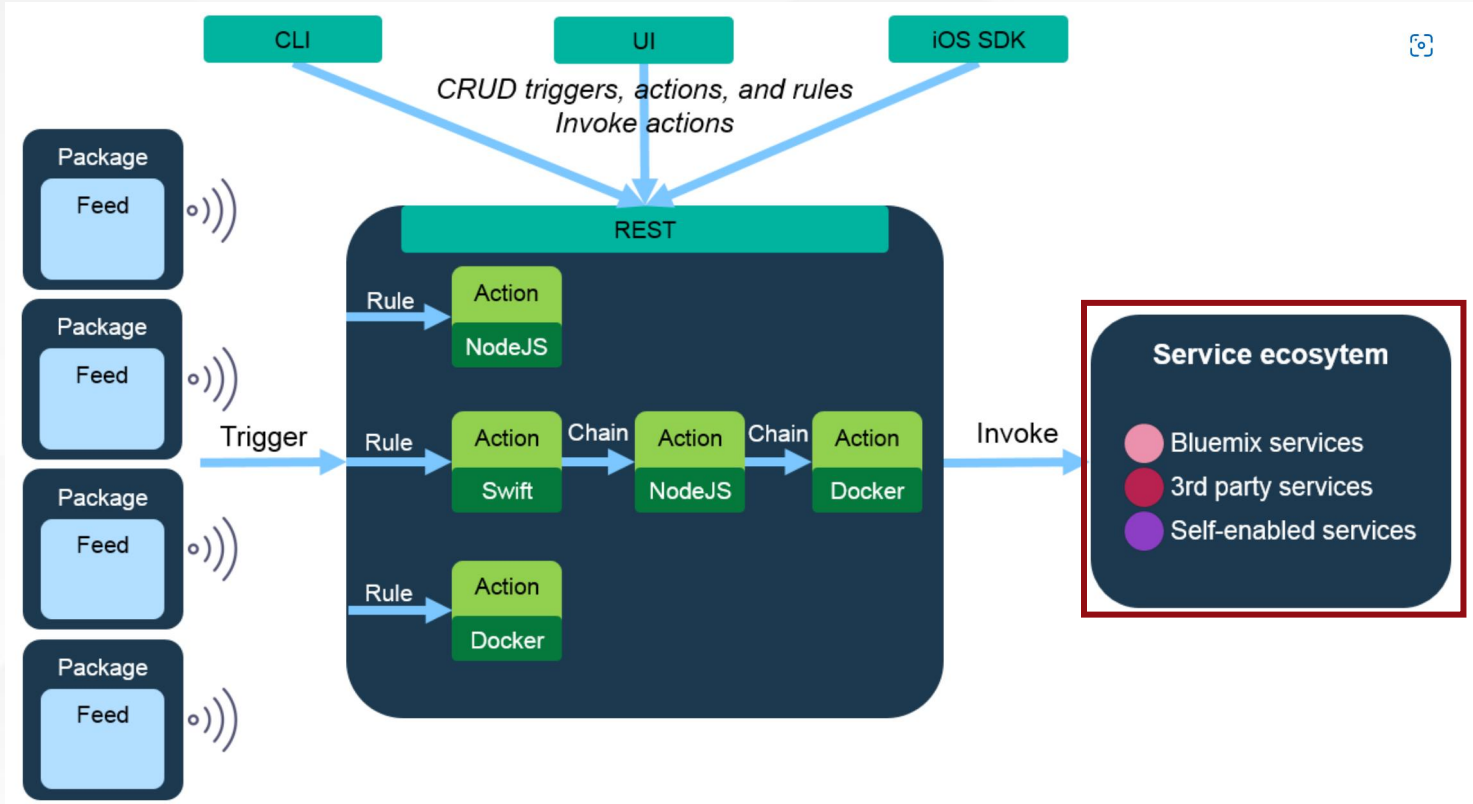
the high-level OpenWhisk architecture

OpenWhisk

An existing catalog of packages offers a quick way to enhance applications with useful capabilities, and to access **external services** in the **ecosystem**.

Examples of external services that are OpenWhisk-enabled include

- Cloudant
- The Weather Company
- Slack
- GitHub.



the high-level OpenWhisk architecture

OpenWhisk

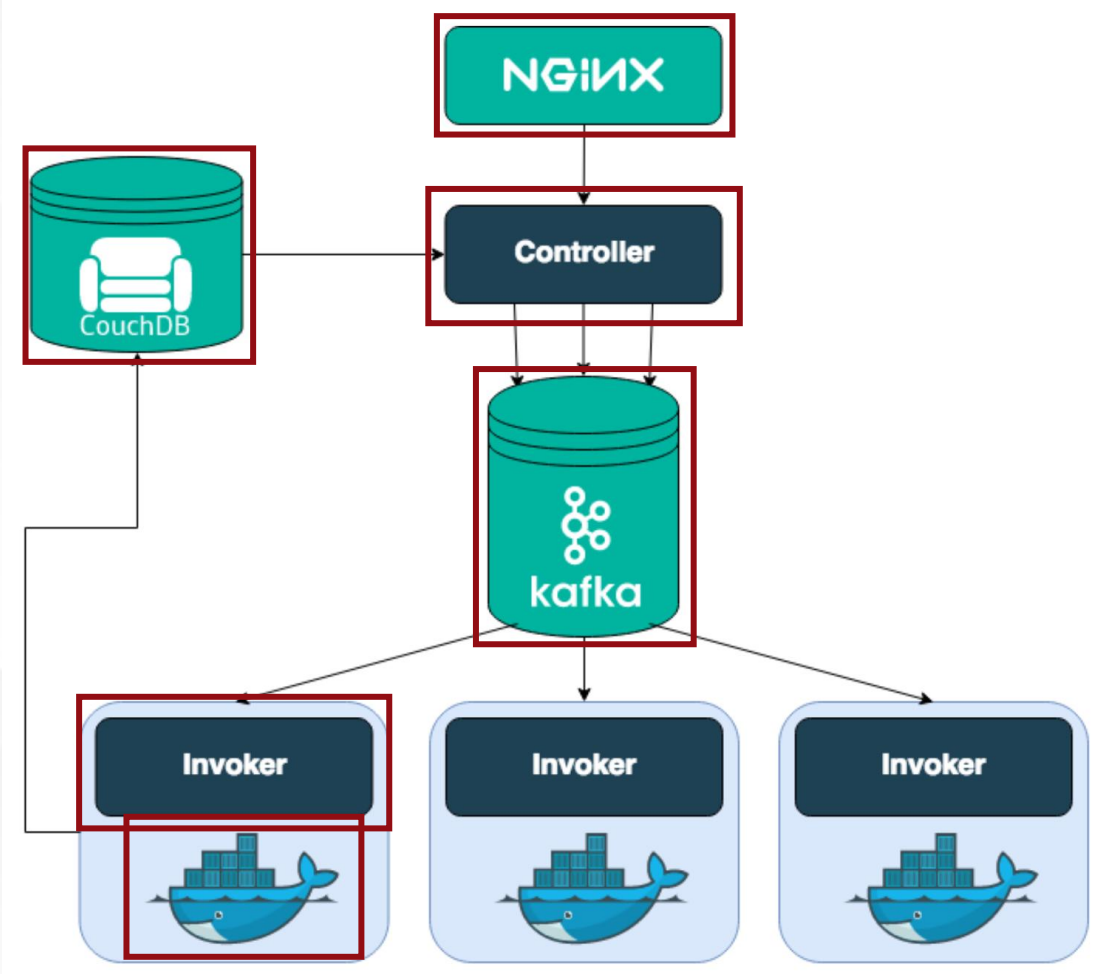
Being an open-source project, OpenWhisk stands on the shoulders of giants, including

- **Nginx**
- **Kafka**
- **Docker**
- **CouchDB**

All of these components come together to form a “serverless event-based programming service”.

