Elements of Scalable Applications

Instructors

Battista Biggio and **Luca Didaci**

M.Sc. in Computer Engineering, Cybersecurity and Artificial Intelligence

University of Cagliari, Italy

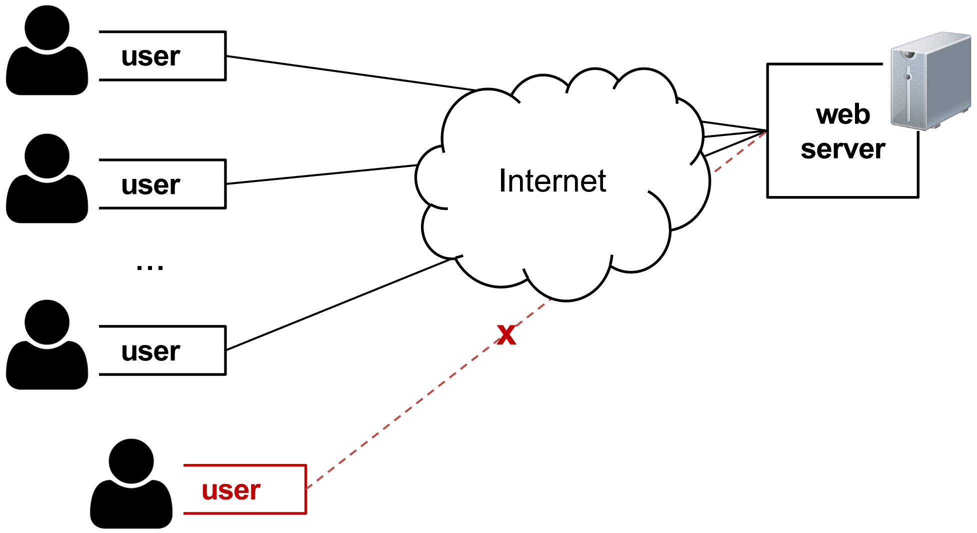
What Is Scalability?

*"**Scalability** is the capability of a system, network, or process **to handle a growing amount of work**, or its potential to be enlarged in order to accommodate that growth."*

Bondi, André B. (2000)
Characteristics of scalability and their impact on performance

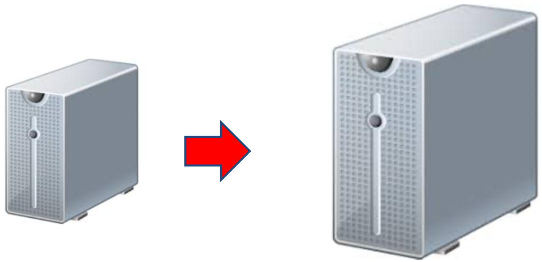
Running Example: Developing a Web Application

- How many users can we serve simultaneously before experiencing a denial of service (DoS)?

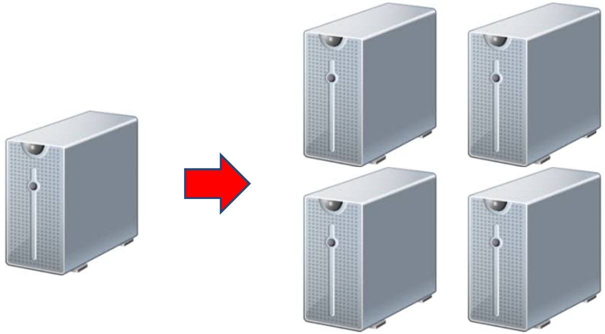


Vertical vs Horizontal Scalability

- **Vertical Scalability (Scale Up)**: increasing resources on a single machine - RAM, CPU, hard disk,...
  - no longer enough to cope with large data volumes and large predictive models (deep networks)



- **Horizontal Scalability (Scale Out)**: execute software and scale its execution on multiple machines, depending on the requested processing power
  - software has to be executed on heterogeneous architectures (e.g., cloud, data centers)

