Proxies, CDNs, and Distributed Databases

Dr. Philipp Leitner

philipp.leitner@chalmers.se

@xLeitix

LECTURE 7

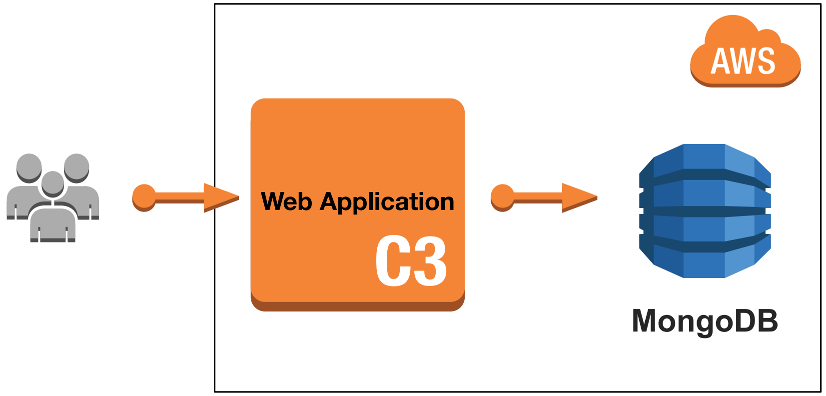
Covers …

Some leftovers related to scalable deployments

(From next week we will move on to the second big part of the course, ops and monitoring)

Architectural Evolution of a Web App
Stage 1 - a MEVN application

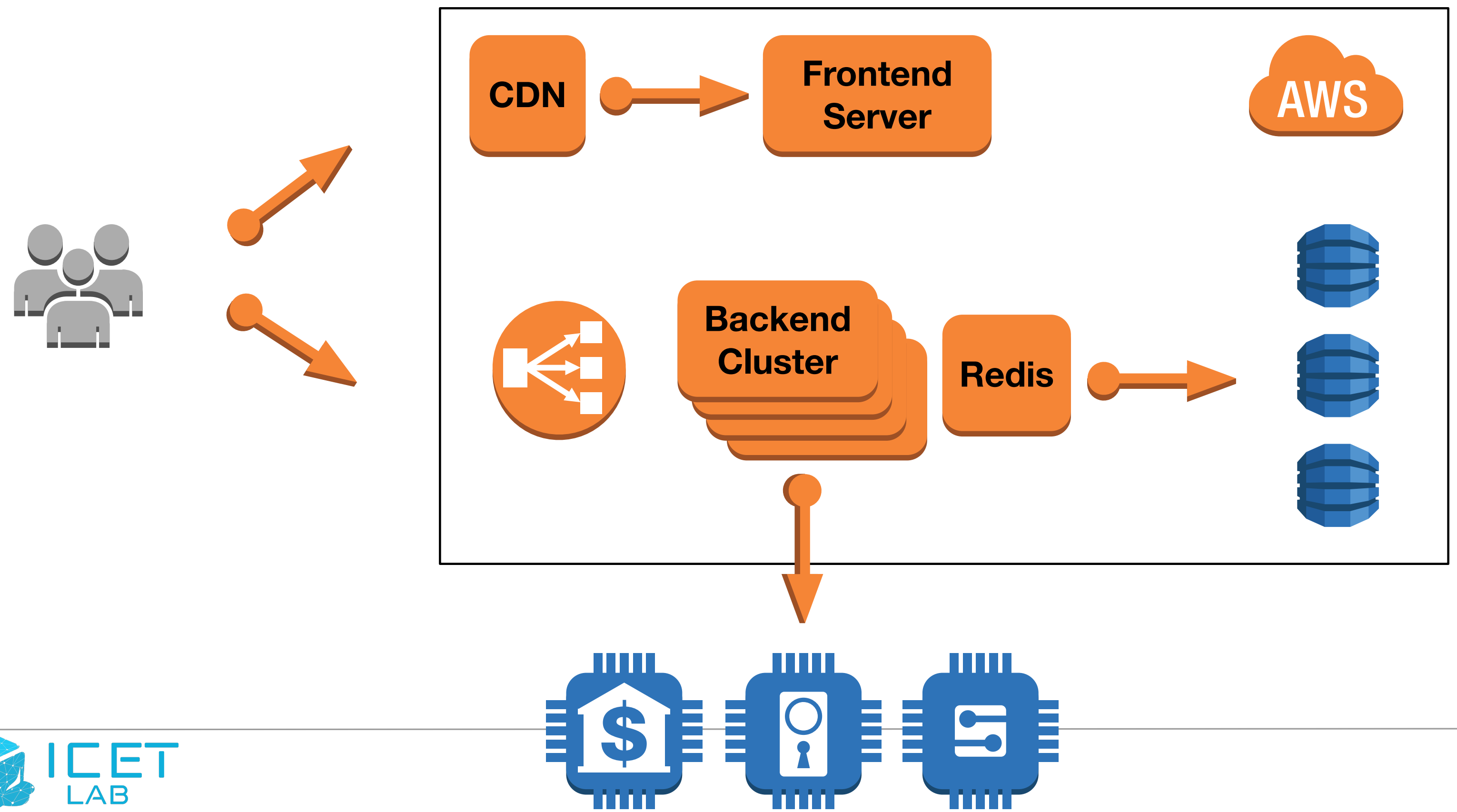
REMINDER



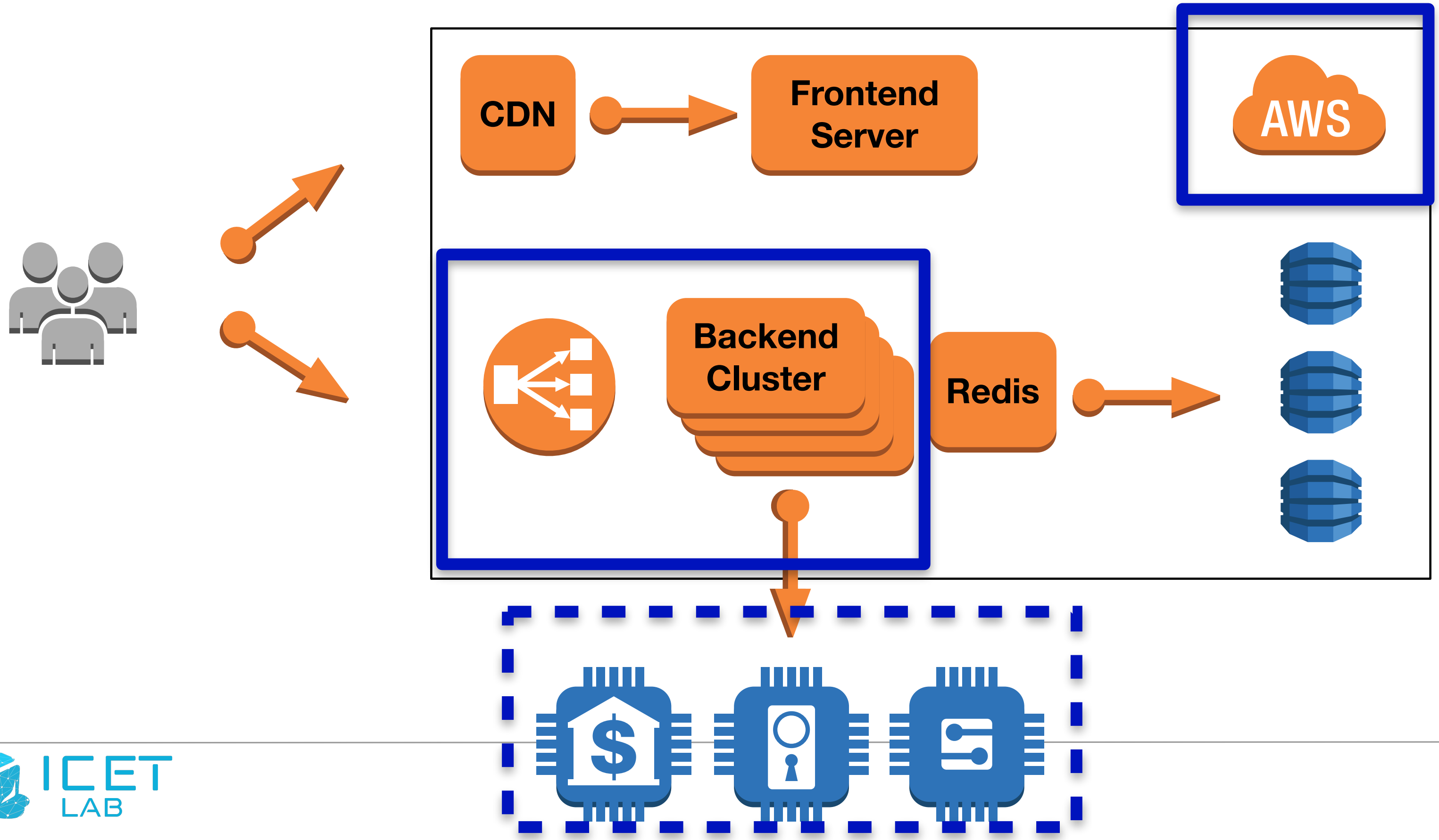
(MEVN - MongoDB, Express, Vue.js, Node)

