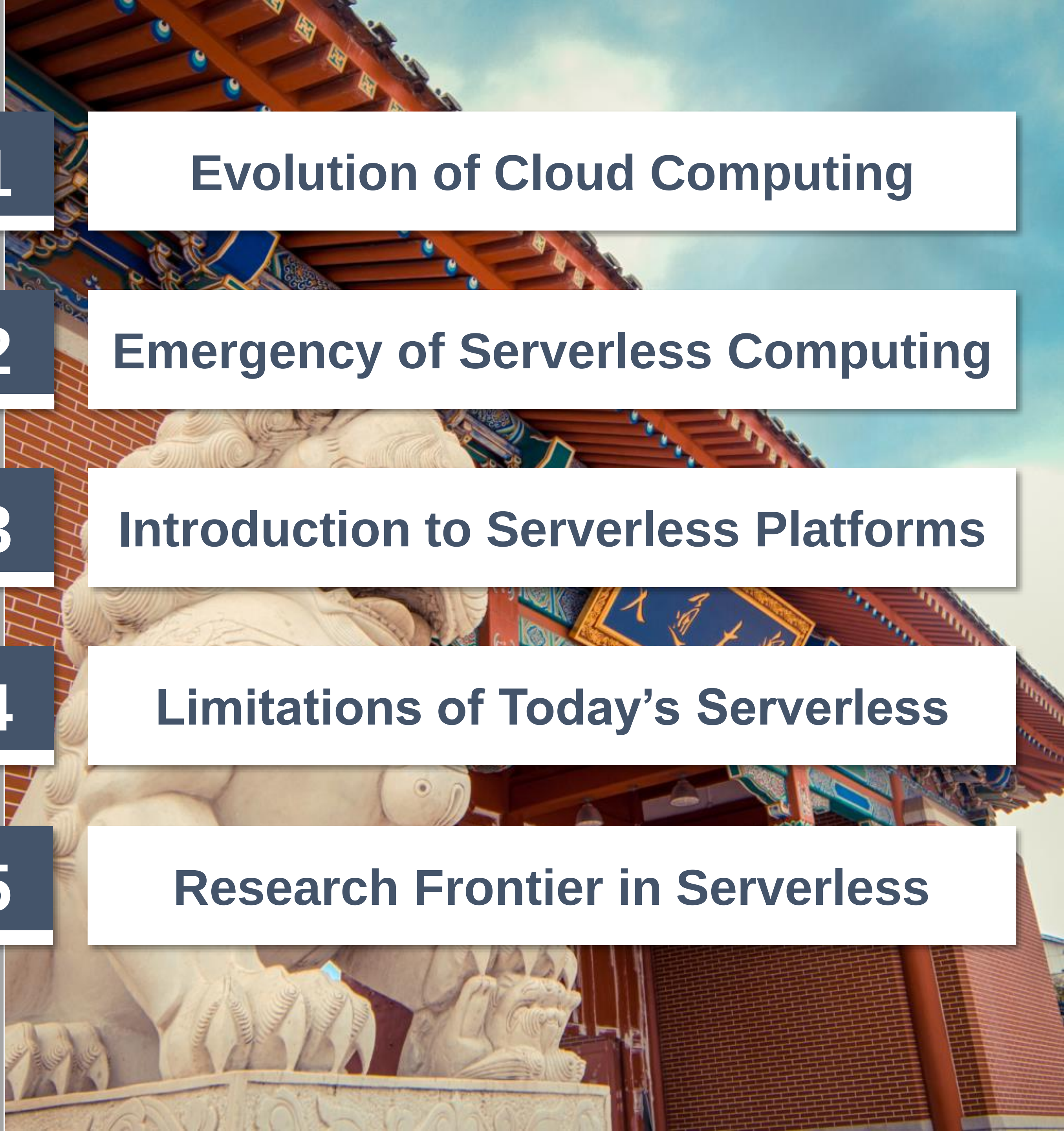
Introduction to Serverless Computing

马汝辉 副教授

计算机科学与工程系

上海交通大学



目 录

01

Evolution of Cloud Computing

What is Cloud Computing?

Cloud Computing refers to

- the applications delivered as services over the Internet
- the hardware and systems software in the datacenters that provide those services

Cloud

Software as a Service (SaaS)

Public Cloud： cloud in a pay-as-you-go manner **available to** the general public

(Utility Computing： the service being sold)

Private Cloud： internal datacenters of a business or other organization **not available to** the general public

Thus, Cloud Computing is the sum of **SaaS** and **Utility Computing**, but does not include Private Clouds.

Why is Cloud Computing?

Six potential advantages in Cloud Computing：　(Berkeley in 2009)

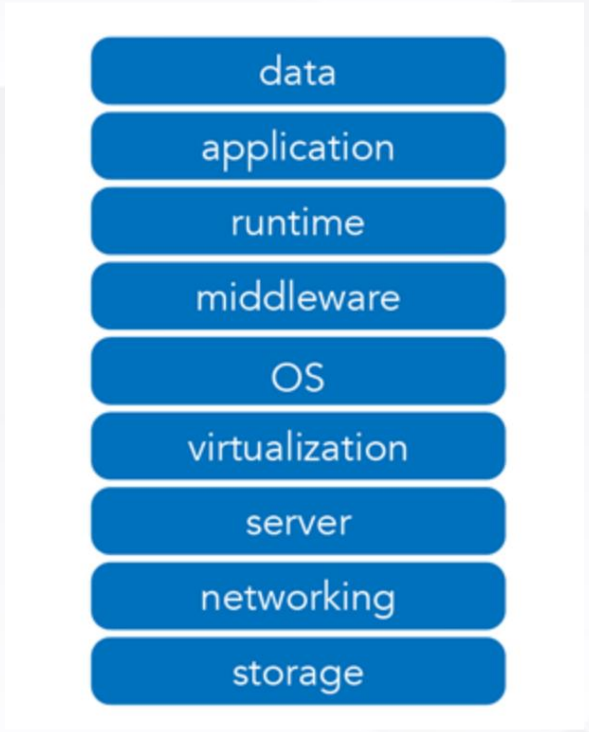
1. The appearance of **infinite computing resources** available on demand. √
2. The **elimination of an up-front commitment** by Cloud users. √
3. The ability to **pay for use** of computing resources on a short-term basis as needed. √
4. **Economies of scale** that significantly reduced cost due to many, very large data centers. √
5. **Simplifying operation and increasing utilization** via resource virtualization. …
6. **Higher hardware utilization** by multiplexing workloads from different organizations. ···

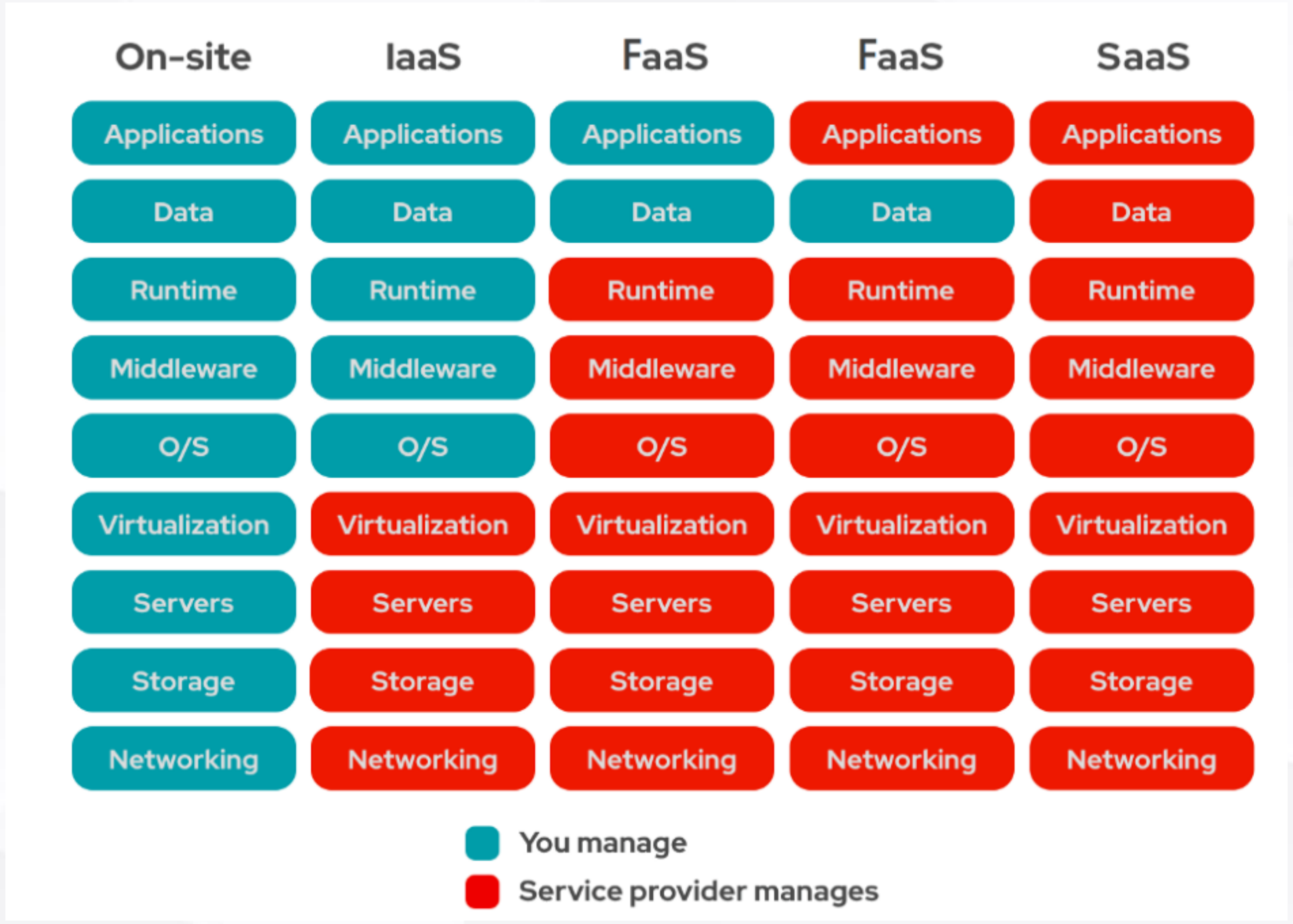
Eight Issues in Setting up Cloud Environment

1. **Redundancy** for availability, so that a single machine failure doesn’t take down the service.
2. **Geographic distribution** of redundant copies to preserve the service in case of disaster.
3. **Load balancing** and request routing to efficiently utilize resources.
4. **Autoscaling** in response to changes in load to scale up or down the system.
5. **Monitoring** to make sure the service is still running well.
6. **Logging** to record messages needed for debugging or performance tuning.
7. **System upgrades**, including security patching.
8. **Migration** to new instances as they become available.

Service Categories

The standard model of organizing all services available in the cloud into categories is the following:





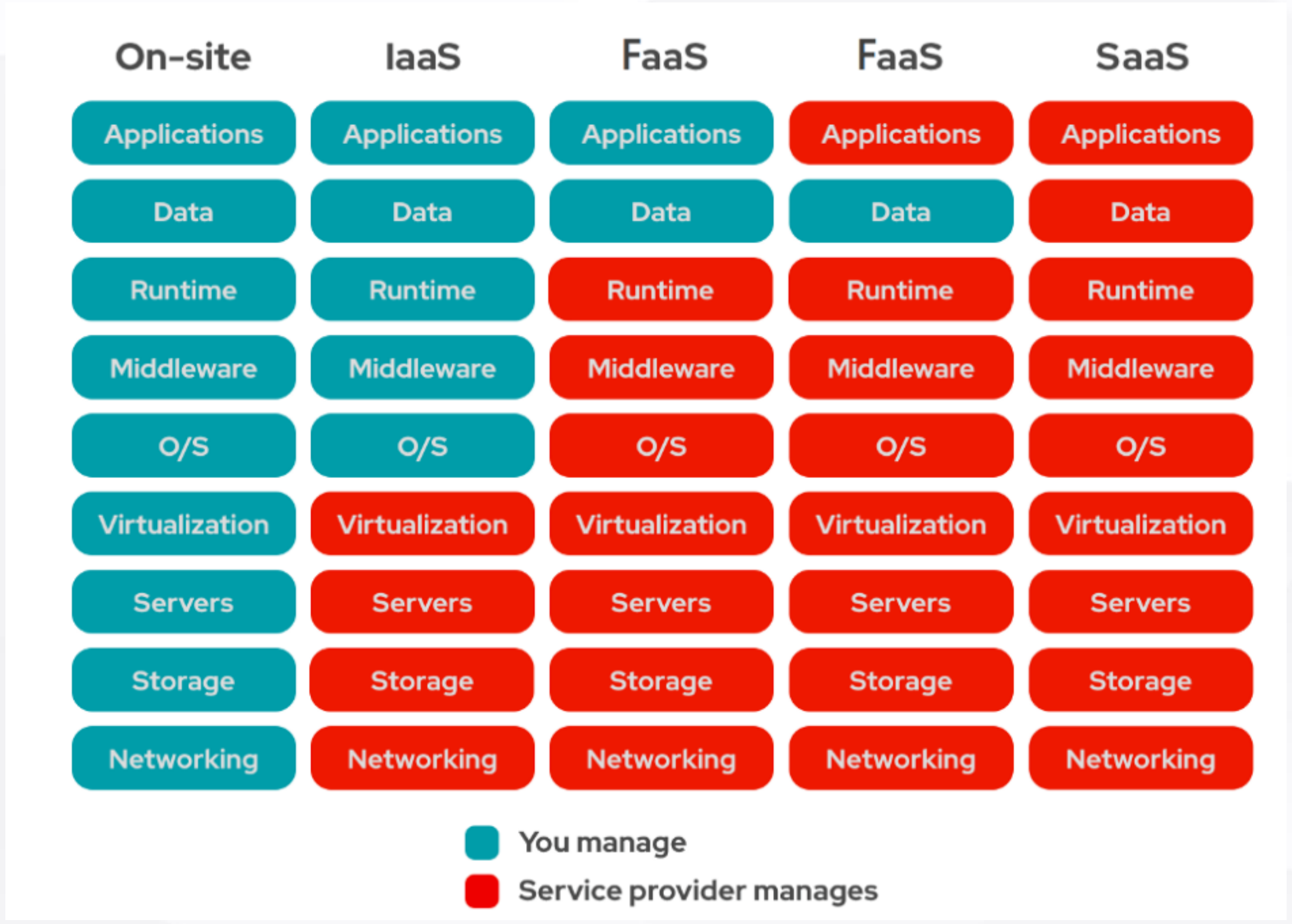
On-site	IaaS	FaaS	FaaS	SaaS
🟦 Applications	🟦 Applications	🟦 Applications	🟥 Applications	🟥 Applications
🟦 Data	🟦 Data	🟦 Data	🟦 Data	🟥 Data
🟦 Runtime	🟦 Runtime	🟥 Runtime	🟥 Runtime	🟥 Runtime
🟦 Middleware	🟦 Middleware	🟥 Middleware	🟥 Middleware	🟥 Middleware
🟦 O/S	🟦 O/S	🟥 O/S	🟥 O/S	🟥 O/S
🟦 Virtualization	🟥 Virtualization	🟥 Virtualization	🟥 Virtualization	🟥 Virtualization
🟦 Servers	🟥 Servers	🟥 Servers	🟥 Servers	🟥 Servers
🟦 Storage	🟥 Storage	🟥 Storage	🟥 Storage	🟥 Storage
🟦 Networking	🟥 Networking	🟥 Networking	🟥 Networking	🟥 Networking

🟦 You manage
🟥 Service provider manages

These 9 categories group services based on their infrastructure level.

Service Models

A service model indicates the number of infrastructure levels which you decide to rent, and which ones you decide to manage.