The system itself mainly consists of only two custom components, the **Controller** and the **Invoker**. Everything else is already there, developed by so many people out there in the open-source community.

To explain all the components in more detail, lets trace **an invocation of an action through the system** as it happens.



The internal flow of processing

OpenWhisk

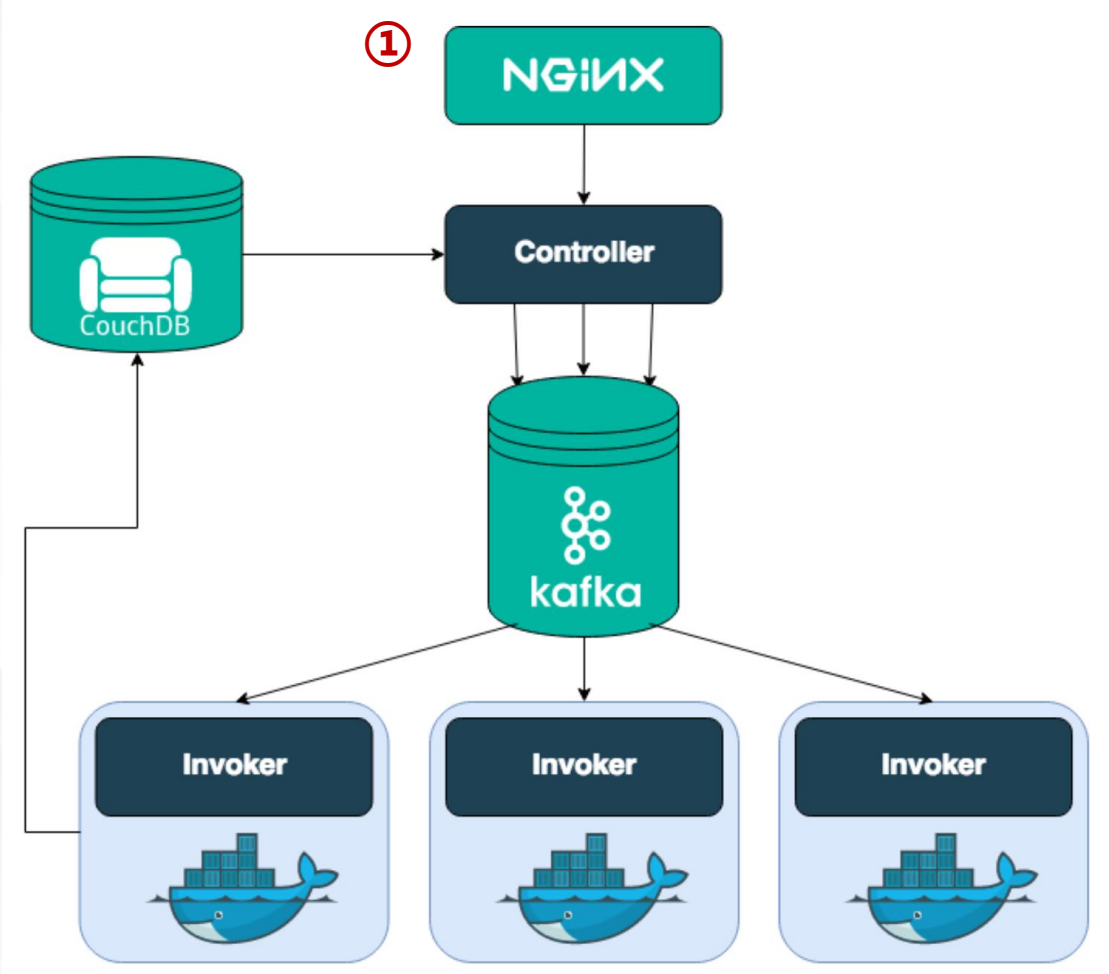
① Entering the system: nginx

OpenWhisk's user-facing API is completely **HTTP based** and follows a **RESTful design**.

As a consequence, the command sent via the *wsk* CLI is essentially an HTTP request against the OpenWhisk system.

The first entry point into the system is through **nginx**, "an HTTP and reverse proxy server".

It is mainly used for SSL termination and forwarding appropriate HTTP calls to the next component.



