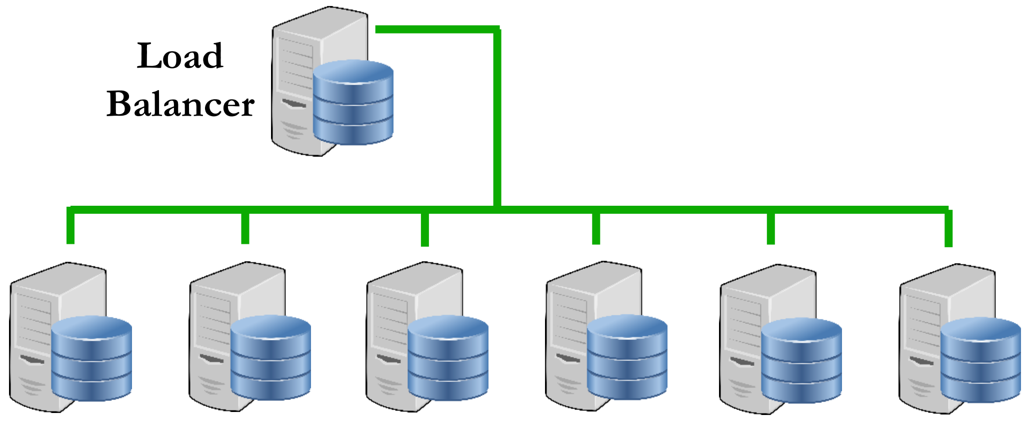
Consistency and CAP

- Types of consistency
- Examples
- CAP theorem

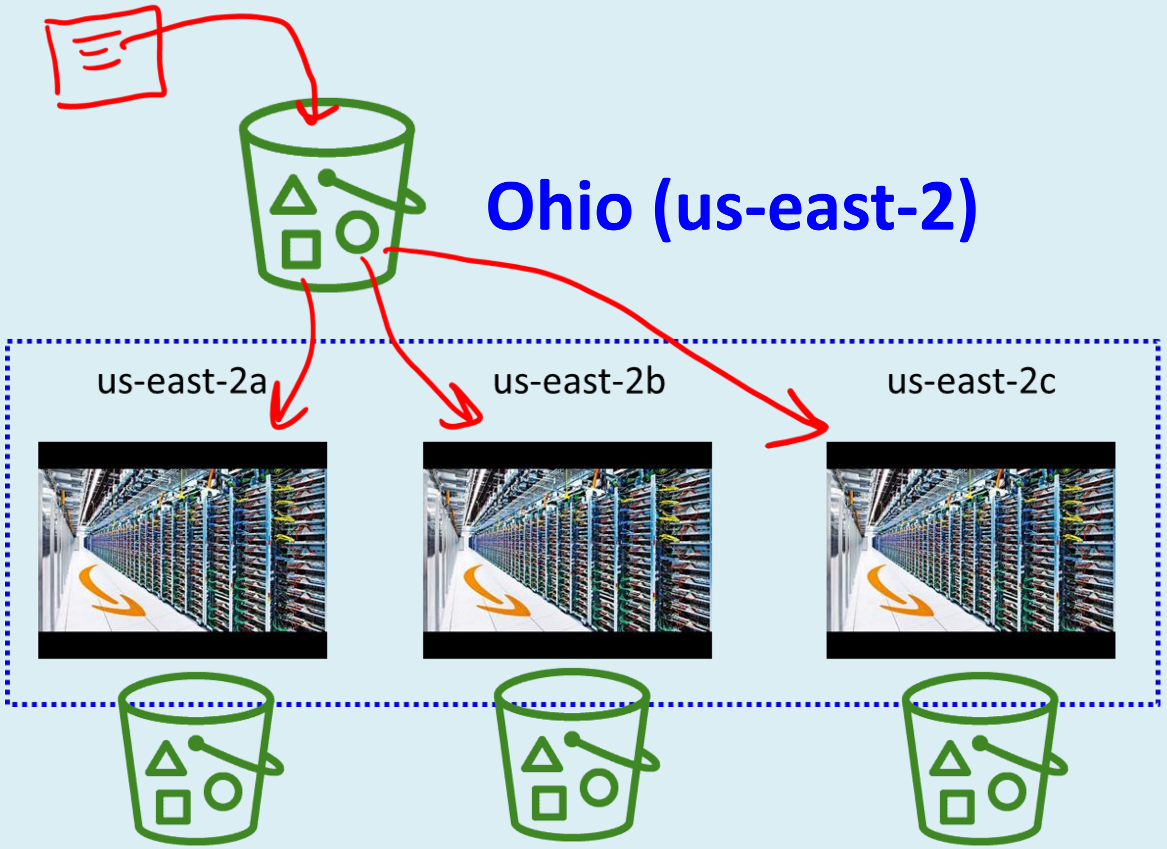
Availability / fault tolerance

- Computers crash, it will happen…
- Only way to keep your system **available** is with multiple computers
- A system that keeps running in presence of failures is **fault tolerant**



Example: S3 is fault tolerant

- Buckets are replicated across all sites in a region in case one of the sites loses power / network connectivity…

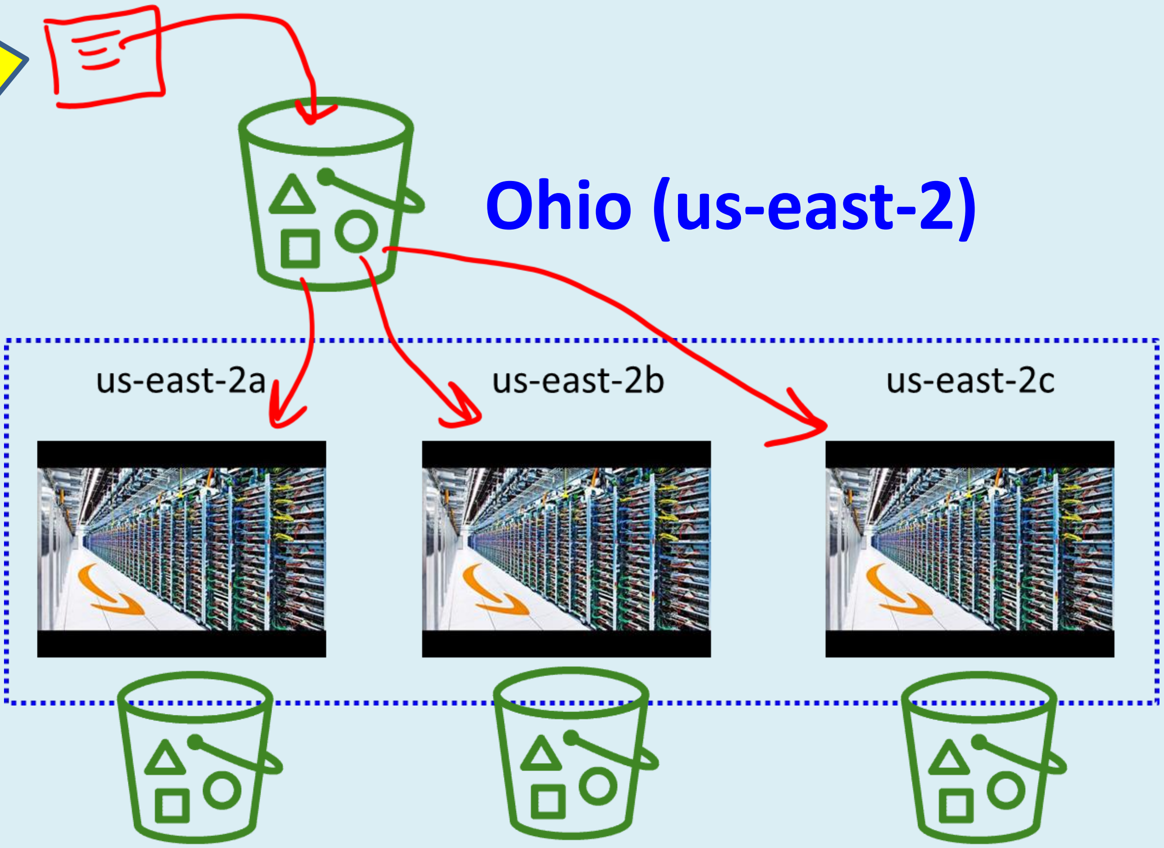


Interesting question…

- **What type of consistency does AWS guarantee?**

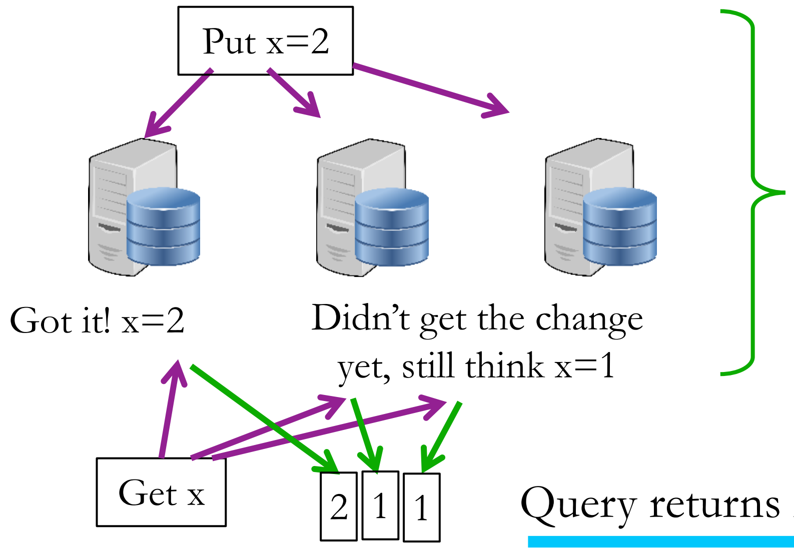
If we upload an image to S3, how soon is it available in all 3 sites?