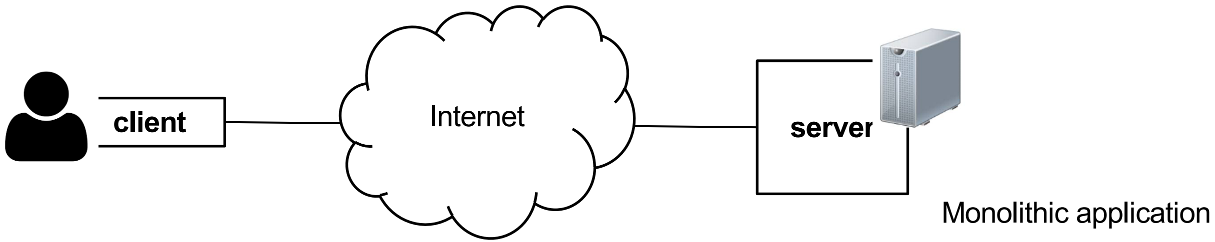


Main Requirements for Scalable Applications

- **High Availability / Redundancy**
  - No Single Point of Failure (SPoF)
  - Fault tolerance
- **Scalability / Elasticity**
  - Ability to scale up (vertically) or out (horizontally), *on demand*
- **Performance**
  - Average Response Time / Latency
  - Resource consumption
- **Security**
  - Ensure privacy and security of each layer/component

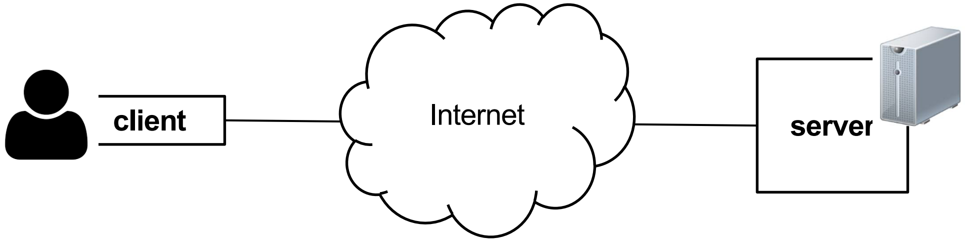
Running Example: Developing a Web Application

- What are the main problems with this simple architecture?



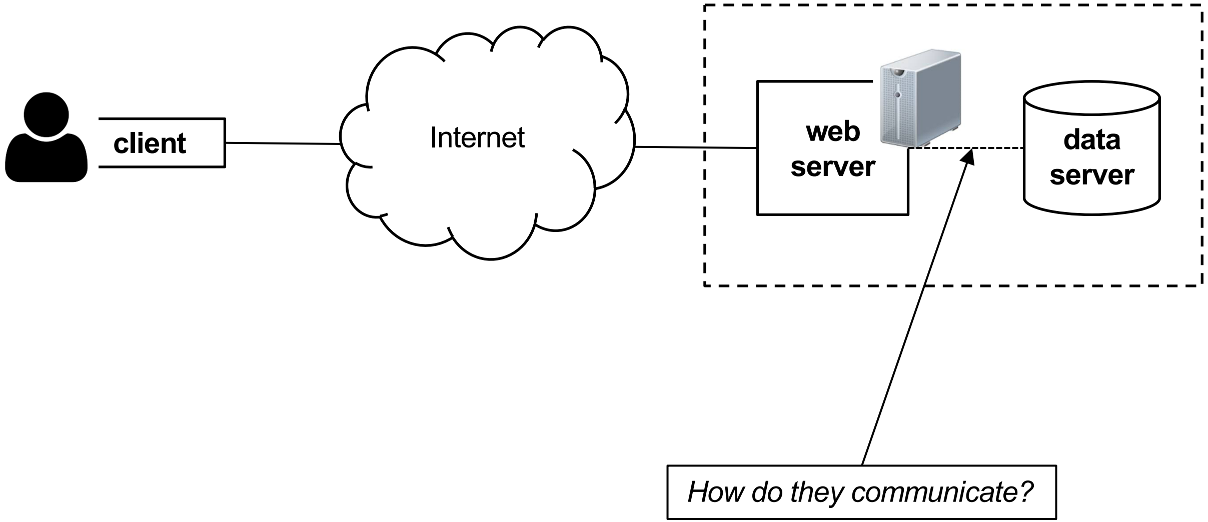
Running Example: Developing a Web Application

1. Storing data for each client on the same server may increase the average response time
2. The server may execute computationally-demanding operations and be slow to respond
3. The server is a Single Point of Failure (SPoF). If it fails, our application won't be accessible