The internal flow of processing

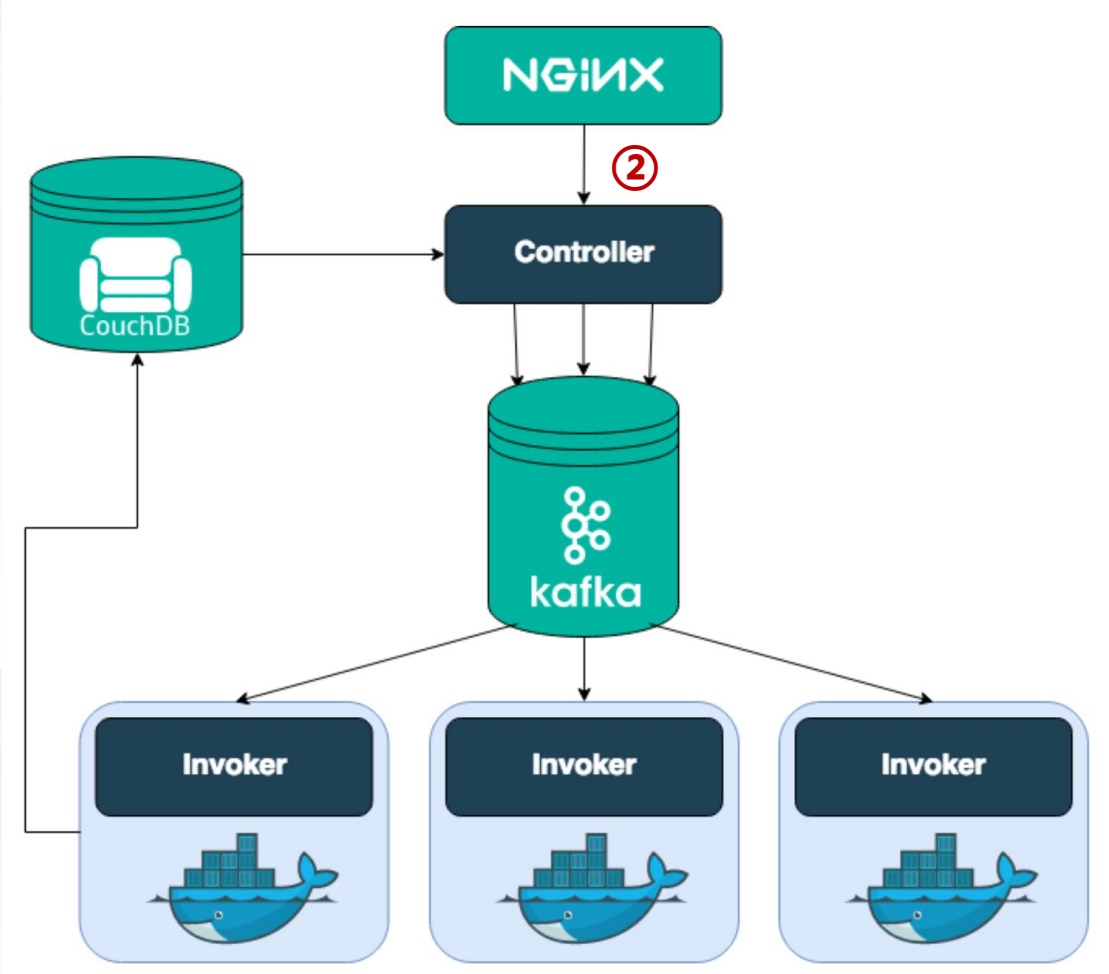
② Entering the system: Controller

Not having done much to our HTTP request, nginx forwards it to the **Controller**.

It is a Scala-based implementation of the actual REST API and thus **serves as the interface for everything a user can do**, including CRUD requests for your entities in OpenWhisk and invocation of actions.

The Controller first disambiguates what the user is trying to do. It does so based on the HTTP method you use in your HTTP request.

As per translation above, the user is issuing a POST request to an existing action, which the Controller translates to **an invocation of an action**.



The internal flow of processing

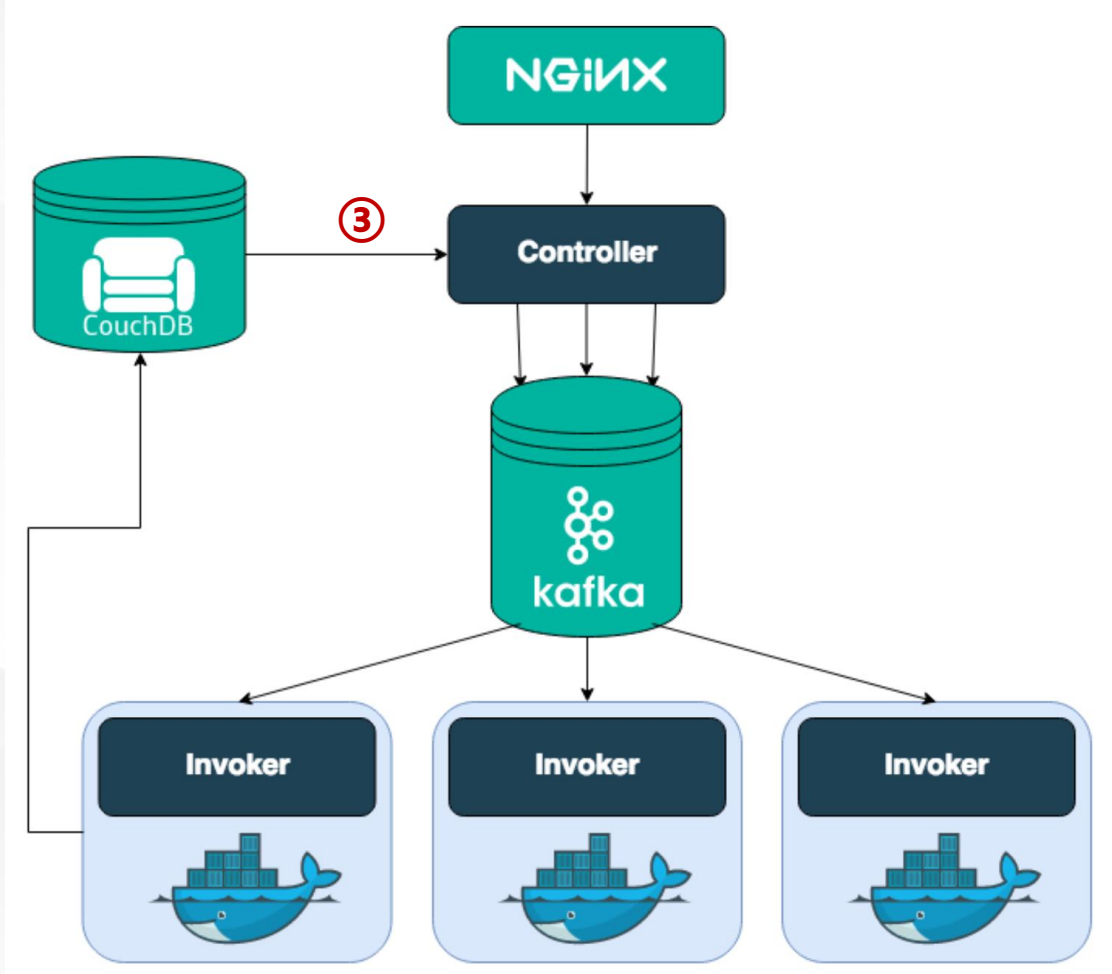
③ Authentication and Authorization: CouchDB

Now the Controller verifies who you are (**Authentication**) and if you have the privilege to do what you want to do with that entity (**Authorization**).

The credentials included in the request are verified against the so-called **subjects** database in a **CouchDB** instance.

It is checked that the user exists in OpenWhisk's database and that it has the privilege to invoke the action. The latter effectively gives the user the privilege to invoke the action, which is what he wishes to do.

As everything is sound, the gate opens for the next stage of processing.