What if a download request is routed to a site that hasn't finished replicating? What happens? What does AWS guarantee, if anything?



Consistency

- Whenever data is replicated, there is a possibility of **inconsistency**.
  - Example: x =1. An update is then sent to three replicas:



Communication and queueing delays are unpredictable. We can send messages, but we don't know exactly when they'll arrive.

Query returns inconsistent answers!

Types of consistency

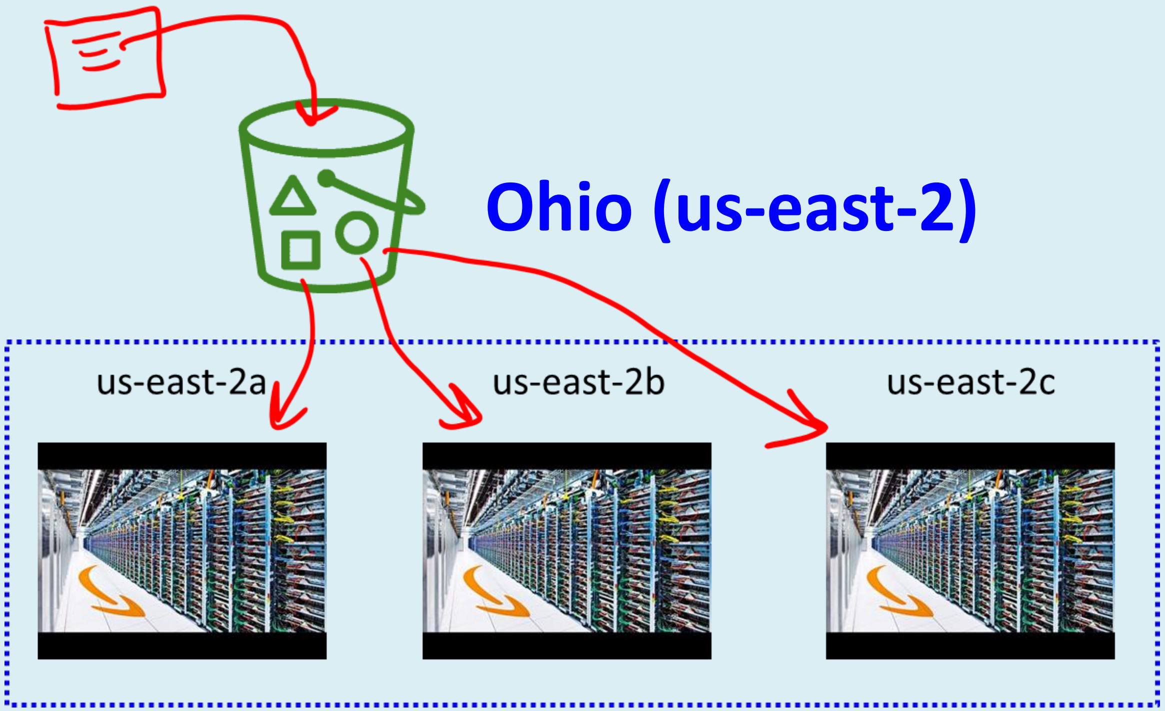
- **Eventual**
  - *An update to a distributed system will *eventually* yield a consistent view*
  - *BASE:*
    - *Basically-available*
    - *Soft-state*
    - *Eventually-consistent*

- **Strong**
  - *A distributed system provides a single, consistent view at all times*
  - *ACID:*
    - *Atomic*
    - *Consistent*
    - *Isolated*
    - *Durable*

The type of consistency impacts how you call / use a given service. Can you trust the response you get?

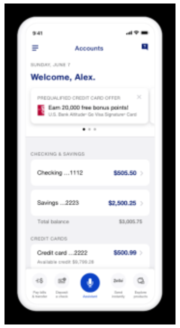
S3? Answer…

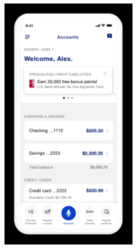
- **The image will be available for download no matter which site is accessed**
  - *AWS S3 guarantees **strong consistency***
  - *The client's view of S3 is THE SAME across all sites*



Banks require strongly consistency

- Suppose a bank account has exactly $1,000
- How does the bank prevent two different people from withdrawing $1,000 from that account at the same time?



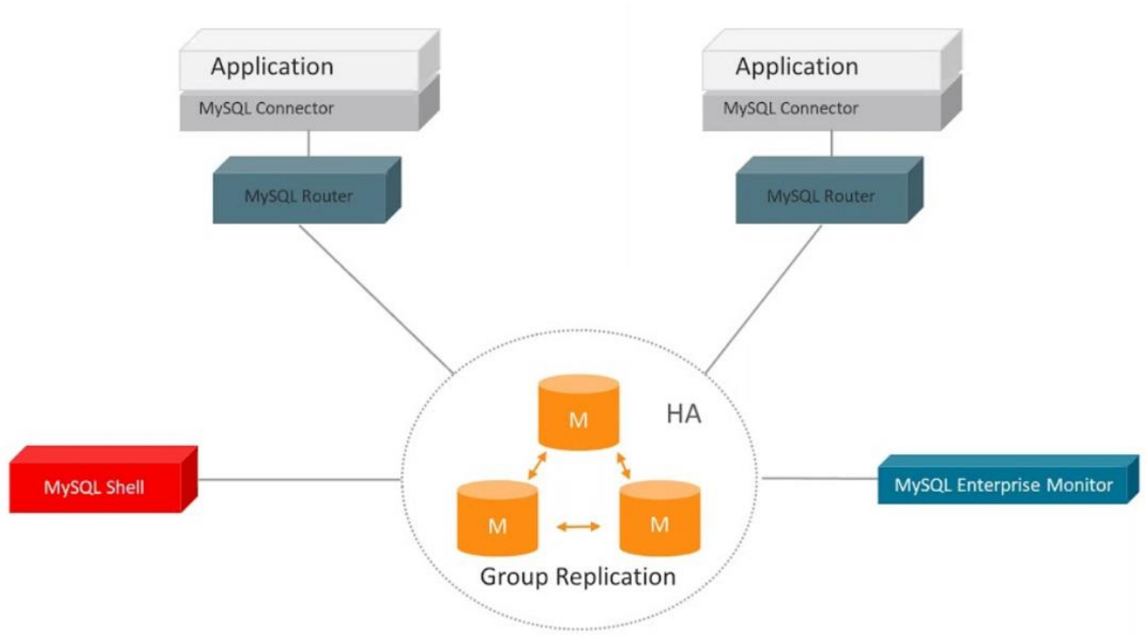


Databases are strongly consistent

- **Most DB systems offer strong consistency**
- **This is HARD to implement:**
  - *Must replicate data so it's accessible at all times anywhere*
  - *Must ensure a strongly consistent view of all data!*

MySQL Cluster

- **MySQL Cluster is a <u>distributed</u> database system**
  - *Replicates the data across multiple servers for fault tolerance*
  - *Still guarantees strong consistency*



MySQL High Availability Cluster

Example

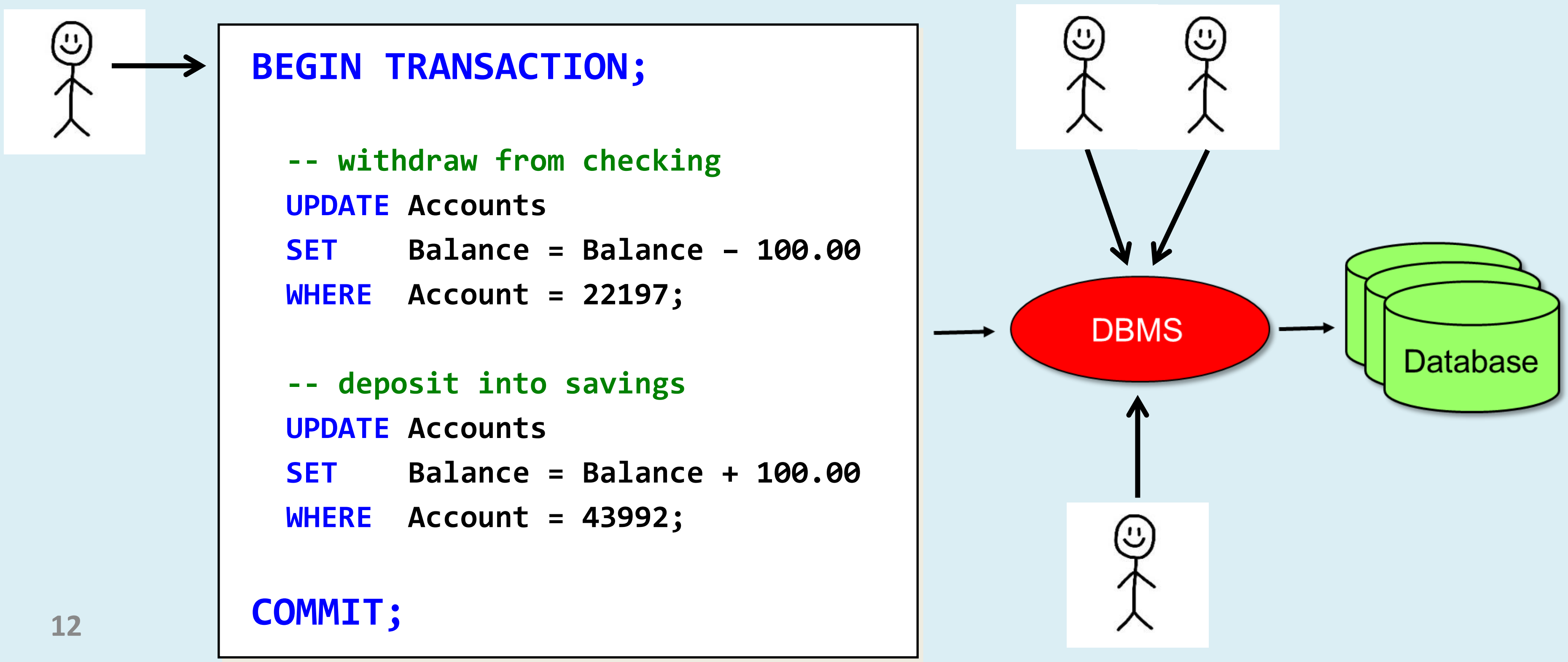
- Banks cannot risk losing track of customer $
- Banks use a DB to keep track…
- Transferring $ requires two SQL queries:

```
-- withdraw from checking
UPDATE Accounts
SET    Balance = Balance – 100.00
WHERE  Account = 22197;

-- deposit into savings
UPDATE Accounts
SET    Balance = Balance + 100.00
WHERE  Account = 43992;
```