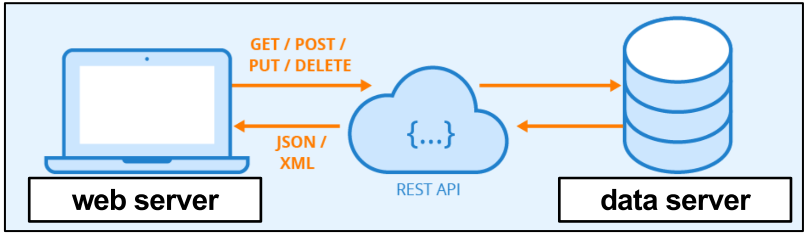
Tackling Problem 1: Separate Web Server and Data Storage

1. Separate web server from data storage (**db**: database)
2. ...
3. ...



How Does the WebServer Communicate with the DB?

- RESTful APIs (via HTTP)



Task	Method	Path
Create a new customer	POST	/customers
Delete an existing customer	DELETE	/customers/{id}
Get a specific customer	GET	/customers/{id}
Search for customers	GET	/customers
Update an existing customer	PUT	/customers/{id}

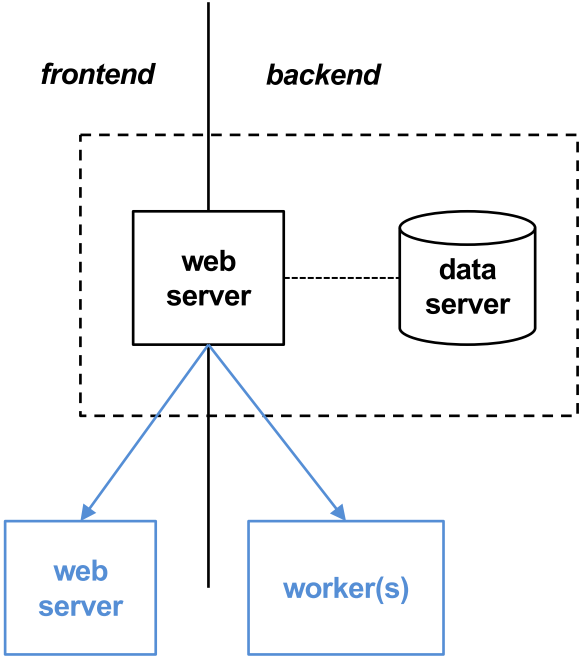
Tackling Problem 2: Separate Frontend and Backend

- What if some components in our backend are slow or take too much time to answer?
  - We cannot expect that the client waits several minutes or hours to get a response

- What can we do?
  - Separate *Frontend* and *Backend*

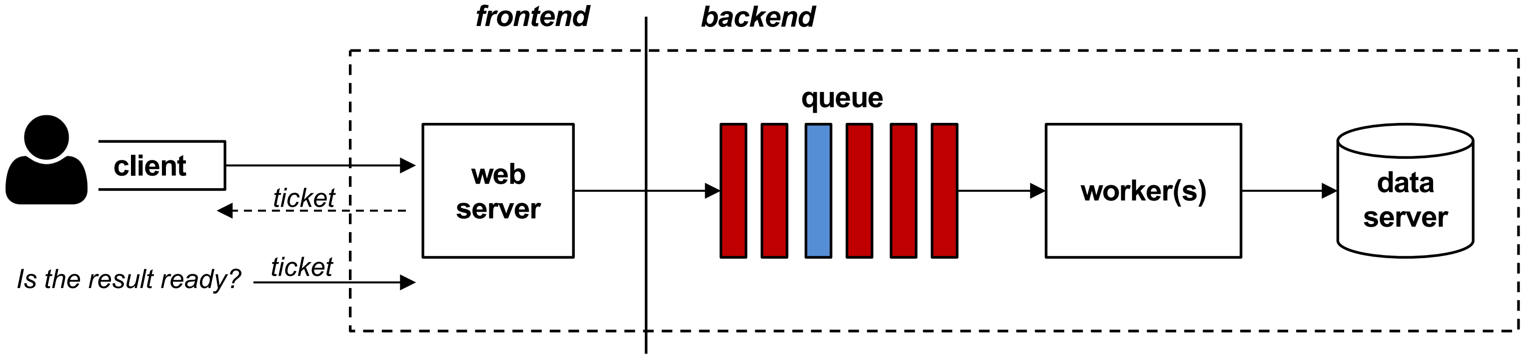
- **Frontend**: the *lightweight* web application seen from the client side
  - i.e., the interface / web page