On-site	IaaS	FaaS	FaaS	SaaS
🟦 Applications	🟦 Applications	🟦 Applications	🟥 Applications	🟥 Applications
🟦 Data	🟦 Data	🟦 Data	🟦 Data	🟥 Data
🟦 Runtime	🟦 Runtime	🟥 Runtime	🟥 Runtime	🟥 Runtime
🟦 Middleware	🟦 Middleware	🟥 Middleware	🟥 Middleware	🟥 Middleware
🟦 O/S	🟦 O/S	🟥 O/S	🟥 O/S	🟥 O/S
🟦 Virtualization	🟥 Virtualization	🟥 Virtualization	🟥 Virtualization	🟥 Virtualization
🟦 Servers	🟥 Servers	🟥 Servers	🟥 Servers	🟥 Servers
🟦 Storage	🟥 Storage	🟥 Storage	🟥 Storage	🟥 Storage
🟦 Networking	🟥 Networking	🟥 Networking	🟥 Networking	🟥 Networking

🟦 You manage
🟥 Service provider manages

IAAS (Infrastructure As A Service)
With IAAS, you only rent the first 4 categories of Services(All machines that perform Computing, Storages, Servers)

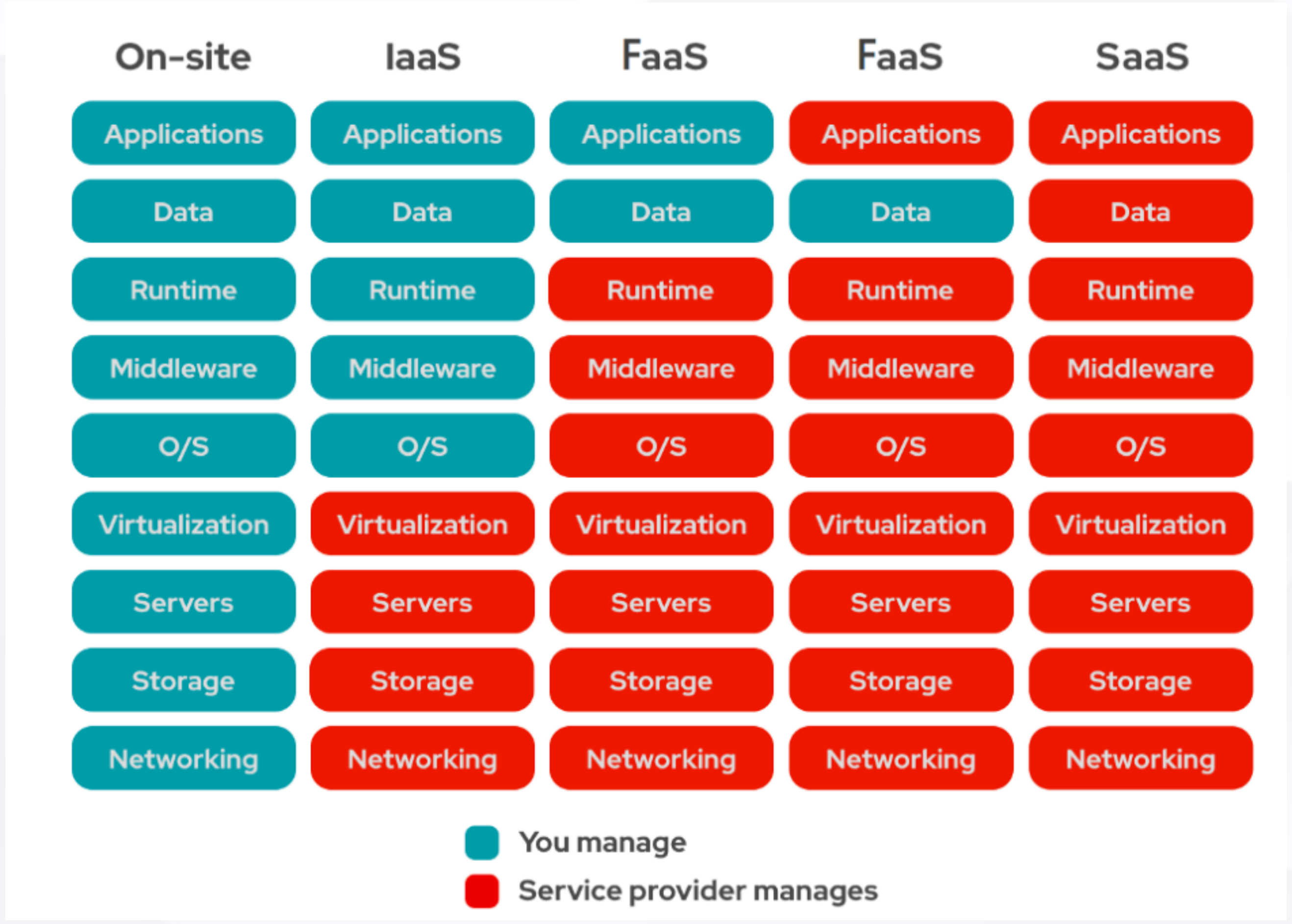
The Cloud itself is an IAAS, which is an overall service that you can rent.

PAAS (Platform As A Service)
With PAAS you rent even more. Because you only need to provide data and application, you can build your own software and launch it on the web, making it available for even millions of people.

PAAS are platforms that you can rent to host your web applications.

Service Models

A service model indicates the number of infrastructure levels which you decide to rent, and which ones you decide to manage.



On-site	IaaS	FaaS	FaaS	SaaS
🟦 Applications	🟦 Applications	🟦 Applications	🟥 Applications	🟥 Applications
🟦 Data	🟦 Data	🟦 Data	🟦 Data	🟥 Data
🟦 Runtime	🟦 Runtime	🟥 Runtime	🟥 Runtime	🟥 Runtime
🟦 Middleware	🟦 Middleware	🟥 Middleware	🟥 Middleware	🟥 Middleware
🟦 O/S	🟦 O/S	🟥 O/S	🟥 O/S	🟥 O/S
🟦 Virtualization	🟥 Virtualization	🟥 Virtualization	🟥 Virtualization	🟥 Virtualization
🟦 Servers	🟥 Servers	🟥 Servers	🟥 Servers	🟥 Servers
🟦 Storage	🟥 Storage	🟥 Storage	🟥 Storage	🟥 Storage
🟦 Networking	🟥 Networking	🟥 Networking	🟥 Networking	🟥 Networking

🟦 You manage

🟥 Service provider manages

FAAS (Function As A Service)
With FAAS, you can run a serverless application. The backend of your website is run with functions, independent pieces of code that serves as a backend.

Especially when you are small, this choice will save you plenty of money.

SAAS (Software As A Service)
With SAAS, you are using a pre-made tool available online. You are not only renting infrastructure, but also a premade application: analytics and graphing services are good examples.

02

Emergency of Serverless Computing

What is Serverless Computing?

Serverless = FaaS + BaaS

FaaS (Function as a Service) offerings：

Cloud Functions (the core of Serverless Computing)

- short-running: functions are expected to complete in a short time period
- stateless: functions are stateless and only describe the application logic for task processing

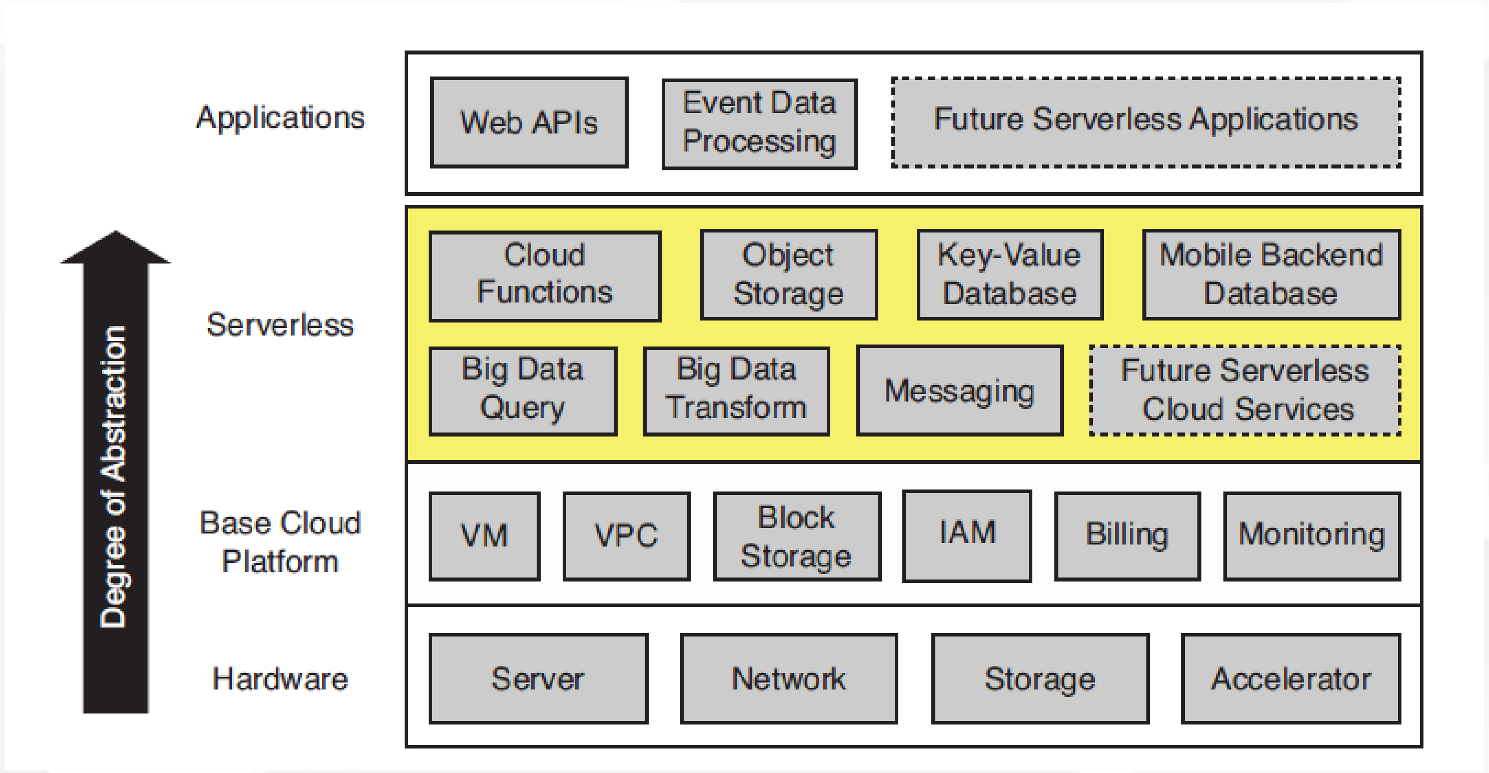
BaaS (Backend as a Service) offerings：

specialized serverless frameworks that cater to specific application requirements provided by cloud platforms

- object storage
- database
- messaging
- big data services
- ...

What is Serverless Computing?

Architecture of the serverless cloud



The serverless layer sits between applications and the base cloud platform, simplifying cloud programming.

Cloud functions (FaaS) provide general compute and are complemented by an ecosystem of specialized **Backend as a Service (BaaS)** offerings such as object storage, databases, or messaging.

Characteristics

A number of properties that a service should have in order to call it a **serverless service**:

(1) **Transparency**: The execution environment must be transparentto the customer. The processing node, the virtual machine, the container, its operating system and etc. are all hidden to the customer.

(2) **Auto-scaling**: The provider should provide an auto-scaling service i.e., the resources should be made available to the customer instantly per demand.

(3) **Pay-as-you-go**: The billing mechanism should only reflect the amount of resources the customer actually used i.e., pay-as-you-go billing model.

(4) **Event-driven**: The provider does its best effort to complete the customer's task as soon as it receives the request and the execution duration is bounded.

(5) **Function-level management:** The basic elements in serverless services are functions. The functions are not transparent to the provider. The provider knows their data dependencies, dependencies to external libraries, run-time environments and state during and after execution.

Characteristics

Three critical distinctions between **serverless** and **serverful** (traditional approach) computing:

1. **Decoupled computation and storage**
The storage and computation scale separately and are provisioned and priced independently. In general, the storage is provided by a separate cloud service and the computation is stateless.

2. **Executing code without managing resource allocation**
Instead of requesting resources, the user provides a piece of code and the cloud automatically provisions resources to execute that code.

3. **Paying in proportion to resources used instead of for resources allocated**
Billing is by some dimension associated with the execution, such as execution time, rather than by a dimension of the base cloud platform, such as size and number of VMs allocated.

The aim and opportunity in serverless computing is to give cloud programmers benefits similar to those in the transition to high-level programming languages.

Pros and Cons

Pros

For developers:

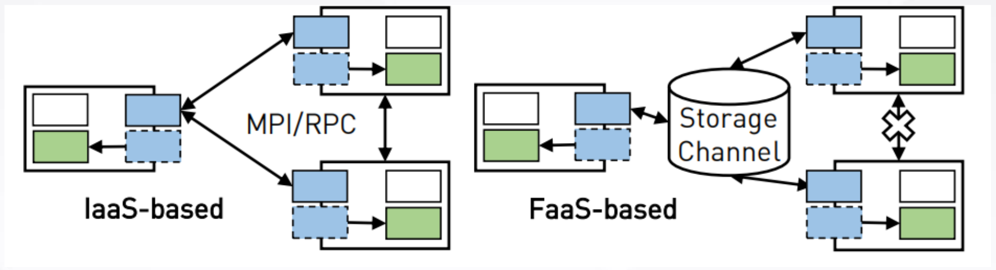
- Cost saving.
- No worrying about deployment and provision.
- Increased programming productivity.

For cloud providers:

- More control over infrastructures.
- Utilize less popular computers.
- Promote business growth.

Cons

- Startup latency.
- Short-lived execution time.
- No direct communication.



- Limited resource (e.g. CPU, memory)
- No specialized hardware.

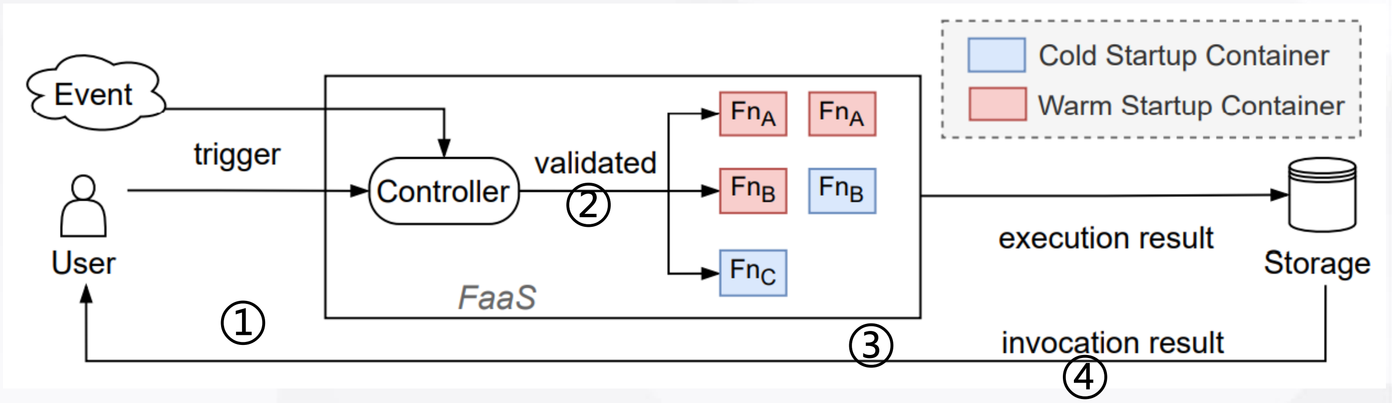
Serverless Applications

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
- ...

