Cloud Computing

Serverless

Thomas Ropars

thomas.ropars@univ-grenoble-alpes.fr

2024

References

Most the of slides are borrowed from the following presentation:

- The lecture on Serverless by Renaud Lachaize
- The lecture on Serverless by Mathieu Bacou

Agenda

- Overview of the serverless paradigm
- Function-as-a-Service
- Backend-as-a-Service: The case of Object Storage

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

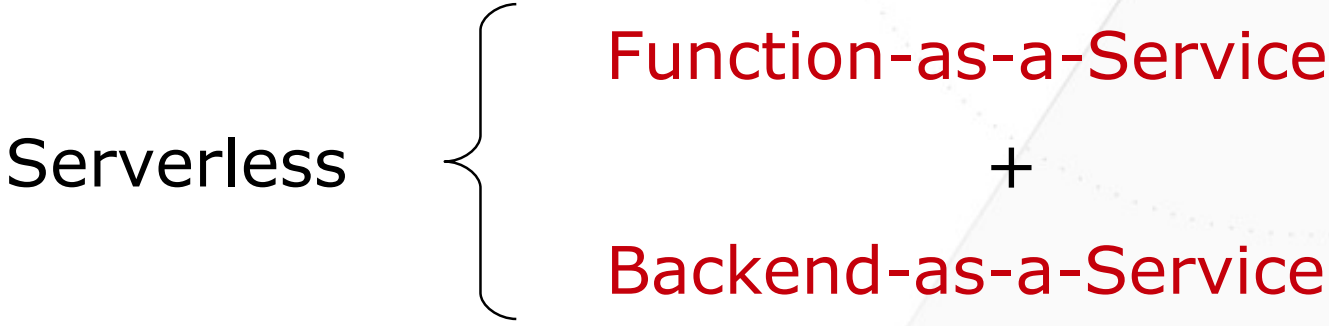
1. Redundancy for availability
2. Geographic distribution of redundant copies (e.g., for disaster recovery)
3. Load balancing and request routing for efficient resource utilization
4. Autoscaling (up or down) in response to changes in input workload
5. Health monitoring
6. Logging events/messages (for 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.)

Introducing: serverless

- Real cloud-native applications: only provide code for the business core features
- All management and execution provided by the cloud platform
  - From execution environment to service availability



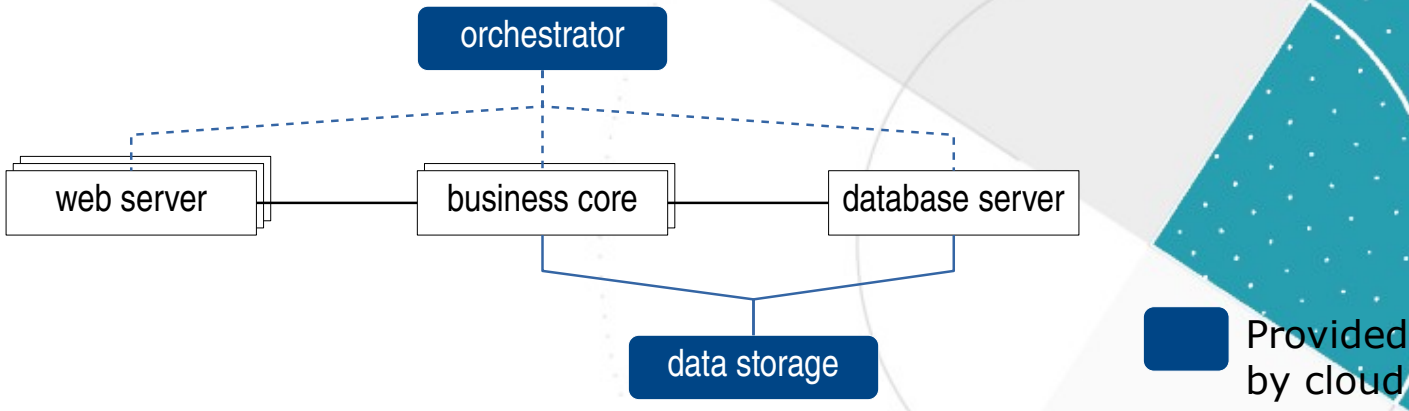
Backend-as-a-Service

- Common backend components in application architectures
  - Database servers, message queues, (object) storage…
- Better served by the cloud provider
  - Mutualized, no overhead for the user, available
  - Provides an ecosystem of components
    - Beware vendor lock-in!
- Elasticity requirement: scale quickly, up and down to zero, with the FaaS workload

