

Workloads in the cloud

- **Types of workloads**
- **Packaging options**
- **Execution options**



New architectural concepts...

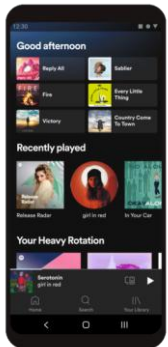
- I want to motivate two design concepts...

1. Lambda functions

2. Serverless computing

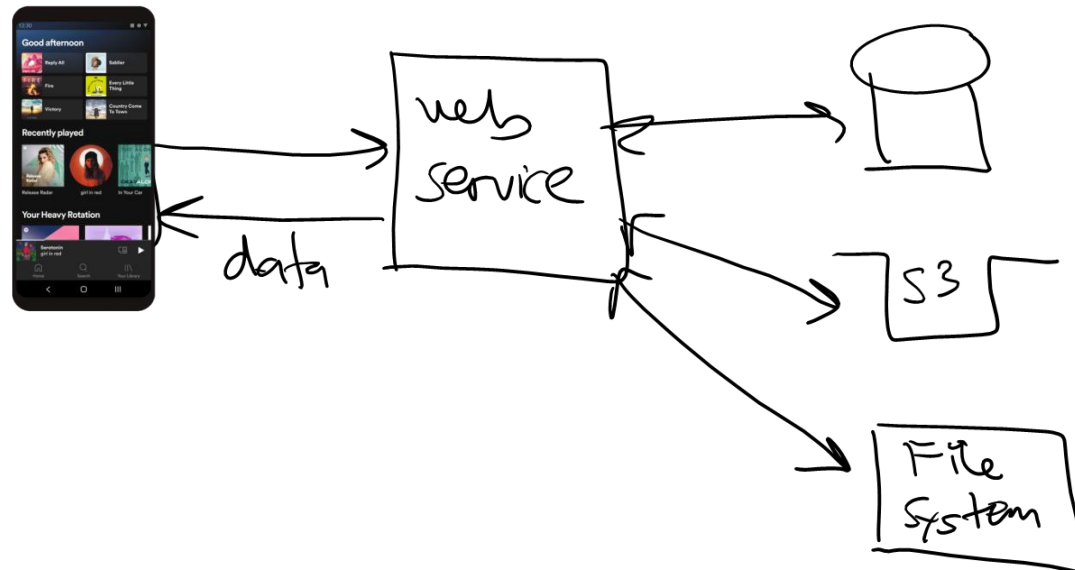
Multi-tier, data-driven apps

- Our examples (so far) have all been data-driven



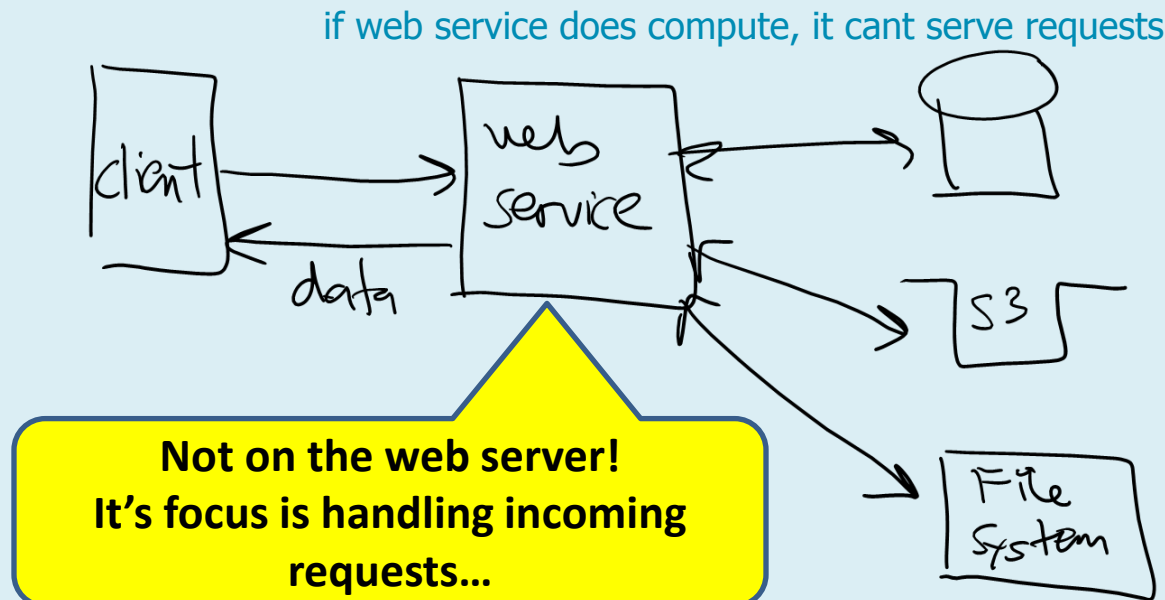
I/O bound

- We call this kind of workload "I/O bound"
 - *Server is spending most of its time waiting for requests / data, i.e. input/output*
 - *This is typically handled via async programming*



Compute-bound

- What if we need to compute something?
 - *Image/video compression, encryption, content analysis*
 - *Stock market simulation*
 - *Run AI / ML training set*
- We call these "compute-bound" workloads due to heavy CPU usage... Where do we execute?



Example: prime factors in Python

<https://2noicxltxjwxxt4ego5d7q4uc40bcgjl.lambda-url.us-east-2.on.aws/?n=600851475143>