The internal flow of processing

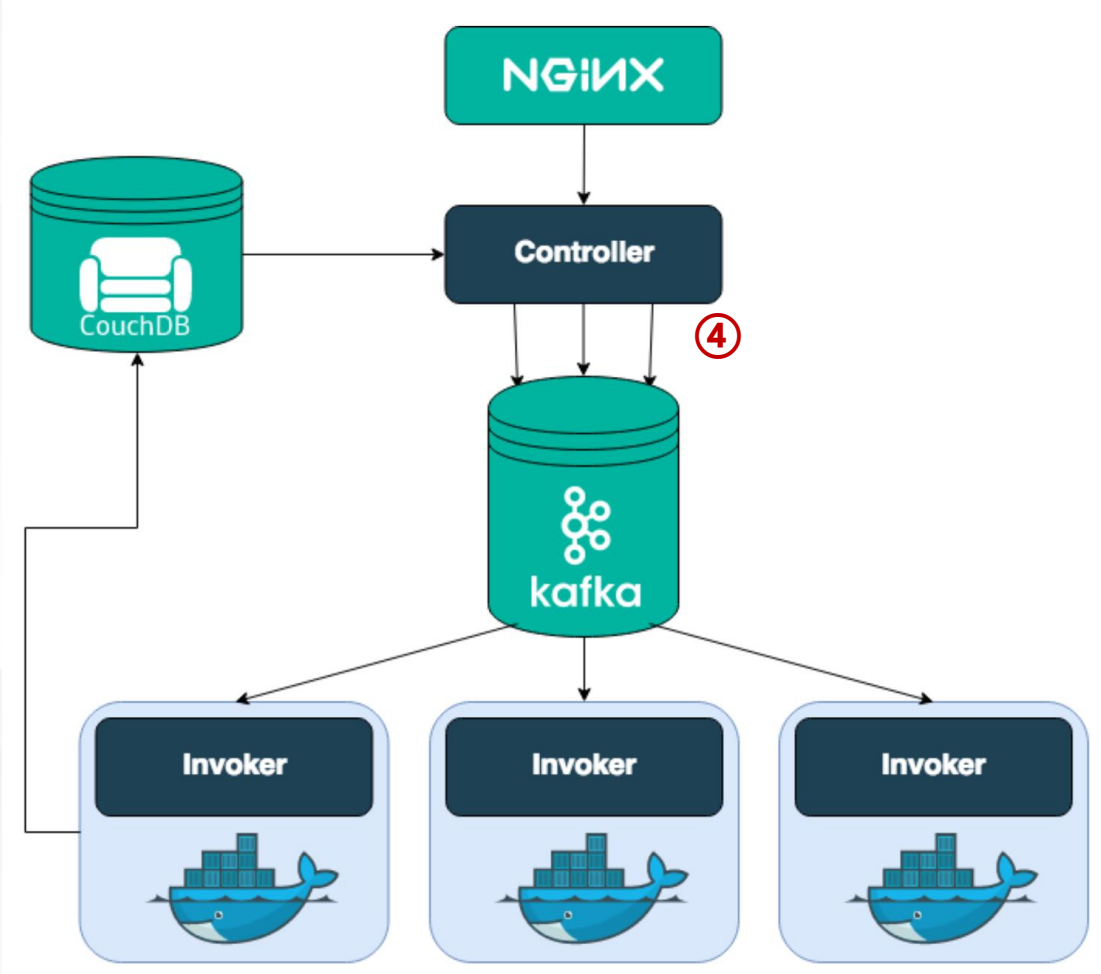
OpenWhisk

④ Who's there to invoke the action: Load Balancer

The **Load Balancer**, which is part of the Controller, has a global view of the executors available in the system by checking their health status continuously.

Those executors are called **Invokers**.

The Load Balancer, knowing which Invokers are available, chooses one of them to invoke the action requested.



The internal flow of processing

⑤ Please form a line: Kafka

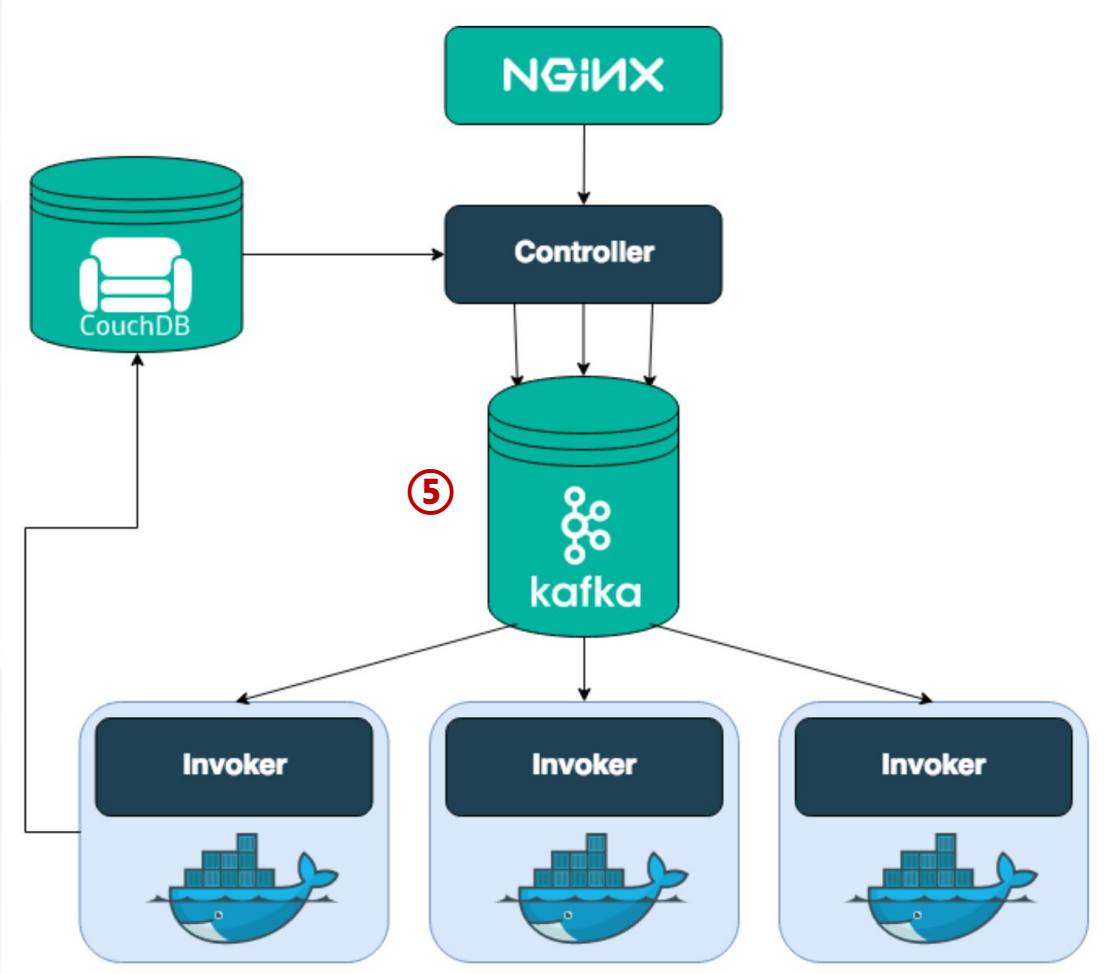
From now on, mainly two bad things can happen to the invocation request sent in:

- The system can crash, losing your invocation.
- The system can be under such a heavy load, that the invocation needs to wait for other invocations to finish first.

The answer to both is **Kafka**, “a high-throughput, distributed, publish-subscribe messaging system”.

Controller and Invoker solely communicate through messages buffered and persisted by Kafka.

Once Kafka has confirmed that it got the message, the HTTP request to the user is responded to with an **ActivationId**. The user will use that later on, to get access to the results of this specific invocation.