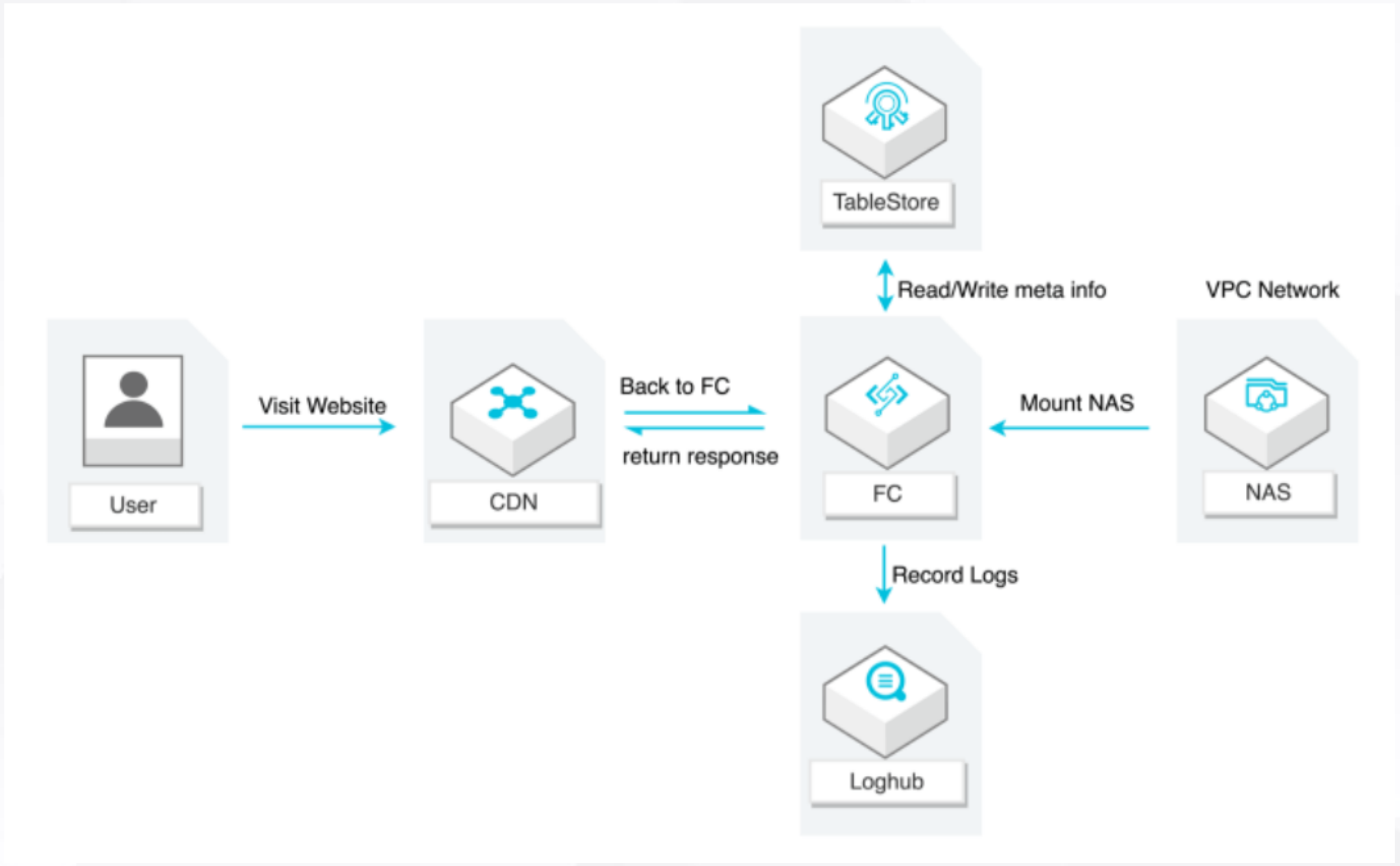
Serverless Applications

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



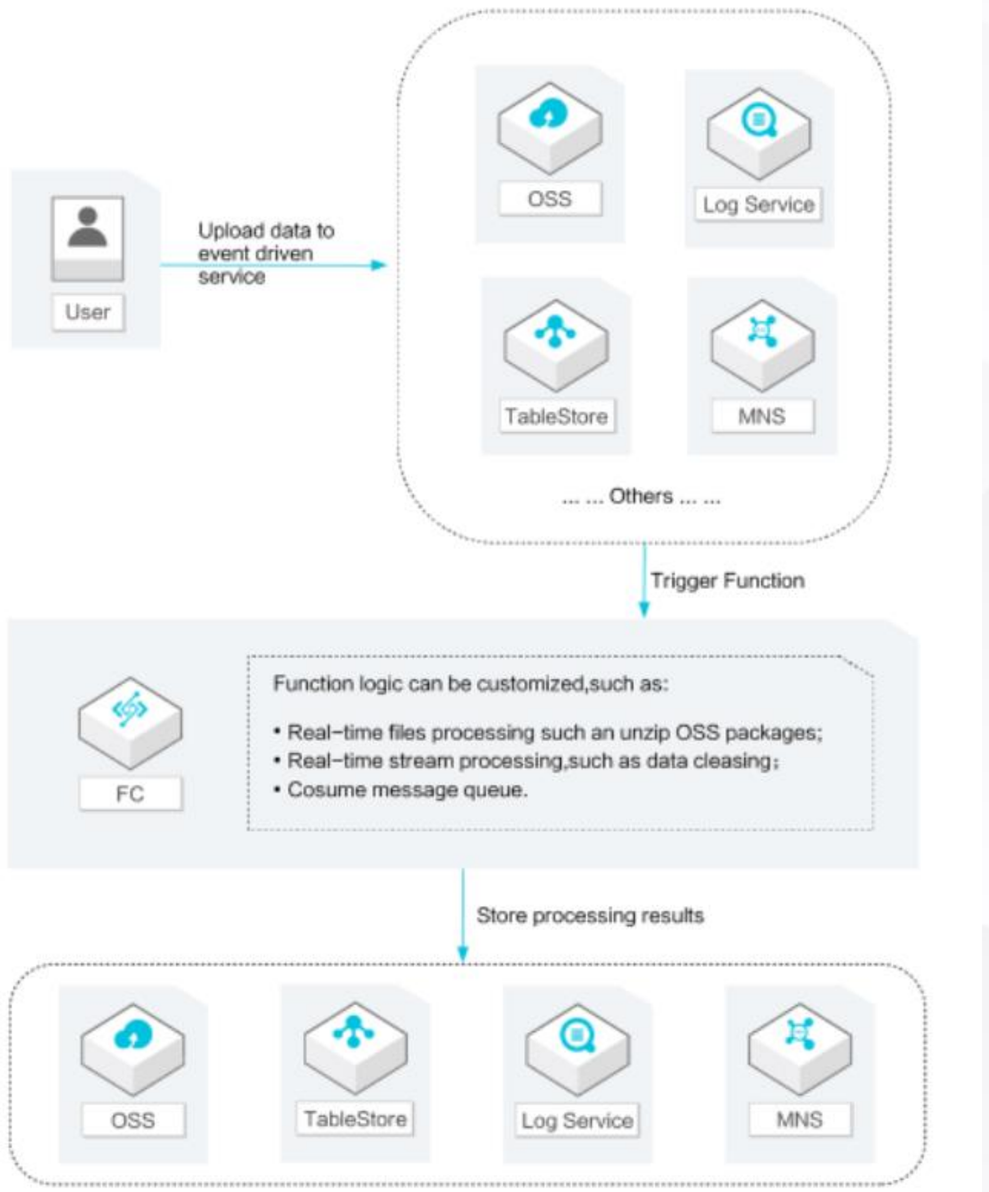
Serverless Applications

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


User
Upload data to event driven service
OSS
Log Service
TableStore
MNS
... ... Others ... ...
Trigger Function
FC
Function logic can be customized,such as:
• Real-time files processing such an unzip OSS packages;
• Real-time stream processing,such as data cleasing;
• Cosume message queue.
Store processing results
OSS
TableStore
Log Service
MNS

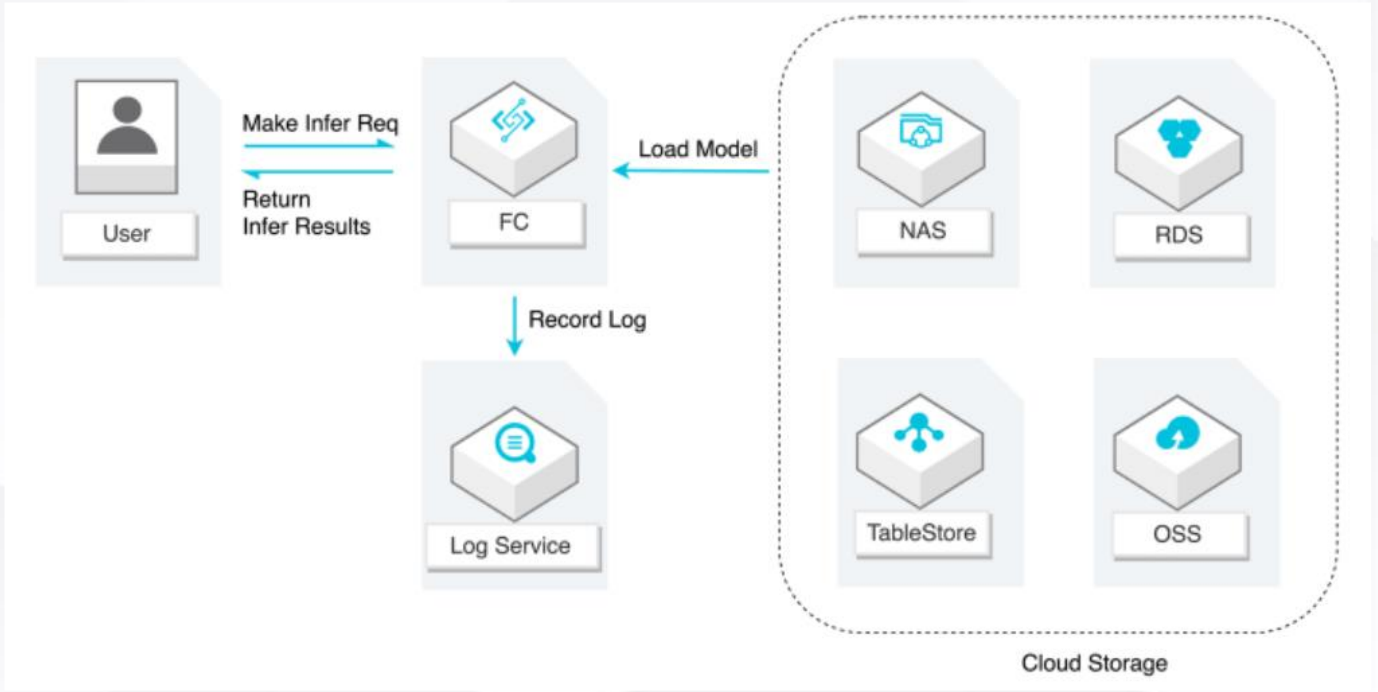
Serverless Applications

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



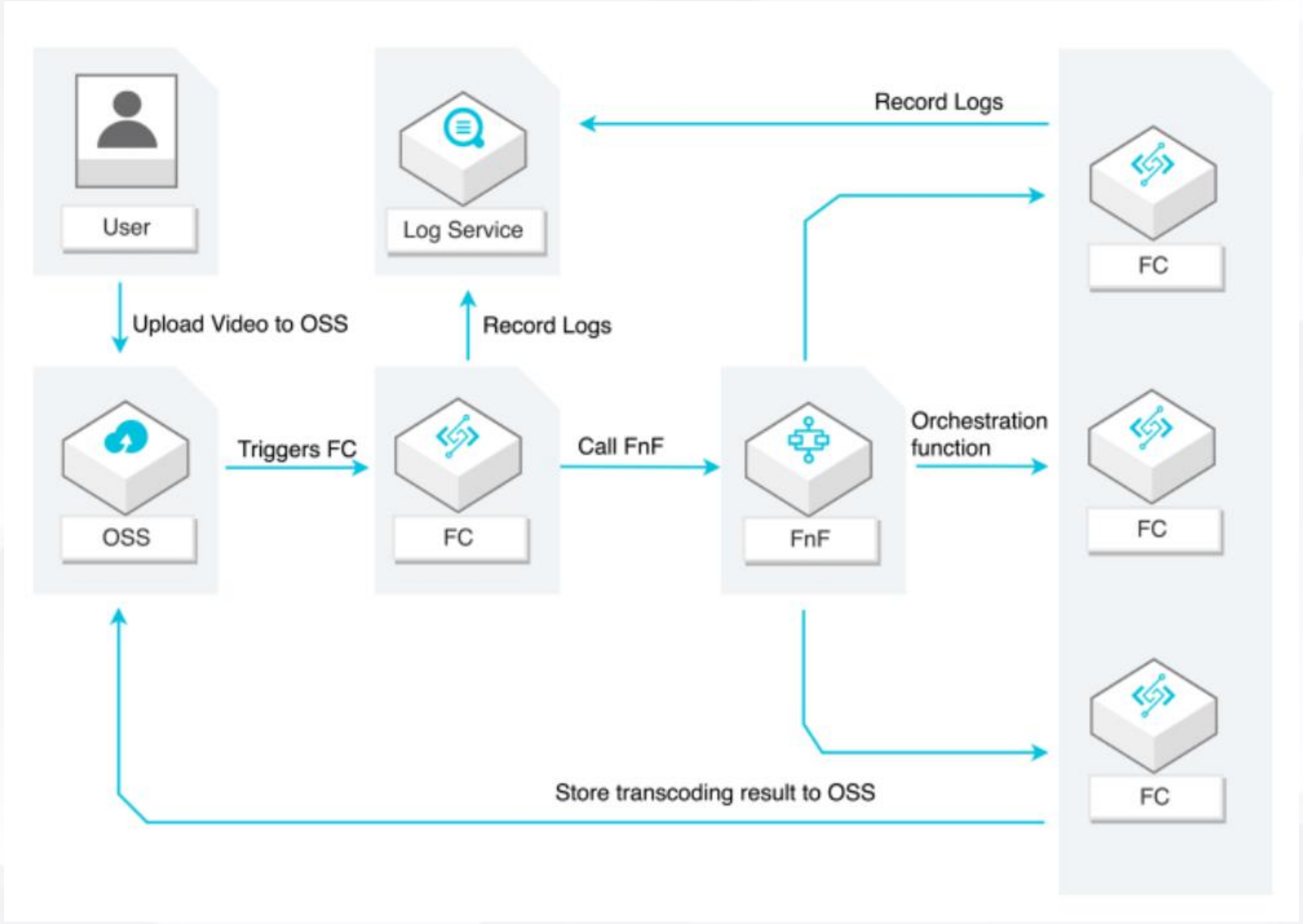
Serverless Applications

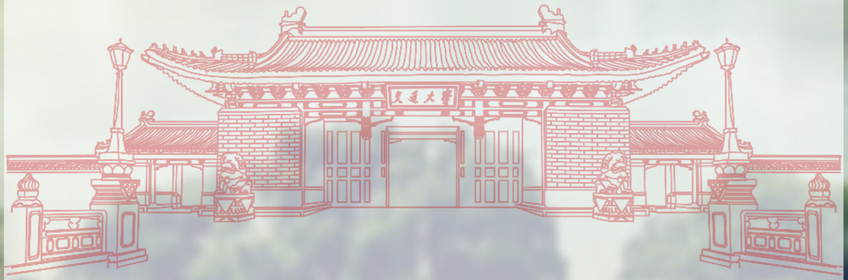
Case 4: Video Transcoding

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

Introduction to Serverless Platforms

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

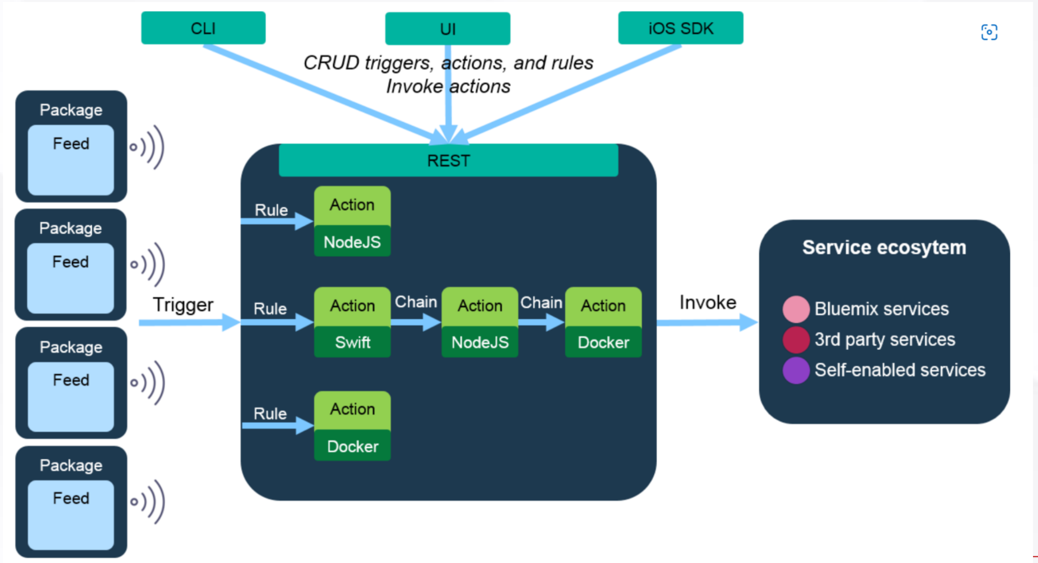
- OpenWhisk
- OpenFaaS
- Kubeless
- Knative
- Fission
- Nuclio
- ...