<https://2noicxltxjwxxt4ego5d7q4uc40bcgjl.lambda-url.us-east-2.on.aws/?n=6008514751439999>

```
import json

def prime_factors(n):
    i = 2

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

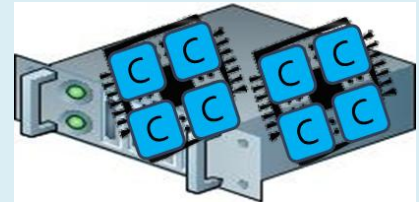
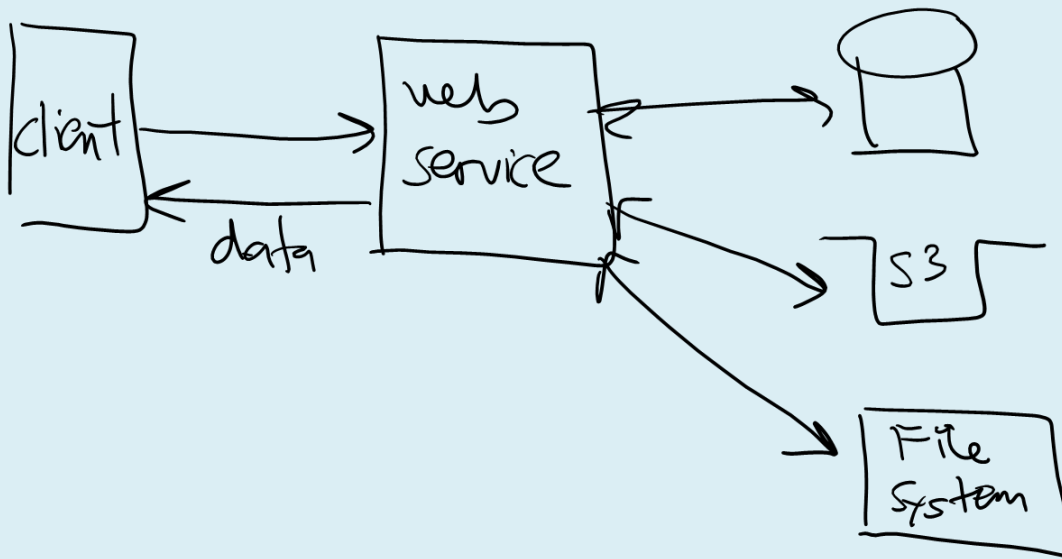
    if n > 1:
        factors.append(n)

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



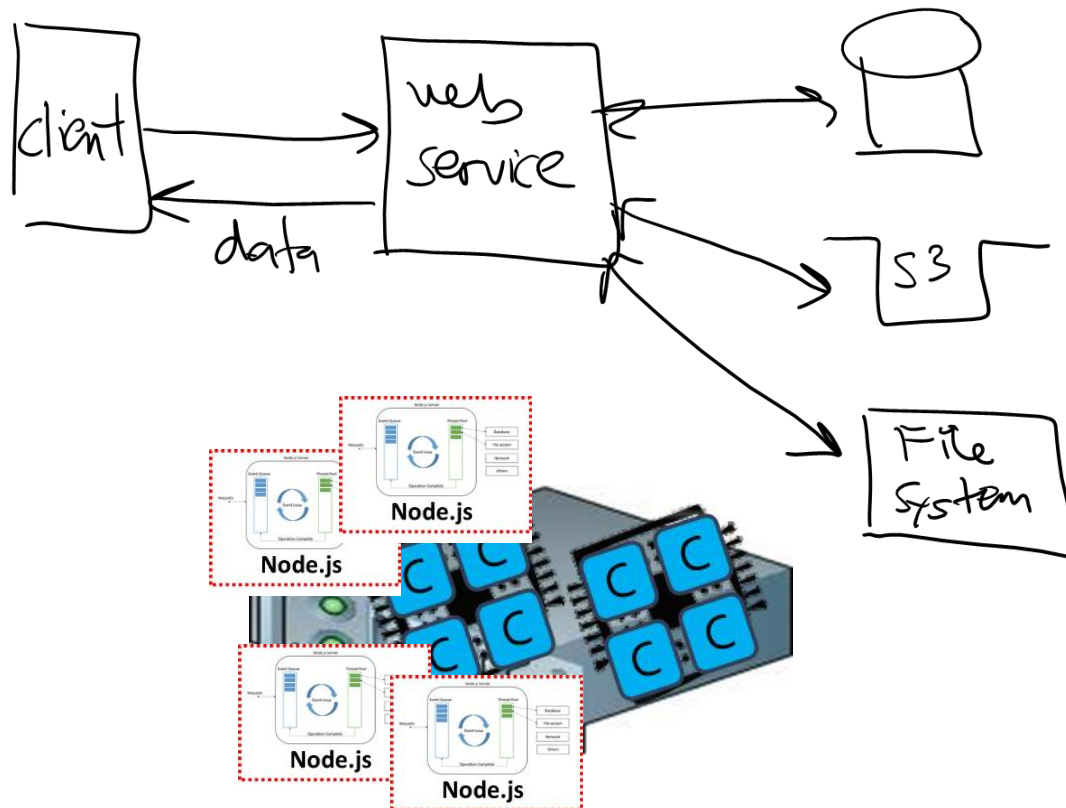
Compute tier

- We need a separate tier for executing compute-bound work
 - *This can be a separate core, CPU, or machine*



Option #1

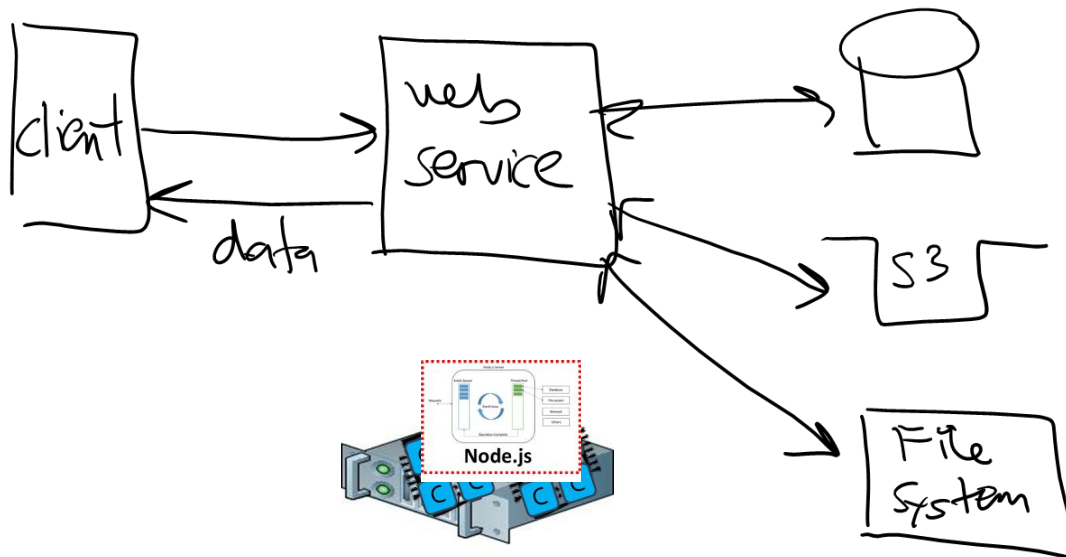
- ***IF*** you have unused cores available, use those