The internal flow of processing

OpenWhisk

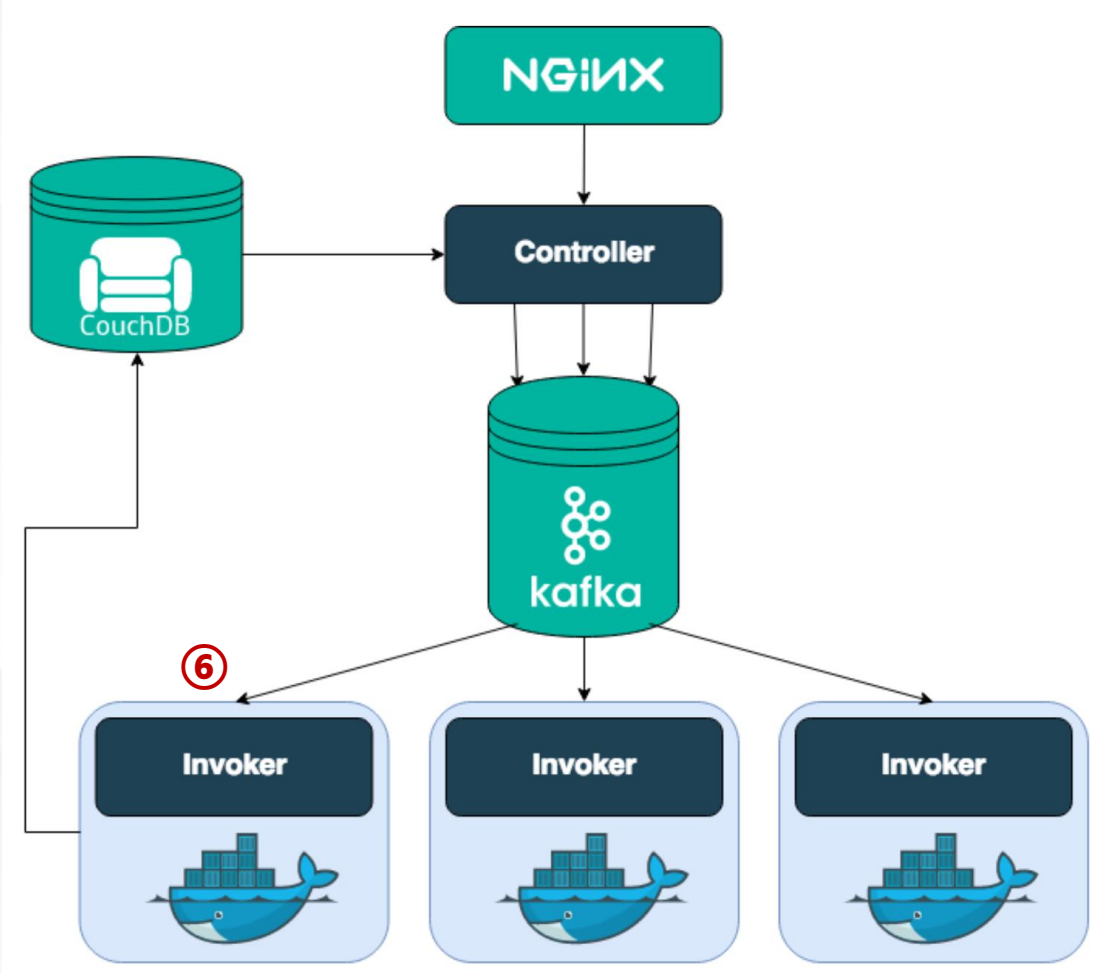
⑥ Actually invoking the code already: Invoker

The **Invoker** is the heart of OpenWhisk. The Invoker's duty is to invoke an action.To execute actions in an isolated and safe way it uses **Docker**.

Docker is used to setup a new self-encapsulated environment (called container) for each action that we invoke in a fast, isolated and controlled way.

In a nutshell, for each action invocation a Docker container is spawned, the action code gets injected, it gets executed using the parameters passed to it, the result is obtained, the container gets destroyed.

This is also the place where a lot of **performance optimization** is done to reduce overhead and make low response times possible.



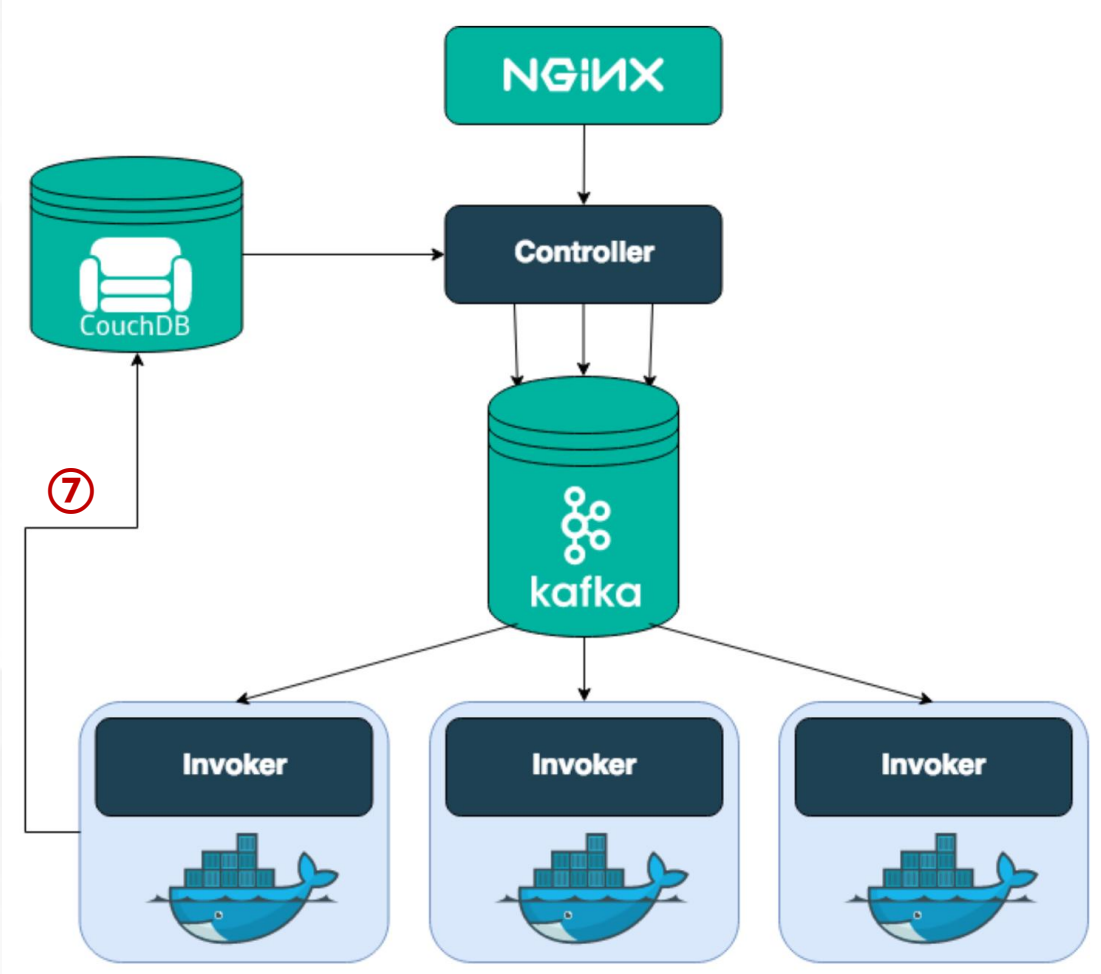
The internal flow of processing

⑦ Storing the results: CouchDB again

As the result is obtained by the Invoker, it is stored into the **activations** database as an activation under the ActivationId mentioned further above. The activations database lives in **CouchDB**.