What if the computer crashes right here? The money would be lost… What must be done to ensure strong consistency?

Answer

- You must wrap the SQL within a transaction
- Databases are strongly consistent only in the presence of properly-written transactions…



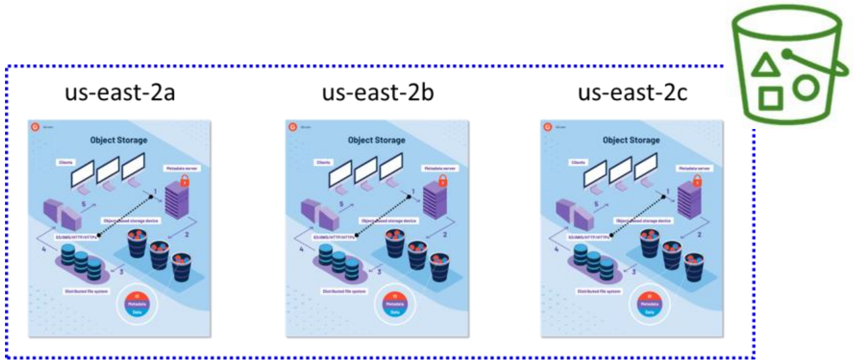
```
BEGIN TRANSACTION;

   -- withdraw from checking
   UPDATE Accounts
   SET    Balance = Balance – 100.00
   WHERE  Account = 22197;

   -- deposit into savings
   UPDATE Accounts
   SET    Balance = Balance + 100.00
   WHERE  Account = 43992;

COMMIT;
```

CAP Theorem

One of the most important results in distributed systems theory

Theorem: a distributed system cannot achieve ***all three*** of the following:

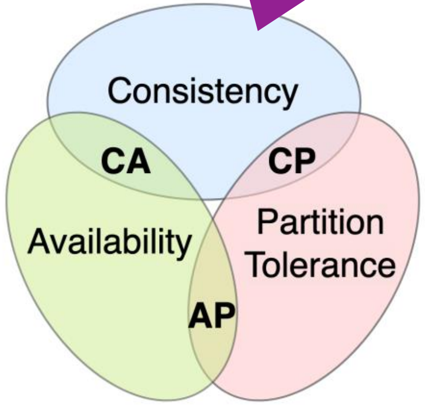
- **Consistency**: reads always return the most recent write (or an error)
- **Availability:** every request receives a timely, non-error response
- **Partition tolerance:** the system continues to operate even in the presence of failures (software, hardware, network, power, etc.)



You have to pick two…

CAP Theorem implications

Pick Two: CA, CP, AP



- **Consistency**: reads always return the most recent write (or error)
- **Availability:** every request receives a timely, non-error response
- **Partition tolerance:** the system continues to operate even in the presence of failures (software, hardware, network, power, etc.)

redundancy k2 partition tolerance
but that means inconsistency if we want results fast
or slow if we want results consistent

Implications?

Redundancy is the only way to achieve partition tolerance, i.e. fault-tolerant, highly-available systems. **So you always choose P.**

This implies architects have to choose between

- **Availability (AP)**: returning an answer to the client that may be inconsistent (old)

or

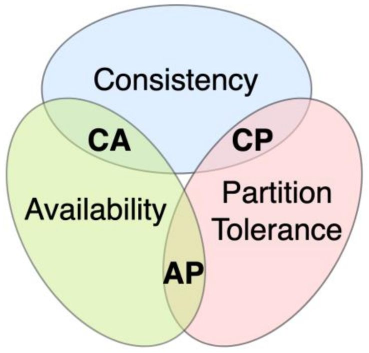
- **Consistency (CP)**: making the client wait until consistent answer is available (in the worst-case, request could error/timeout & client will have to try again)

Client-centric consistency models

- The CAP theorem gives us a tradeoff between **consistency & delay**
- Inconsistency is bothersome --- makes client-side programming harder
  - *Example: imagine if the database gave different answers to the same query?!*
- Delay is usually something client-side apps can handle…

slower, but consistent and fault tolerant

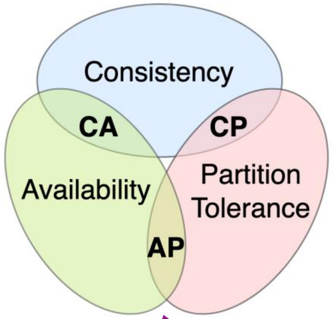
Most architects agree that CP is the right choice



- Client-side code is less complex
- In almost all systems, you want correct responses over fast (but wrong) responses
- Good summary from a Google architect

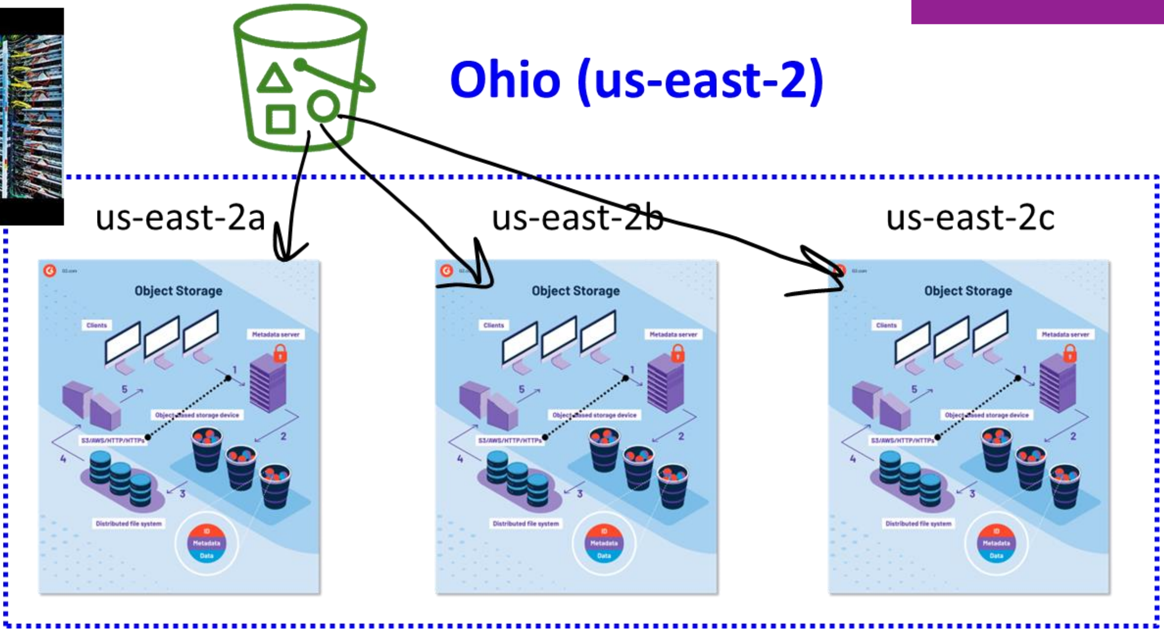
S3: eventual to strongly consistent

- In 2020, S3 was redesigned to be strongly consistent
  - *https://www.allthingsdistributed.com/2021/04/s3-strong-consistency.html*