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

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



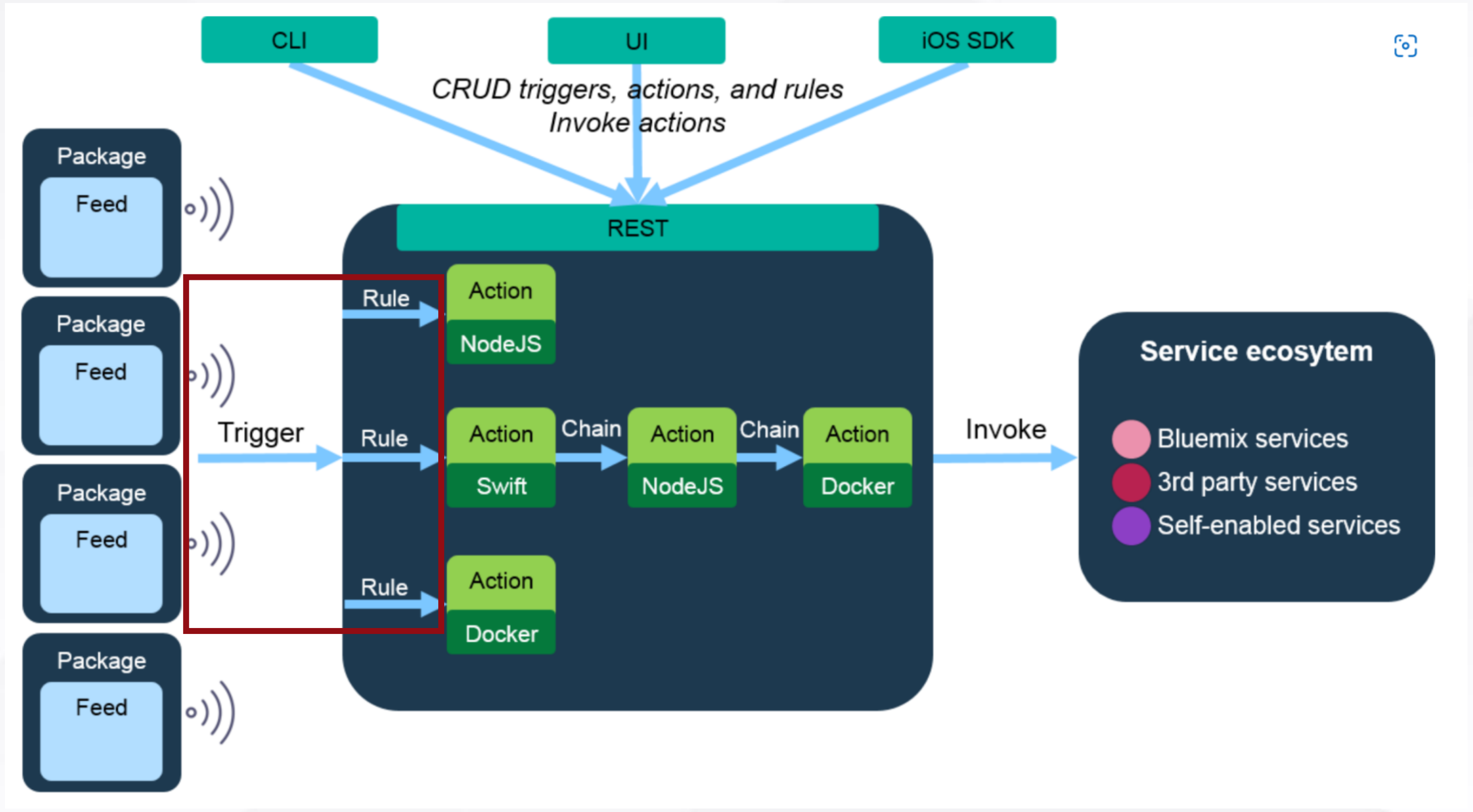
the high-level OpenWhisk architecture

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

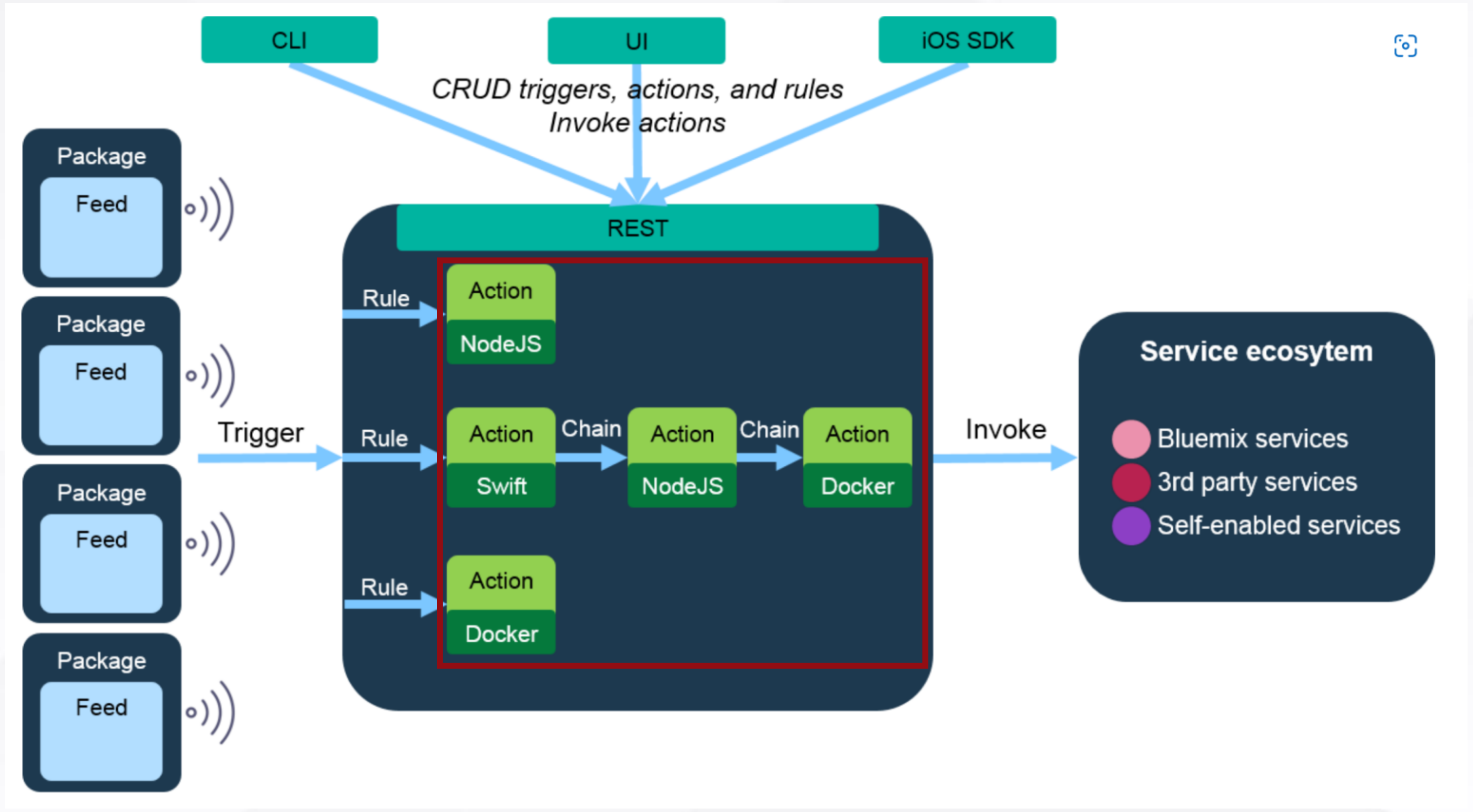
OpenWhisk

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

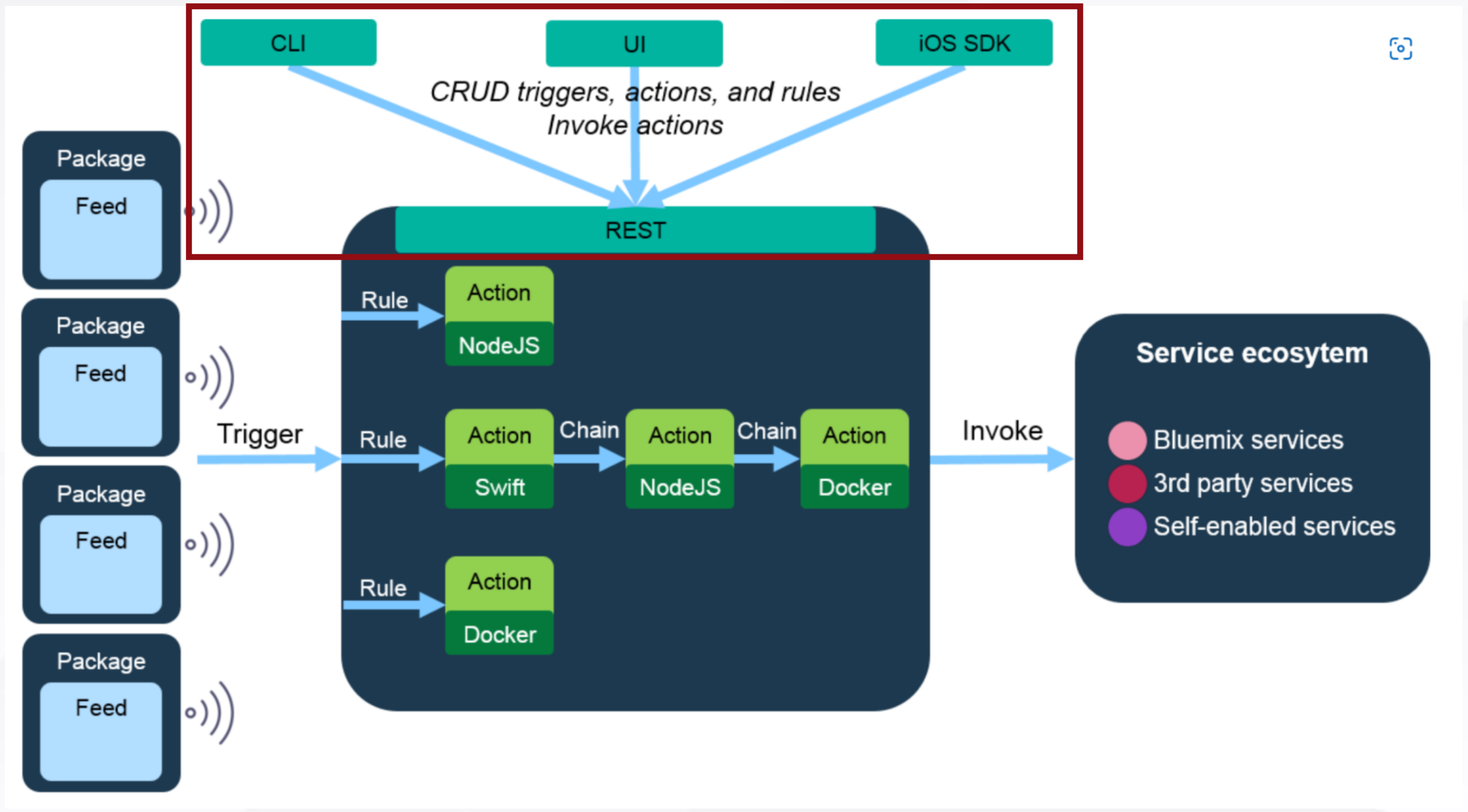
OpenWhisk

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.

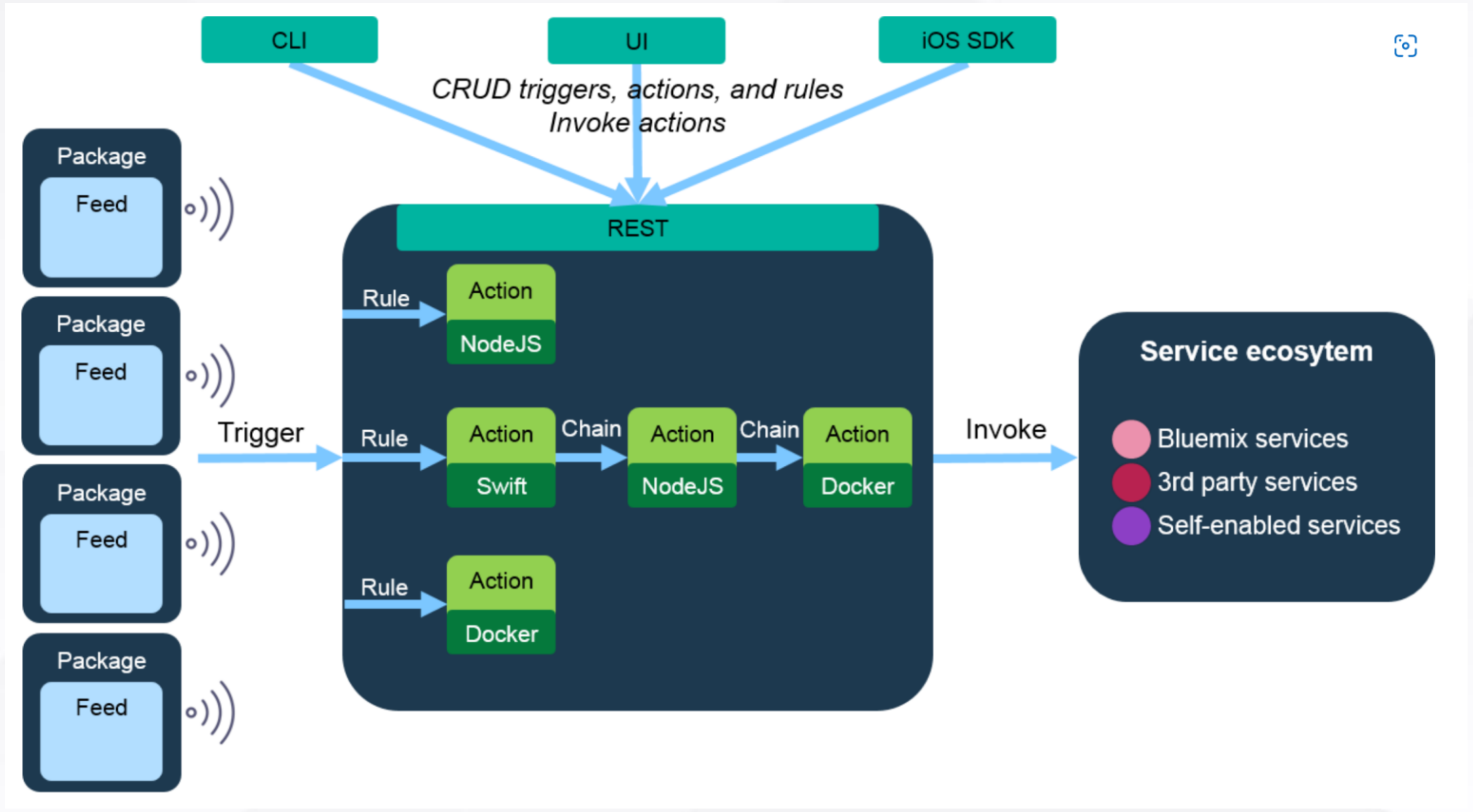


the high-level OpenWhisk architecture

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.



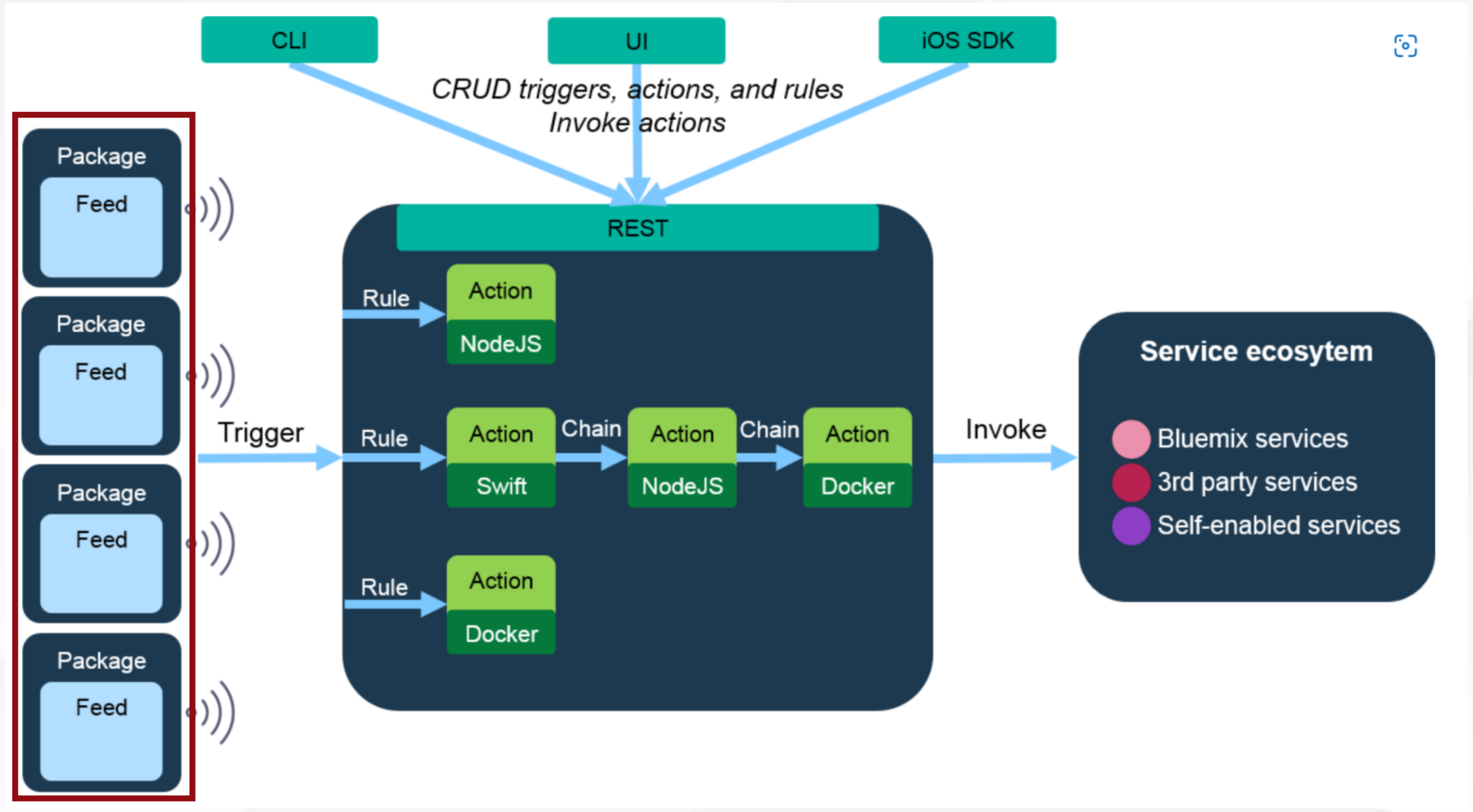
the high-level OpenWhisk architecture

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.