The Invoker gets the resulting JSON object back from the action, grabs the log written by Docker, puts them all into the **activation record** and stores it into the database.

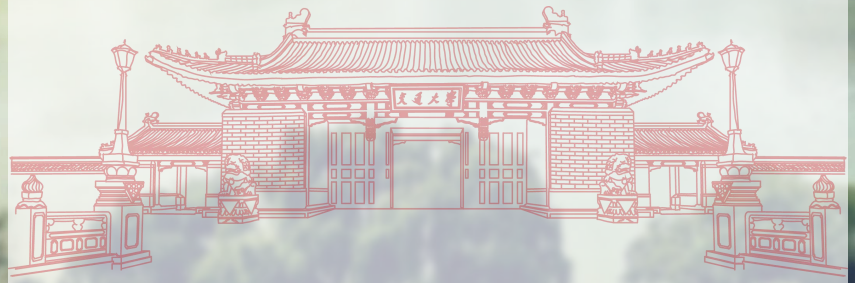
The record contains both the returned result and the logs written. It also contains the start and end time of the invocation of the action.



The internal flow of processing

04

Challenges of Serverless Adoption



Limitations of Serverless Computing Platforms

Four limits in the current state of serverless computing:

1. Inadequate storage for fine-grained operations.
2. Lack of fine-grained coordination.
3. Poor performance for standard communication patterns.
4. Predictable Performance.

Limitation 1: Storage

The stateless nature of serverless platforms

difficult to support

The fine-grained state sharing needs of applications

		Block Storage (e.g., AWS EBS, IBM Block Storage)	Object Storage (e.g., AWS S3, Azure Blob Store, Google Cloud Storage)	File System (e.g., AWS EFS, Google Filestore)	Elastic Database (e.g., Google Cloud Datastore, Azure Cosmos DB)	Memory Store (e.g., AWS ElastiCache, Google Cloud Memorystore)	"Ideal" storage service for serverless computing
Cloud functions access		No	Yes	Yes[13]	Yes	Yes	Yes
Transparent Provisioning		No	Yes	Capacity only[14]	Yes[15]	No	Yes
Availability and persistence guarantees		Local & Persistent	Distributed & Persistent	Distributed & Persistent	Distributed & Persistent	Local & Ephemeral	Various
Latency (mean)		< 1ms	10 – 20ms	4 – 10ms	8 – 15ms	< 1ms	< 1ms
Cost[16]	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

Persistence and availability guarantees describe how well the system tolerates failures:

- **Local** provides reliable storage at one site
- **Distributed** ensures the ability to survive site failures
- **Ephemeral** describes data that resides in memory and is subject to loss

green for good
orange for medium
red for poor

The properties of existing storage services offered by cloud providers

Limitation 1: Storage

		Block Storage (e.g., AWS EBS, IBM Block Storage)	Object Storage (e.g., AWS S3, Azure Blob Store, Google Cloud Storage)	File System (e.g., AWS EFS, Google Filestore)	Elastic Database (e.g., Google Cloud Datastore, Azure Cosmos DB)	Memory Store (e.g., AWS ElastiCache, Google Cloud Memorystore)	"Ideal" storage service for serverless computing
Cloud functions access		No	Yes	Yes[13]	Yes	Yes	Yes
Transparent Provisioning		No	Yes	Capacity only[14]	Yes[15]	No	Yes
Availability and persistence guarantees		Local & Persistent	Distributed & Persistent	Distributed & Persistent	Distributed & Persistent	Local & Ephemeral	Various
Latency (mean)		< 1ms	10 – 20ms	4 – 10ms	8 – 15ms	< 1ms	< 1ms
Cost[16]	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

The properties of existing storage services offered by cloud providers

Object storage services

- Such as AWS S3, Azure Blob Storage, and Google Cloud Storage
- Highly scalable and provide inexpensive long-term object storage
- High access costs and high access latencies

Limitation 1: Storage

		Block Storage (e.g., AWS EBS, IBM Block Storage)	Object Storage (e.g., AWS S3, Azure Blob Store, Google Cloud Storage)	File System (e.g., AWS EFS, Google Filestore)	Elastic Database (e.g., Google Cloud Datastore, Azure Cosmos DB)	Memory Store (e.g., AWS ElastiCache, Google Cloud Memorystore)	"Ideal" storage service for serverless computing
Cloud functions access		No	Yes	Yes[13]	Yes	Yes	Yes
Transparent Provisioning		No	Yes	Capacity only[14]	Yes[15]	No	Yes
Availability and persistence guarantees		Local & Persistent	Distributed & Persistent	Distributed & Persistent	Distributed & Persistent	Local & Ephemeral	Various
Latency (mean)		< 1ms	10 – 20ms	4 – 10ms	8 – 15ms	< 1ms	< 1ms
Cost[16]	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

The properties of existing storage services offered by cloud providers

Key-value databases

- Such as AWS DynamoDB, Google Cloud Datastore, and Azure Cosmos DB
- Provide high IO Per Second (IOPS)
- Expensive and can take a long time to scale up
- Not fault tolerant and not autoscale

Limitation 1: Storage

		Block Storage (e.g., AWS EBS, IBM Block Storage)	Object Storage (e.g., AWS S3, Azure Blob Store, Google Cloud Storage)	File System (e.g., AWS EFS, Google Filestore)	Elastic Database (e.g., Google Cloud Datastore, Azure Cosmos DB)	Memory Store (e.g., AWS ElastiCache, Google Cloud Memorystore)	"Ideal" storage service for serverless computing
Cloud functions access		No	Yes	Yes[13]	Yes	Yes	Yes
Transparent Provisioning		No	Yes	Capacity only[14]	Yes[15]	No	Yes
Availability and persistence guarantees		Local & Persistent	Distributed & Persistent	Distributed & Persistent	Distributed & Persistent	Local & Ephemeral	Various
Latency (mean)		< 1ms	10 – 20ms	4 – 10ms	8 – 15ms	< 1ms	< 1ms
Cost[16]	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

The properties of existing storage services offered by cloud providers

"Ideal" storage service for serverless computing

- Transparent provisioning
- Equivalent of compute autoscaling
- Different applications will likely motivate different guarantees of persistence and availability
- Low access costs and low access latencies