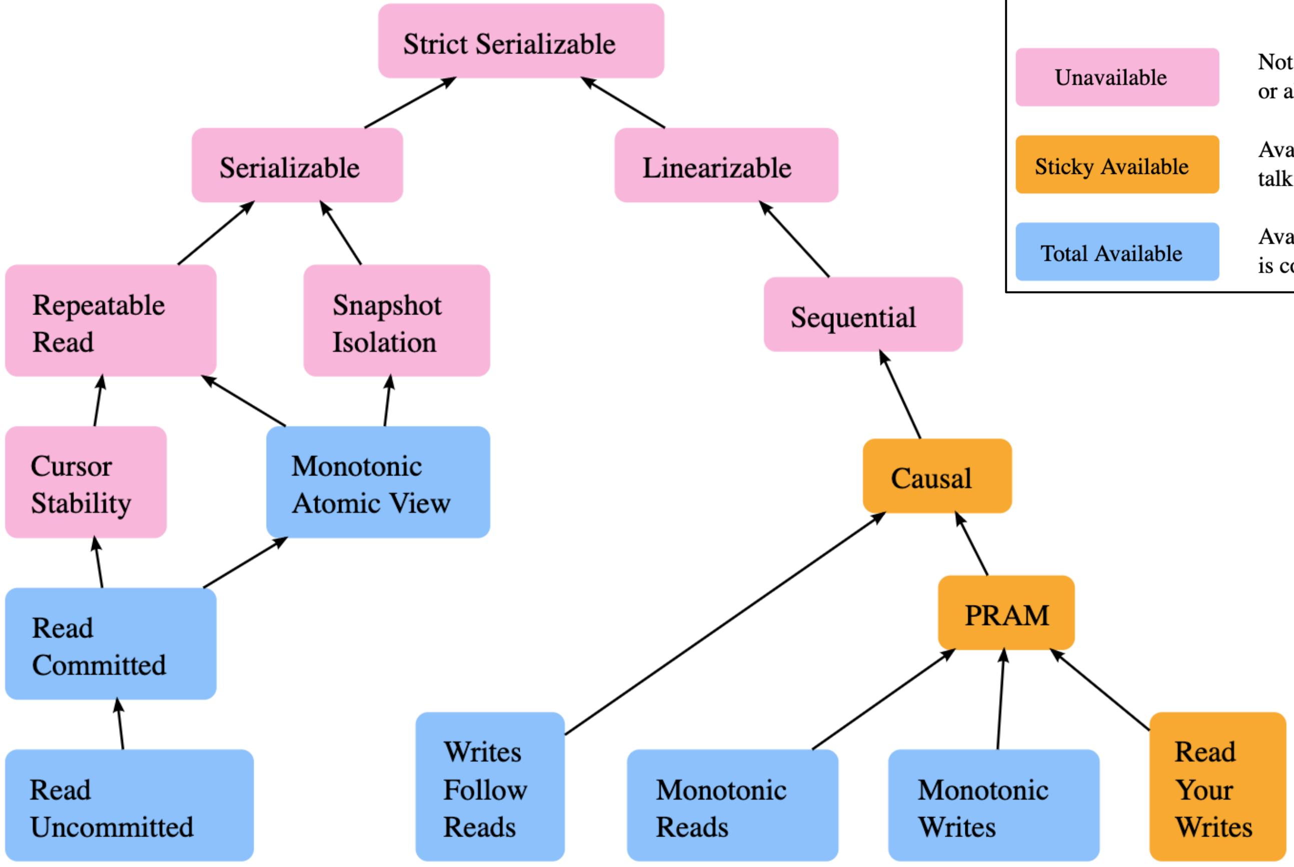


What did they give up?

avail

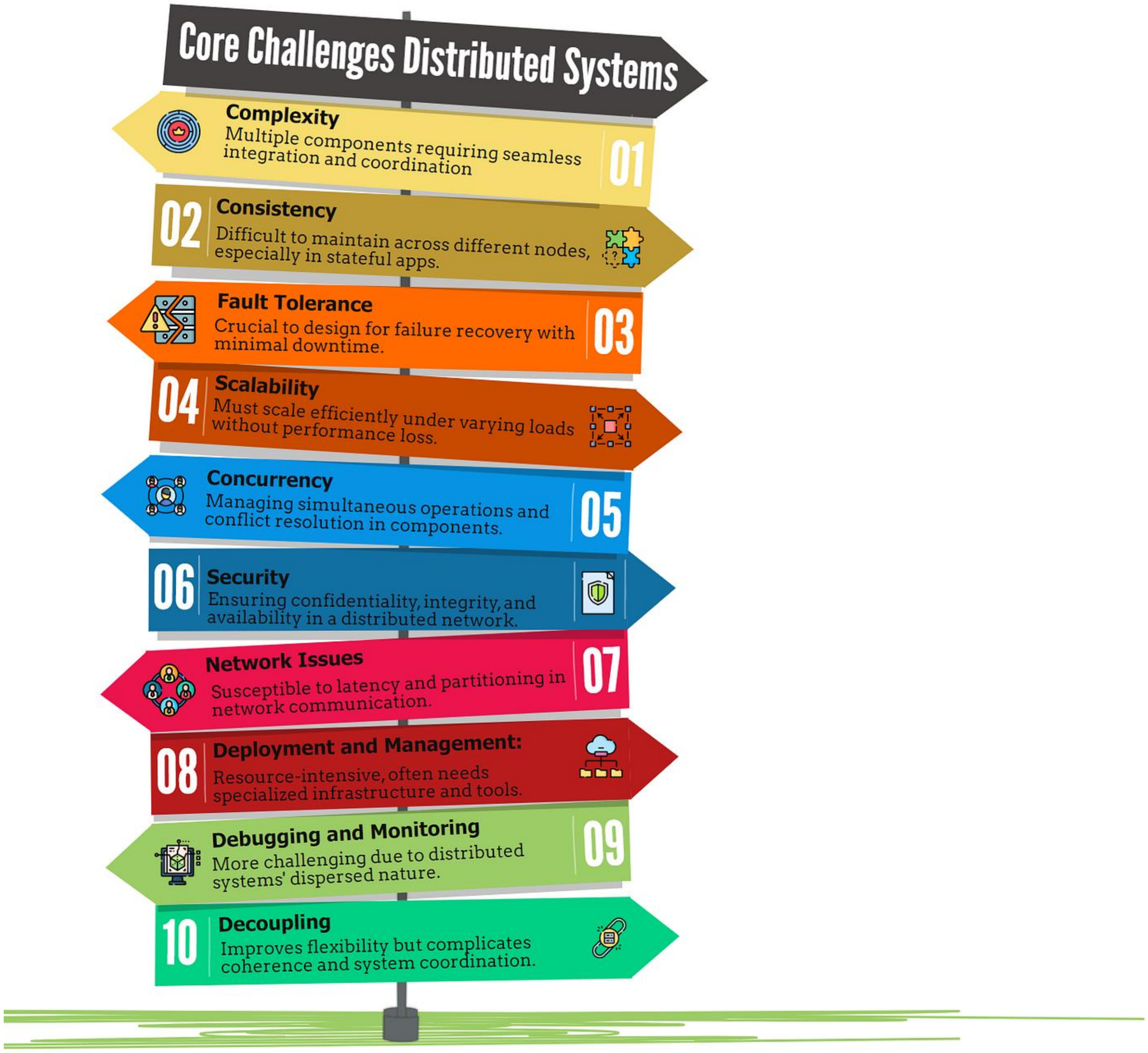


Consistency is a subtle topic, with many models.



Legend

🟪 Unavailable	Not available during some types of network failures. Some or all nodes must pause operations in order to ensure safety.
🟧 Sticky Available	Available on every non-faulty node, so long as clients only talk to the same servers, instead of switching to new ones.
🟦 Total Available	Available on every non-faulty node, even when the network is completely down.



Core Challenges Distributed Systems

01 **Complexity**
Multiple components requiring seamless integration and coordination

02 **Consistency**
Difficult to maintain across different nodes, especially in stateful apps.

03 **Fault Tolerance**
Crucial to design for failure recovery with minimal downtime.

04 **Scalability**
Must scale efficiently under varying loads without performance loss.

05 **Concurrency**
Managing simultaneous operations and conflict resolution in components.

06 **Security**
Ensuring confidentiality, integrity, and availability in a distributed network.

07 **Network Issues**
Susceptible to latency and partitioning in network communication.

08 **Deployment and Management:**
Resource-intensive, often needs specialized infrastructure and tools.

09 **Debugging and Monitoring**
More challenging due to distributed systems' dispersed nature.

10 **Decoupling**
Improves flexibility but complicates coherence and system coordination.

That's it, thank you!

Lambda

- Lambda functions
- Intro to serverless computing

Execution continuum

↔

EC2, EKS, ECS, Fargate	Elastic Beanstalk	API Gateway + Lambda	Lambda
• *Run any software you want for as long as you want* • *Complete control over HW and SW* • *Hardest to config*	• *Upload .zip file* • *Limited software choices* • *Some control over HW and SW*	• *Function based* • *Near-zero config* • *Web service + functions (15-min limit)*	• *Function based* • *Near-zero config* • *Short execution (< 15 mins)*

AWS lambda

- **By far the simplest, least expensive way to compute**

EC2, EKS, ECS, Fargate

Elastic Beanstalk

API Gateway + Lambda

Lambda

function

Scalable! Each call executes independently on different HW => horizontal scaling

AWS lambda / Azure functions / Google functions

- **Standalone functions executed on demand**
  - *Can be written in JavaScript, Python, Java, C++, etc.*
  - *Execution time is limited (AWS => 15 minutes)*

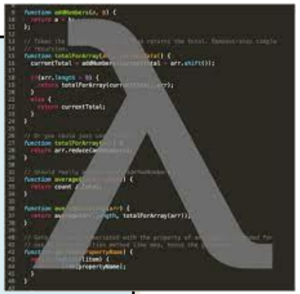
- **Callable in a variety of ways:**
  - *Like a traditional function( ) using AWS library*
  - *Based on* ***events*** *that occur (e.g. uploading an item into S3)*
  - *Via* ***function URL*** *through AWS-managed web server*
  - *Via* ***API Gateway*** *offering a more customizable AWS-managed web server (e.g. test vs. production, more authentication options, …)*