- **Backend**: server-side (computationally-demanding) operations
  - e.g., database queries, machine learning, …



Decouple Frontend from Backend and Use a Message Broker

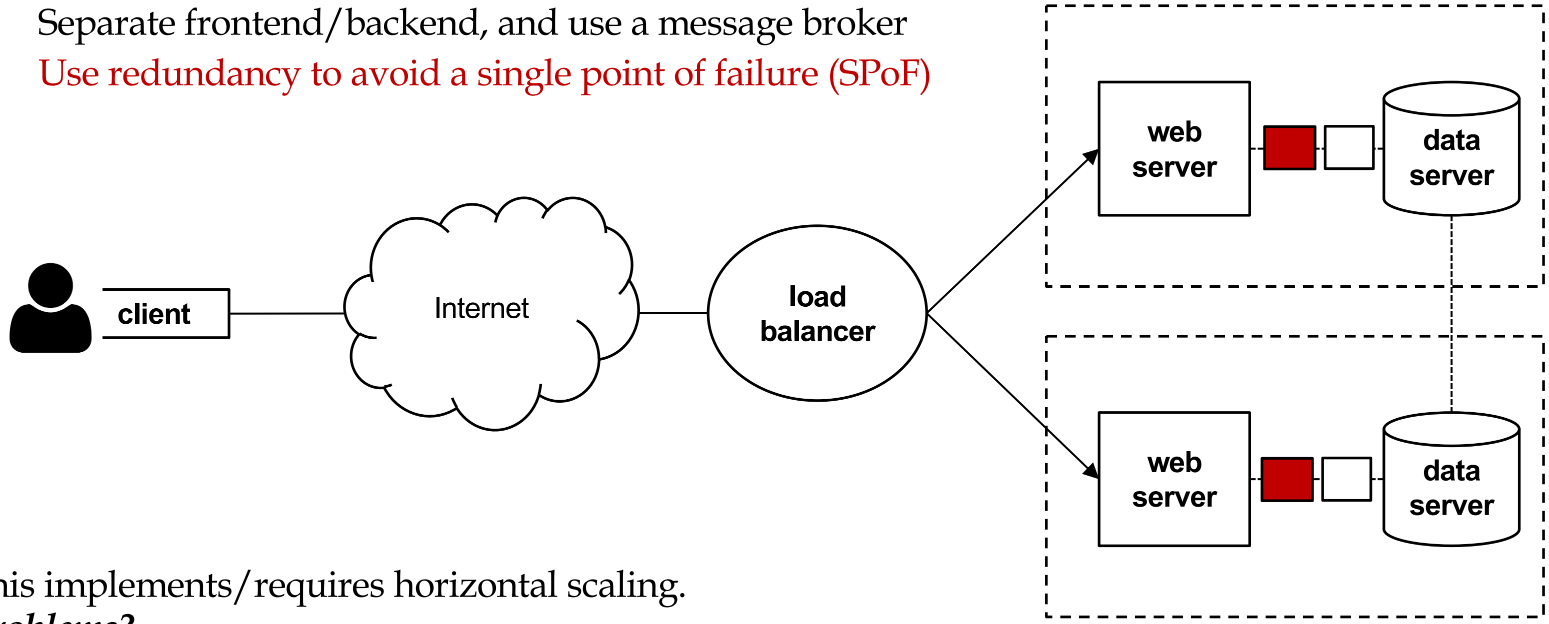
- We can use a message broker to queue requests
  - e.g., RabbitMQ - https://www.rabbitmq.com/getstarted.html



- The client sends a request. The web server receives it and returns a «ticket» to the client
- The request is added to the queue, and then served by a worker
- The result is stored in the data server (or a cache db)
- The client tries to retrieve the final result using the «ticket» given before

Tackling Problem 3: Redundancy vs SPoF

1. Separate web server from data storage
2. Separate frontend/backend, and use a message broker
3. Use redundancy to avoid a single point of failure (SPoF)



This implements/requires horizontal scaling.
Problems?