Architectural Evolution of a Web App
Stage 2 - architecting for (reasonable) scale

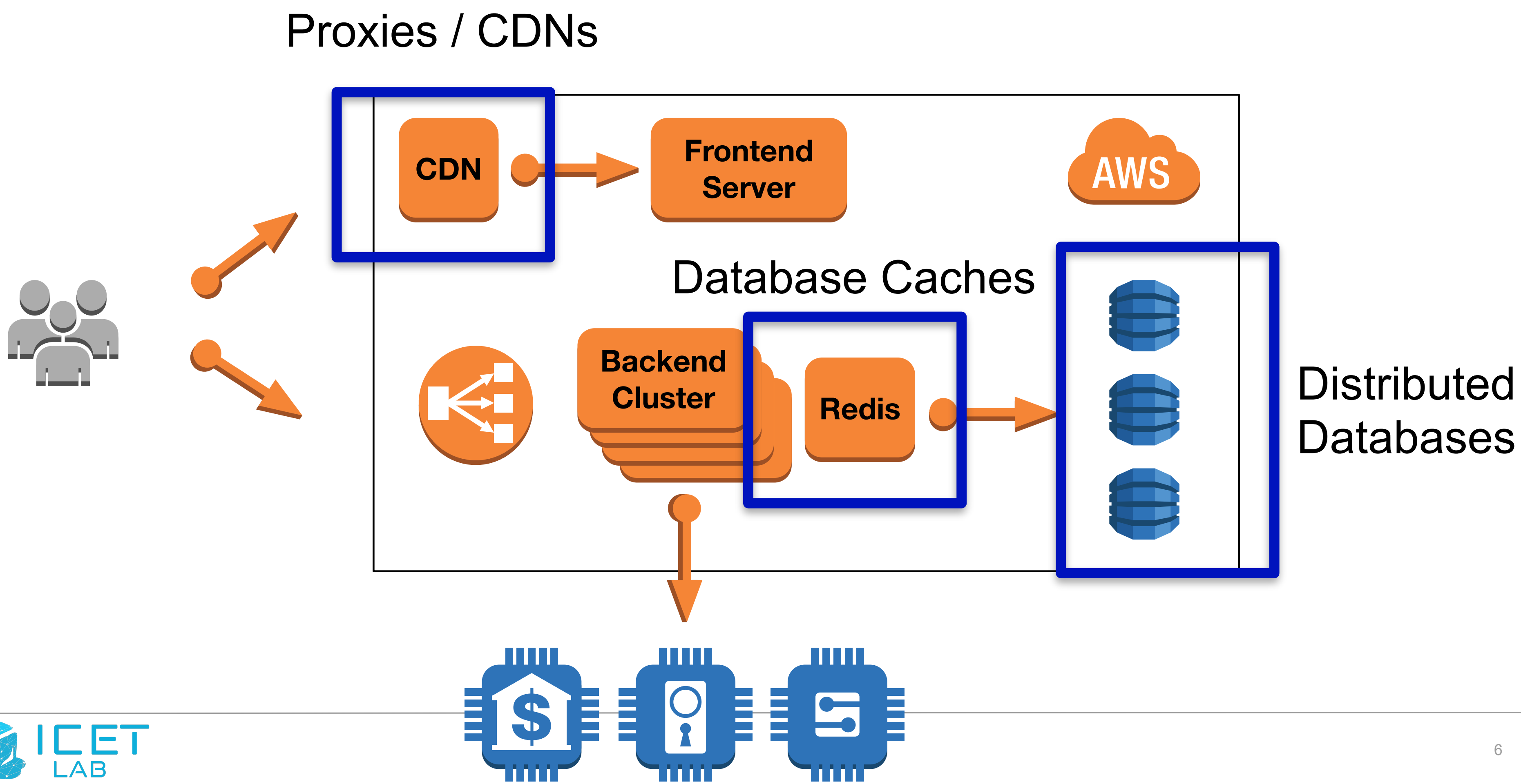
REMINDER



We already covered some of these …



Goal for Today: briefly talk about the rest



An important intuition to get started

Performance in Web systems is dominated by **network delays:**

In-memory function calls (delay <1ns)

much faster than

Inter-Process Communication, IPC (a few ns)

much faster than

Local networking (<1ms)

much faster than

“Close” Internet (delay 20+ milliseconds)

much faster than

“Far” Internet (delay 100+ milliseconds)

1ms = 10^6ns

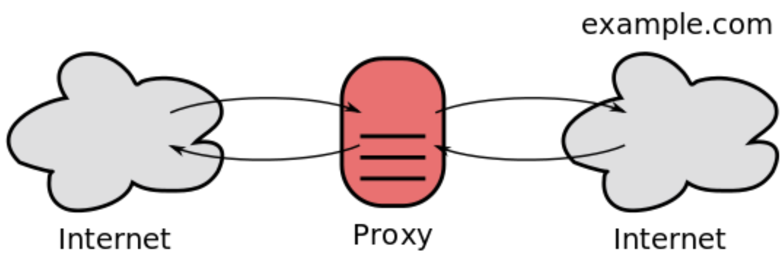
Proxying

REST Constraints

REMINDER FROM LECTURE 2

- Client-Server
- Stateless
- Cacheable
- Layered System
- Uniform Interface
- Code-On-Demand (optional)

Proxy Servers



Source: https://en.wikipedia.org/wiki/Proxy_server

Any intermediary component in a Web system whose responsibility is to accept requests / responses and forward them to a different target system.

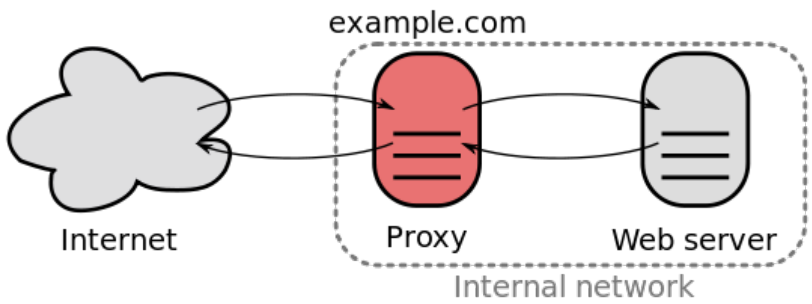
A "middle layer" in REST terminology

Types:

Forward proxies (proxy on client side)

Reverse proxies (proxies that sit in front of one or multiple servers)