Example: prime factors in Python

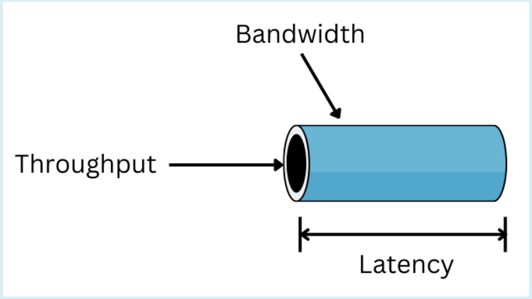
https://2noicxltxjwxxt4ego5d7q4uc40bcgjw.lambda-url.us-east-2.on.aws/?n=600851475143

https://2noicxltxjwxxt4ego5d7q4uc40bcgjw.lambda-url.us-east-2.on.aws/?n=6008514751439999

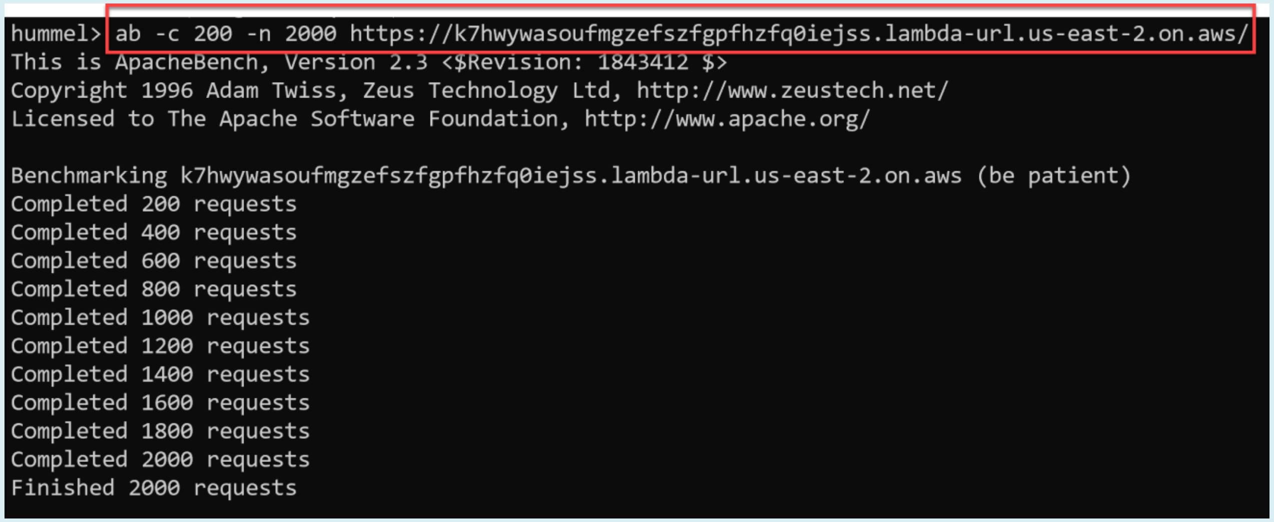
```python
import json

def prime_factors(n):
  i = 2

  factors = []
  while i * i <= n:
    if n % i:
      i += 1
    else:
      n //= i
      factors.append(i)

  if n > 1:
    factors.append(n)

  return {
    'statusCode': 200,
    'body': json.dumps(factors)
  }
```



Latency vs. Throughput



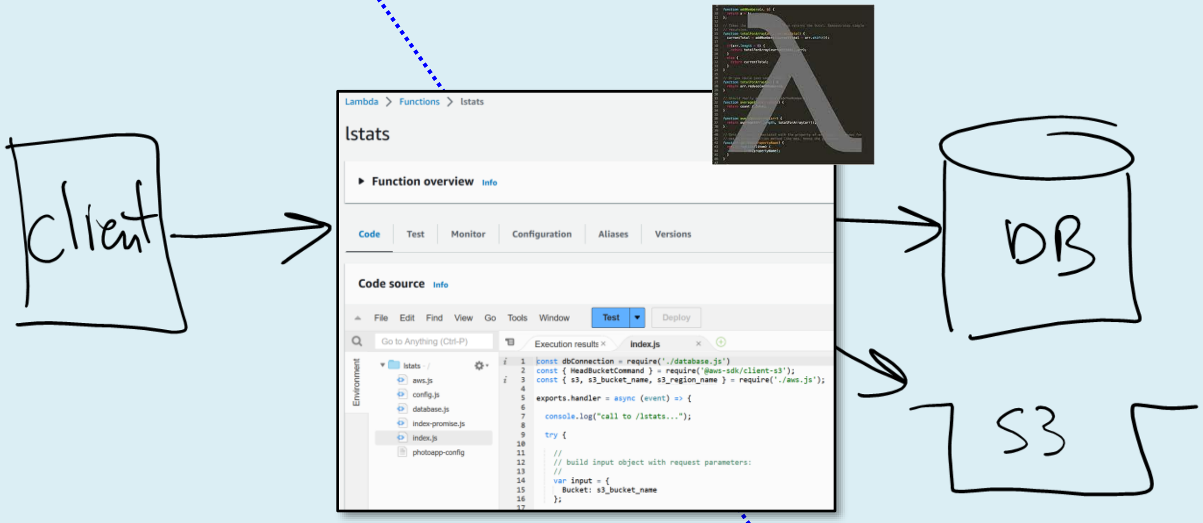
- Lambda trades latency for throughput
- Lambda functions have a longer latency (i.e. slower)
  - *Takes time to load function + support libraries*
- Lambda offers higher throughput (supports more clients) by automatically scaling calls across EC2



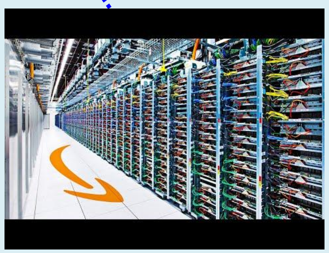
```
hummel> ab -c 200 -n 2000 https://k7hwywasoufmgzefszfgpfhzfq0iejss.lambda-url.us-east-2.on.aws/
This is ApacheBench, Version 2.3 <$Revision: 1843412 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking k7hwywasoufmgzefszfgpfhzfq0iejss.lambda-url.us-east-2.on.aws (be patient)
Completed 200 requests
Completed 400 requests
Completed 600 requests
Completed 800 requests
Completed 1000 requests
Completed 1200 requests
Completed 1400 requests
Completed 1600 requests
Completed 1800 requests
Completed 2000 requests
Finished 2000 requests
```

Demo #2: lambda-based stats

https://k7hwywasoufmgzefszfgpfhzfq0iejss.lambda-url.us-east-2.on.aws/



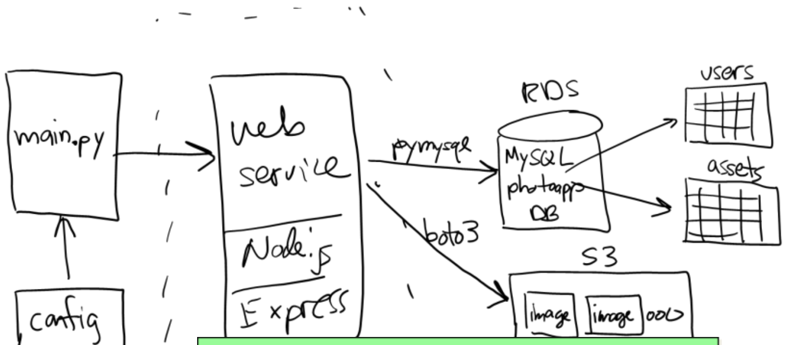
```
const dbConnection = require('./database.js')
const { HeadBucketCommand } = require('@aws-sdk/client-s3');
const { s3, s3_bucket_name, s3_region_name } = require('./aws.js');

exports.handler = async (event) => {

  console.log("call to /lstats...");

  try {

    //
    // build input object with request parameters:
    //
    var input = {
      Bucket: s3_bucket_name
    };
```



Let's compare our project 02 web service /stats vs. a lambda-based version of /stats

