

263-3855-00: Cloud Computing Architecture

Section 01: Cloud Service Architectures

Swiss Federal Institute of Technology Zurich (ETH Zürich)

Last Edit Date: 05/31/2025

Disclaimer and Term of Use:

1. We do not guarantee the accuracy and completeness of the summary content. Some of the course material may not be included, and some of the content in the summary may not be correct. You should use this file properly and legally. We are not responsible for any results from using this file.
2. This personal note is adapted (derived) from the slides and codes from *Professor Ana Klimovic and people from ETH Zurich Systems Group*. Please [contact us](#) to delete this file if you think your rights have been violated.
3. This work is licensed under a [Creative Commons Attribution 4.0 International License](#).

I. The cloud in an economic context

The cloud turns them into services: made available by a provider, pay per-use, ownership does not shift to the client.

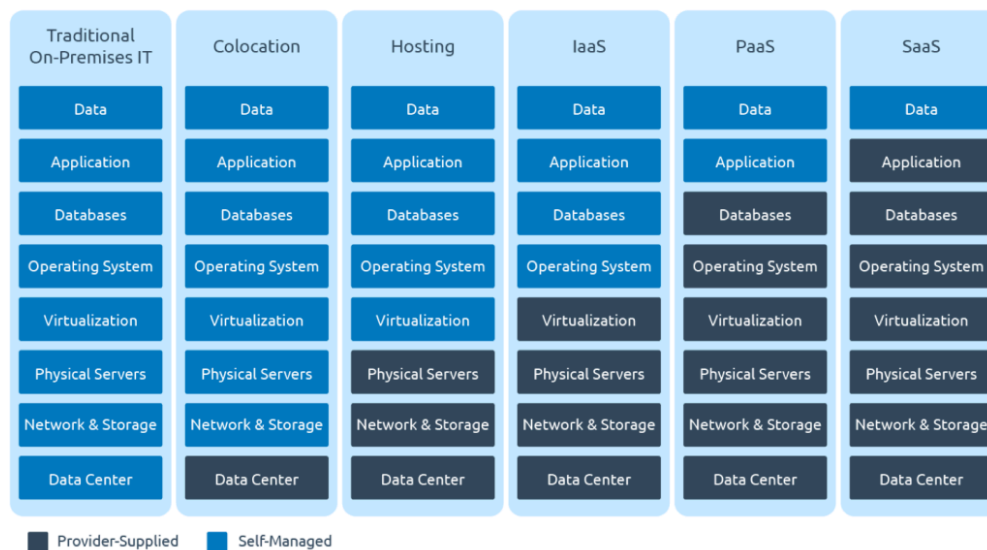
The magnificent 7: Amazon, Microsoft, Google, Alibaba, Baidu, Tencent, Meta.

Cloud economics for the user:

- Capital expenses are an investment
 - They become an asset of the company
 - Cost spread over time through depreciation
- Operational expenses are cost
 - They are the price of doing business
 - Typically tax deductible

The cloud turns from capital expenses into operational expenses, enabling many companies to scale sizes that would have been difficult before because of the upfront investment required.

II. The cloud as a service



1. Infrastructure as a Service (IaaS)

Equivalent to renting a server, enabled through virtualization. Useful to those who need access to servers but can / want to build the rest of the stack themselves.

Examples:

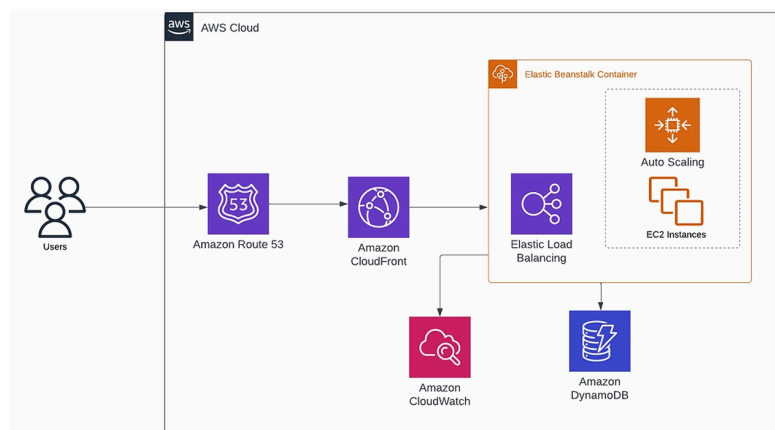
- Amazon Elastic Compute Cloud (EC2)
 - Many different instances
 - Several different pricing models

- Additional services available
 - Persistent storage volumes
 - Autoscaling
 - Monitoring and alerts

2. Platform as a Service (PaaS)

In PaaS, what is “rented” is not just a VM but the whole software and hardware infrastructure needed to run an application. The user still puts the application together using the services provided but PaaS reduces the amount of code and takes advantage of cloud features.