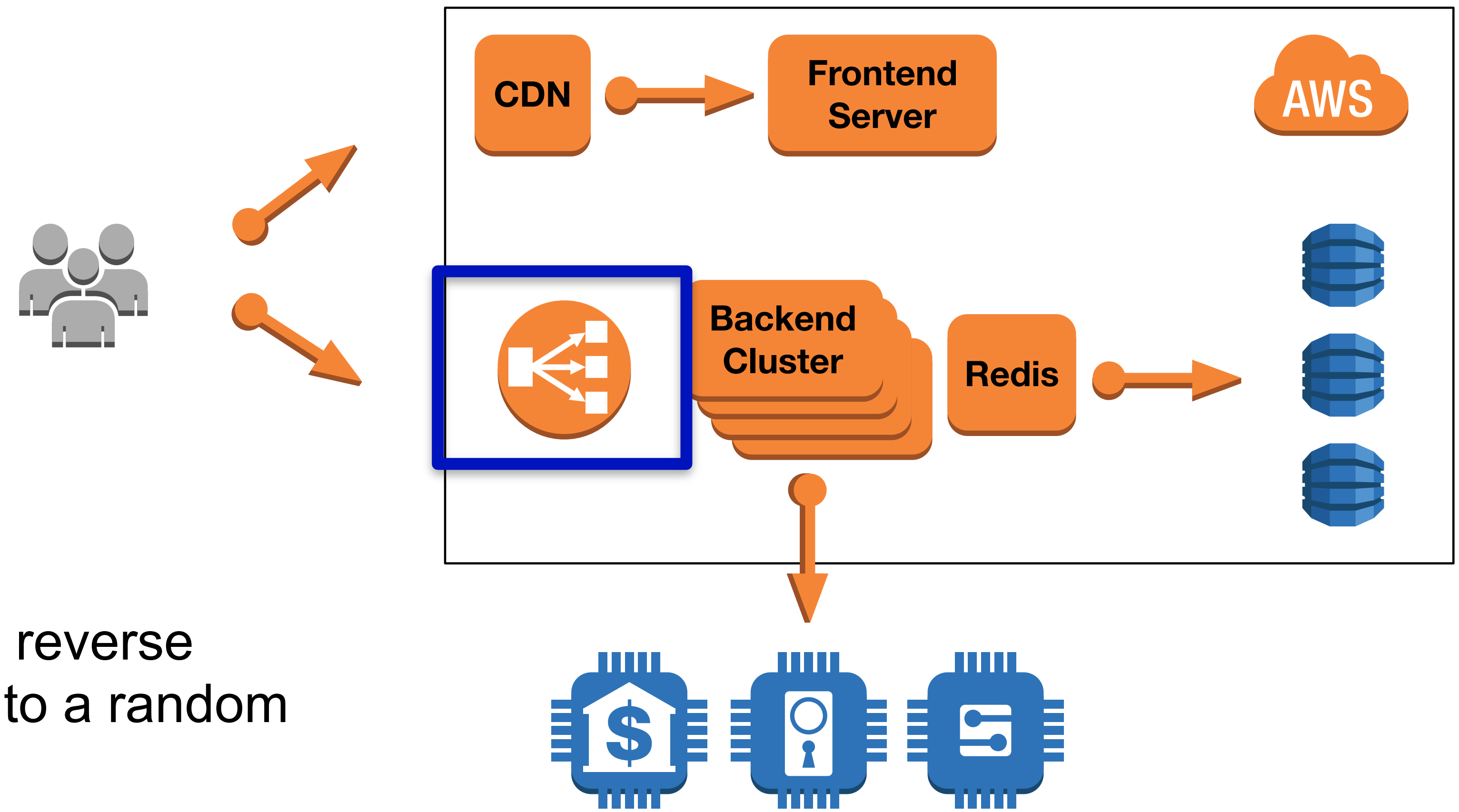
Reverse Proxies



Source: https://en.wikipedia.org/wiki/Proxy_server

Very useful general principle when building scale-out systems:

- To address multiple servers with one address (e.g., Kubernetes Ingress, load balancers)
- As circuit breaker in a microservice based system
- To hide system internals
- To provide HTTPS endpoints
- To detect and mitigate DDoS attacks
- ……..



Example:
A load balancer is a reverse proxy that forwards to a random backend instance