Stateful vs Stateless

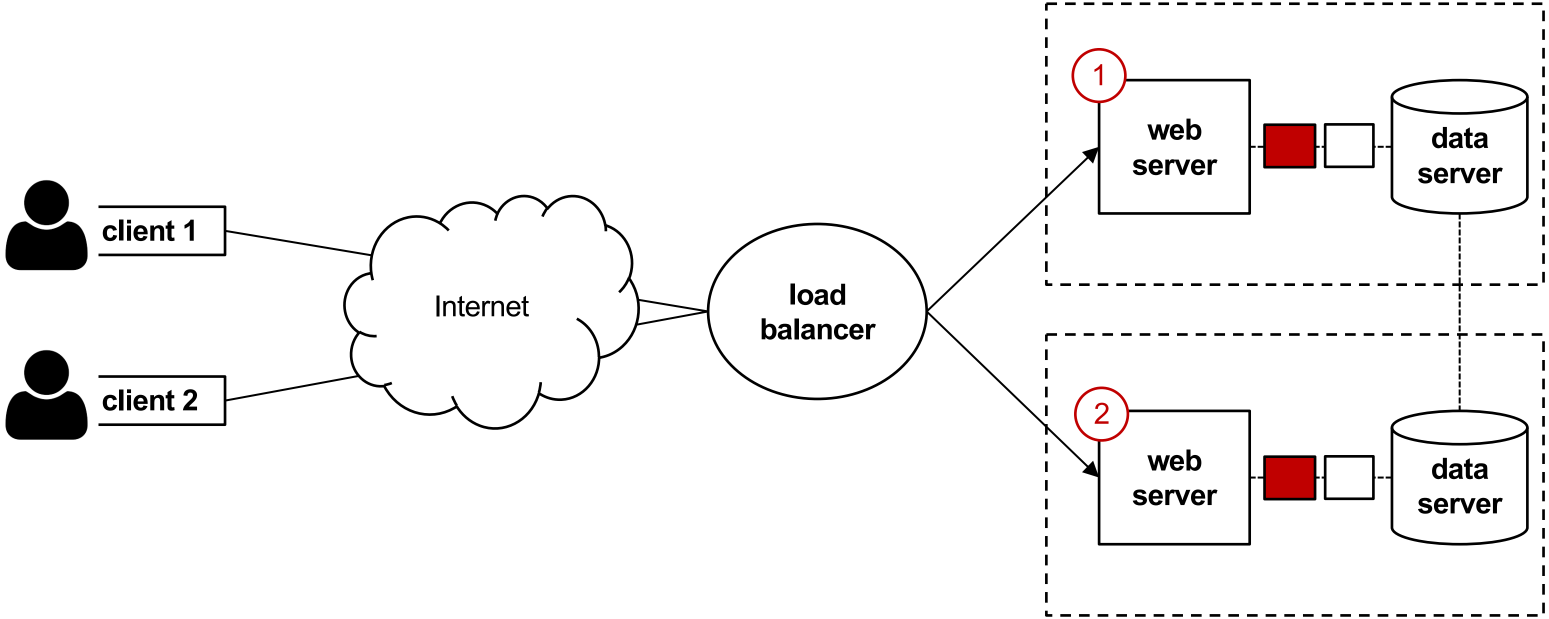
- **Stateful**: the application requires maintaining a given state
  - e.g., our web application may require a login and keep a user session alive (via HTTP cookies)

- **Stateless**: no state information is required to execute the application

What if our web application has *user sessions* (i.e., it is stateful)?
Can we scale horizontally, or not?

Stateful vs Stateless

- Is sticking user sessions to a given web server a good idea?



Stateful vs Stateless

- Is sticking user sessions to a given web server a good idea?
  - No, it is not.

- Being stateful hinders horizontal scalability.
  - We cannot assign the same user to another web server.

- **Workaround** *(problematic)***:** we may make the application stateless by letting the user submit their HTTP session cookie to the server each time. However:
  1. this may enable an attacker to impersonate the user by just stealing/forging the HTTP cookie;
  2. if the cookie is large in size, resending it every time increases latency unnecessarily.