Example: Amazon Elastic Beanstalk



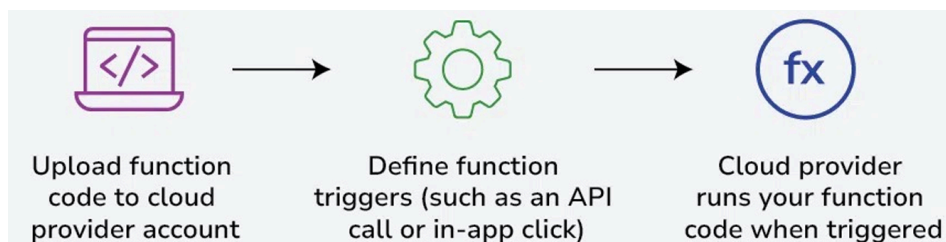
3. Software as a Service (SaaS)

With SaaS, you get the application, you provide the data to populate it. SaaS can be offered by companies other than the cloud providers, using the cloud infrastructure or their own cloud.

Examples: Microsoft Office 365, Google GMail, DropBox, Snowflakes, Salesforce, etc.

4. Functions as a Service (FaaS)

FaaS allows users to encapsulate a small application into a function (e.g., a container image) and the cloud platform takes care of scheduling, allocating resources for, and executing the function.



FaaS has the following properties:

- No reservation of VM or provisioning of instances

- Automatic scalability
- Automatic triggering
- Charged by millisecond

Examples: AWS Lambda, Google Cloud Functions, Microsoft Azure Functions, IBM Cloud Functions, etc.

III. Serverless computing

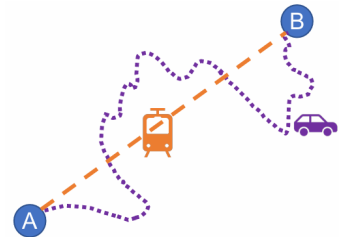
Serverless computing is a general concept, FaaS is just one popular example of serverless computing.

Infrastructure as a service is like renting a car

- Rent on-demand, insurance included
- Plan route, fill up gas, concentrate on the road, monitor traffic
- Pay even when not driving

Serverless computing is like a train

- A high-level abstraction to the cloud
- User specify what need to be done, not how



2. Why serverless

Cloud user perspective:

- Simple to use (do not need to be a cloud / system expert)
- Reduced cost when there is no load
- Handle sporadic bursts of requests by quickly spinning up infrastructure
- Automatic resource scaling, based on events that trigger the function

Cloud provider perspective:

- Serverless gives cloud providers the opportunity to optimize resource efficiency (i.e., cost) under the hood, since they control resource allocation for tasks and have a global view across users.
- However, today's FaaS platforms are highly inefficient, with lots of room for improvement in the system software design.

2. Limitations and challenges of FaaS

Limitations:

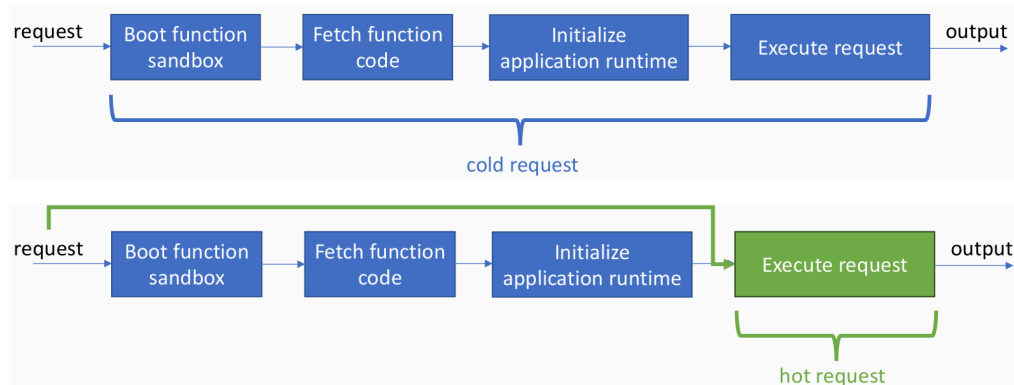
- Functions are stateless (no persistent state across invocations)
- Functions cannot establish direct network connections between each other (IP address not exposed)
- Limited execution time per function invocation (e.g., 15 min in AWS Lambda)
- Limited compute / memory / network bandwidth per function, scaled in fixed ratio