Implementing a Reverse Proxy

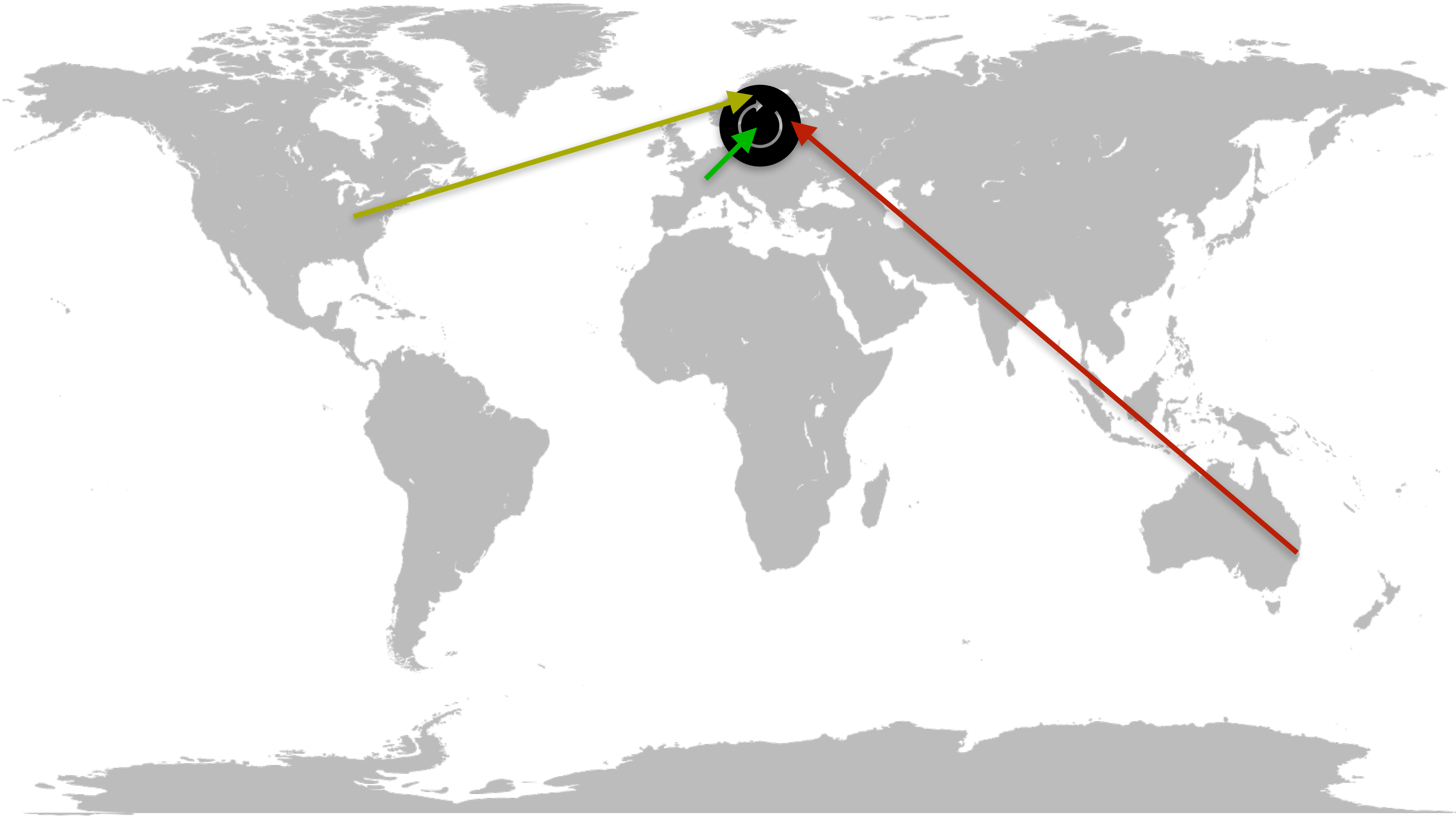
Any web server can be a proxy server

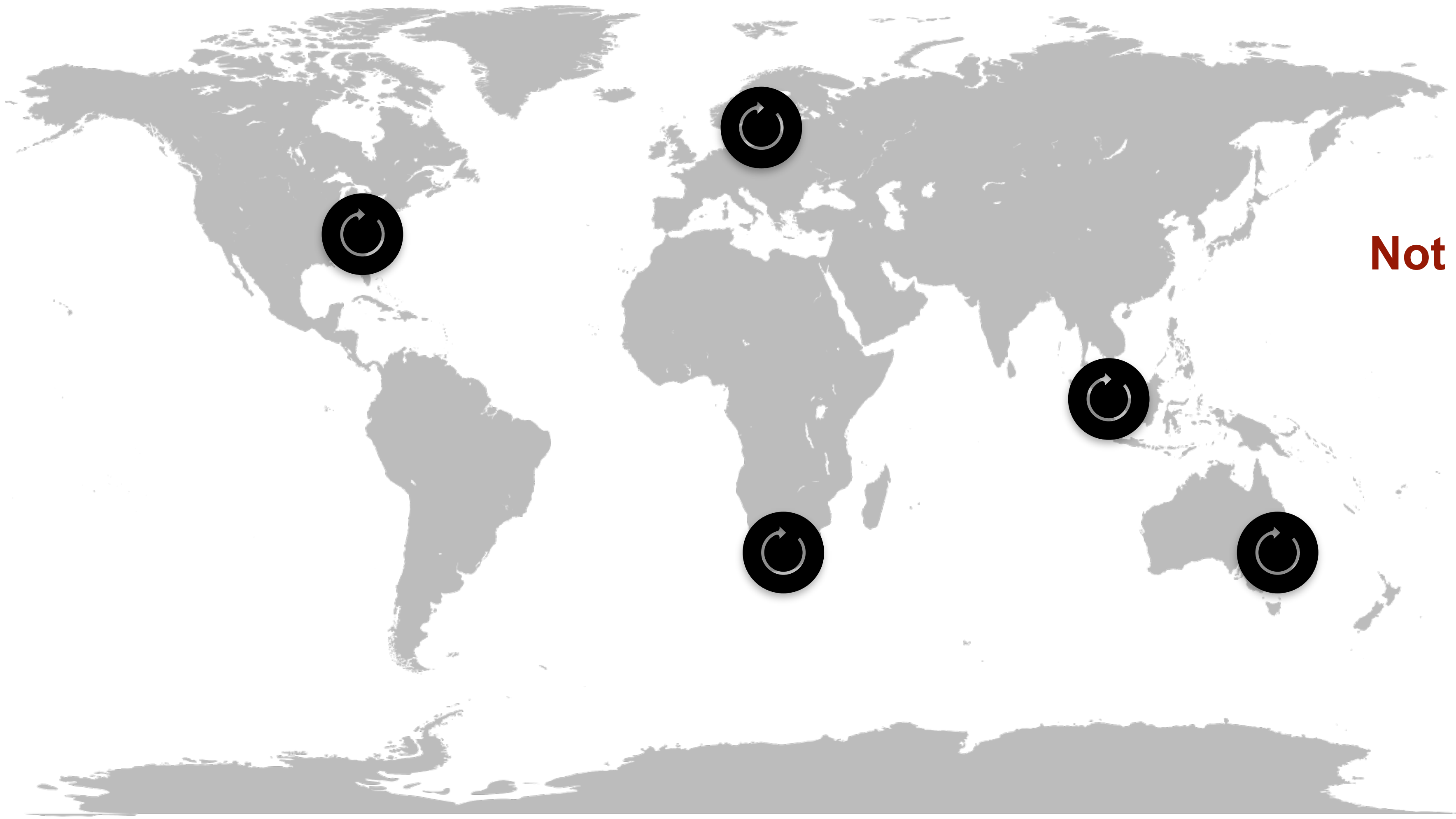
Common choice:

Nginx (https://www.nginx.com)

(fast, production-ready open source web server)

Content Delivery Networks





Not ideal …

Content [Delivery | Distribution] Networks

"A group of geographically distributed servers that speed up the delivery of web content by bringing it closer to where users are."

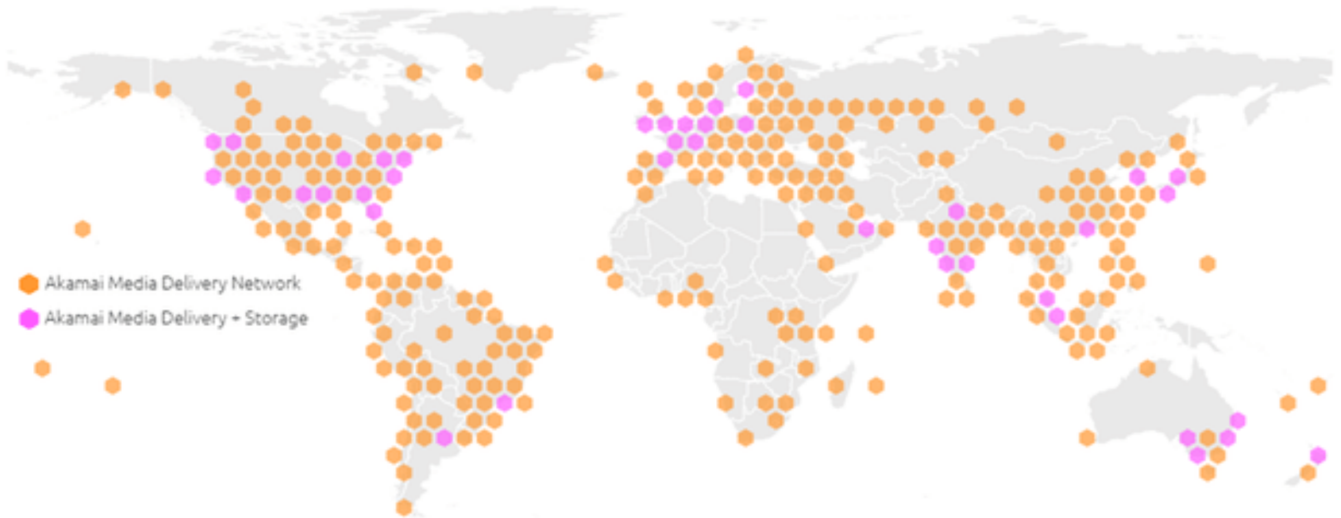
Basically a cache / reverse proxying system that has the explicit goal of shortening the (physical) distance between users and content

Hence: speeding up page load times

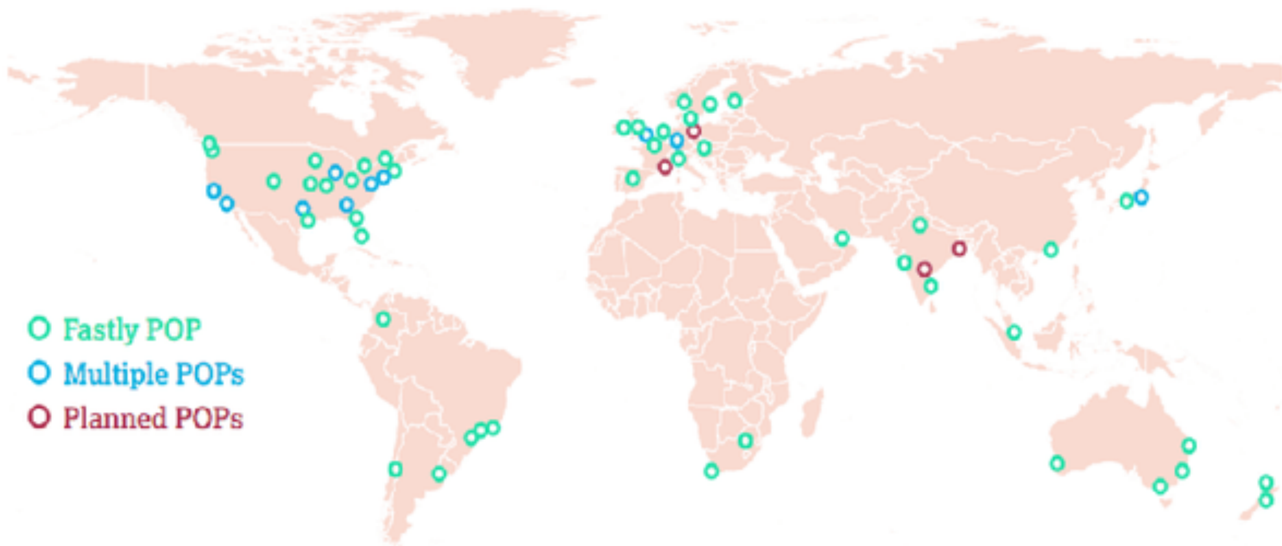
Examples:

Akamai and Fastly

Akamai



Fastly



Source: https://seekingalpha.com/article/4379686-akamai-granddaddy-of-cdn-is-well-positioned-for-next-generation-applications

Static Assets

CDNs are only useful for **static assets**

(any content of your application that *is not* dynamically generated)

E.g.,

HTML pages and CSS stylesheets

Javascript code and libraries (e.g., the ScalyShop Vue.JS frontend)

Images

…

But *not* dynamic service endpoints

`http://scalyshop.com/api/products/13`

(in practice these are often less latency sensitive, as they are called asynchronously)

CDN Advantages

1. Improves **page load time**
   Esp. for customers far away from your data center
2. Improves **scalability** of your frontend
   Since the CDN will handle most requests
   Usually no need to scale your frontend server
3. Improves **availability** of your frontend
   Since the CDN can still serve cached requests while your frontend server is restarting
4. Improves **security** and **resilience**
   CDN can detect, block, or absorb (D)DoS attacks

Disadvantages?