 - *Better for small-scale work, i.e. small tasks that only run for a few seconds / minutes*



Option #2

- What if my task takes longer to run, or needs lots of RAM?
- Run on separate hardware...

The problem? Installing the software you want to run...



Elastic Compute Cloud (EC2)

Everything runs on an EC2 instance...

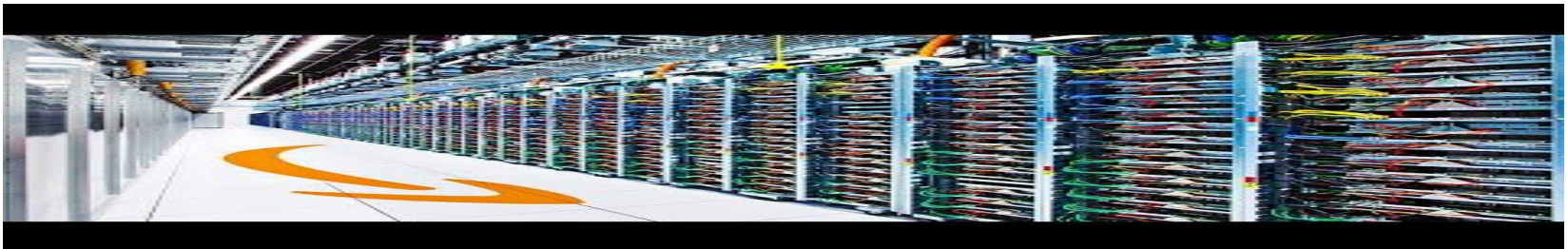
EC2 is AWS machine rental service, started in 2006

- Outsourcing hardware is an old idea. Amazon's innovation was to charge by the **hour**, not month, and this started cloud revolution



Lambda

function

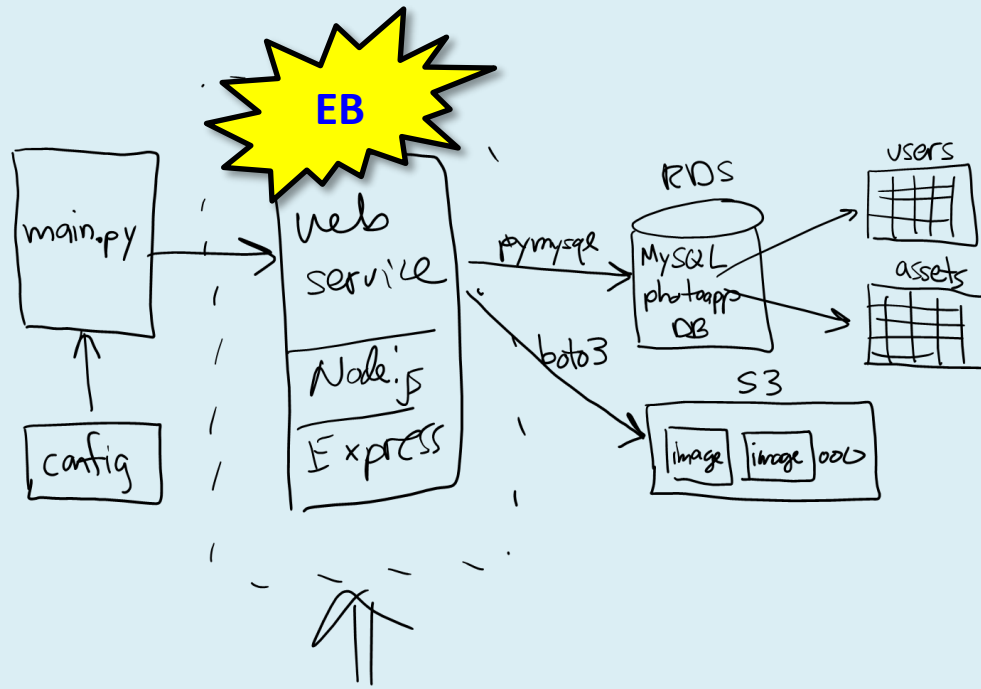


Example

Elastic Beanstalk

Program
(e.g. web service)