Serverless computing

- Architects realized the web server is often just a "gateway" to the functions

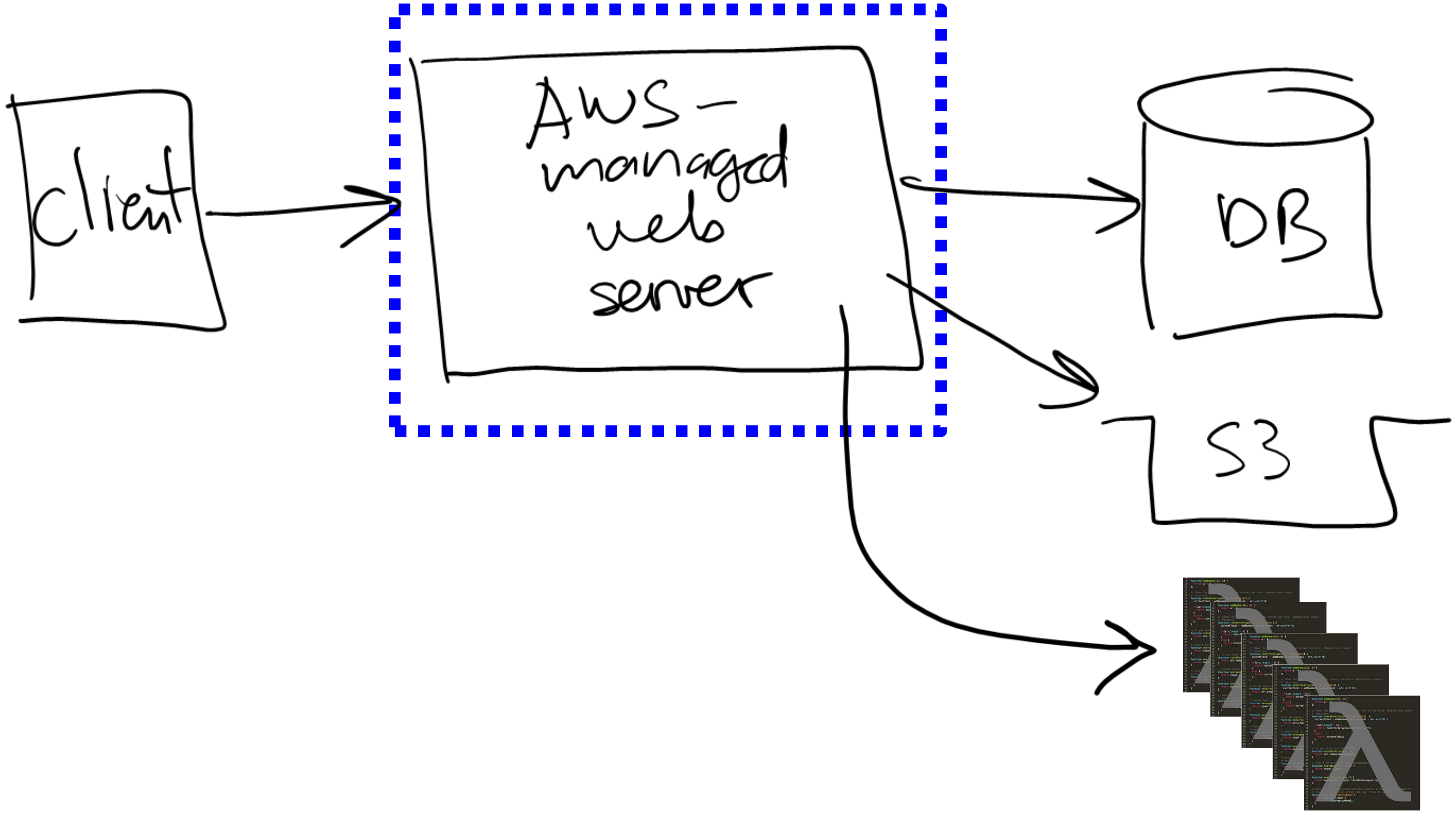


```
//
// PhotoApp web service
//
app.get('/stats', (req, res) => {…});
app.get('/users', (req, res) => {…});
app.get('/assets', (req, res) => {…});
app.get('/bucket', (req, res) => {…});
app.get('/download/:assetid', (req, res) => {…});
```

- Break the monolithic code base => functions or microservices…
- … and let AWS manage the web server!

Serverless doesn't mean no server

- We still have a web server…
- We just don't manage it

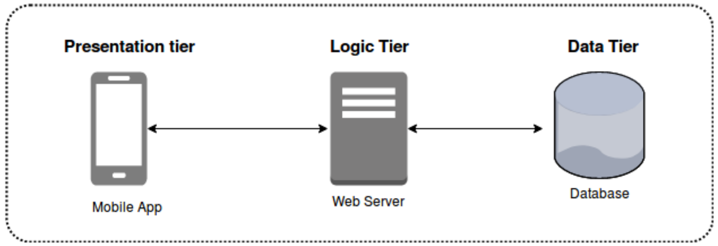


That's it, thank you!

Serverless

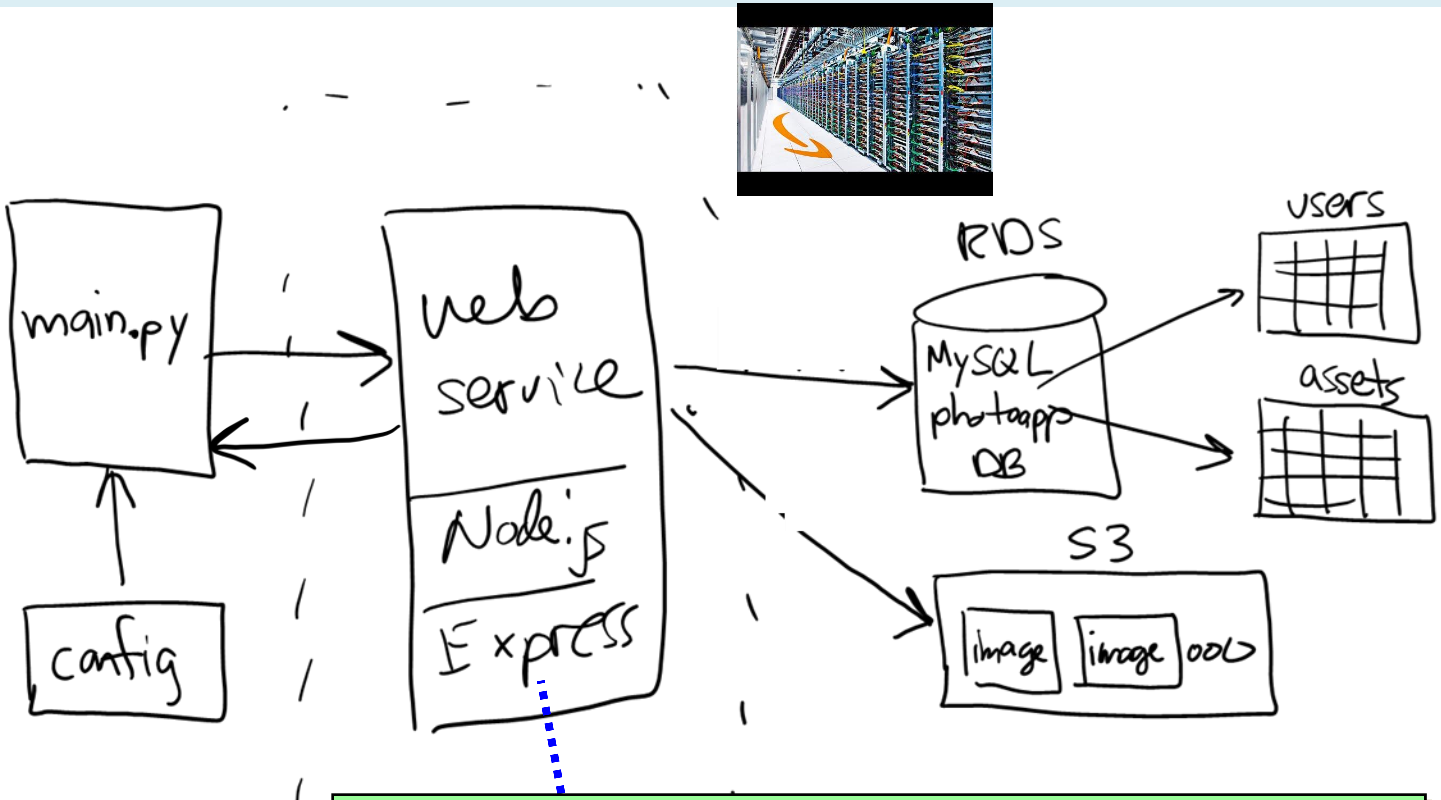
- Serverless computing
- API Gateway + lambda

Monolithic multi-tier



- **Traditional software design for the cloud**
- **Monolithic approach --- one large code base on server**
  - *Safe, conservative engineering*
  - *No one gets fired for building systems this way :-)*

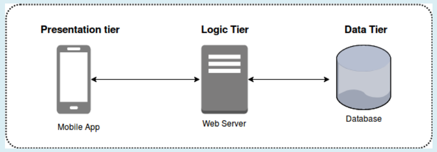
Project 02 --- monolithic web service



```
//
// PhotoApp web service
//
app.get('/stats', (req, res) => {…});
app.put('/user', (req, res) => {…});
app.get('/users', (req, res) => {…});
app.get('/assets', (req, res) => {…});
app.get('/bucket', (req, res) => {…});
app.get('/image/:assetid', (req, res) => {…});
app.post('/image/:userid', (req, res) => {…});
```

JavaScript

Alternative designs?