Not so many on a technical level, but …

… due to their prominence / usefulness CDN providers are quickly becoming a **single point of failure** for the entire Internet (next to DNS)

TECH

What is Fastly and why did it just take a bunch of major websites offline?

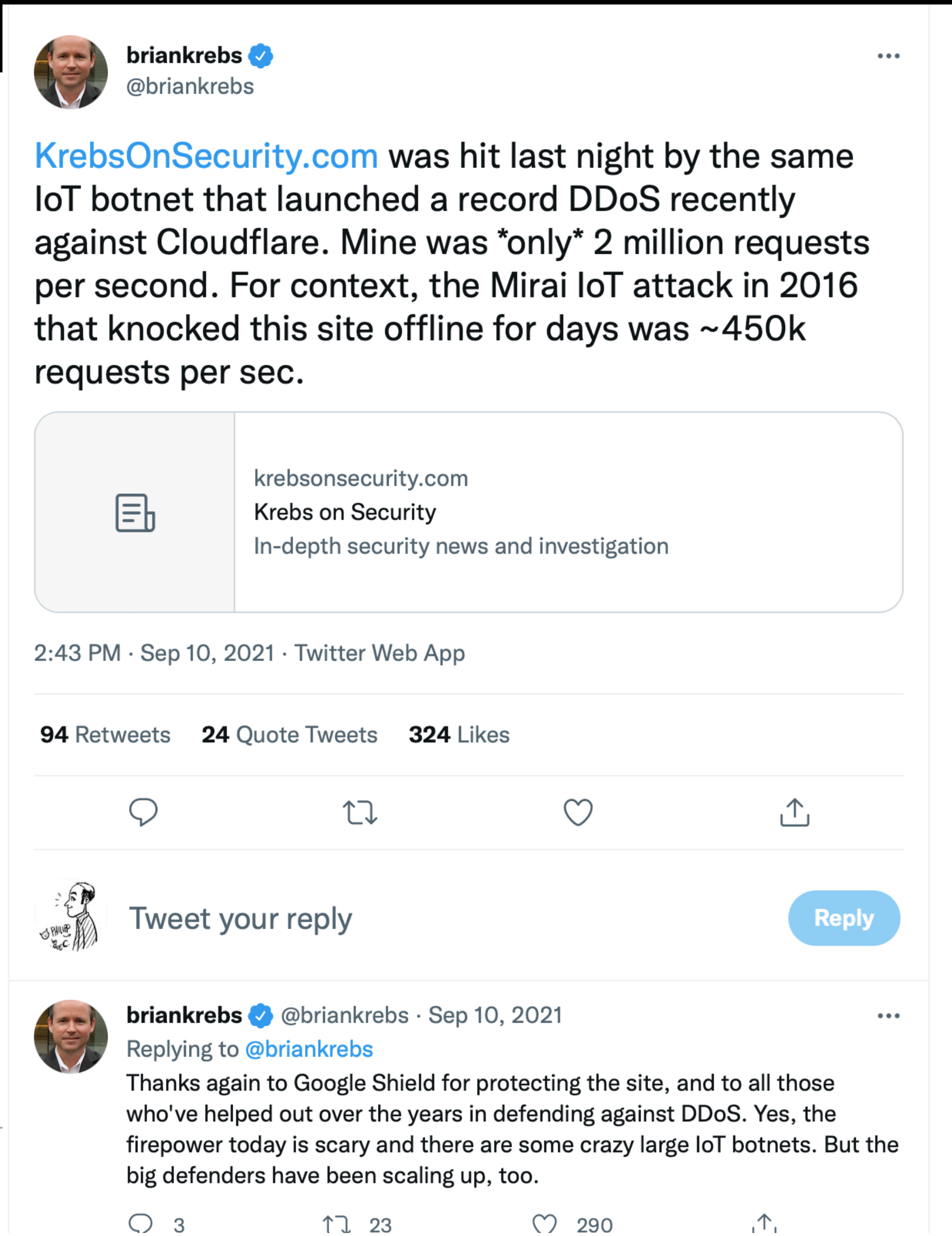
PUBLISHED TUE, JUN 8 2021•10:44 AM EDT | UPDATED TUE, JUN 8 2021•11:26 AM EDT

Ryan Browne @RYAN_BROWNE_ | Sam Shead @SAM_L_SHEAD

SHARE

https://www.cnbc.com/2021/06/08/fastly-outage-internet-what-happened.html

Security

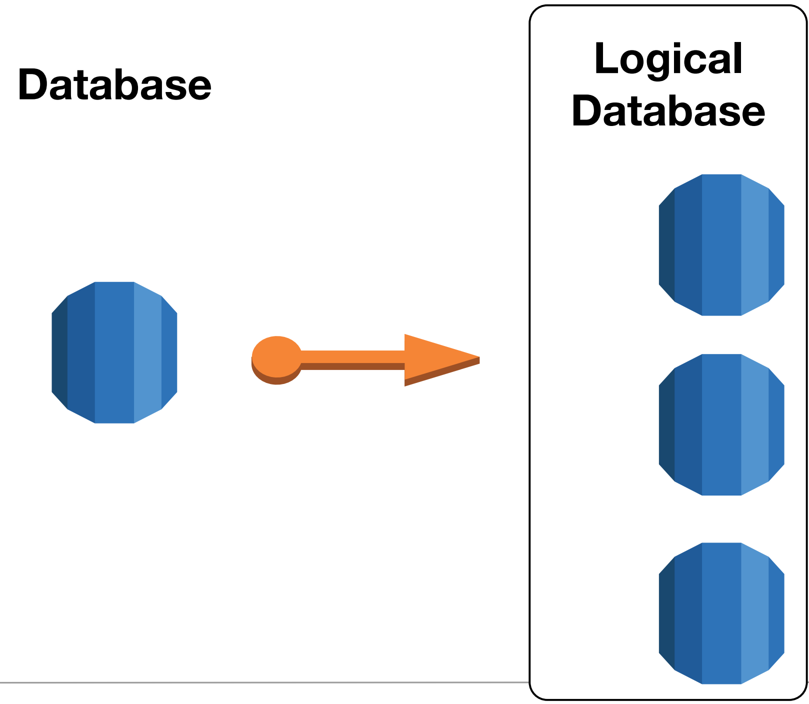

briankrebs
@briankrebs
KrebsOnSecurity.com was hit last night by the same IoT botnet that launched a record DDoS recently against Cloudflare. Mine was *only* 2 million requests per second. For context, the Mirai IoT attack in 2016 that knocked this site offline for days was ~450k requests per sec.
krebsonsecurity.com
Krebs on Security
In-depth security news and investigation
2:43 PM · Sep 10, 2021 · Twitter Web App
94 Retweets 24 Quote Tweets 324 Likes
Tweet your reply
Reply
briankrebs @briankrebs · Sep 10, 2021
Replying to @briankrebs
Thanks again to Google Shield for protecting the site, and to all those who've helped out over the years in defending against DDoS. Yes, the firepower today is scary and there are some crazy large IoT botnets. But the big defenders have been scaling up, too.
3
23
290

Distributed Databases

Distributed Databases

Basic idea very similar to scaling out the backend:

Go from one database instance to *n* database instances



Basic Challenge - Database instances cannot be stateless!

When discussing scaling out the backend, we made our lives easier by requiring **stateless** backend instances.

(so that a load balancer can arbitrarily schedule requests to instances)

The only purpose of a database is to keep state. We **cannot** require our database instances to be stateless.

Only real option is to have them **synchronise** their state.