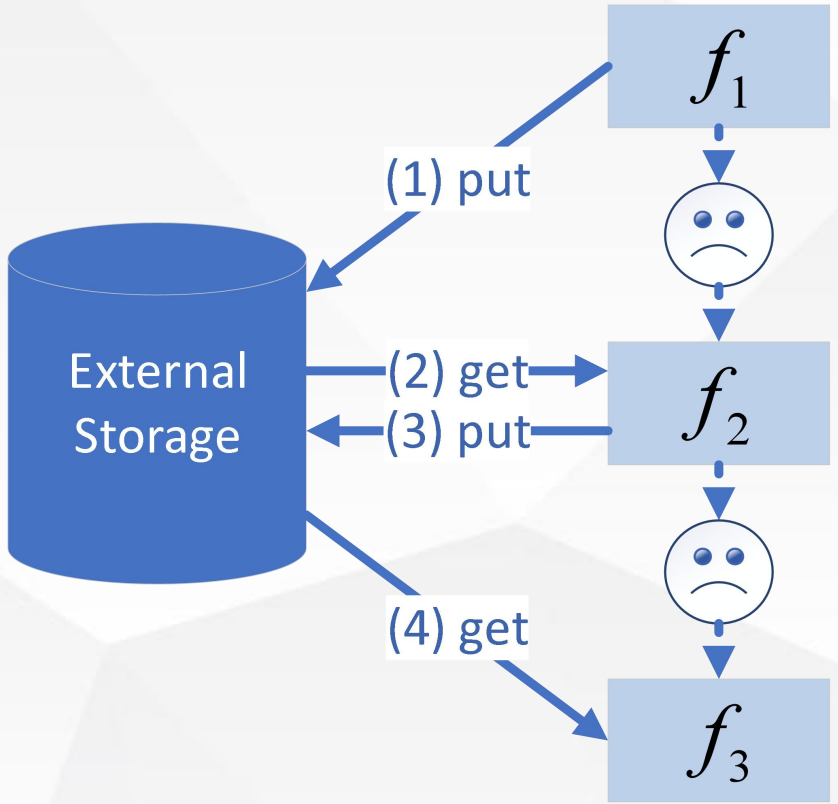
Limitation 2: Coordination

If task A uses task B's output, there must be a way for A to know when its input is available.

However, none of the existing cloud storage services come with **notification capabilities**.

Current methods:

- Cloud providers offer stand-alone notification services, such as SNS and SQS, but with significant latency and be costly when used for fine grained coordination.
- Applications themselves manage a VM-based system that provides notifications, as in ElastiCache and SAND.
- Applications themselves implement their own notification mechanism, such as in ExCamera.

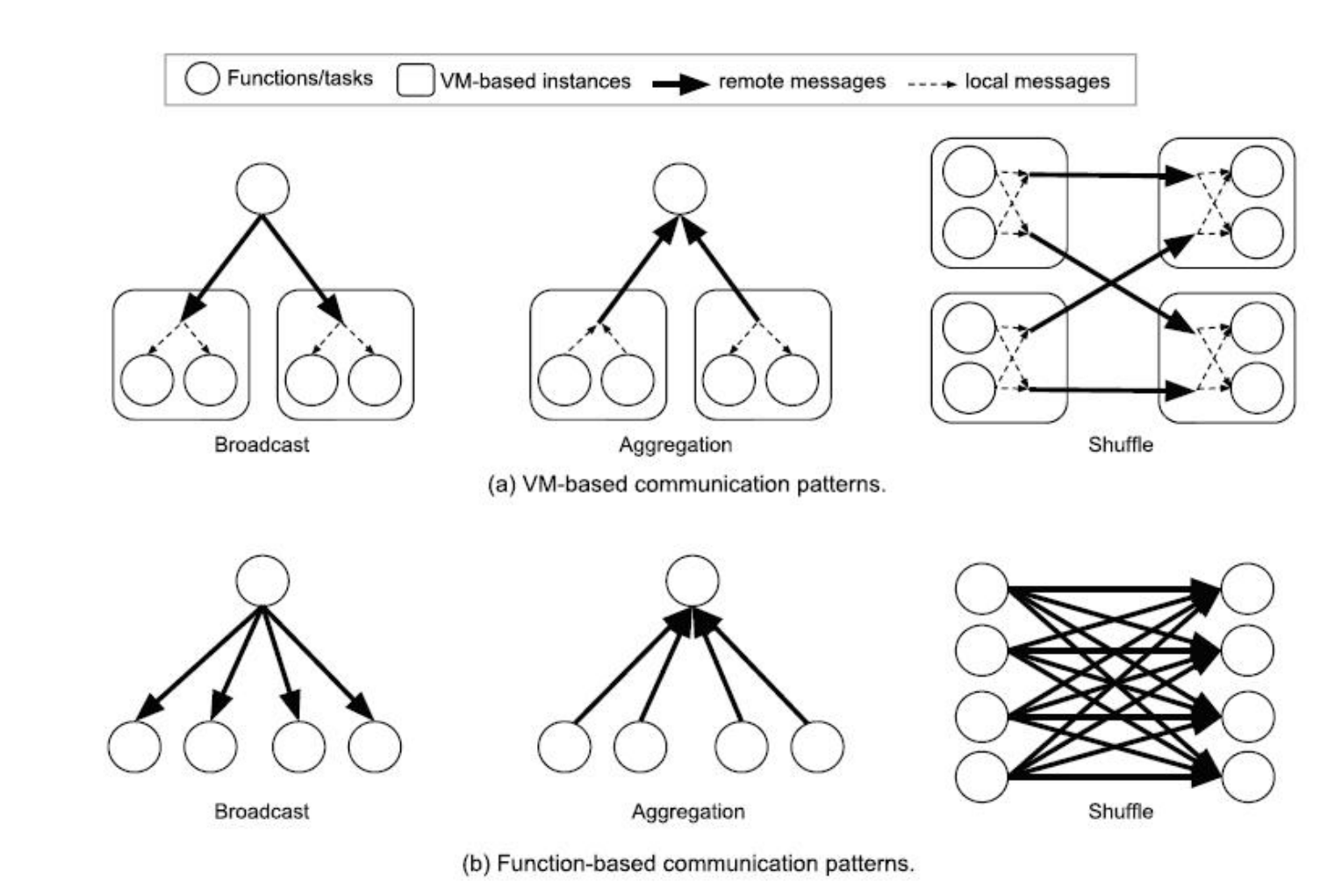


Limitation 3: Communication

Broadcast, aggregation, and shuffle are some of the most common communication primitives in distributed systems.

Communication patterns for these primitives for both **VM-based** and **function-based** solutions.