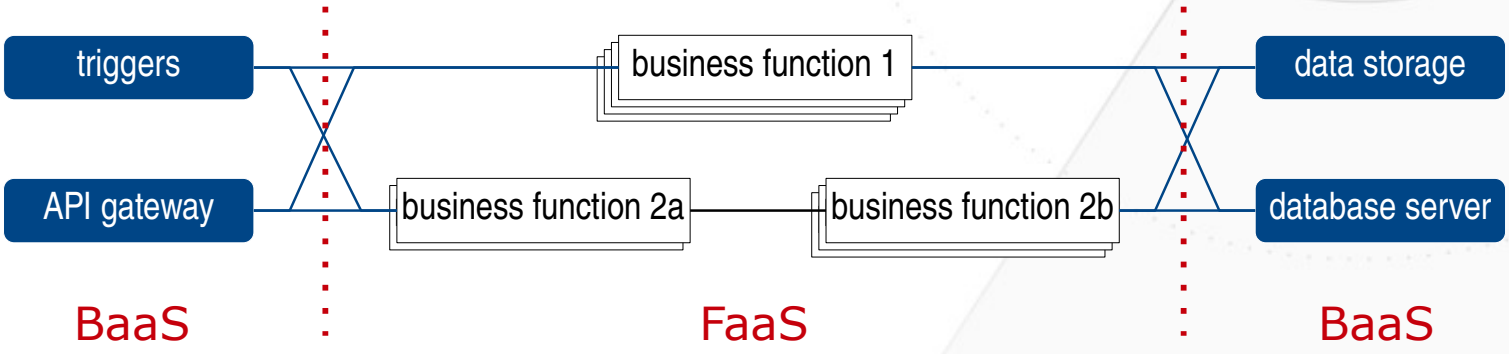
Function-as-a-Service

- Run backend code without long-lived servers
  - Execution environments are spawned on-demand
  - All managed by the cloud platform
- The unit of execution is a code block: the function
  - Applications are mostly event-driven
  - Parallelism at the cloud function level
  - Technically, also concurrency inside the cloud function
- Central feature of serverless

Comparison with micro-services



Micro-services architecture in the cloud



Serverless architecture in the cloud

Benefits of FaaS

- Elasticity: granularity of the request handler
  - Quick scaling, down to zero
- Deployment: just write code and upload
  - Quick experimentation, update
- Cost: pay only the compute time you need
  - No request = no function running = no resource = no cost
  - Roughly: Cost = Compute Time x Reserved Memory

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).