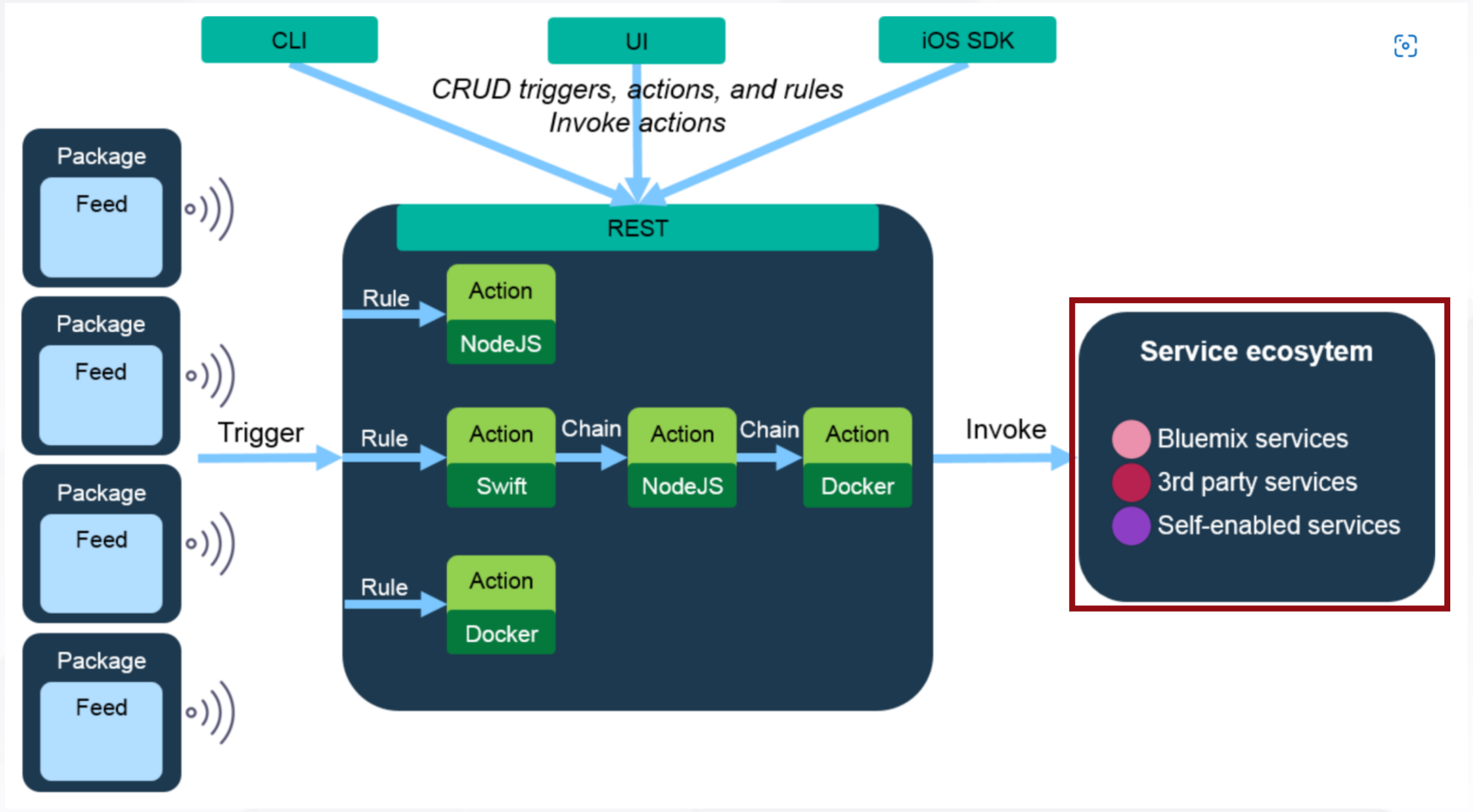
the high-level OpenWhisk architecture

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

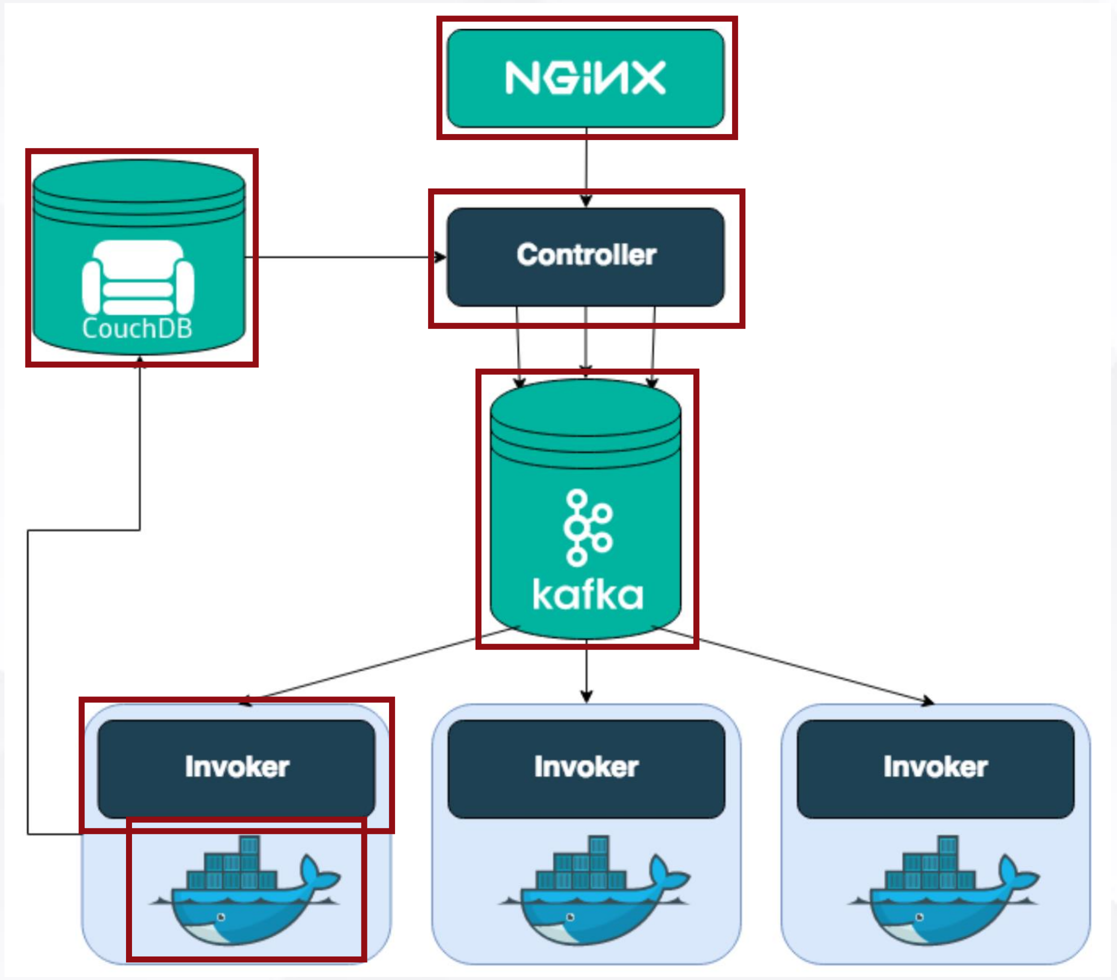
Being an open-source project, OpenWhisk stands on the shoulders of giants, including

- **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

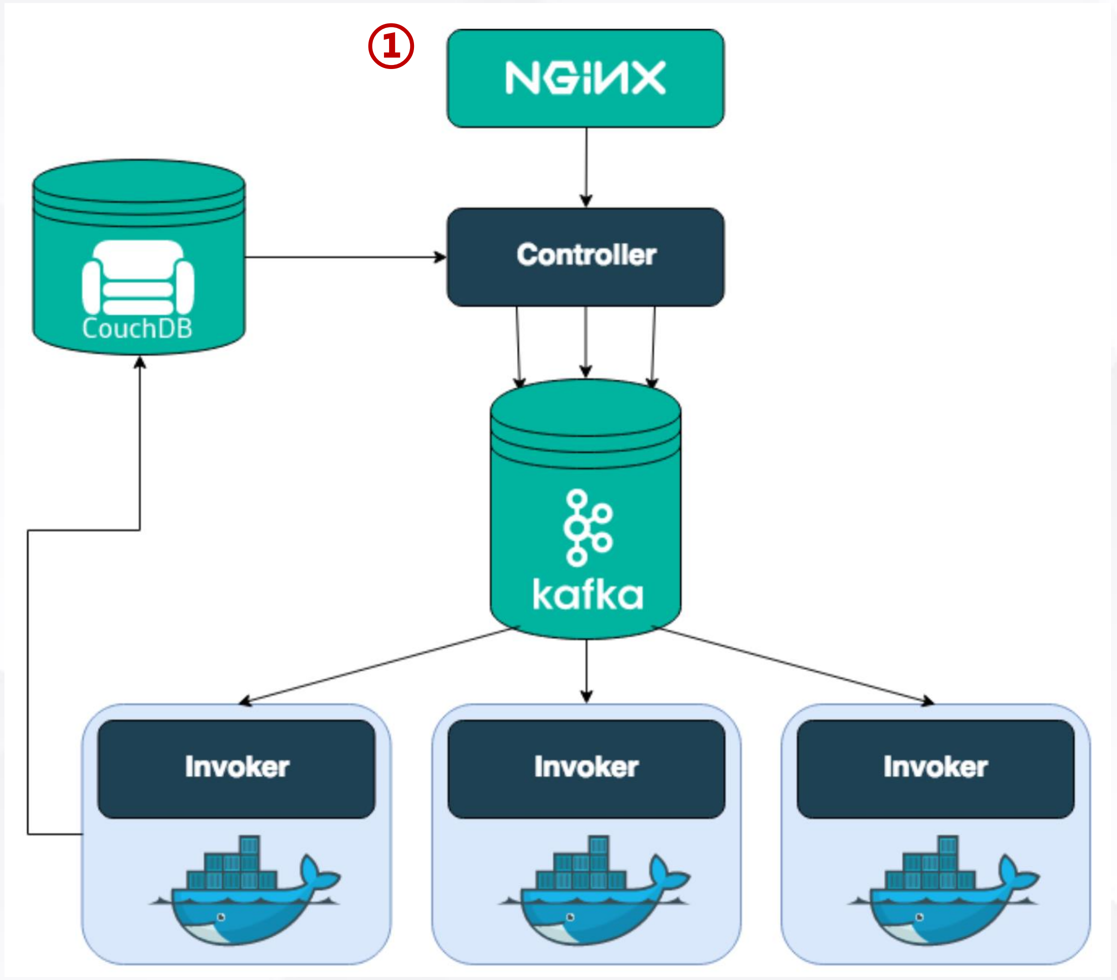
① Entering the system: nginx

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

OpenWhisk

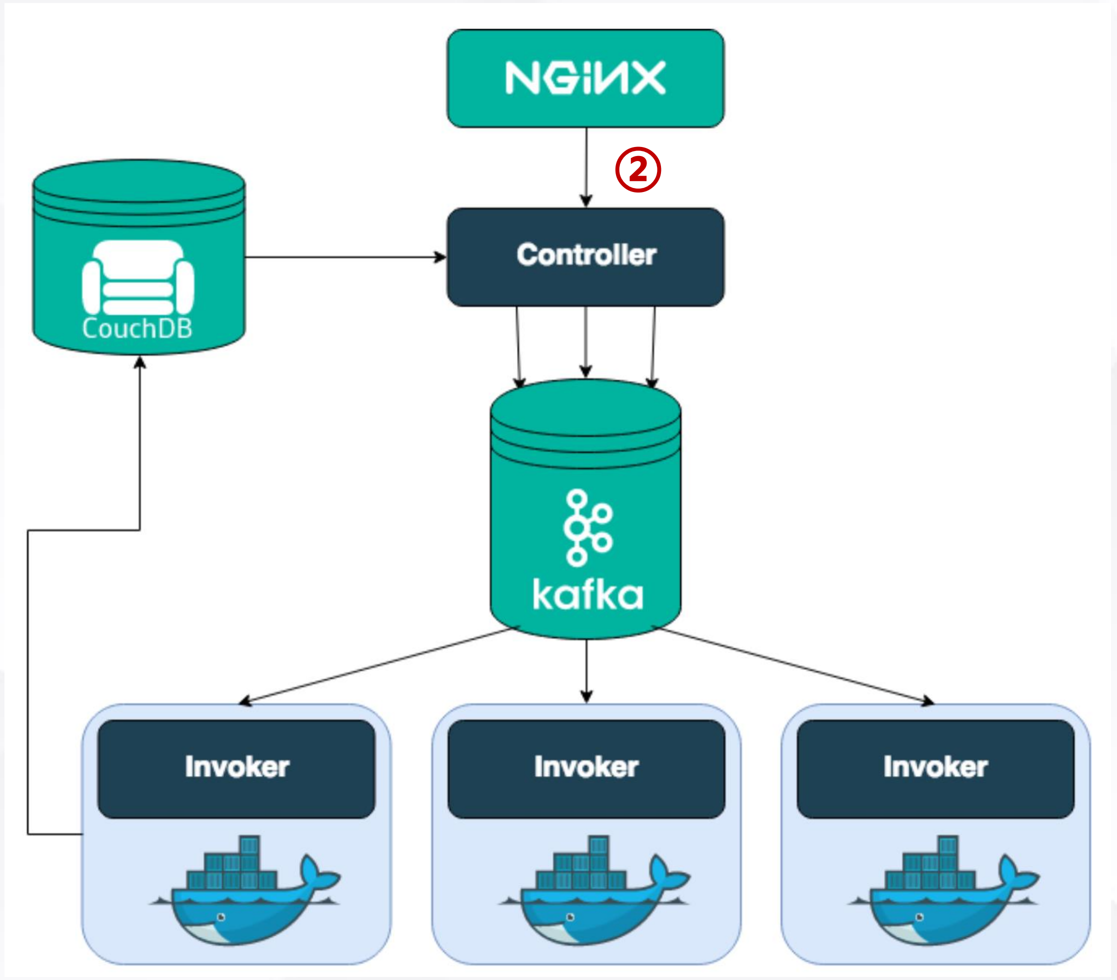
② Entering the system: Controller

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

OpenWhisk

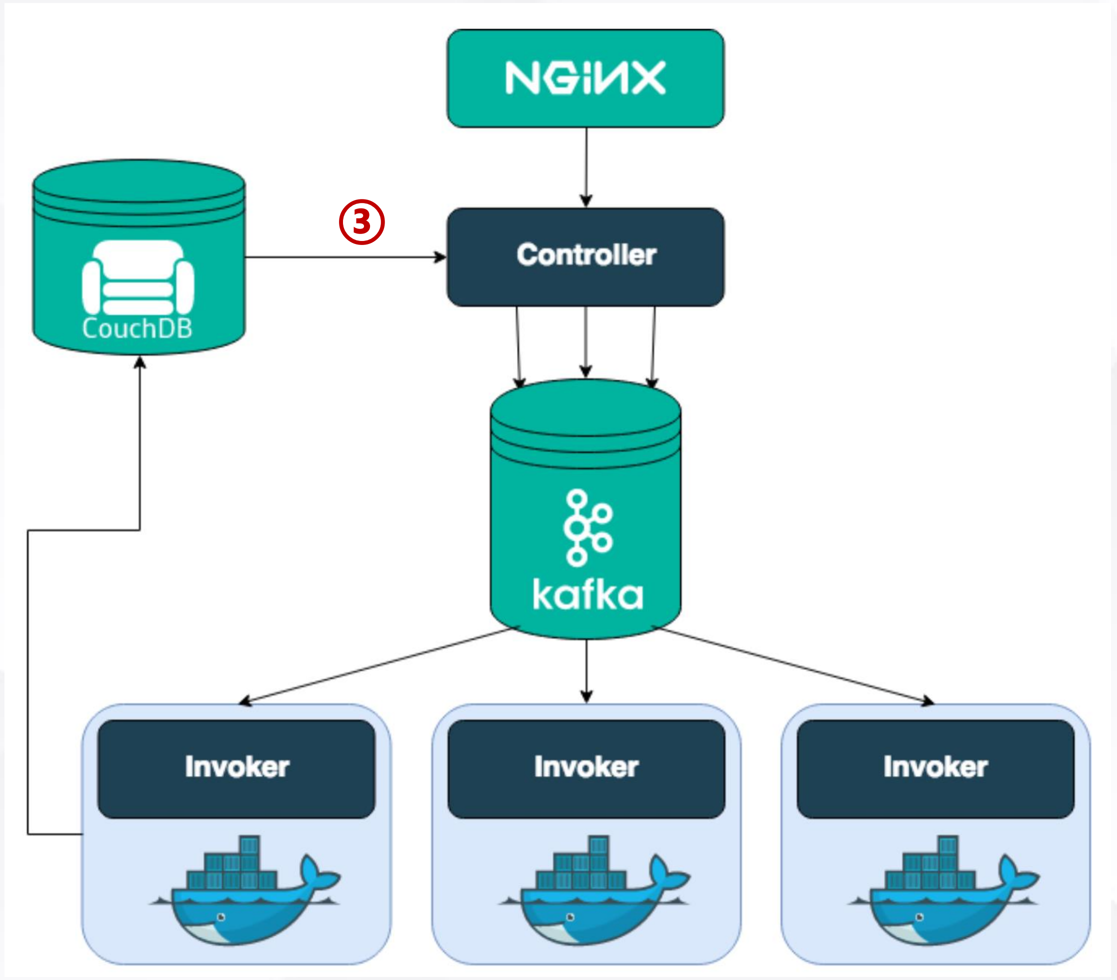
③ Authentication and Authorization: CouchDB