- In project 02, EB allowed us to have a web service up and running with .zip and a few button clicks...



Execution continuum



- Trade-offs:



EC2, EKS, ECS, Fargate

- *Run any software you want for as long as you want*
- *Complete control over HW and SW*
- *Hardest to config*

Elastic Beanstalk

- *Server-based*
- *Upload .zip file*
- *Limited software choices*
- *Some control over HW and SW*

API Gateway + Lambda

- *Function based*
- *Near-zero config*
- *Multi-tier web service + functions (15-min limit)*

Lambda

- *Function based*
- *Near-zero config*
- *Short execution (< 15 mins)*

That's it, thank you!

Lambda

- **Lambda functions**
- **Intro to serverless computing**



Execution continuum



EC2, EKS, ECS, Fargate

- *Run any software you want for as long as you want*
- *Complete control over HW and SW*
- *Hardest to config*

Elastic Beanstalk

- *Upload .zip file*
- *Limited software choices*
- *Some control over HW and SW*

API Gateway + Lambda

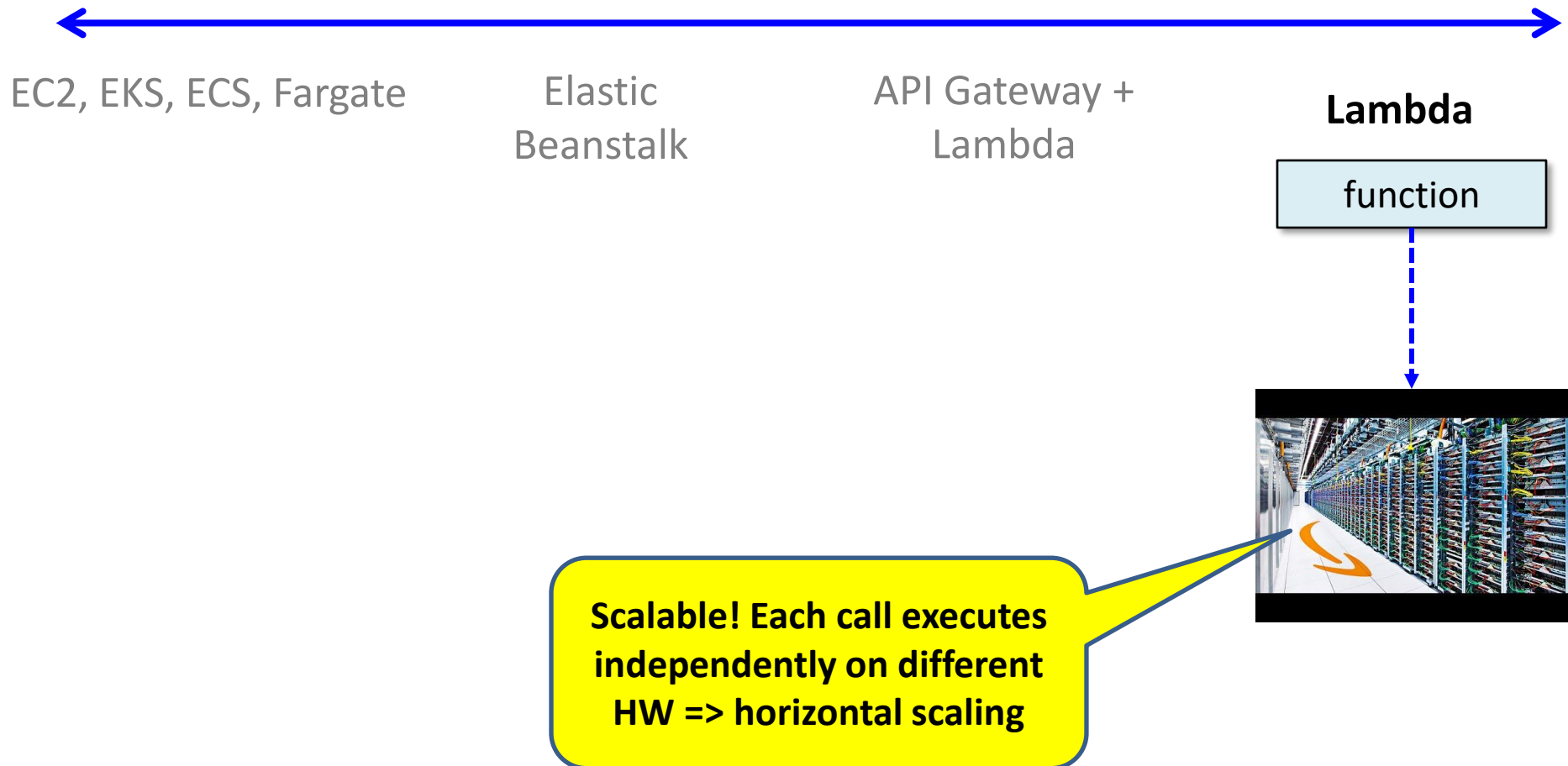
- *Function based*
- *Near-zero config*
- *Web service + functions (15-min limit)*

Lambda

- *Function based*
- *Near-zero config*
- *Short execution (< 15 mins)*

AWS lambda

- By far the simplest, least expensive way to compute



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>

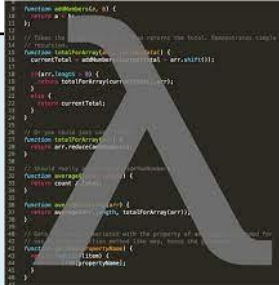
```
import json