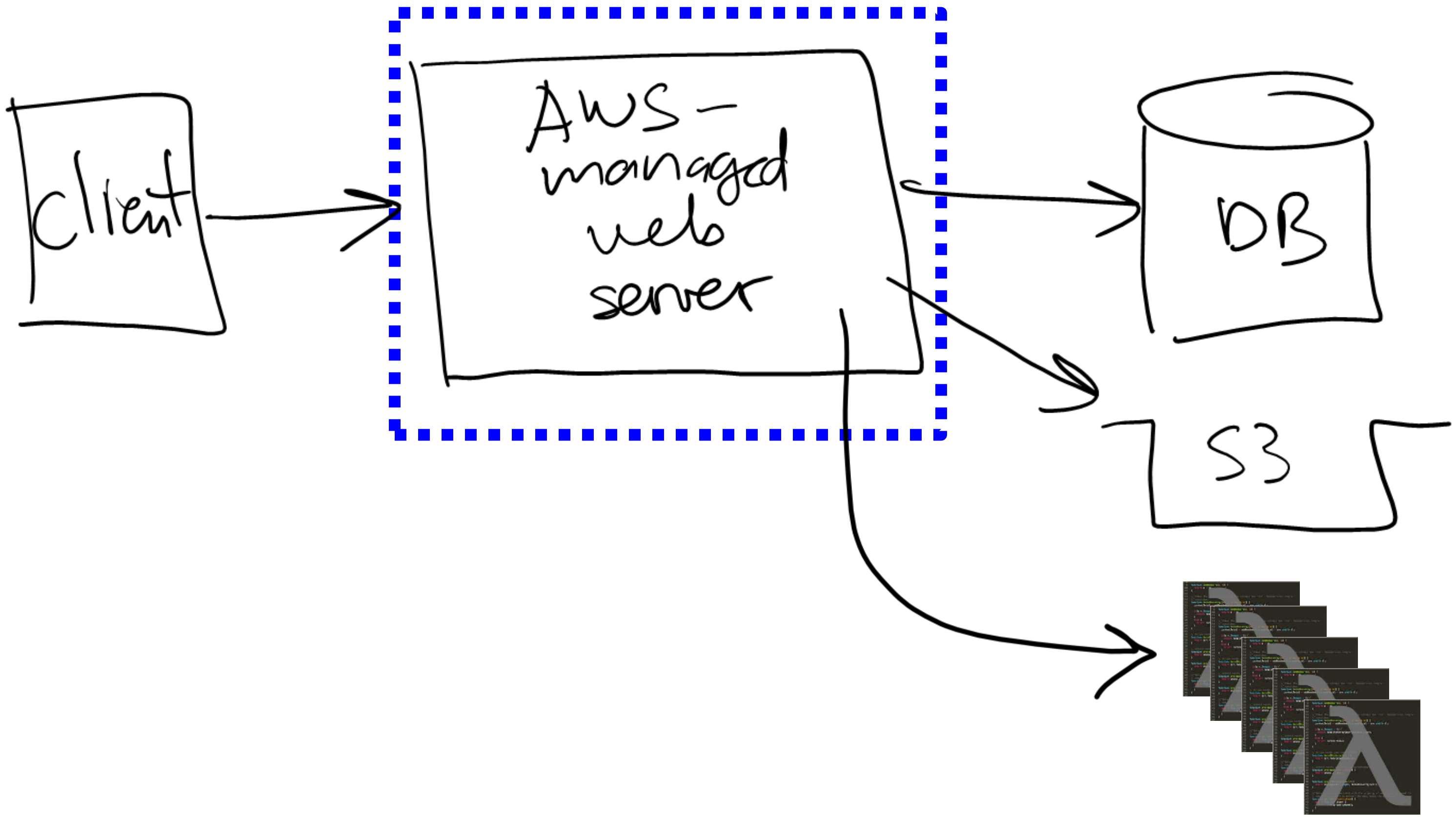
1. **Microservices**
   – *Break monolithic system apart --- easier to develop, update, release, but more moving parts to manage*
   – *Example: **Netflix** was one of the first to do this*

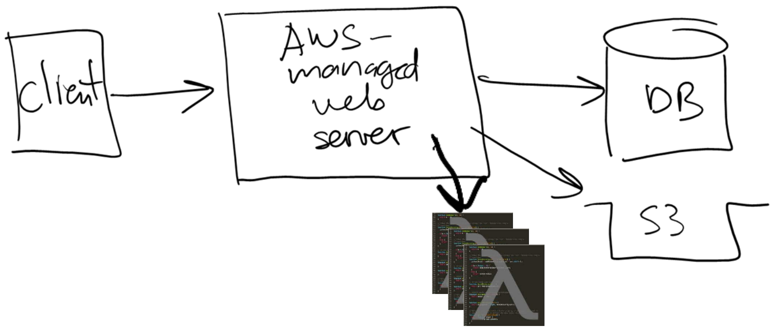
2. **Event-driven**
   – *Design based on events that occur / application states*
   – *Example: food delivery => menu, order, purchase, prepare, deliver*

3. **Serverless computing…**

Serverless doesn't mean no server

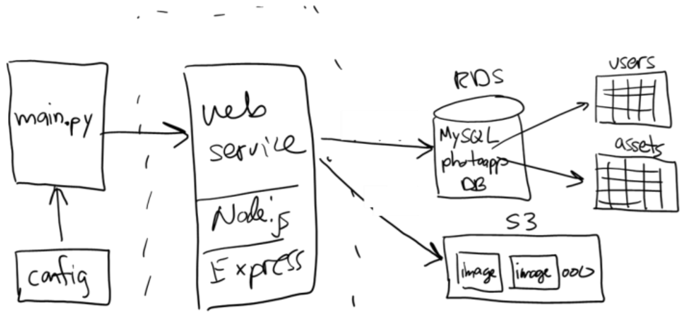
- We still have a web server…
- We just don't manage it

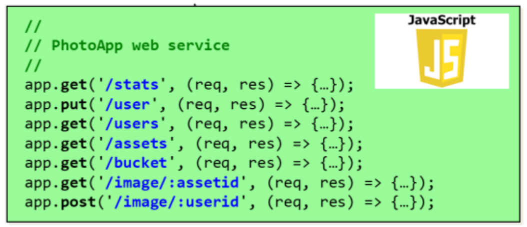




Why serverless?

- **Advantages:**
  1. Break monolithic code base into microservices / functions
     - *Enabling development in whatever language / platform makes the most sense --- JavaScript, Python, Java, …*
     - *Quicker to update and release functions / add new functionality*
  2. Retain advantages of web server tier but let AWS manage
  3. Scalability of server & functions w/o idle capacity (saving $)



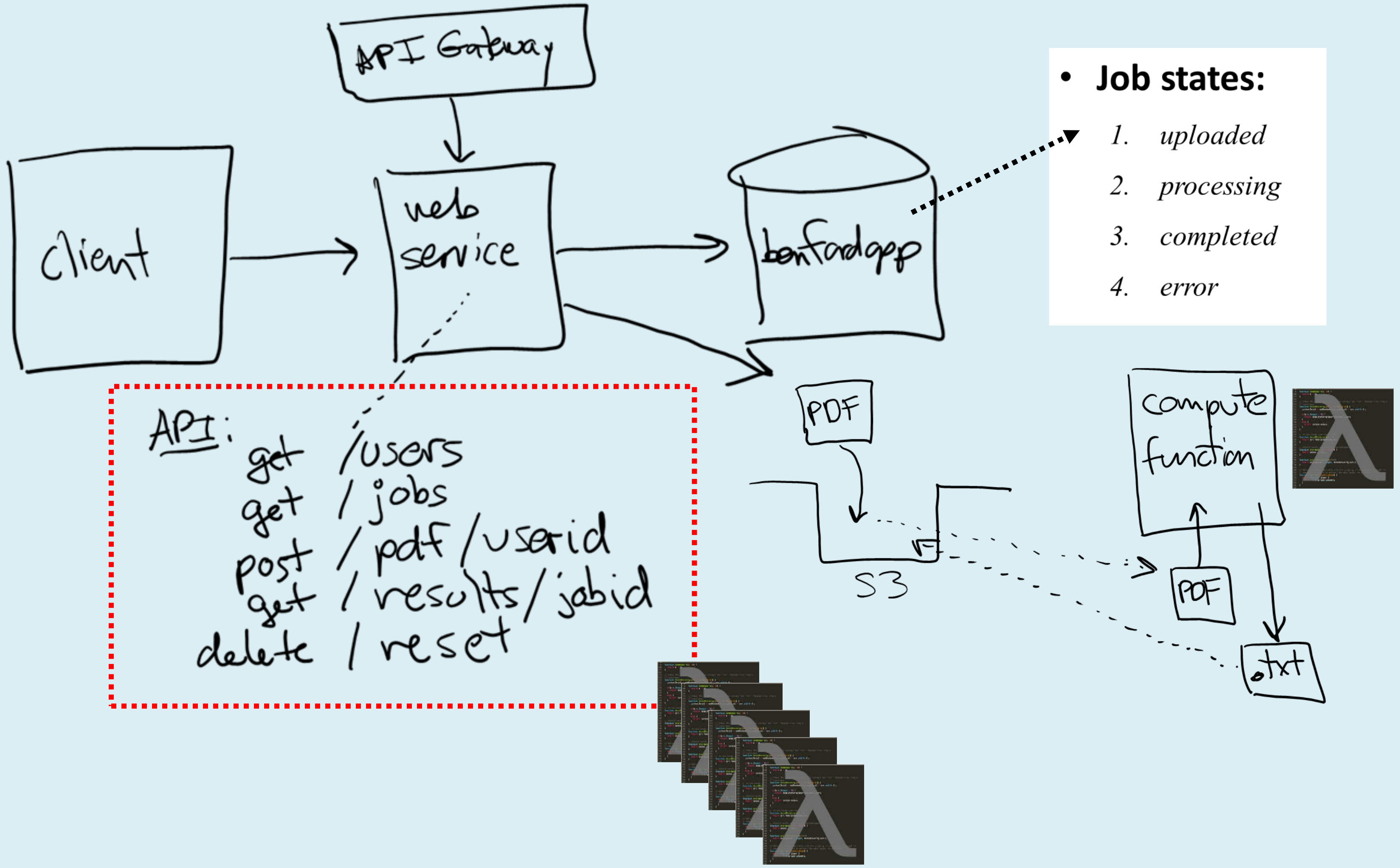


```
//
// PhotoApp web service
//
app.get('/stats', (req, res) => {…});
app.put('/user', (req, res) => {…});
app.get('/users', (req, res) => {…});
app.get('/assets', (req, res) => {…});
app.get('/bucket', (req, res) => {…});
app.get('/image/:assetid', (req, res) => {…});
app.post('/image/:userid', (req, res) => {…});
```

longer latency ☹

Example: Project 03

- *Serverless and event-driven…*



- **Job states:**
  1. *uploaded*
  2. *processing*
  3. *completed*
  4. *error*