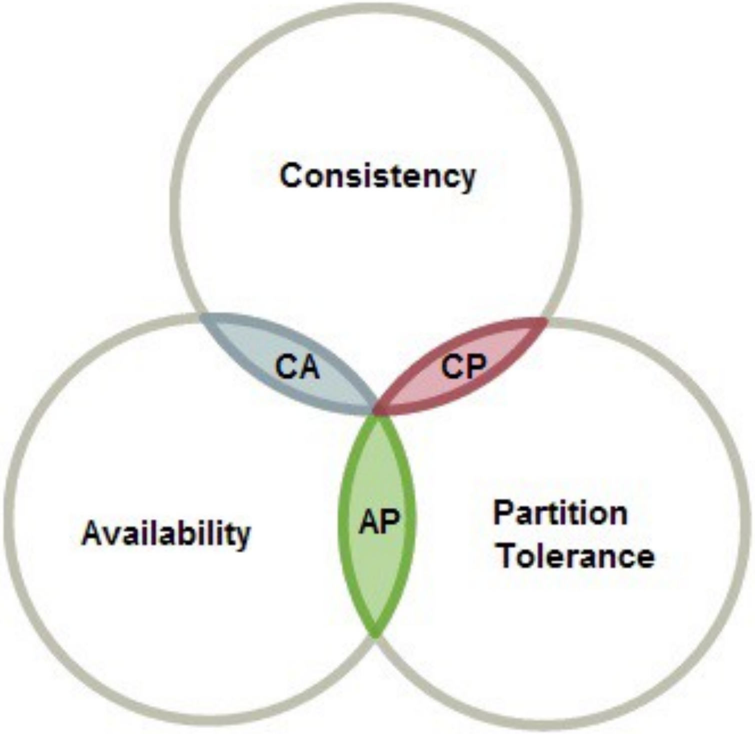
CAP Theorem

Well-known theoretical result for distributed databases:

CAP Theorem

You can have max. two of the following three properties at the same time in a distributed database

Consistency (all nodes have same copy of data)
Availability for Updates (you can at all times **write** to the database)
Partition tolerance (the system can deal graciously with a nodes being unable to talk to each other for some time)


Consistency
CA
CP
Availability
AP
Partition Tolerance

Rule of Thumb

Relational databases (SQL) are **C-databases**
- Strong focus on consistency
- ACID properties
- Example: Postgres, MySQL, Microsoft SQL Server

(Many) NoSQL databases are **AP-databases**
- Strong focus on availability and partition tolerance
- No ACID guarantees
- Eventual consistency (BASE properties)
- Example: MongoDB, myriad of cloud database services

Eventual Consistency

AP databases typically use a concept of **eventual consistency**

- You can always read/write to the system, and the system can deal with partitions
- But you don't have a guarantee of consistency between all nodes in practice
  - Asking two replicas for the same data can lead to different results

Strictly speaking:

- **An eventually consistent database is guaranteed to reach a consistent state in finite time**
  - However, little theoretical guarantees how long it will take
  - And other inconsistencies may come up while the previous ones are being fixed

ACID Properties (C-Databases)

Atomicity

All changes are applied, or nothing is applied

Consistency

Database is always in a consistent state (no “in-between” time)

Isolation

Concurrency control - guarantees that concurrent transactions are not interfering

Durability

Once a change is applied, it remains even through failures

BASE Properties (many AP-Databases)

Tongue-in-cheek alternative to ACID:

BASE

(**b**asically **a**vailable, **s**oft state, **e**ventually consistent)

So why should we distribute the database in the first place?

Two **orthogonal** reasons:

(1) Avoiding the DB to be a single point of failure (increasing **availability**)
(2) Increasing DB response time under high load (increasing **scalability**)

These two purposes require **entirely different** mechanisms!

Distribution for Availability

Avoiding single points of failure (i.e., making the application fault tolerant) means that **the same data needs to be available** in multiple instances

Actually makes it **harder** to manage large volumes of data

Basic principle:

Replication

(mechanism to keep data in sync between different database instances)

Replication Models

Inherent trade-off of replication:

- It's not possible to have two or more DB instances that are always in sync

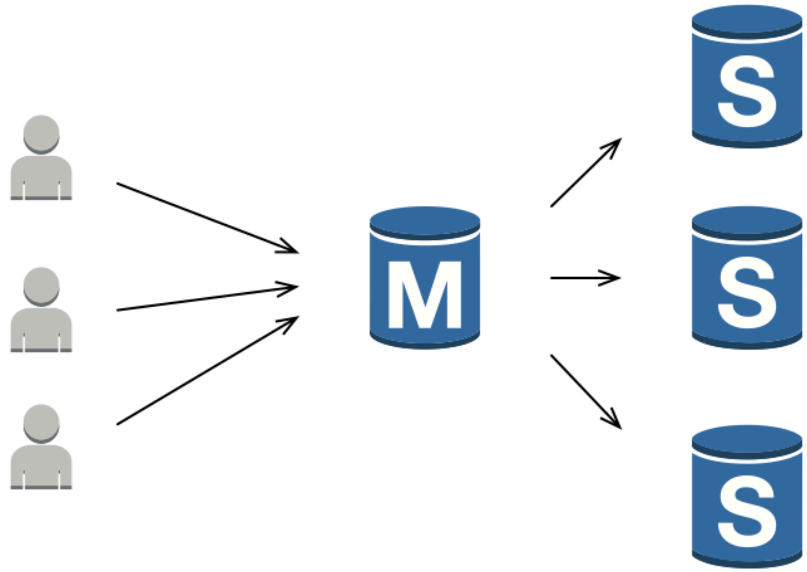
Common models:

- **primary/secondary replication**
- **primary/primary replication**

Primary / Secondary Replication

Primary/secondary replication

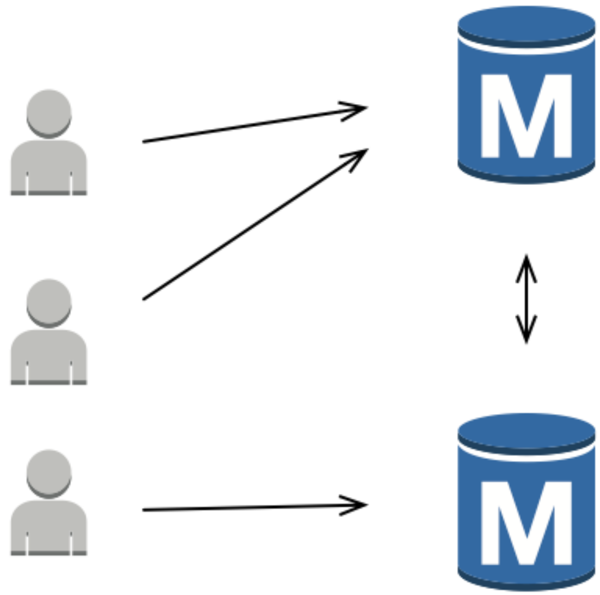
- 1 primary (authoritive copy), 1..N secondary (backups, get activated if the primary is unavailable)
- Election procedure is used to promote one secondary to primary if previous primary becomes unavailable



Primary / Primary Replication

Primary/primary replication

- Or: decentralised replication
- Multiple primaries
- Voting is used to resolve inconsistencies



Distribution for Scalability

Imagine you are a company like Facebook

- It’s evident that your data won’t all fit into a single database instance

Basic mechanism for dealing with data at scale: **sharding**

- Fancy name for splitting up data between multiple instances

Like in replication, you have multiple database instances

Unlike in replication, the goal is **explicitly not** that they keep the same data