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)

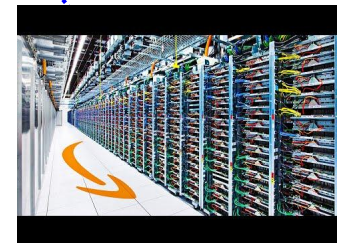
    return {
        'statusCode': 200,
        'body': json.dumps(factors)
    }
```



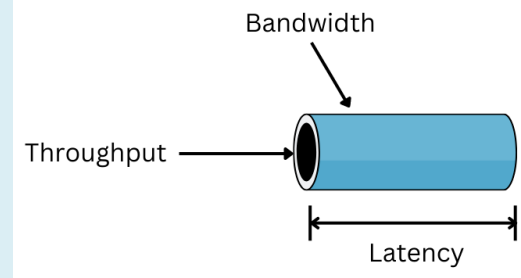
```
function handler(event, context) {
    // Parse the input
    const { n } = JSON.parse(event.body);

    // Calculate the prime factors
    const factors = prime_factors(n);

    // Return the result
    return {
        statusCode: 200,
        body: JSON.stringify(factors)
    };
}
```



Latency vs. Throughput



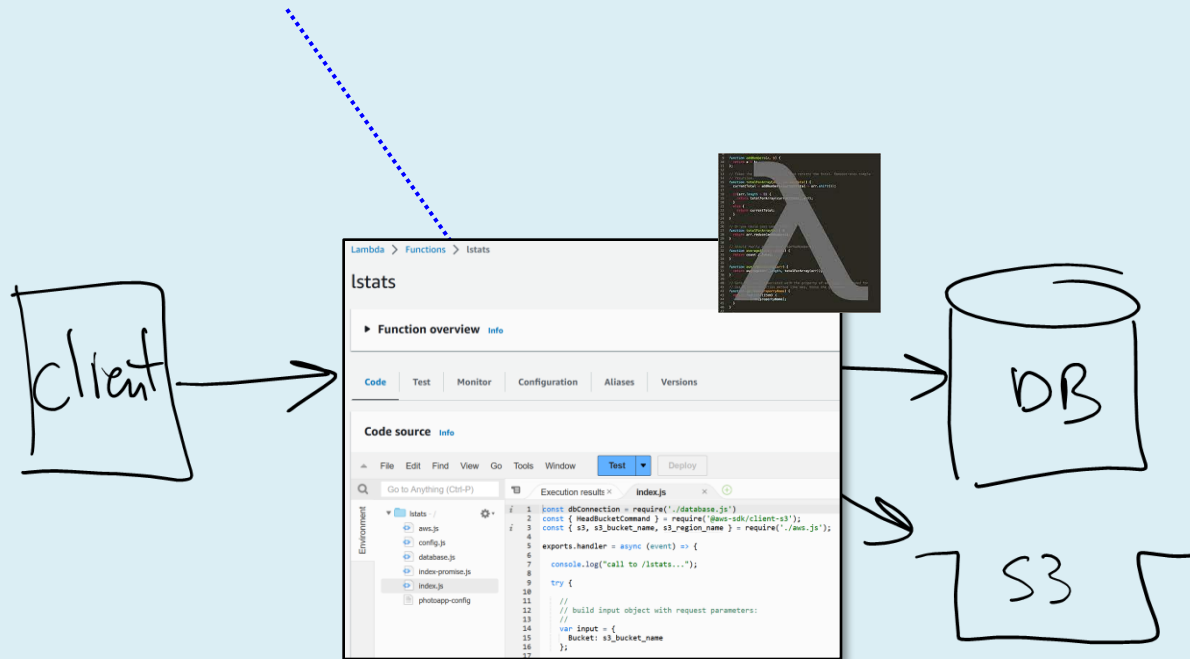
- **Lambda trades latency for throughput**
- **Lambda functions have a longer latency (i.e. slower)**
 - *Takes time to load function + support libraries*