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

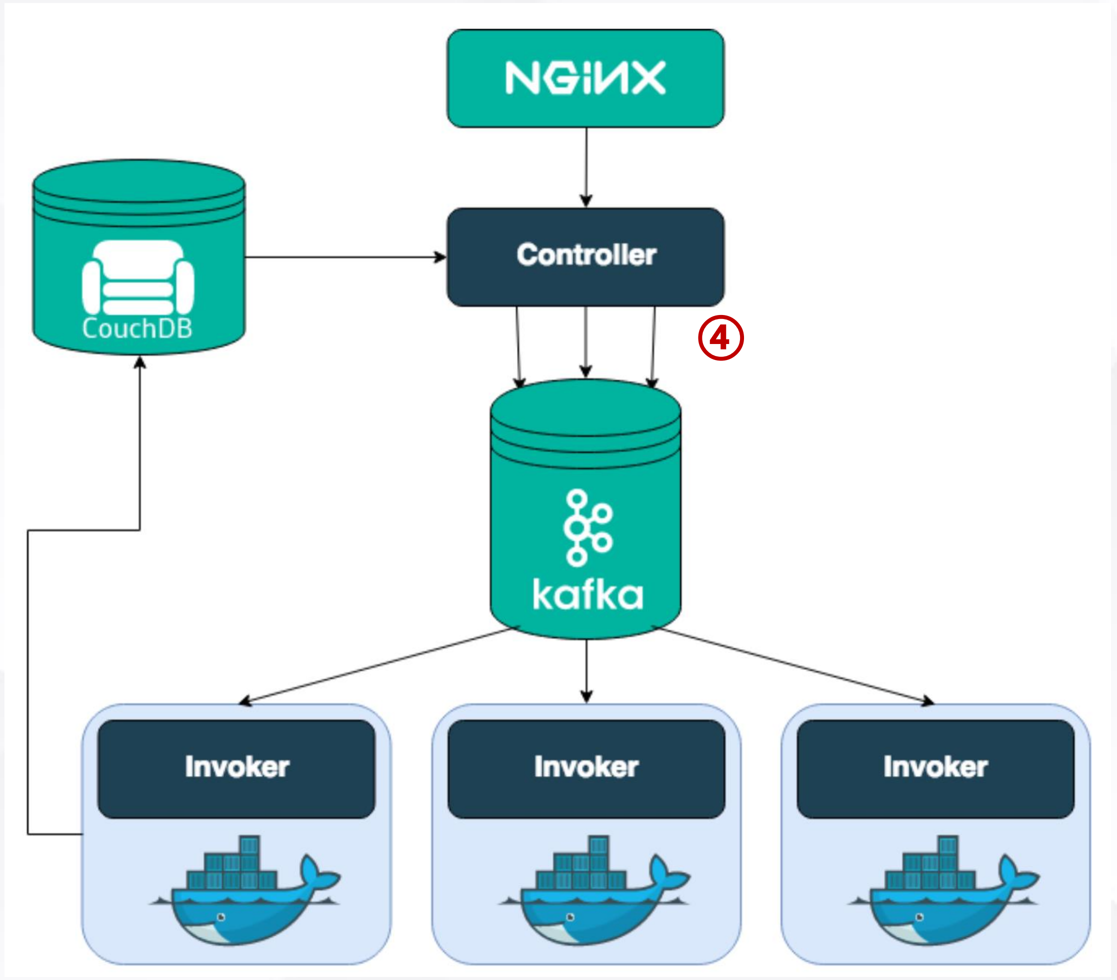
OpenWhisk

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

OpenWhisk

⑤ Please form a line: Kafka

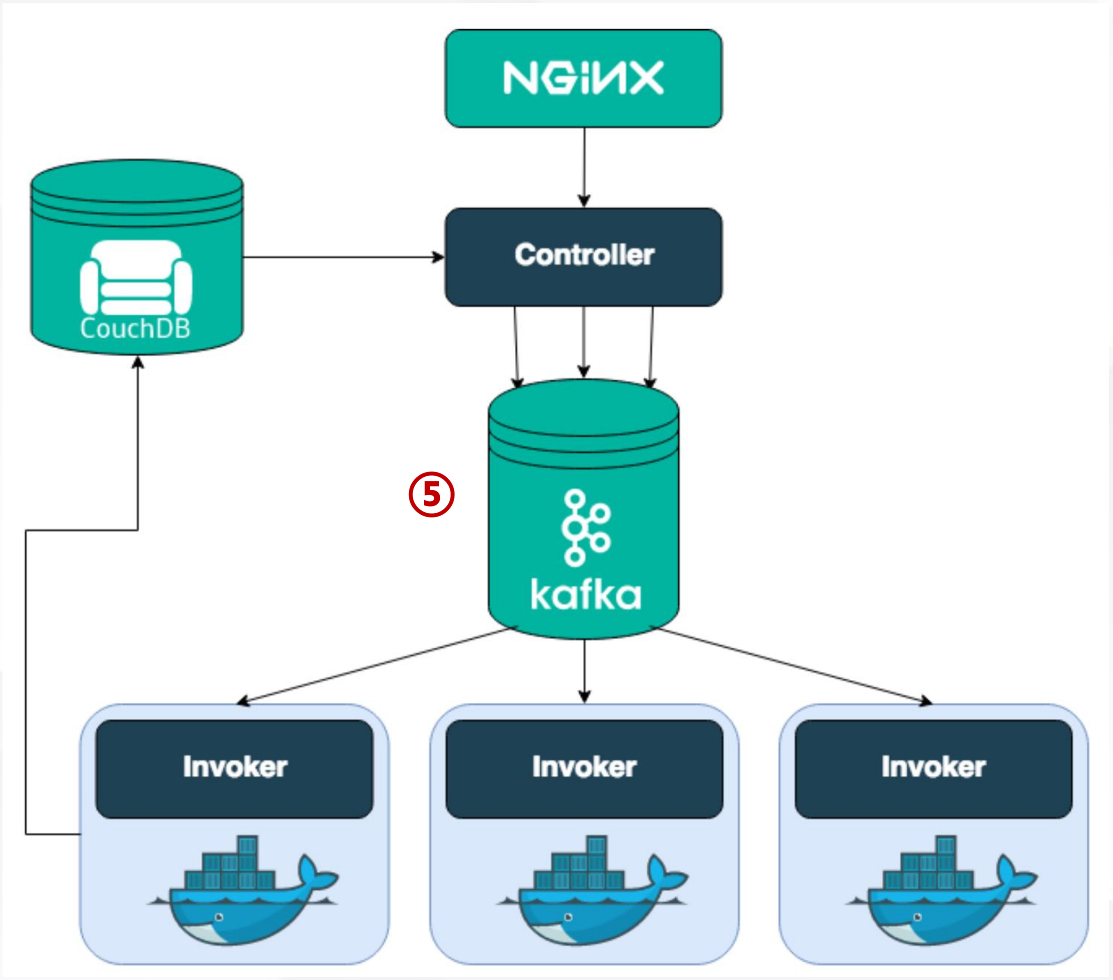
From now on, mainly two bad things can happen to the invocation request sent in:

- 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

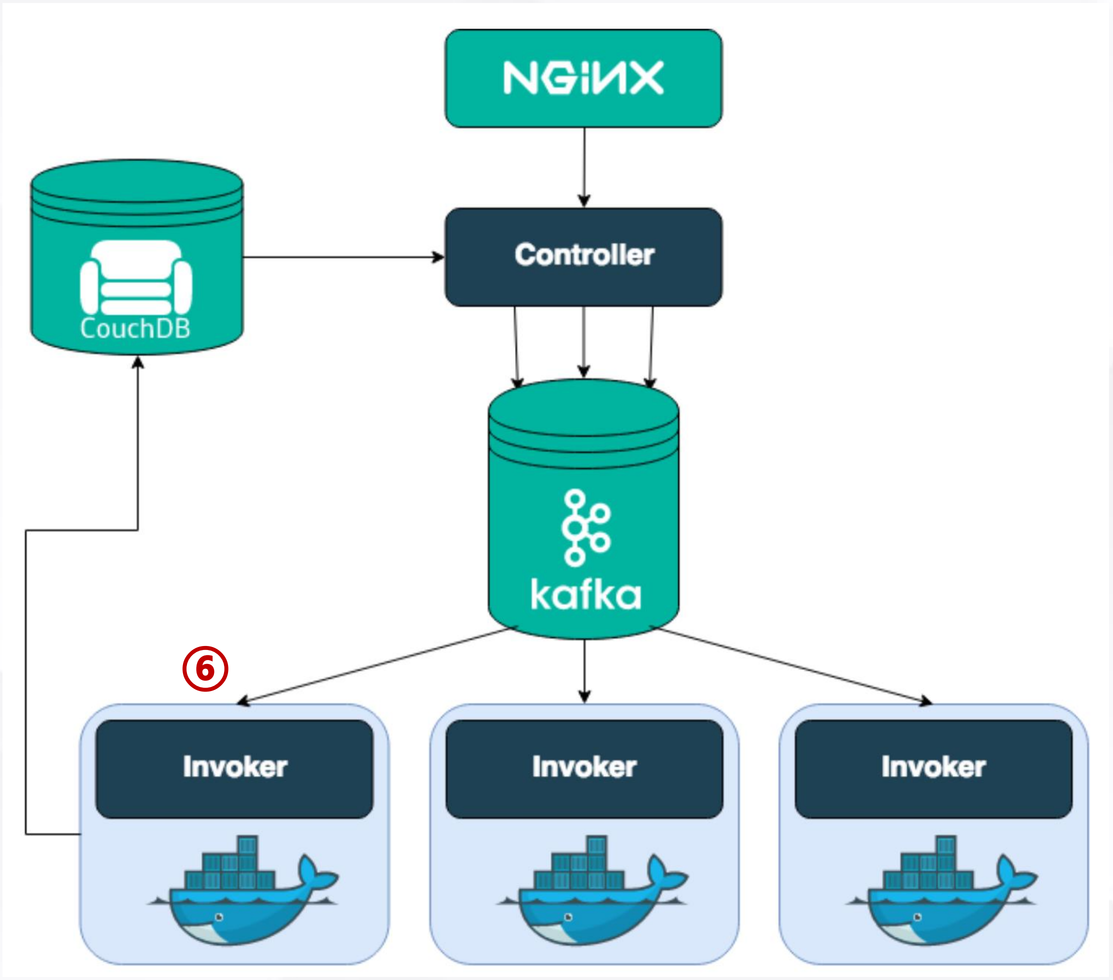
⑥ Actually invoking the code already: Invoker

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

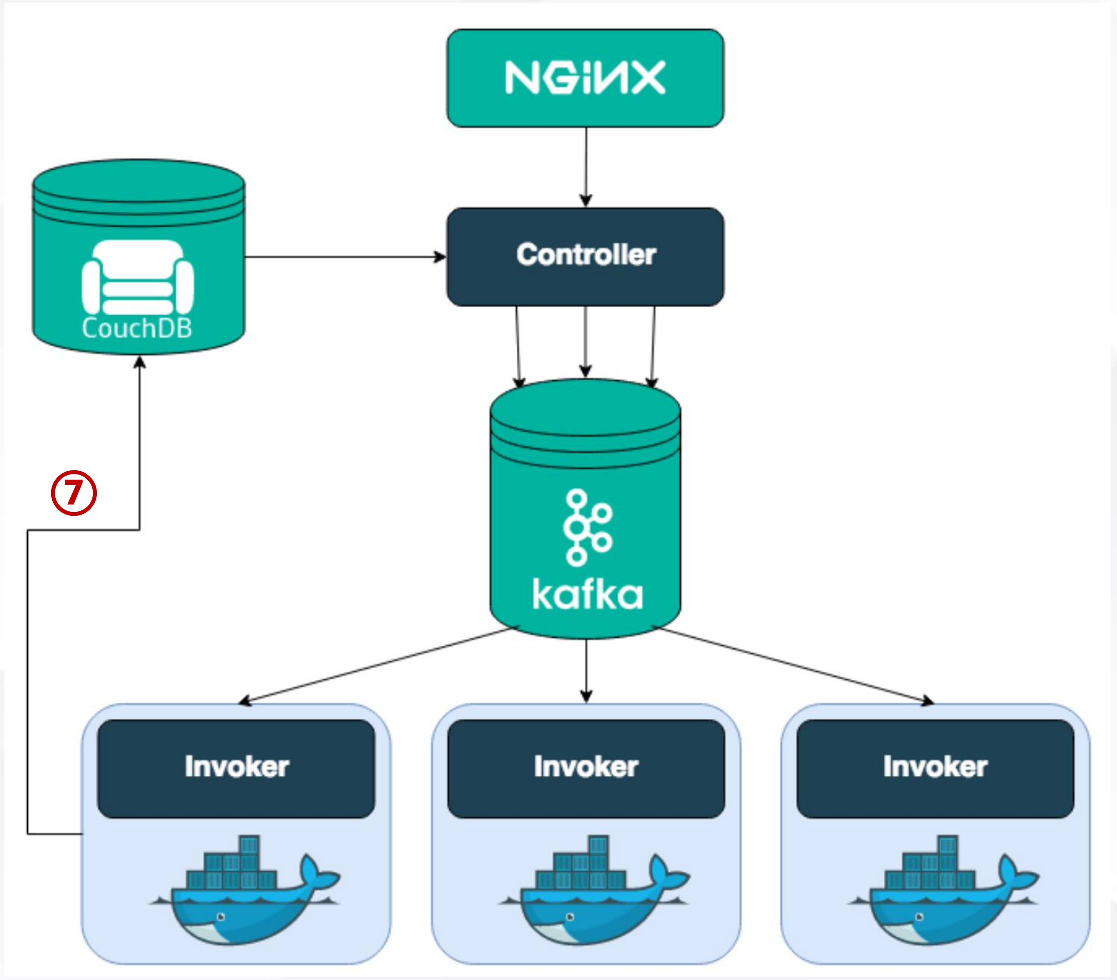
OpenWhisk

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

Limitations of Today’s Serverless

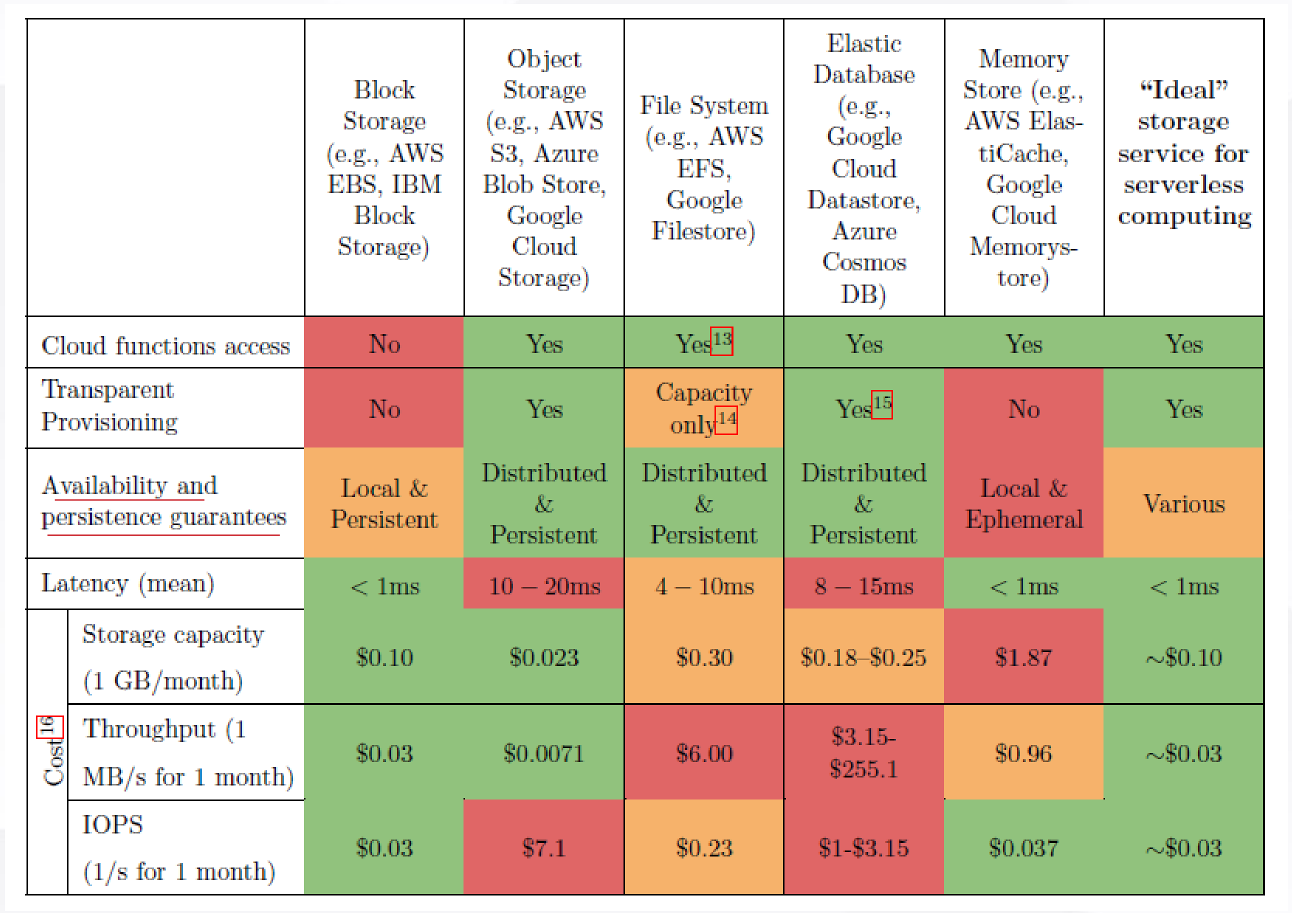
Limitations of Serverless Computing Platforms

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 $\xrightarrow{\text{difficult to support}}$ 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

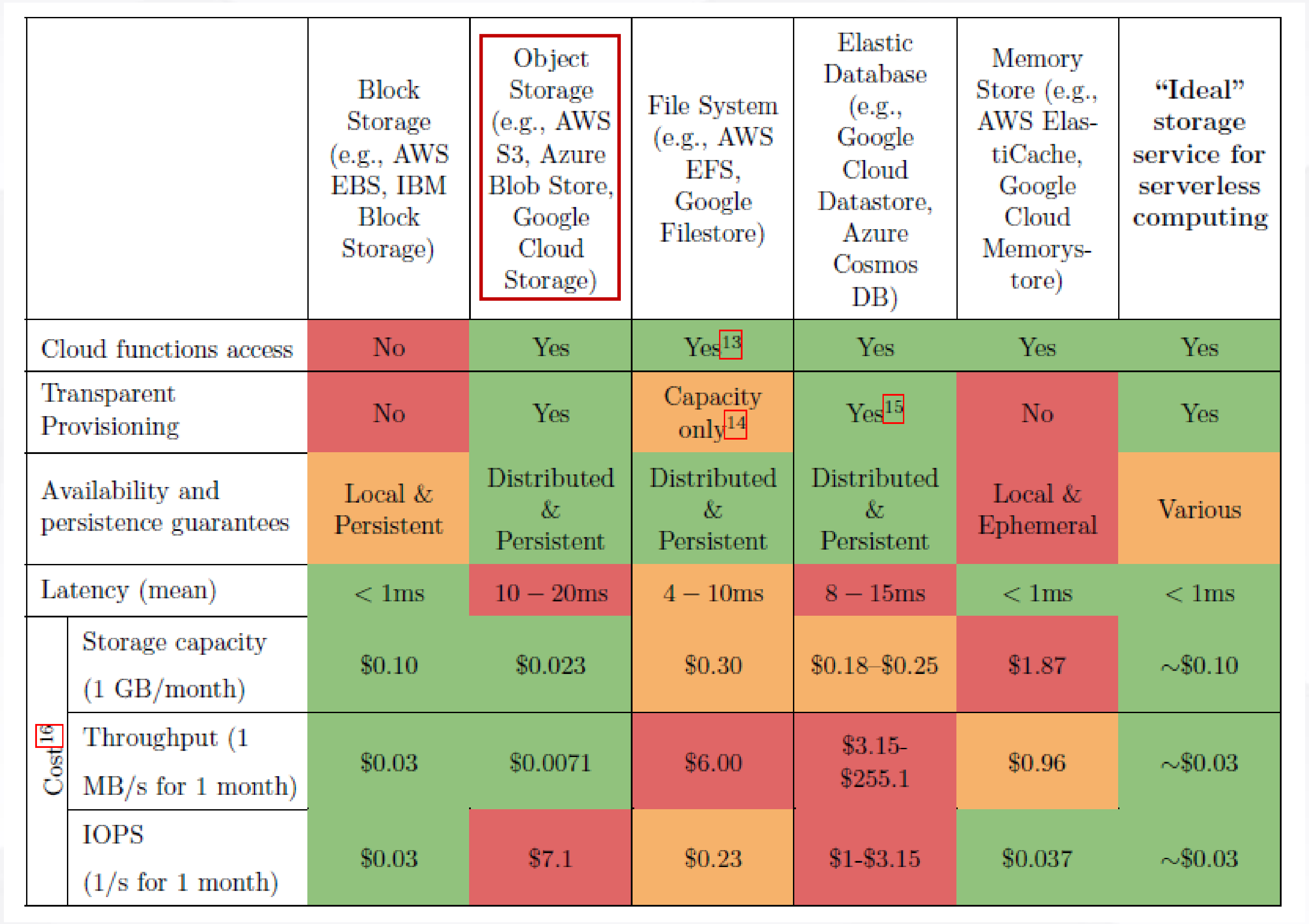
Persistence and availability guarantees describe how well the system tolerates failures:

- **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



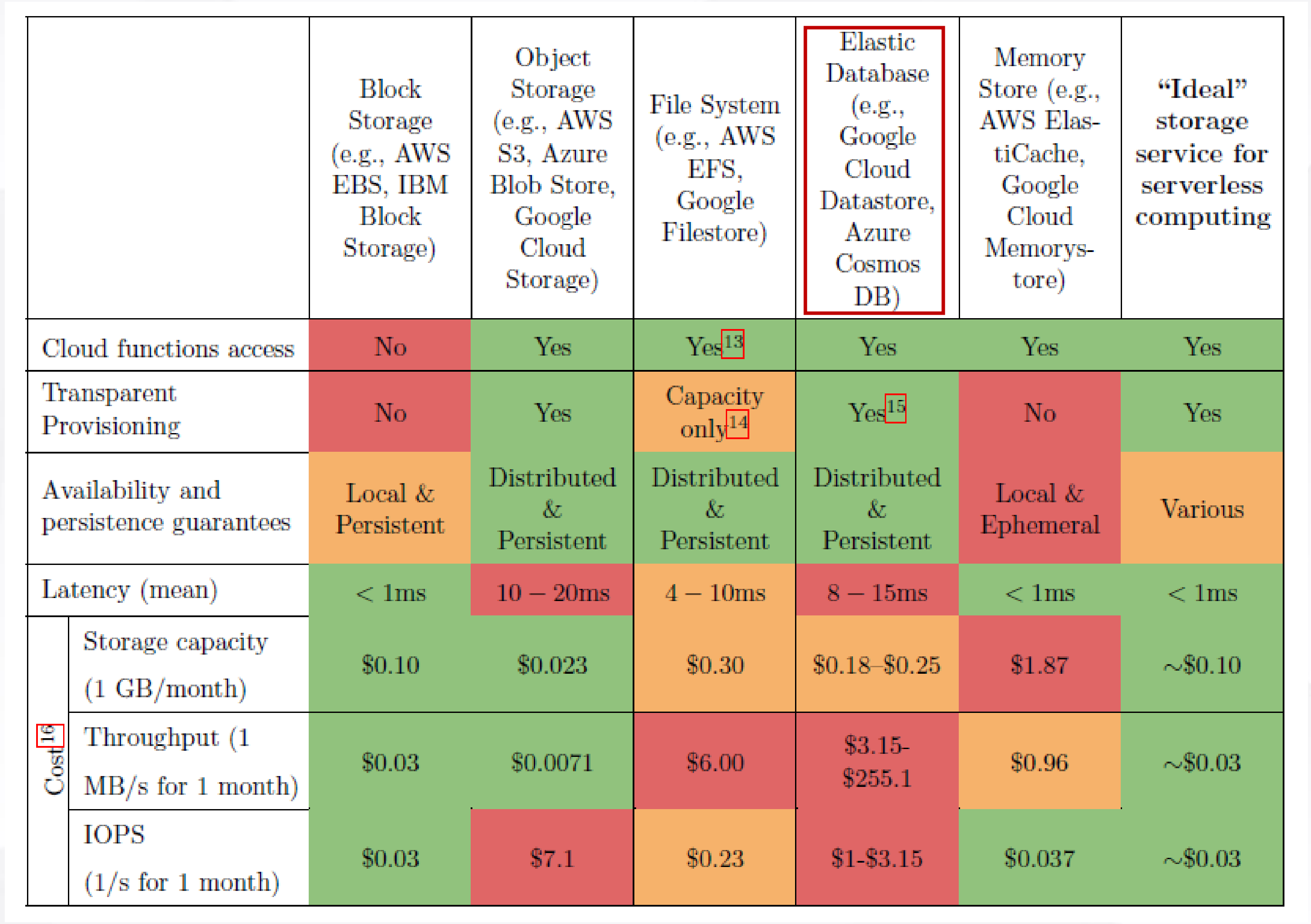
		Block Storage (e.g., AWS EBS, IBM Block Storage)	Object Storage (e.g., AWS S3, Azure Blob Store, Google Cloud Storage)	File System (e.g., AWS EFS, Google Filestore)	Elastic Database (e.g., Google Cloud Datastore, Azure Cosmos DB)	Memory Store (e.g., AWS ElasticCache, Google Cloud Memorystore)	"Ideal" 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



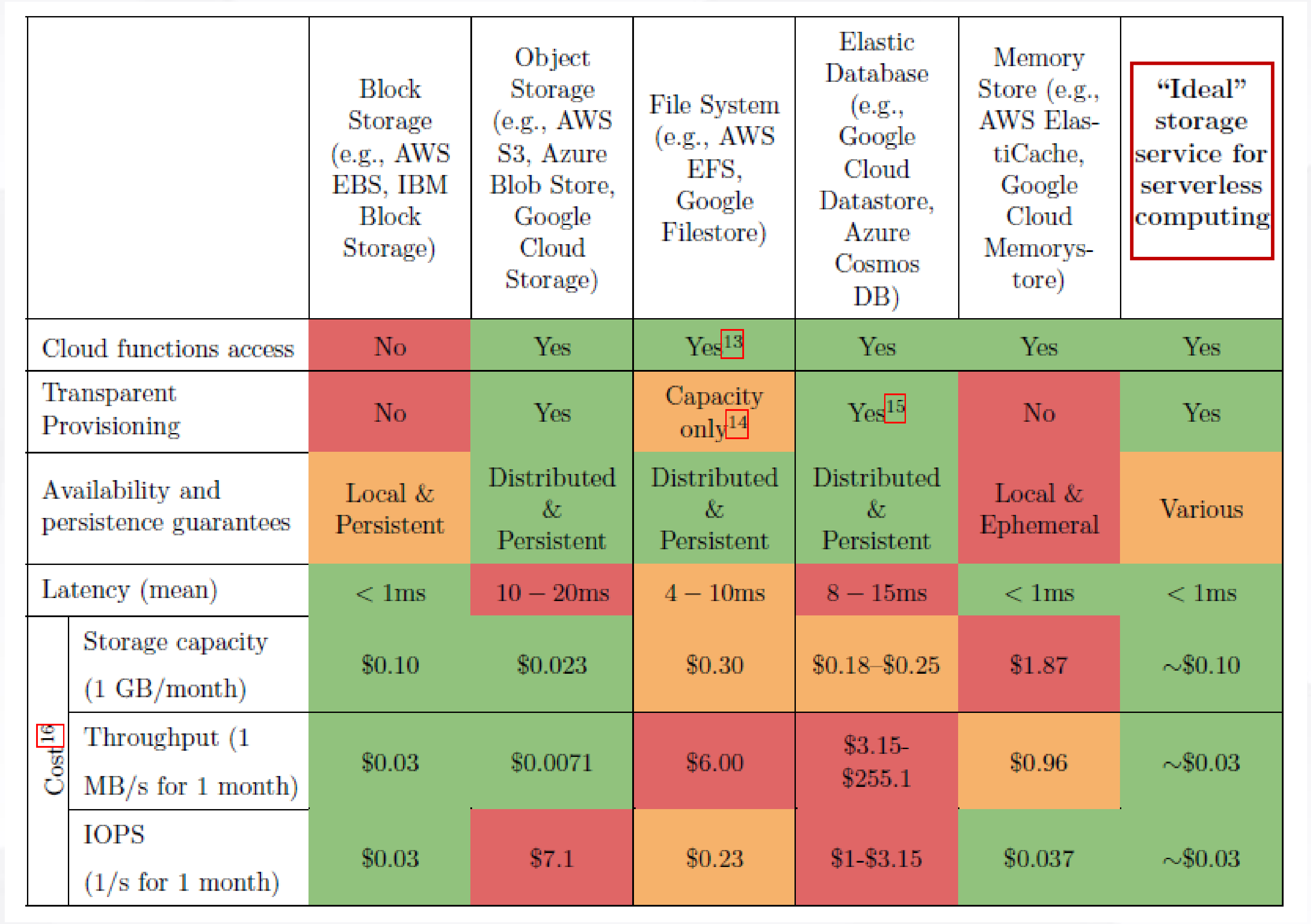
		Block Storage (e.g., AWS EBS, IBM Block Storage)	Object Storage (e.g., AWS S3, Azure Blob Store, Google Cloud Storage)	File System (e.g., AWS EFS, Google Filestore)	Elastic Database (e.g., Google Cloud Datastore, Azure Cosmos DB)	Memory Store (e.g., AWS ElasticCache, Google Cloud Memorystore)	"Ideal" 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	$10-20$ms	$4-10$ms	$8-15$ms	< 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

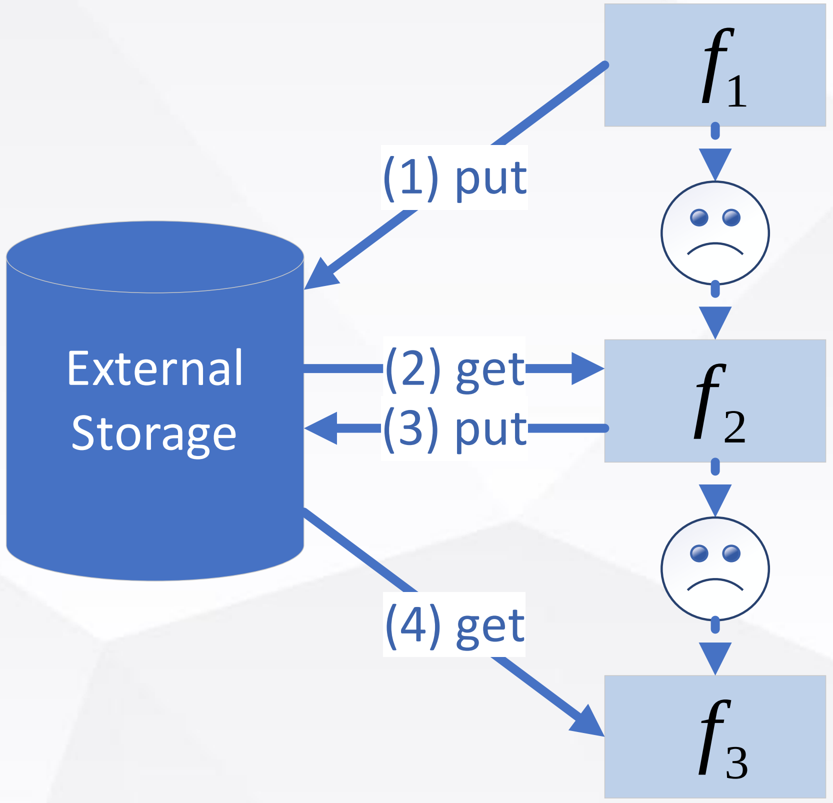
Limitation 2: Coordination

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.

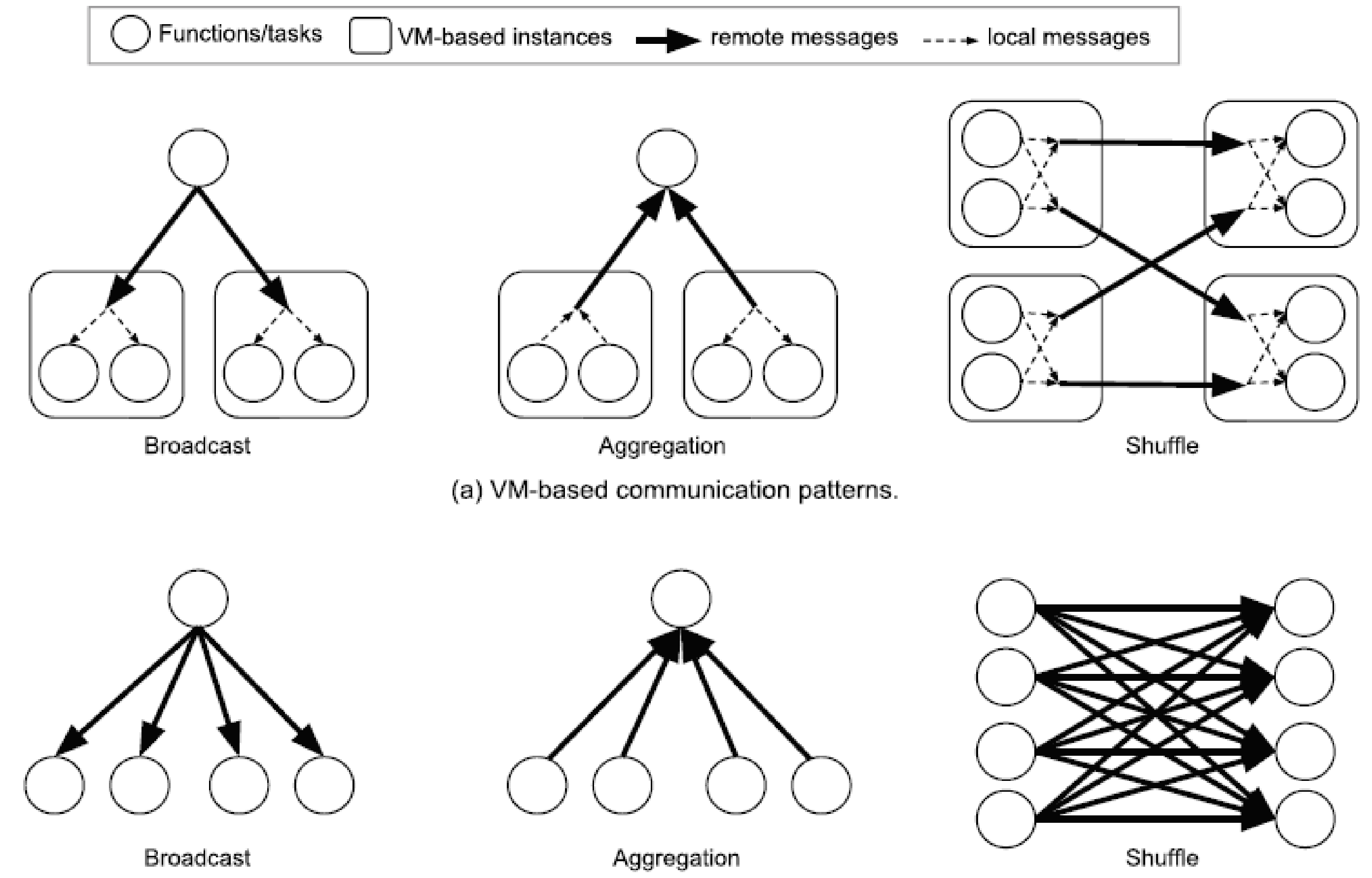


Limitation 3: Communication

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**.



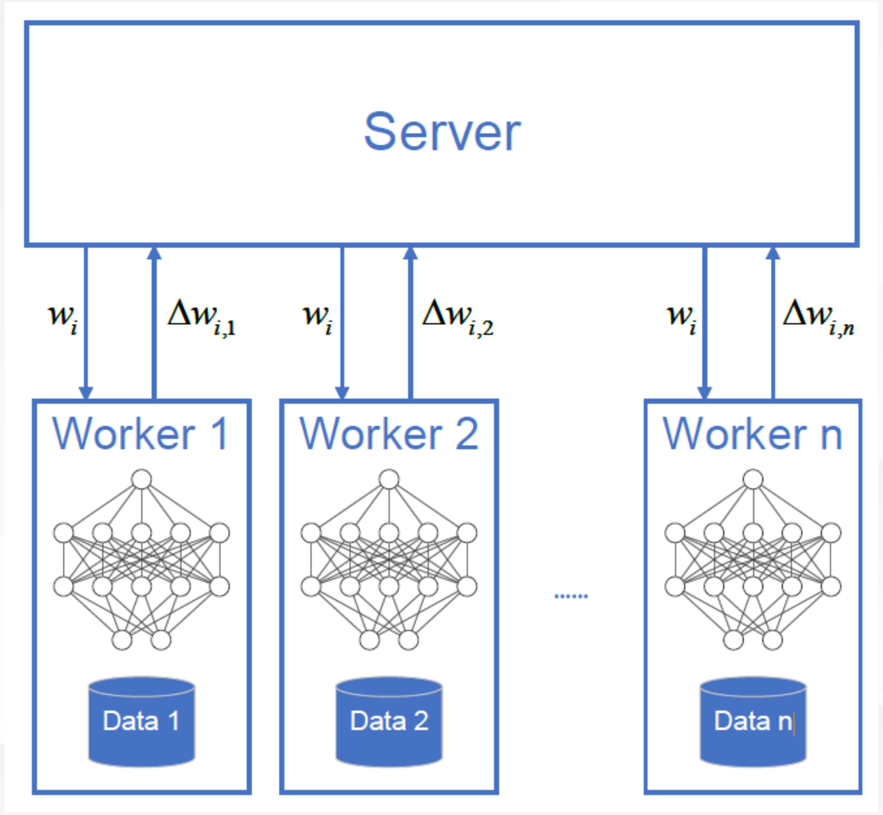
(a) VM-based communication patterns.

(b) Function-based communication patterns.