- **Workaround** *(recommended)***:** store a unique session identifier in a HTTP cookie, and more detailed user session information server-side

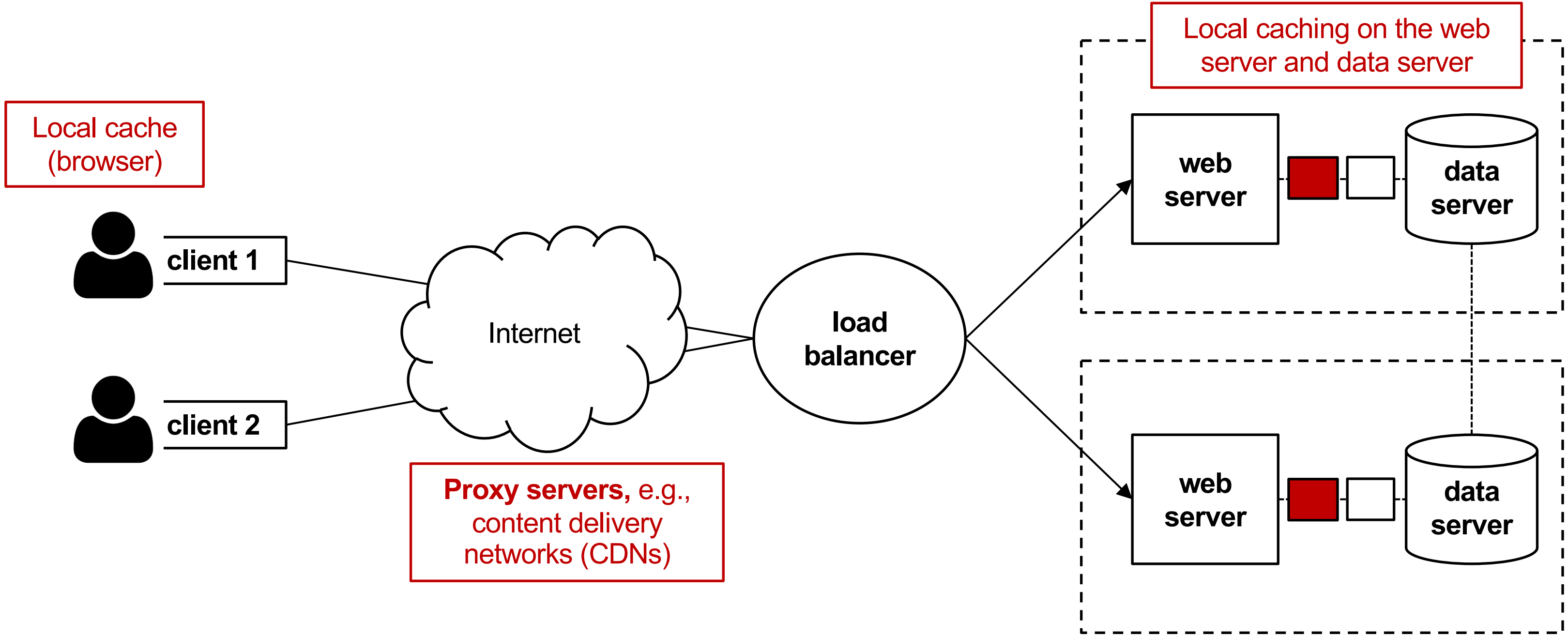
High Scalability: Stateless Applications and Loose Coupling

- **Stateless applications.** A stateless application requires no knowledge of the previous interactions and stores no session information. It can scale horizontally because any request can be served by any of the available system compute resources

- **Loose coupling.** As application complexity increases, a desirable characteristic is to be able to break it into smaller, loosely-coupled components
  - Each component should be designed as a *black box* to reduce interdependencies
  - A change or a failure in one component does not cascade to other components
  - The more loosely system components are coupled, the larger they scale