Challenges - scheduling fine-grain, short-running functions from different users is hard:

- Need to densely bin-pack functions per server to get high resource utilization
- Need to securely isolate functions in sandboxes (e.g., VM)
- Function sandboxes need to boot fast and be scheduled fast, since the functions themselves may only run for a few milliseconds

3. Warm (hot) v.s. cold start

To optimize performance, providers often keep function sandboxes “warm” in memory. The cloud platform needs to commit DRAM for each hot sandbox, but memory is expensive.



4. FaaS paradox

- Serverless gives providers the opportunity to be more efficient by optimizing scheduling decisions.
- But current FaaS is highly inefficient
 - Simply retrofitting the traditional software stack originally designed for coarse-grained, long-running applications leads to high overhead
 - The provider has little info about application characteristics to optimize scheduling

III. Designing cloud applications

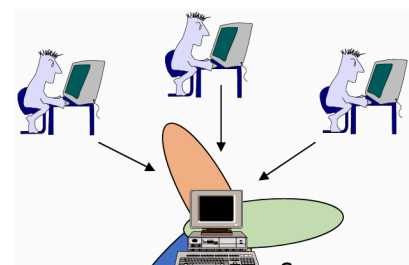
1. Tiers (layers)

Cloud software systems are typically divided into different tiers. Tiers can be conceptual or real.

Basic tiers:

- Presentation layer (client, external API layer): Enables interaction with the outside world (end-user, other apps)
- Application logic (business rules, business processes): Defines the system functionality
- Data layer (database, storage, business objects): This is where the data being used and processed is stored and managed

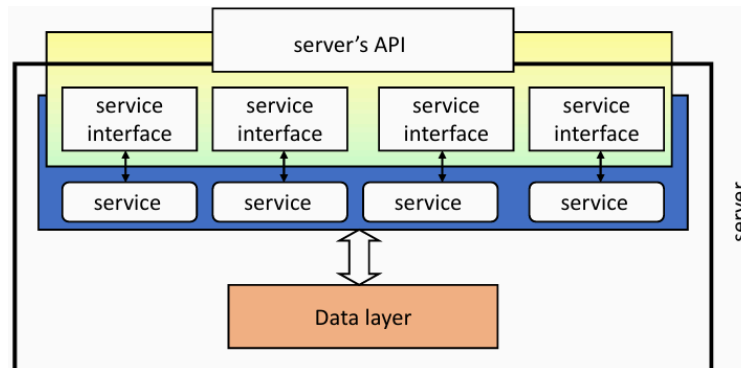
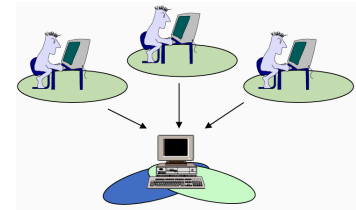
2. 1-tier architecture - fully centralized



- The beginning of computing: mainframe architecture
- The presentation layer, application logic and data are built as a monolithic entity
- Users / programs access the system through display terminals but what is displayed and how it appears is controlled by the server (“dumb” terminals)
- Advantages:
 - No forced context switches in the control flow (everything happens within the system)
 - All is centralized, managing / controlling resources is easier
 - The design can be highly optimized by blurring by blurring the separation between layers

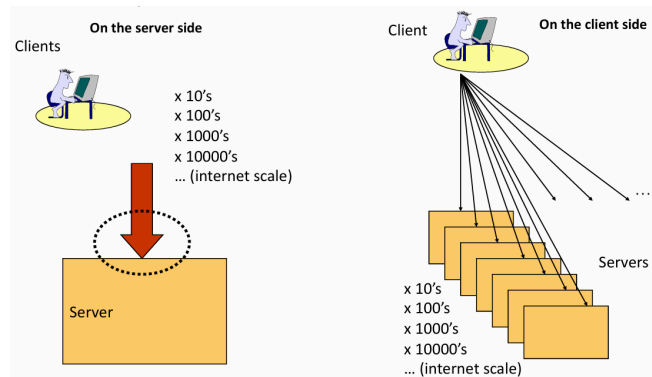
3. 2-tier architecture - client / server