- **Lambda offers higher throughput (supports more clients) by automatically scaling calls across EC2**

```
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/

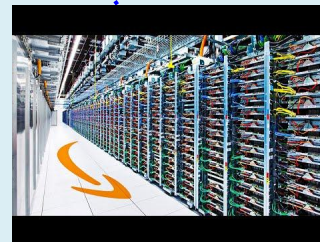
Benchmarking k7hwywasoufmgzefszfgpfhzfq0iejss.lambda-url.us-east-2.on.aws (be patient)
Completed 200 requests
Completed 400 requests
Completed 600 requests
Completed 800 requests
Completed 1000 requests
Completed 1200 requests
Completed 1400 requests
Completed 1600 requests
Completed 1800 requests
Completed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

Demo #2: lambda-based stats

<https://k7hwywasoufmgzefszfgpfhzhfq0iejss.lambda-url.us-east-2.on.aws/>

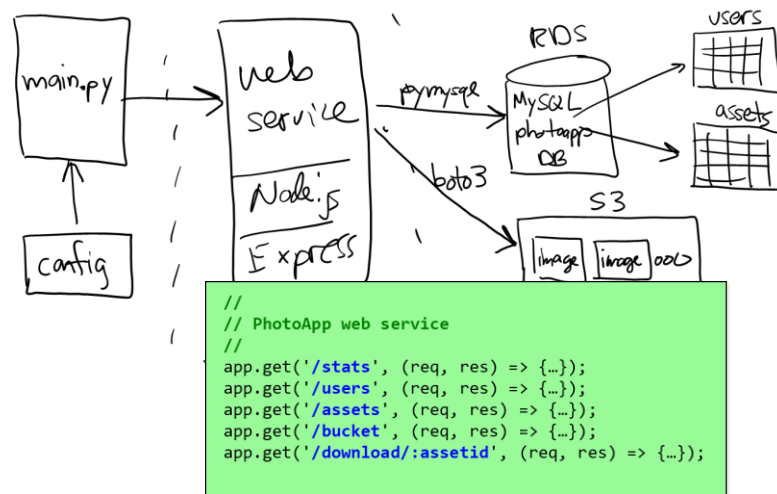


Let's compare our project 02 web service /stats vs. a lambda-based version of /stats