High Scalability: Reduce Latency via Caching

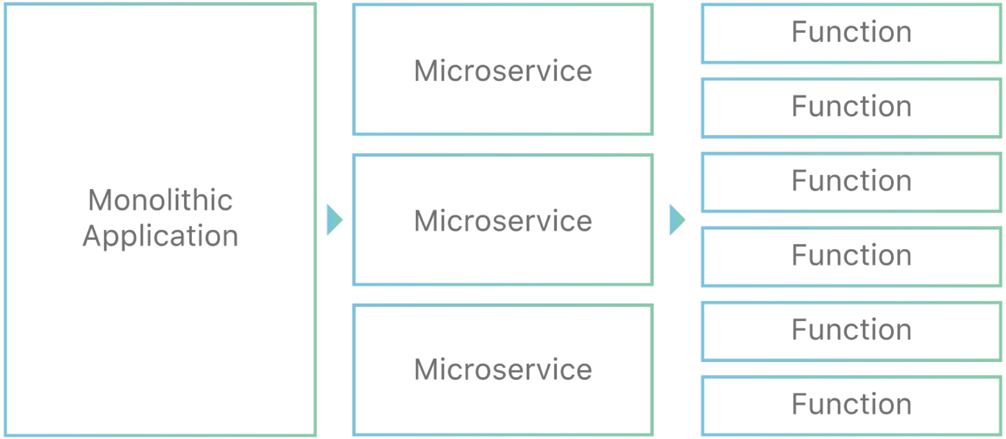
- If the application is still too slow client-side, caching (at different levels) may help



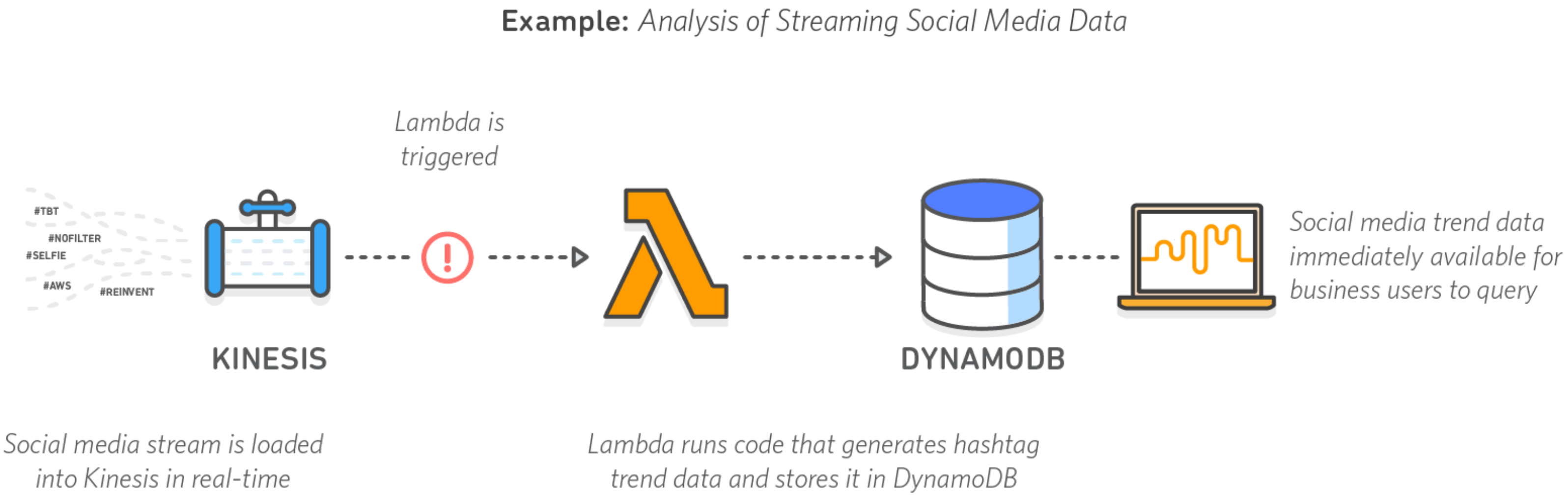
From Microservices to Serverless Applications

- We have broken an application into *stateless* and *loosely-coupled* «microservices»
  - Each microservice is basically deployed as a container (e.g, docker), i.e., a «lighter» virtual machine

- Cloud providers have become even more efficient and offer now FaaS (Functions as a Service)
  - This enables implementing *serverless* applications easily via third-party (*cloud-based*) services



An Example with AWS Lambda



https://aws.amazon.com/it/lambda/data-processing/

Serverless: *Pros* and *Cons*

- Serverless is good for
  - Short-running
  - Event-driven
  - Stateless

- Examples
  - Machine-learning classification
  - Microservices
  - Mobile backends
  - Modest data stream processing
  - Service integration

- Serverless is not good for
  - Long-running
  - Number-crunching
  - Stateful

- Examples
  - Machine-learning training
  - Databases
  - Numerical simulations
  - Video streaming

Slide credit: 3rd Int'l Workshop on Serverless Computing (WoSC) 2018