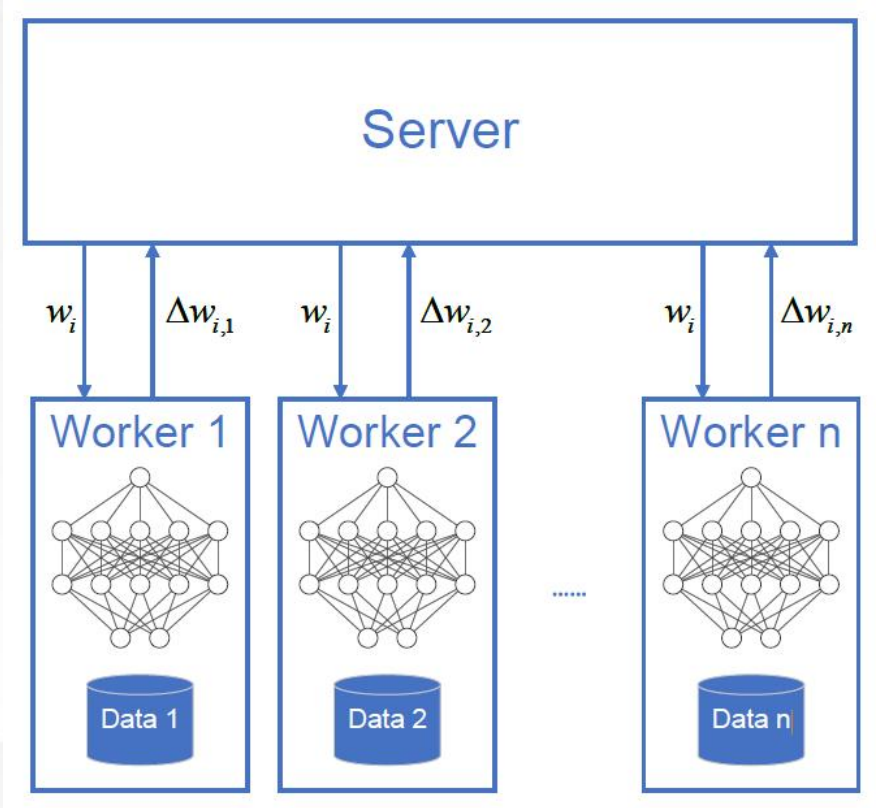
Note the significantly lower number of remote messages for the VM-based solutions. This is because VM instances offer ample opportunities to **share, aggregate, or combine data locally across tasks before sending it or after receiving it**.



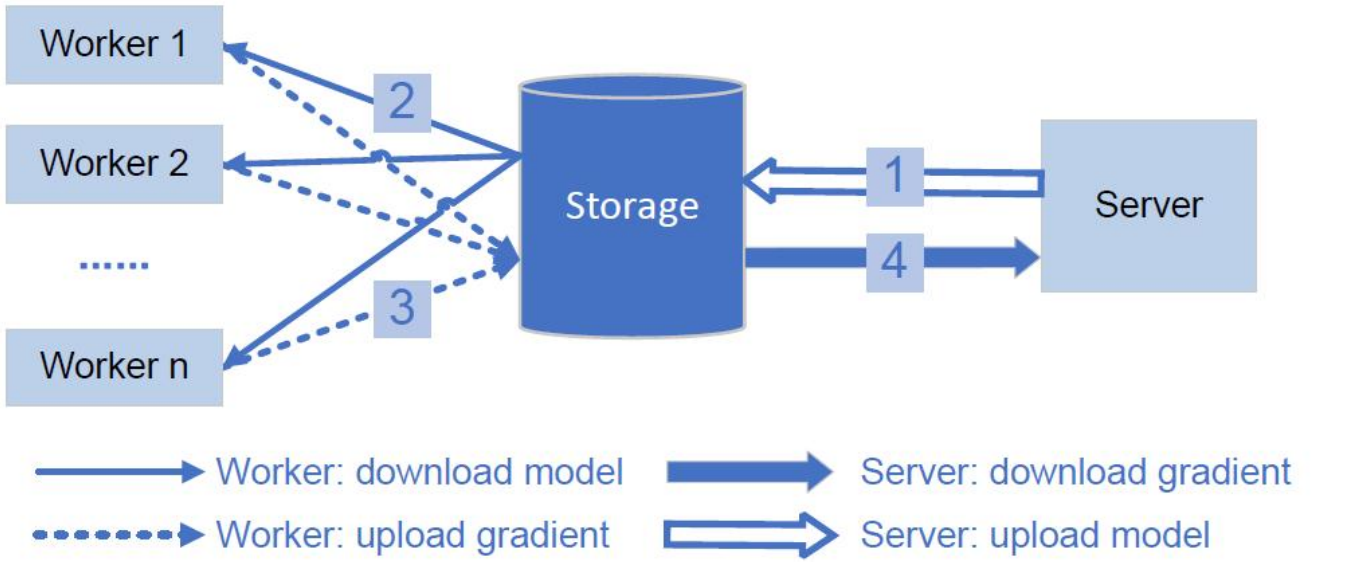
Limitation 3: Communication

Case: Distributed Machine Learning

Parameter Server



Serverless Parameter Server



Limitation 3: Communication

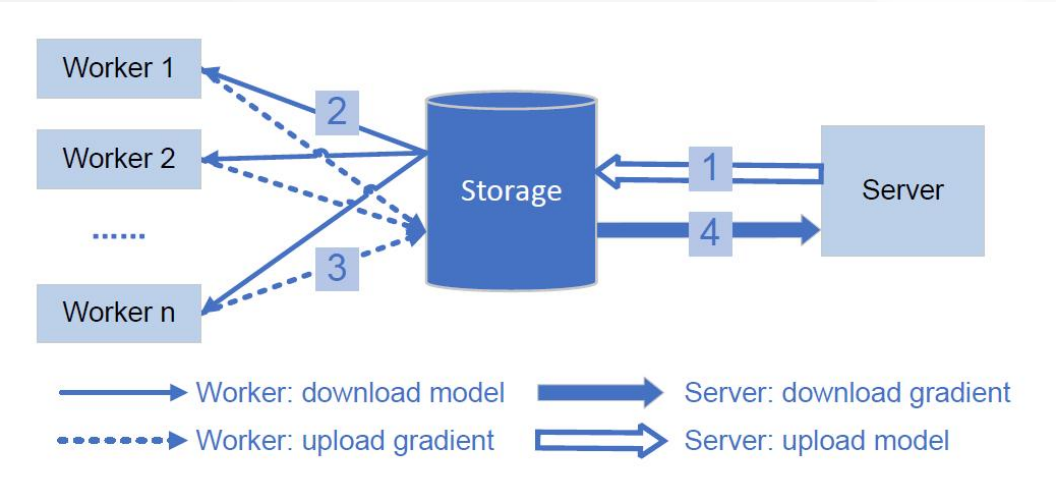
Feasible Optimization for Communication

(1)Optimizing the **storage server**

- Current storage services designed for short-running functions and thus become a performance bottleneck.

- *Pocket* introduces multi-tier storage including DRAM, SSD and HDD.

- *Locus* also combines different kinds of storage devices to achieve both performance and cost-efficiency for serverless analytics



(2)Optimizing the **communication path**

- Optimize the communication path when the relationship between functions is known in advance.

- Another line of work tries to kick the storage server out of the communication path with network mechanisms.

Limitation 4: Cold Start

Although cloud functions have a much lower startup latency than traditional VM-based instances, the delays incurred when starting new instances can be high for some applications.

Three factors impacting cold start latency:

(1) the time it takes to start a cloud function
(2) the time it takes to initialize the software environment
(3) application-specific initialization in user code

Feasible optimization for cold start

- **Container cache**: When a function is finished, the serverless framework can retain its runtime environment.
- **Pre-warming**: OpenWhisk can pre-launch Node.js containers if it has observed that the workload mainly consists of Node.js-based functions.
- **Container optimization**: Provide lean containers with much faster boot time than vanilla ones
- **Looking for other abstractions**: Google gVisor, AWS FireCracker, Unikernel

Thanks

饮 水 思 源　爱 国 荣 校