- As computers became more powerful, we could move the presentation layer to the client.
- Advantages:
 - Clients are independent of each other: can have several presentation layers depending on what each client wants.
 - Can use computing power at the client machine to have more sophisticated presentation layers without requiring extra resources at the server machine.
- API in 2-tier architecture
 - Introduce the notion of a service: the client invokes a service implemented by the server
 - Also comes the notion of a service interface: how the client can invoke a given service
 - The interfaces to all the services provided by a server: the server’s app - program interface (API)
 - Many standardization efforts to agree on common APIs for each type of server



- Advantages / disadvantages of 2-tier architecture as a system design pattern
 - Advantages
 - Leverage resources at the client
 - Customization by modifying the client

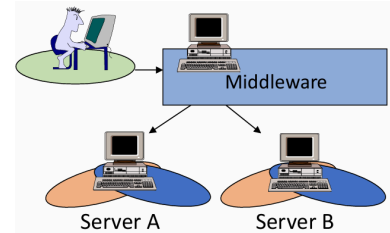
- Forces the application to have an interface
 - Integration is easier through the client
 - Centralized control for server (1-tier)
- Disadvantages
 - Software at the (remote) client is part of the system
 - Maintenance requires coordinating server and client
 - Backward compatibility issues for older clients
 - Stateless v.s. Stateful clients
 - Managing connections at server becomes bottleneck
 - Performance loss through context switch client-server and networking
- Limitations
 - Scalability limitations of client-server



- Architecture limitations of client server
 - If clients want to access 2 or more servers, a 2-tier architecture causes several problems: servers don't know about each other, the client is the point of integration, increasingly "fat" clients; there is no common business logic between the servers
 - This is very inefficient from all points of view (software design, portability, code reuse, performance since the client capacity is limited, etc.)
- Example of a client system: web browser
 - A standardized platform for clients
 - Web browsers (HTTP, HTML, and everything that came afterwards)
 - Provided the universal client platform
 - Decouple the client from the server as the client side is delivered on demand
 - Reduced the overhead of managing clients (managing only browsers)

- Standardized the presentation layer language and communication protocol with the presentation layer (needed for the internet)
- Initially for reading content and simple forms, today a full computing platform (JavaScript)

4. 3-tier architecture - middleware



- In a 3 tier system, the three layers are fully separated
- The layers are also typically distributed taking advantage of the complete modularity of the design (in two tier systems, the server is typically centralized)
- Once distributed, there is the possibility of combining components from many different systems
- Middleware is used to connect the components
 - Middleware is a level of indirection between clients and other layers of the system
 - An additional layer of business logic encompassing all underlying systems
 - Simplifies the design of the clients by reducing the number of interfaces
 - Acts as the platform for inter-system functionality and high-level application logic
 - Takes care of locating resources, accessing them, and gathering results
 - A middleware system is just a system like any other - it can be 1 tier, 2 tier, 3 tier, etc.
- Advantages:
 - Reduce the number of necessary interfaces: clients and local applications see only one system (the middleware)
 - Centralize control and provide a common integration platform
 - Make necessary functionality widely available to all clients: can implement functionality that otherwise would be difficult (e.g. transactions)
 - Help deal with application heterogeneity and integration
- Disadvantages:
 - Yet another indirection level - extra complexity, extra latency
 - Needs to be standardized

5. Tail latency

Due to high fan-out in multi-tier cloud applications, it matters how long the slowest requests and services take. Analyzing average latency is not enough. Service level objective is often expressed as p99 latency < X.

- Scale up

- Attaining higher capacity by making a single component layer
- Worked well for decades: Moore's Law and Dennar scaling
- A single component cannot deal with today's loads
- Other system considerations (reliability, replication, scalability)
- Scale out
 - Attaining larger capacity by using many instances of the same (smaller) component
 - Relies on distribution (racks, clusters, data centers) and the necessary underlying infrastructure
 - Provides more flexibility (easier to provision and adjust to workload)
 - Introduces all the problems of distributed computing (additional infrastructure, overheads)
- Why scale out v.s. just scale up?
 - No way to store all of the web in a single computer, scale up simply does not apply
 - Increases parallelism, parallel search allows to go through huge amounts of data in a shorter time
 - Best effort, if some nodes do not reply, there is still some answer (works in some applications)

IV. Related readings

1. [What Serverless Computing Is and Should Become](#)
2. [Firecracker: Lightweight Virtualization for Serverless Applications](#)
3. [The Tail at Scale](#)
4. [Always Measure One Level Deeper](#)
5. (Optional) [Serverless Computing: One Step Forward, Two Steps Back](#)