AWS lambda

- **By far the simplest, least expensive way to compute**

EC2, EKS, ECS, Fargate

Elastic Beanstalk

API Gateway + Lambda

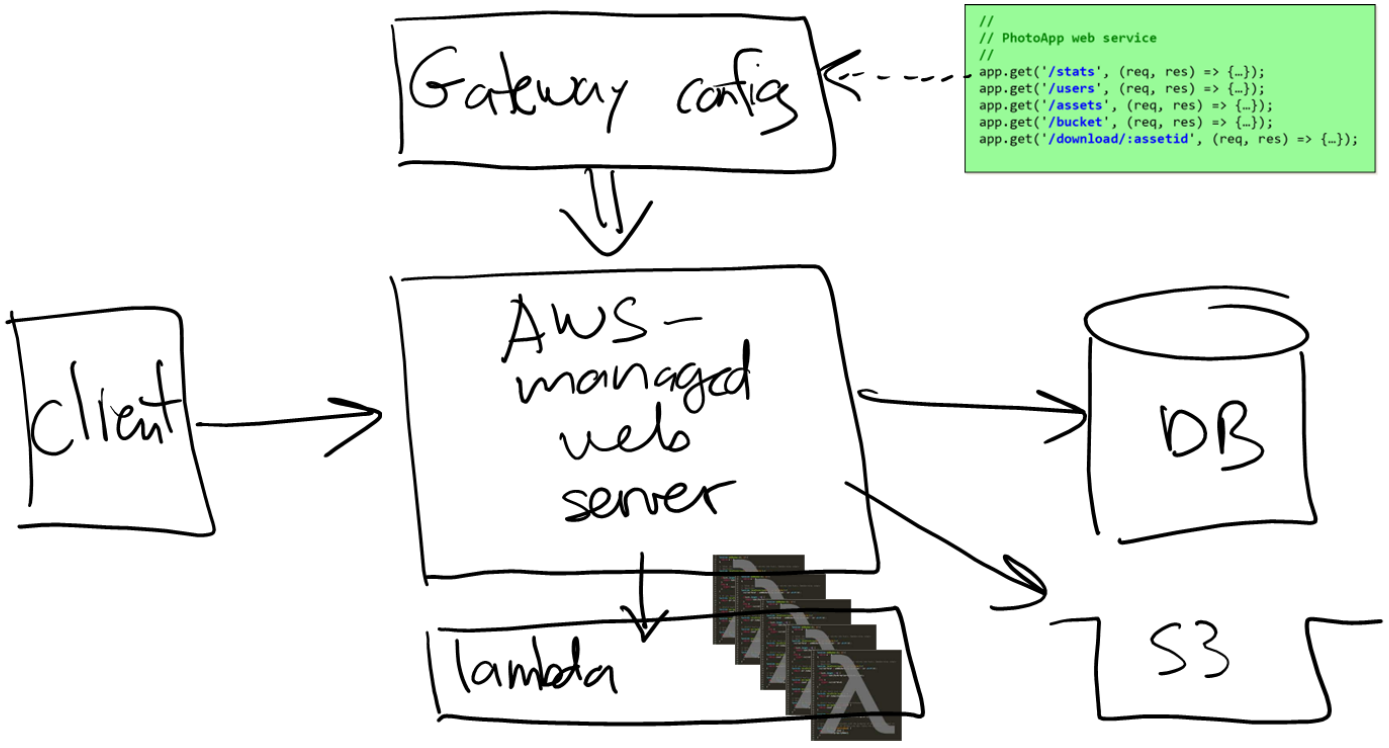
Lambda

function

Serverless computing with API Gateway

- **Just another step in the evolution of making AWS easier:**

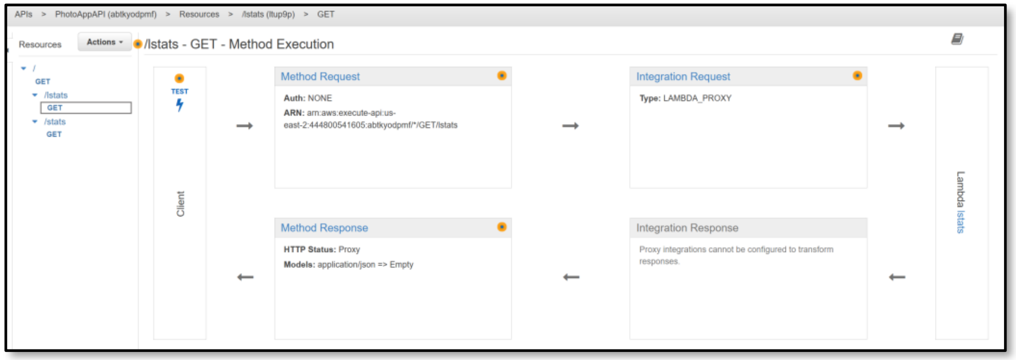
EC2, EKS, ECS, Fargate   Elastic Beanstalk   **API Gateway + Lambda**



```
//
// PhotoApp web service
//
app.get('/stats', (req, res) => {…});
app.get('/users', (req, res) => {…});
app.get('/assets', (req, res) => {…});
app.get('/bucket', (req, res) => {…});
app.get('/download/:assetid', (req, res) => {…});
```

API Gateway

- **API Gateway allows you to define a RESTful API that forwards to other services / lambdas**
  - *Define HTTP verb and URL path (e.g. GET /movies)*
  - *Specify target…*



APIs > PhotoAppAPI (abtkyodpmf) > Resources > /lstats (ltup9p) > GET

Resources Actions

/lstats - GET - Method Execution

/
GET
/lstats
GET
/stats
GET

TEST

Client

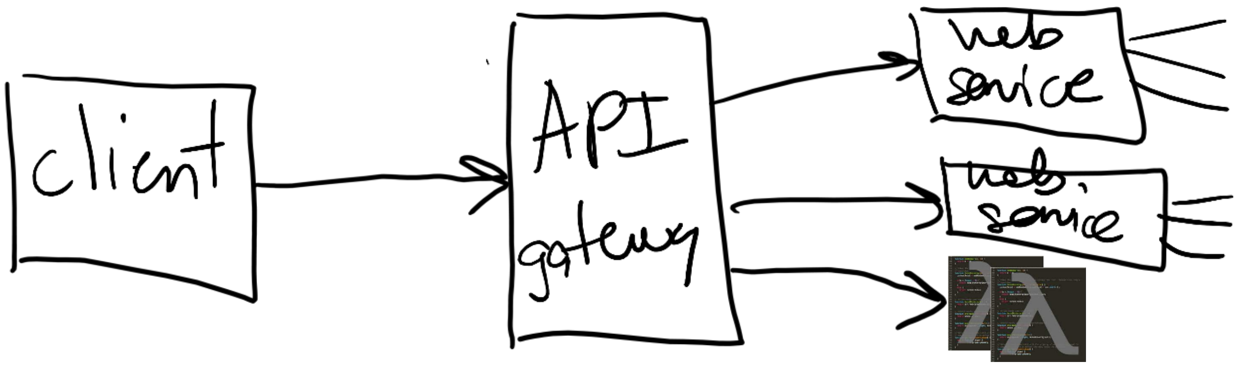
Method Request
Auth: NONE
ARN: arn:aws:execute-api:us-east-2:444800541605:abtkyodpmf/*/GET/lstats

Integration Request
Type: LAMBDA_PROXY

Method Response
HTTP Status: Proxy
Models: application/json => Empty

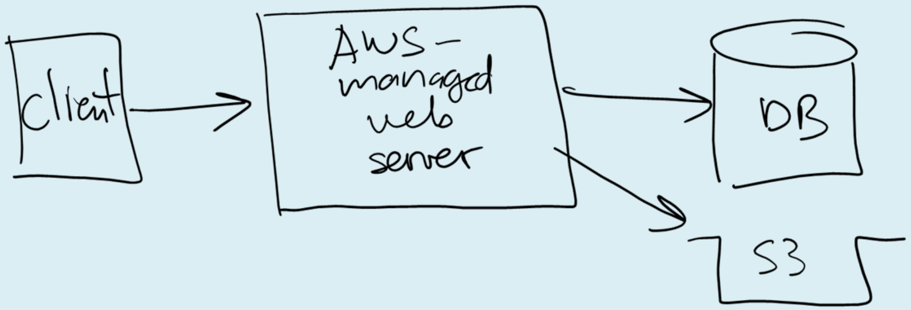
Integration Response
Proxy integrations cannot be configured to transform responses.

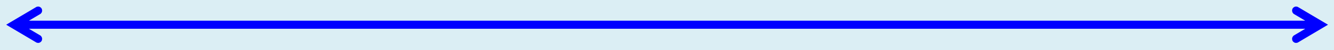
Lambda lstats



Need faster response (lower latency)?

- ***Replace lambda with faster technology (more $)…***





EC2, EKS, ECS, Fargate **Elastic Beanstalk** … Lambda

That's it, thank you!