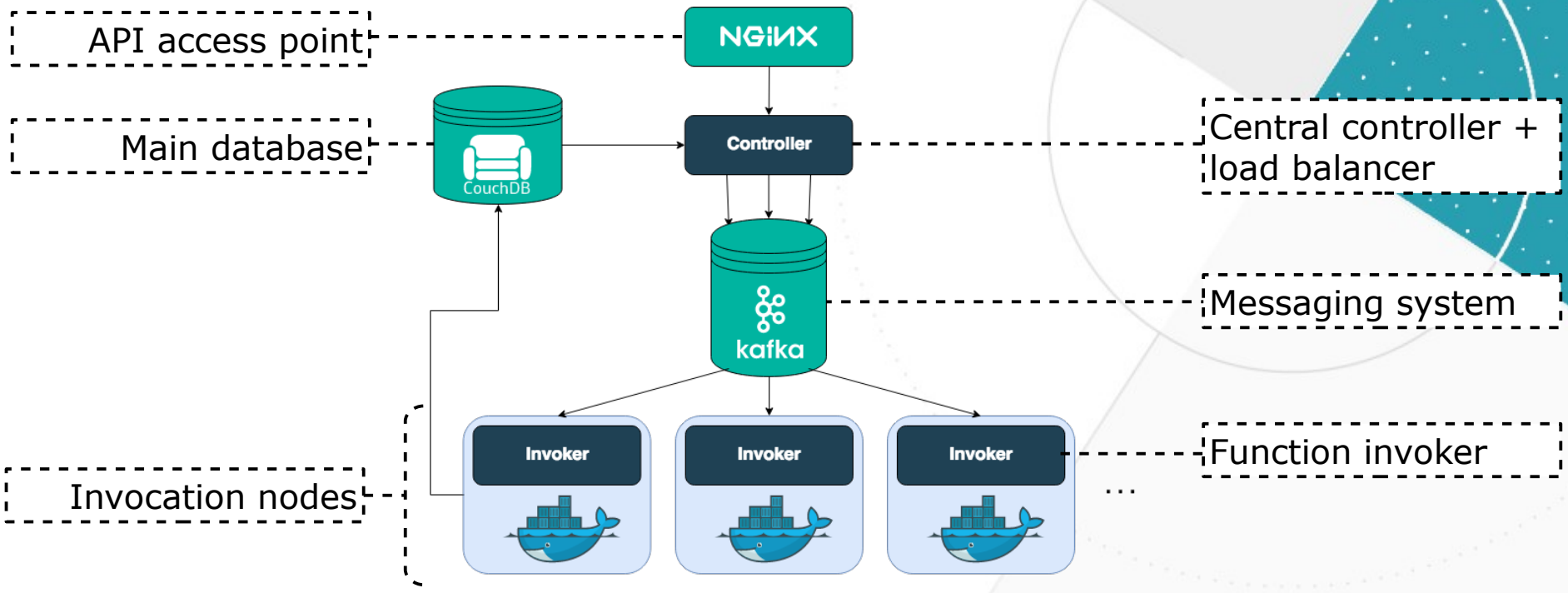
FaaS – Workflows

- 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

Internals of Apache OpenWhisk



Architecture of Apache OpenWhisk

Internals: function invocation

0) (after authentication and other tasks)
1) Spawn new Docker container with runtime
2) Inject action code
3) Execute action with parameters
4) Retrieve result
5) Destroy Docker container

OPTIMIZED

Function container management

- Very slow (for serving request) to spin up new container
  - Around 400ms
- Reuse existing containers!
  - Functions are stateless
- Cold starts and warm starts
  - No runtime container available: cold start
  - Available runtime container: warm start

40 times faster!

- Smart management of container pool
  - Pool of pre-warmed containers
  - Trade-off between occupied resources and execution latency
    - Containers kept warm use resources but are not billed to the user!

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.

Object storage

- (Sometimes also called “blob storage”)
- **Examples:**
  - Cloud services: AWS S3 (“Simple Storage Service), Google Cloud Storage
  - Open-source frameworks: OpenStack Swift, MinIO
- **Main goals:**
  - Scalability (number of data items, size of data items, number of clients)
  - Fault tolerance, durability and high availability (through redundancy)
  - Simplicity (simple to use & to scale)
  - Genericity (very diverse use cases: web sites/applications, multimedia, system images, backups, data lakes, analytics …)
  - Cost effectiveness
  - **Typically available in the form of several storage “classes”**, corresponding to various requirements in terms of performance, durability, availability, cost.

Object storage – Main characteristics (1/2)

- **Relies on the notion of ”object”**
  - Somewhat different from other traditional abstractions (block, file, database tuple).
  - Some similarities with the concept of key-value database (yet with some differences in features).
  - **An object is a generic container of bytes, with a variable size (potentially very big)**
  - **Versatile support for very different object sizes (from bytes to several TBs)**
  - Each object is also associated with a few kBs of metadata stored in a key-value format. Some of these metadata are system defined (e.g., last modification date). Others are flexible (user defined).

Object storage – Main characteristics (2/2)

- **Loosely structured namespace**
  - Each object is identified by a globally unique, user-defined key (text string of variable length)
  - Each object belongs to a given storage container named “bucket”
  - The bucket namespace is flat (i.e., there is no hierarchy, unlike in a file system)

- **Optional features:**
  - Versioning (per object)
  - Access logging (per-object)
  - Access control lists (typically per-bucket)
  - Public access from the Internet, via a URL

Object storage – Interface (1/4)

- Main interface: accessible via a ”Web service” interface (**HTTP-based, REST API**)
- (A client application can also mount a bucket as a file system … but not fully compliant with POSIX file system semantics, and also some features of object storage will not be available)
- Main operations (for buckets, objects and metadata) : **“CRUD”**
  - Create, Read, Update, Delete
  - In the case of objects, “Update” = full overwrite

- Each object is uniquely identified with the combination of bucket name + key (+ optional version ID)
- Although a bucket is flat namespace, it is somehow possible to “recreate the notion of folder by using of a common prefix in the name of various objects (e.g., “logs/”)
- Data at rest can be encrypted on the client side or on the server side

Object storage – Interface (2/4)

- **Other operations:**
  - Partial read of an object
  - Conditional read of an object (if-modified-since, if-unmodified-since …)
  - Copy of an object
  - Delete multiple objects at once
  - Set object/bucket in “lock mode”, preventing future deletion or update (e.g., to enforce a given retention threshold)

Object storage – Interface (3/4)

- The AWS S3 API (although not standardized) has become a de-facto standard, implemented by most systems
  - Note however that, although this facilitates porting applications from one system to another, object storage systems are often not fully compatible and thus not always easily interchangeable, because advanced details/features have significant differences.
  - For example, see the following comparison between AWS S3 and Google Cloud Storage: https://www.zenko.io/blog/four-differences-google-amazon-s3-api/

Object storage – Interface (4/4)

- **Additional details** - For most object storage systems:
  - Updates to a single object are guaranteed to be atomic (no partial writes)
  - ... However, they may not become immediately visible (see description of consistency semantics on the next slides)
  - There are no atomicity/transactional guarantees across several objects
  - The system does not provide any object locking facility to manage concurrent requests.
  - Note that, given the weak consistency model provided by most object storage systems, it is generally not possible/practical to build a locking service on top of them.
  - Instead, if a locking service is needed, it is more appropriate to use an external locking service.