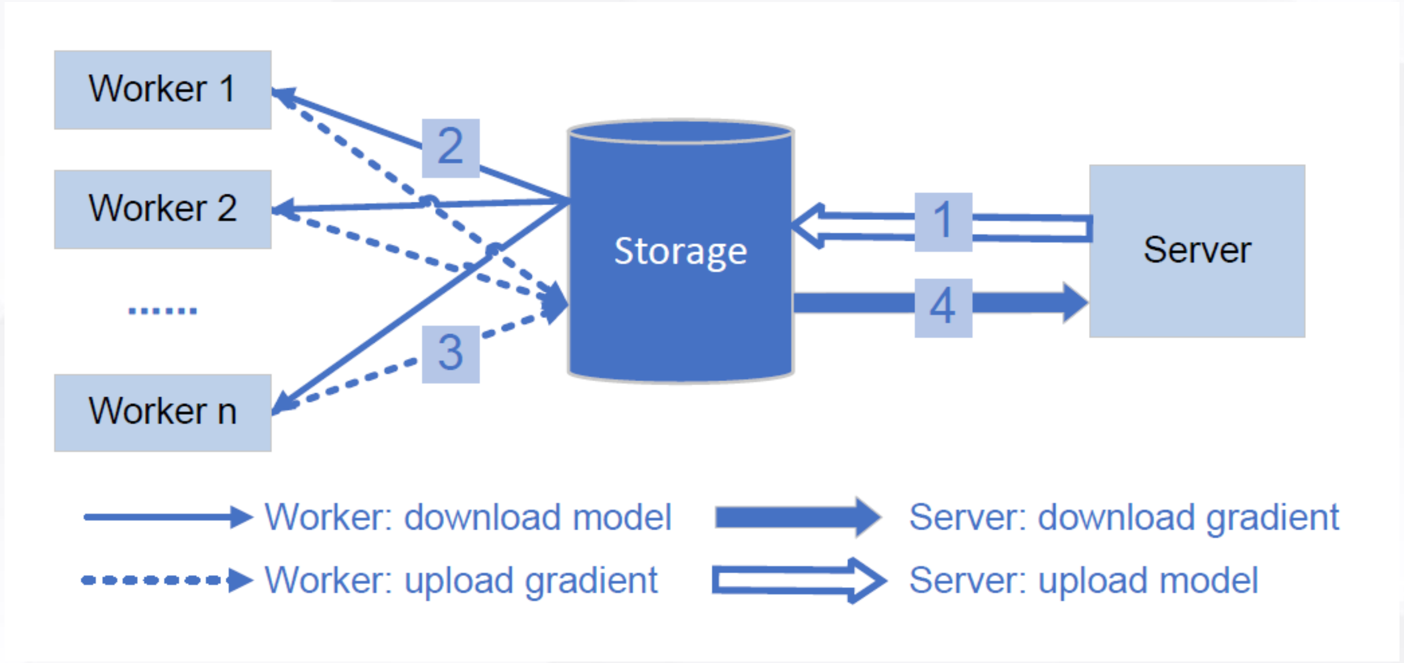
Limitation 3: Communication

Case: Distributed Machine Learning

Parameter Server



Serverless Parameter Server



Limitation 3: Communication

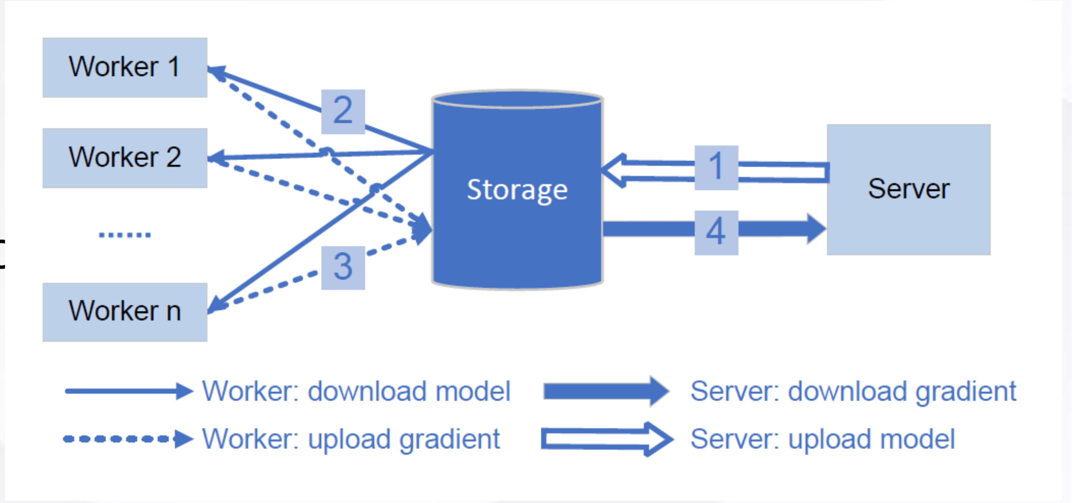
Feasible Optimization for Communication

(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 tc 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：

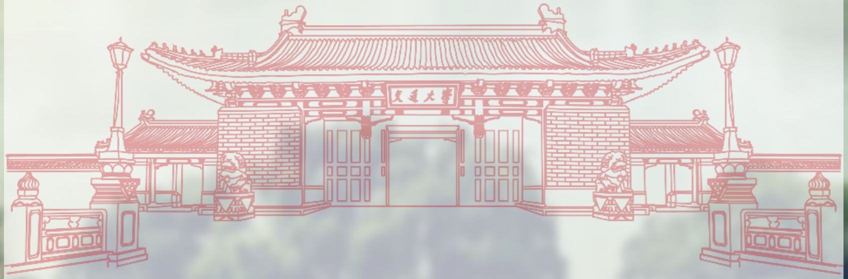
(1) the time it takes to start a cloud function
(2) the time it takes to initialize the software environment
(3) application-specific initialization in user code

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

05

Research Frontier in Serverless



What Serverless Computing Should Become?

After taking a broad view on increasing the types of applications and hardware that work well with serverless computing, identifing research challenges in five areas:

(1) **Abstractions**
- Resource Requirements
- Data Dependencies

(2) **Systems**
- High-performance, Affordable, Transparently Provisioned Storage
- Coordination/Signaling Service
- Minimize Startup Time

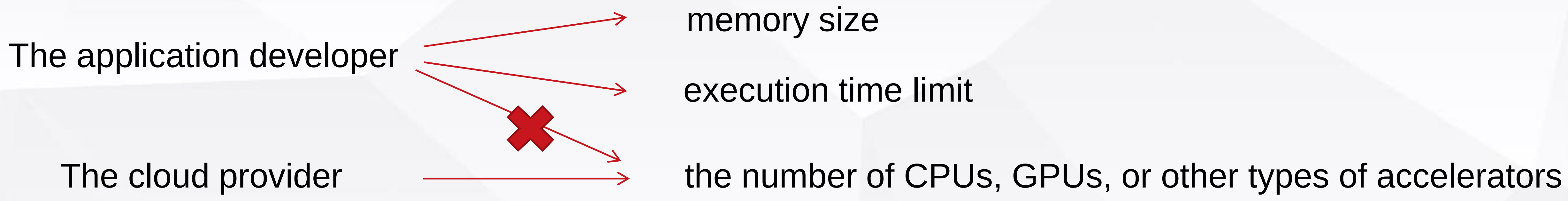
(3) **Networking**

(4) **Security**
- Resource Isolation
- Security Monitoring
- Security Management
- Data Protection

(5) **Architecture**
- Hardware Heterogeneity, Pricing, and Ease of Management

Abstraction Challenge

Resource Requirements



The application developer
The cloud provider
memory size
execution time limit
the number of CPUs, GPUs, or other types of accelerators

A better alternative would be to raise the level of abstraction, having the cloud provider infer resource requirements instead of having the developer specify them.

Methods:

- static code analysis
- profile previous runs
- dynamic (re)compilation to retarget the code to other architectures

Abstraction Challenge

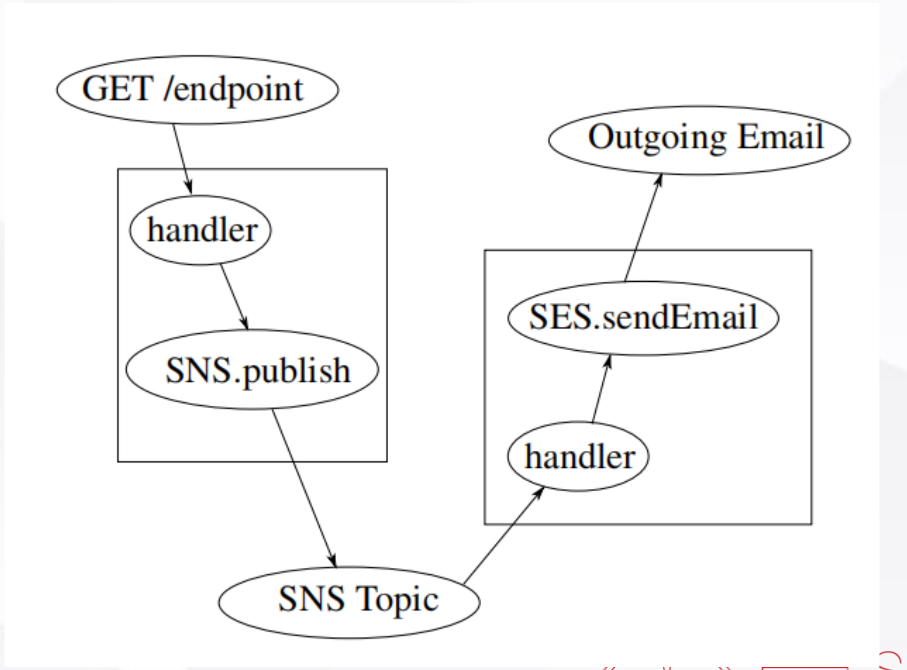
Data Dependencies

Today’s cloud function platforms have no knowledge of the data dependencies between the cloud functions, let alone the amount of data these functions might exchange.

↓

Suboptimal placement that could result in inefficient communication patterns.

One approach to address this challenge would be for the cloud provider to expose an API that allows an application to specify its **computation graph**, enabling better placement decisions that minimize communication and improve performance.



System Challenge

High-performance, Affordable, Transparently Provisioned Storage

(1)Serverless Ephemeral Storage

- Such emphmeral storage are needed to maintain application state during the application lifetime.
- Once the application finishes, the state can be discarded.
- Such ephemeral storage might also be configured as a cache in other applications.
- The speed and latency of the storage system used to transfer state between cloud functions can be a limitation.

One approach to providing ephemeral storage for serverless applications would be to build **a distributed in-memory service with an optimized network stack** that ensures microsecond-level latency.

- Enable the functions of an application to efficiently store and exchange state during the application's lifetime.
- Automatically scale the storage capacity and the IOPS with the application's demands.
- Provide access protection and performance isolation across applications.

System Challenge

High-performance, Affordable, Transparently Provisioned Storage

(2)Serverless Durable Storage.

- Such emphmeral storage are needed as long term data storage and the mutable-state semantics of a file system.
- With longer retention and greater durability than the Serverless Ephemeral Storage.

One approach to providing durable storage for serverless applications would be to leverage **an SSD-based distributed store paired with a distributed in-memory cache**.

- Transparently provisioned.
- Ensure isolation across applications and tenants for security and predictable performance.
- Only free resources explicitly (e.g., as a result of a “delete” or “remove” command), just like in traditional storage systems.
- Ensure durability, so that any acknowledged writes will survive failures.

System Challenge

Coordination/Signaling Service

Some scenarios requiring **coordination/signaling service**

- Sharing state between functions often uses a producer-consumer design pattern, which requires consumers to know as soon as the data is available from producers.
- One function might want to signal another when a condition becomes available.
- Multiple functions might want to coordinate, e.g., to implement data consistency mechanisms.

Note that cloud function instances are **not individually addressable** and **can’t communicate directly**.

Such signaling systems would benefit from

- Microsecond-level latency
- Reliable delivery
- Broadcast or group communication

System Challenge

Minimize Startup Time

Three parts of startup time:
(1) Scheduling and starting resources to run the cloud function.
(2) Downloading the application software environment (e.g., operating system, libraries) to run the function code.
(3) Performing application-specific startup tasks such as loading and initializing data structures and libraries.

Resource scheduling and initialization can incur significant delays and overheads from creating an isolated execution environment, and from configuring customer’s VPC and IAM policies. One approach to **reduce (1)** is to develop new **lightweight isolation mechanisms**.

One approach to **reduce (2)** is leveraging **unikernels**, being preconfigured for the hardware they are running on, statically allocating the data structures and including only the drivers and system libraries strictly required by the application.

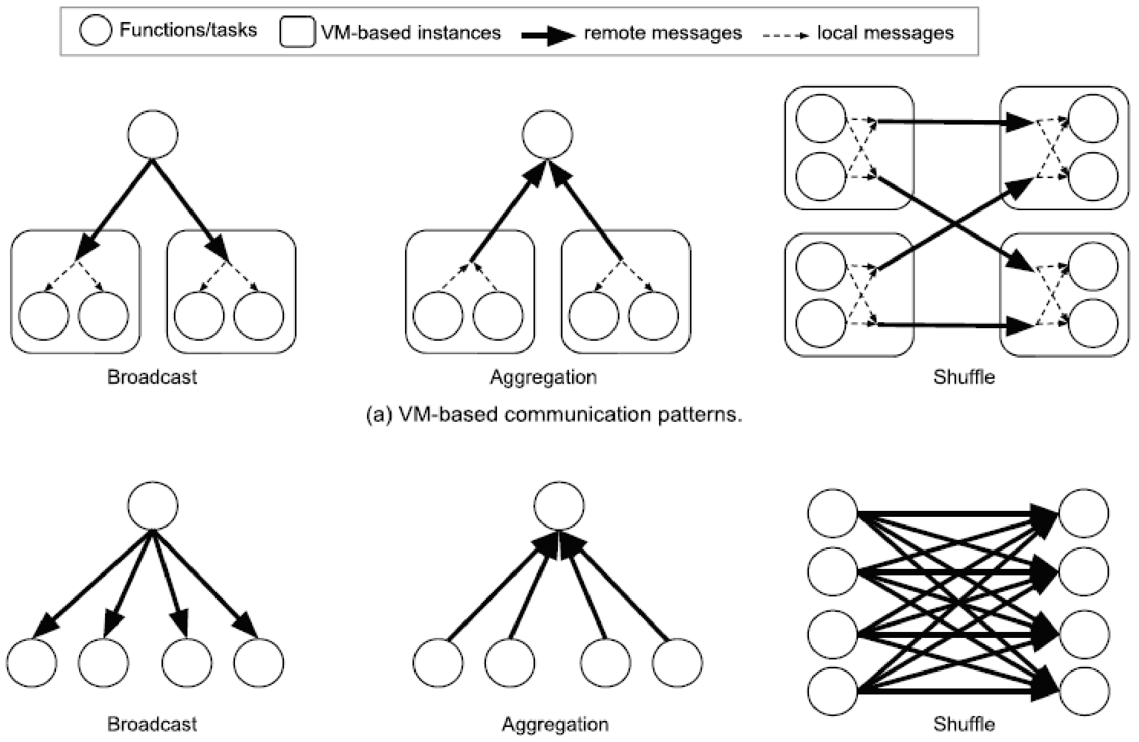
To **reduce (3)**, cloud providers can seek to **perform startup tasks ahead of time**, particularly powerful for customer-agnostic tasks.

Networking Challenge

Cloud functions can impose **significant overhead** on popular communication primitives such as broadcast, aggregation, and shuffle.

Several ways to address this challenge：

- Provide cloud functions with **a larger number of cores**, similar to VM instances.
- Allow the developer to explicitly place the cloud functions **on the same VM instance**.
- Let applications provide **a computation graph**, enabling the cloud provider to co-locate the cloud functions to minimize communication overhead



(a) VM-based communication patterns.

(b) Function-based communication patterns.

Note that the first two proposals could reduce the flexibility of cloud providers to place cloud functions, and consequently reduce data center utilization.

Security Challenge

Resource Isolation

Root Cause	The weakness of lightweight virtualization technologies in isolation
Industrial Solutions	Secure containers and network boundaries (e.g., Firecracker, gVisor, and VPCs)
Literature Work	Virtual-machine-based secure containers (e.g., minimized VMMs and unikernels)
Research Opportunities	Secure containers with better isolation, better performance, and lower overhead

Security Challenge

Security Monitoring

Root Cause	The ephemerality of functions and the broken boundaries of serverless applications
Industrial Solutions	Security scanning and monitoring tools (e.g., Azure Monitor, AWS X-Ray, and DevSecOps tools)
Literature Work	Information flow mining and tracing (e.g., static analysis and dynamic tracing)
Research Opportunities	Nonintrusive tracing schemes and advanced insight capabilities such as diagnosis and forensics

Security Challenge

Security Management

Root Cause	The distributed nature of functions and the fragmented application boundaries
Industrial Solutions	Fine-grained authentication and authorization (e.g., IAM systems and role-based access control)
Literature Work	Advanced management capabilities (e.g., workflow-sensitive authorization and secure container network stacks)
Research Opportunities	Automatic security configuration and auditing tools

Security Challenge

Data Protection

Root Cause	Platforms' control of the function lifecycle and BaaS services' participation in business logic
Industrial Solutions	Encryption in transit and at rest (e.g., TLS/SSL, code/data encryption, and key management services)
Literature Work	Hardware-based trusted execution environments (e.g., protecting running code with Intel SGX)
Research Opportunities	Other confidential computing solutions such as homomorphic encryption

Computer Architecture Challenge

Hardware Heterogeneity, Pricing, and Ease of Management

There are two paths forwarding the demand for faster computation:

- For functions written in high-level scripting languages like JavaScript or Python, hardware-software co-design could lead to **language-specific custom processors** that run one to three orders of magnitude faster.
- **Domain Specific Architectures(DSAs)** are tailored to a specific problem domain and offer significant performance and efficiency gains for that domain, but perform poorly for applications outside that domain (e.g. Graphical Processing Units (GPUs) used to accelerate graphics, Tensor Processing Units (TPUs) for machine learning).

Two paths for serverless computing to support **the upcoming hardware heterogeneity**:

- Serverless could **embrace multiple instance types**, with a different price per accounting unit depending on the hardware used.
- The cloud provider could **select language-based accelerators and DSAs automatically**, implicitly based on the software libraries or languages used in a cloud function (say GPU hardware for CUDA code and TPU hardware for TensorFlow code) or explicitly monitor the performance of the cloud functions and migrate them to the most appropriate hardware the next time they are run.

感谢聆听

饮 水 思 源　爱 国 荣 校