Sharding strategies

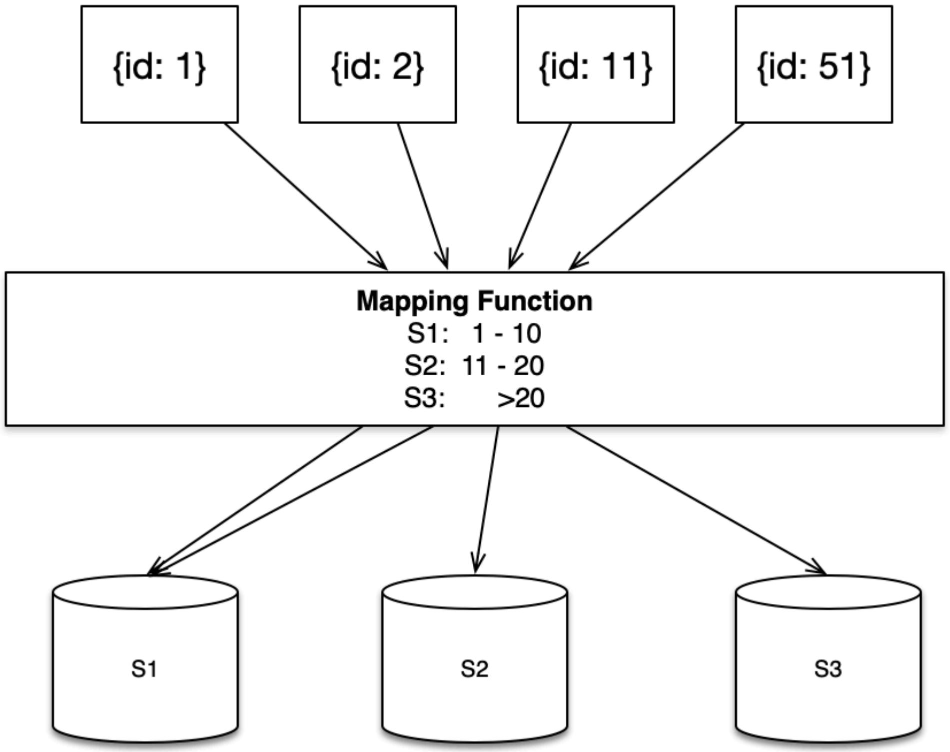
Basic principle:

- Map content to instances (shards) based on a **mapping function**

- Most common:
  - Select an attribute / field value that all content to map has, figure out the domain of this field, and distribute all possible values of the domain evenly across shards

Distribution for Scalability

Sharding



Mapping Functions

Criteria for a good **mapping** function:

- Operates on field(s) all documents have
- Fast to compute
- Splits the documents ~ evenly

Common choice:

Hash function on the document id

Example for *n* shards:

```
shard_id = document_id % n
```

Combining Sharding and Replication

In practice we often want to **combine** sharding and replication

Improve scalability **and** availability

Option 1 (combining primary/primary replication with sharding)

(1) Use sharding through a different mapping function

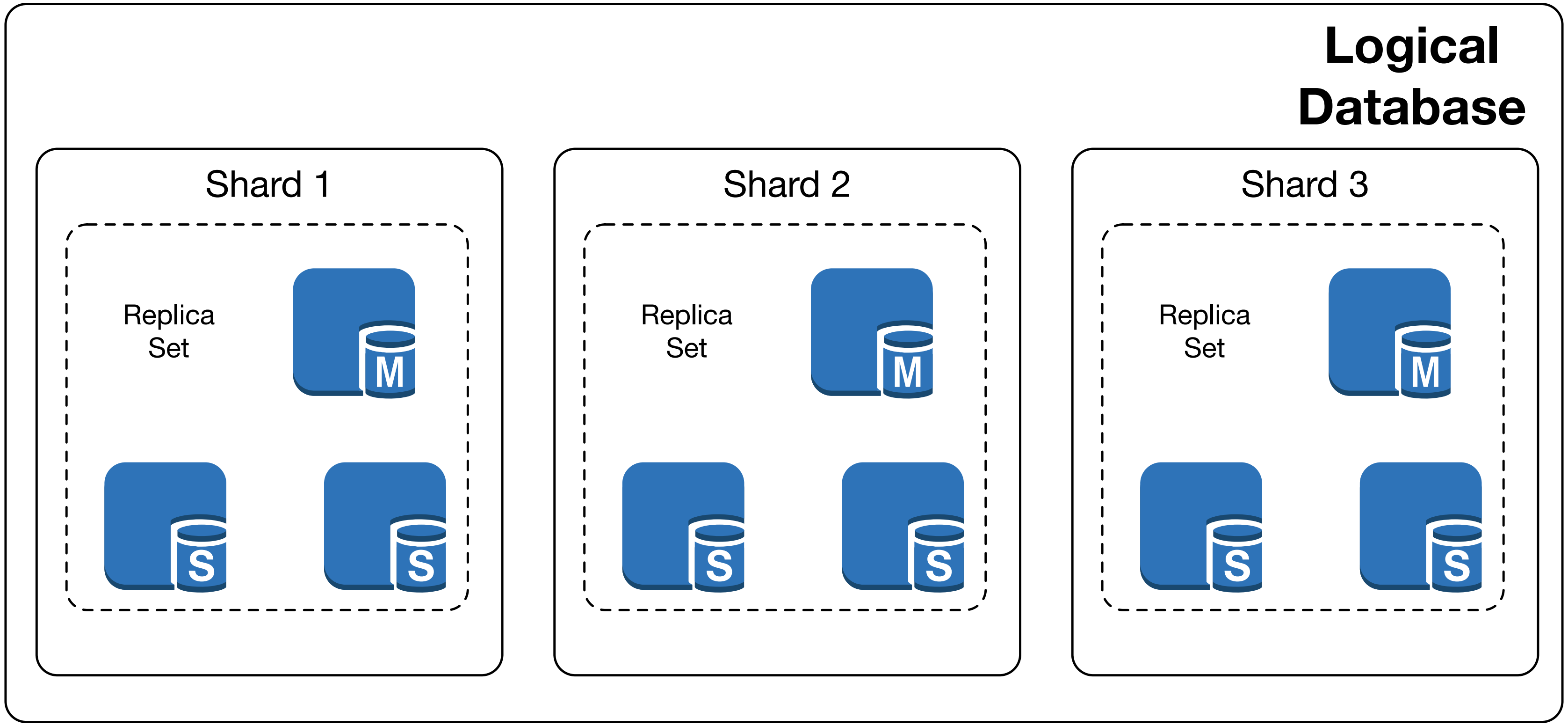
(2) Map each element to **2+ shards**

Option 2 (combining primary/secondary replication with sharding)

(1) Use sharding through a standard mapping function

(2) **Replicate each shard** using a primary/secondary scheme

Distribution in MongoDB



Distribution in MongoDB

Sharding:

https://docs.mongodb.com/manual/sharding/

Replication:

https://docs.mongodb.com/manual/replication/

Database Caching

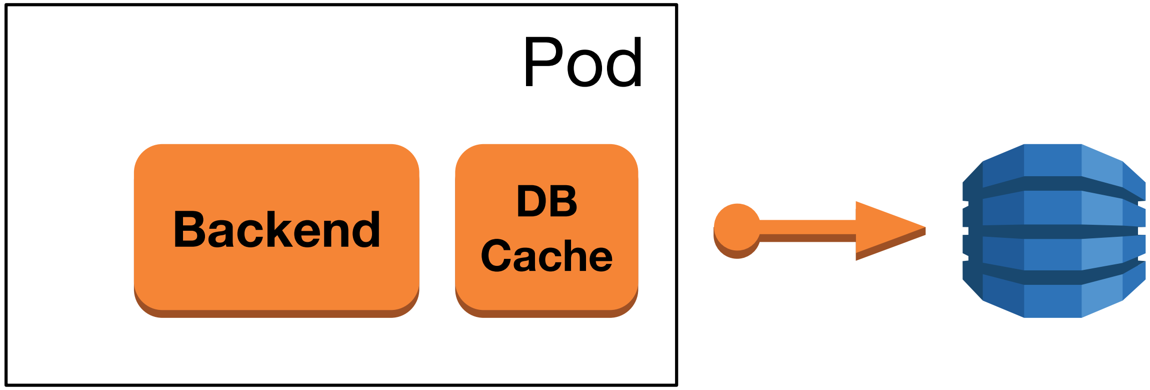
Sharding allows us to speed up database instances at scale

But even a **fast** database call is still **very costly** in terms of response time

Some basic observations:

(1) Most database requests are **read requests** (queries)
(2) Most read requests are **repetitive**
(3) Read requests are (sometimes) **expensive**

Use **caching** to avoid repetitive database reads


Pod
Backend
DB
Cache

Two issues you need to solve:

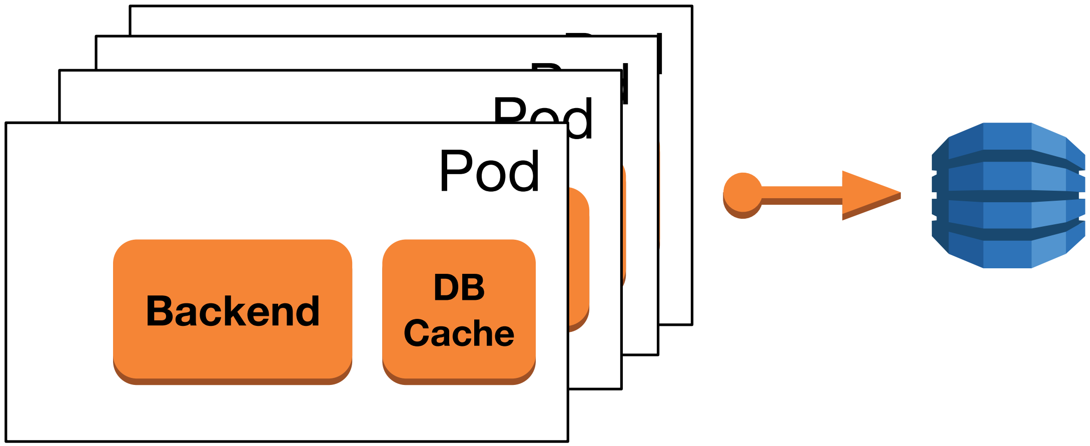
(1) **Cache Invalidation** - when does a cache entry become invalid?

- **Active** - explicitly invalidate tainted entries after write requests
- **Periodic** - cache entries have a validity date and are invalidated after

(2) **Cache Expunge** - which cache entries to prioritise when expunging?

- **LRU** - when space runs out, expunge the entries that have last been used the longest time ago

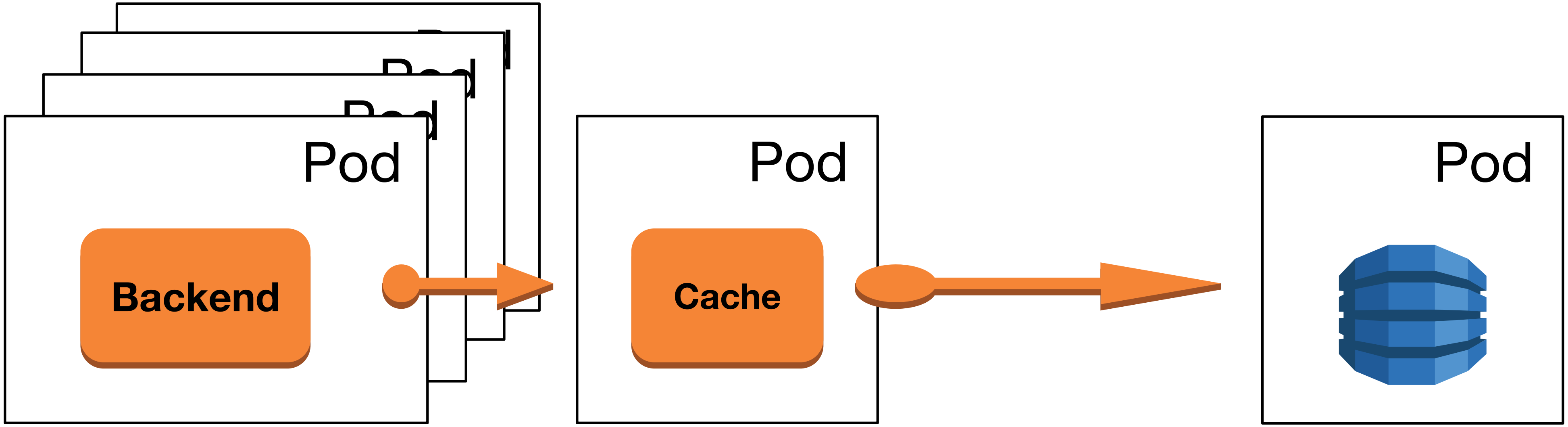
Problem - how does this work with a scaled backend?



Not very well :(

- Cache isn't shared between instances (inefficient)
- Active cache invalidation is a problem

Solution: Distributed Caching



A distributed cache is really just a very fast database (fast in terms of read speed)

Achieved by:

In-memory database (does not write to disk, keeps data in memory)

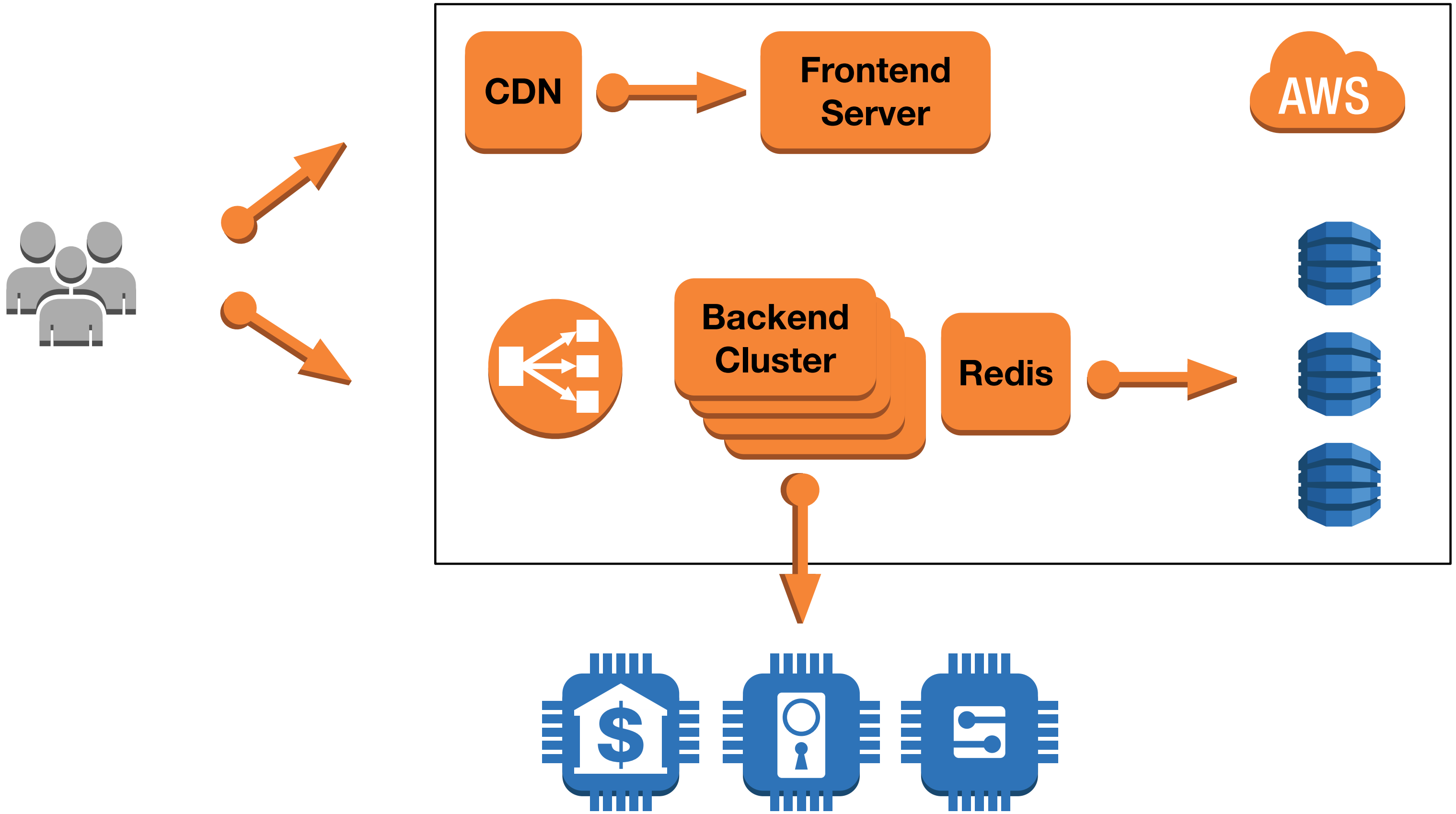
Very **simple** query model (key/value)

No features for consistency, replication, sharding, transactions, …

Potentially: **co-locating** with backend instance pods (same Kubernetes nodes, if possible)

Common choice for caching: Redis

"Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache, and message broker."


CDN
Frontend Server
AWS
Backend Cluster
Redis