Object storage – Consistency (1/4)

- **Some Cloud-based object storage systems only provide limited/weak consistency guarantees.** This means, for example, that, in some situations, a read request may return an outdated view of the data.

- This is due to designed choices related to the management of replication, and the emphasis given to scalability.

- However, consistency guarantees of object storage are being improved (i.e., becoming stronger) due to the growing popularity and diversity of use cases for object storage, and the productivity/correctness benefits for application developers.

Object storage – Consistency (2/4)

- Terminology

  - **“Eventual consistency”** corresponds to a weak level of consistency. It only provides the following guarantees:

    - An attempt to read an existing object may return the current value of the object, or an older version, or NULL.

    - If we stop updating the object, all its copies (“replicas”) will eventually converge to the same state, after a finite amount of time (but we do not know the precise upper bound for this time interval).

  - In contrast, **“read-after-write”** semantics provide stronger consistency guarantees.

Object storage – Consistency (3/4)

- **Case study #1: AWS S3 – Before December 2020**
  - **S3 generally provides read-after-write consistency**
  - … **except in the following situations, which only guarantee eventual consistency**:
    - Modification of an existing object (overwrite or delete) followed by a read
    - Attempting to read an object that does not exist, followed by a creation of the object, followed by a read of the object

Illustration (quotes from the old version of the S3 official documentation):

- *"S3 achieves high availability by replicating data across multiple servers within Amazon's data centers. If a PUT request is successful, your data is safely stored.* ***However, information about the changes must replicate across S3, which can take some time, and so you might observe the following behaviors****:*
  - *A process writes a new object to S3 and immediately lists keys within its bucket. Until the change is fully propagated, the object might not appear in the list.*
  - *A process replaces an existing object and immediately attempts to read it. Until the change is fully propagated, S3 might return the prior data.*
  - *A process deletes an existing object and immediately attempts to read it. Until the deletion is fully propagated, S3 might return the deleted data.*
  - *A process deletes an existing object and immediately lists keys within its bucket. Until the deletion is fully propagated, S3 might list the deleted object."*

Object storage – Consistency (4/4)

- **Case study #2: AWS S3 – Since December 2020**
  - "**all S3 GET, PUT, and LIST operations, as well as operations that change object tags, ACLs, or metadata, are now strongly consistent**. What you write is what you will read, and the results of a LIST will be an accurate reflection of what's in the bucket."

Illustration (quotes from the current version of the S3 official documentation):

- *"S3 achieves high availability by replicating data across multiple servers within Amazon's data centers. If a PUT request is successful, your data is safely stored.* ***Any read (GET or LIST request) that is initiated following the receipt of a successful PUT response will return the data written by the PUT request.*** *Here are examples of this behavior:*
  - *A process writes a new object to Amazon S3 and immediately lists keys within its bucket. The new object appears in the list.*
  - *A process replaces an existing object and immediately tries to read it. Amazon S3 returns the new data.*
  - *A process deletes an existing object and immediately tries to read it. Amazon S3 does not return any data because the object has been deleted.*
  - *A process deletes an existing object and immediately lists keys within its bucket. The object does not appear in the listing."*

- For more details:
  - https://docs.aws.amazon.com/AmazonS3/latest/dev/Introduction.html#ConsistencyModel
  - https://www.allthingsdistributed.com/2021/04/s3-strong-consistency.html
  - https://blog.kylegalbraith.com/2021/01/12/the-s3-consistency-model-got-an-upgrade/

Object storage – Wrap-up

- **Main strengths**
  - Simplicity
  - Versatility (use cases, scale, guarantees, access model …)
  - Cost
- **Main weaknesses**
  - Some systems **only provide limited guarantees regarding data consistency and the management of concurrent operations**. This may not be appropriate for certain kinds of applications, in particular with the following characteristics:
    - Frequently modified objects and freshness requirements
    - Concurrent read-write accesses to the same objects
  - Poorer performance (w.r.t. latency and bandwidth) compared to some other types of storage services (especially lower-level and/or more closely-coupled interfaces like block storage)