Serverless computing

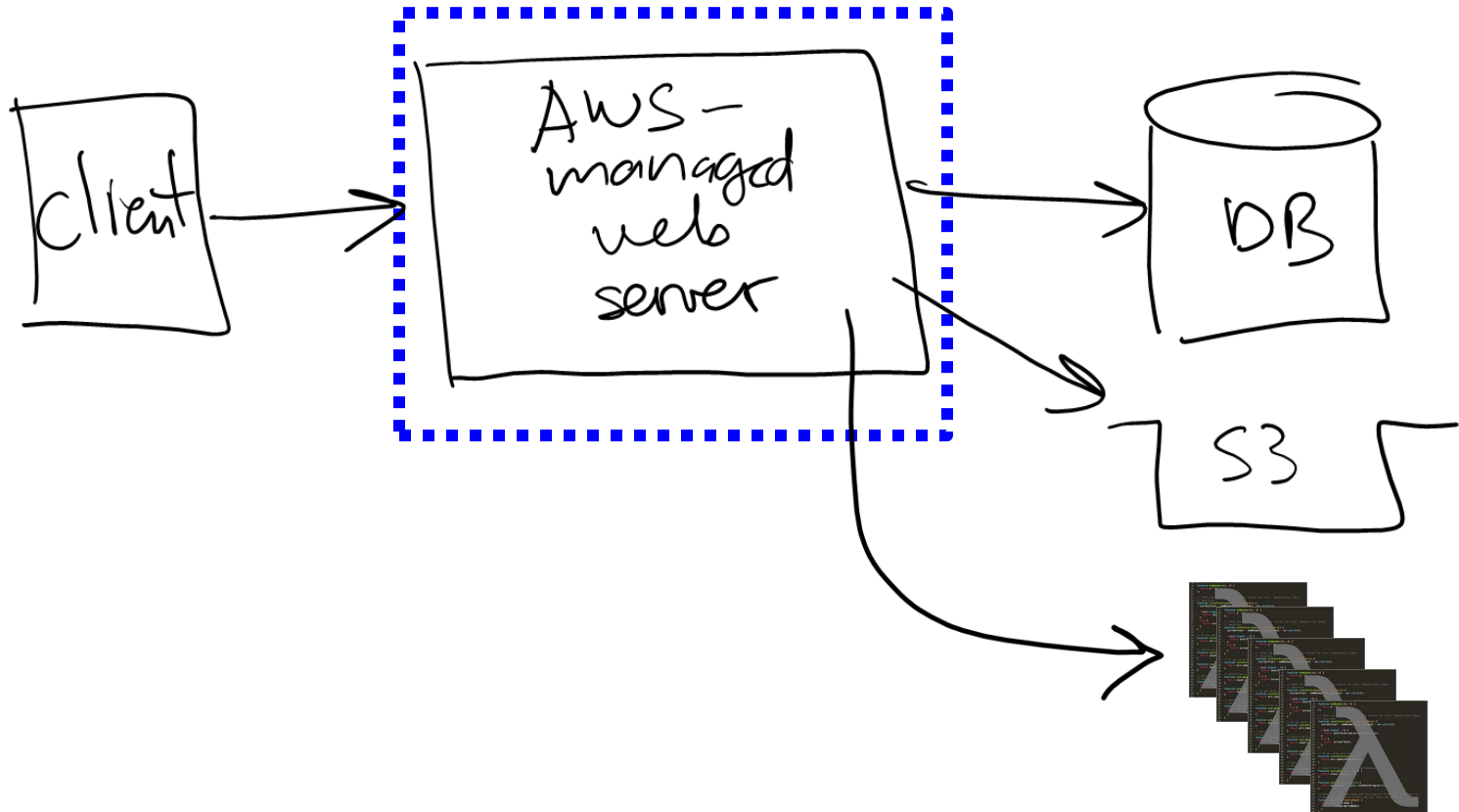
- Architects realized the web server is often just a "gateway" to the functions



- 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



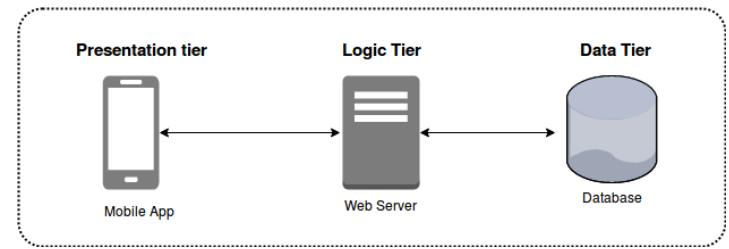
That's it, thank you!

Serverless

- **Serverless computing**
- **API Gateway + lambda**

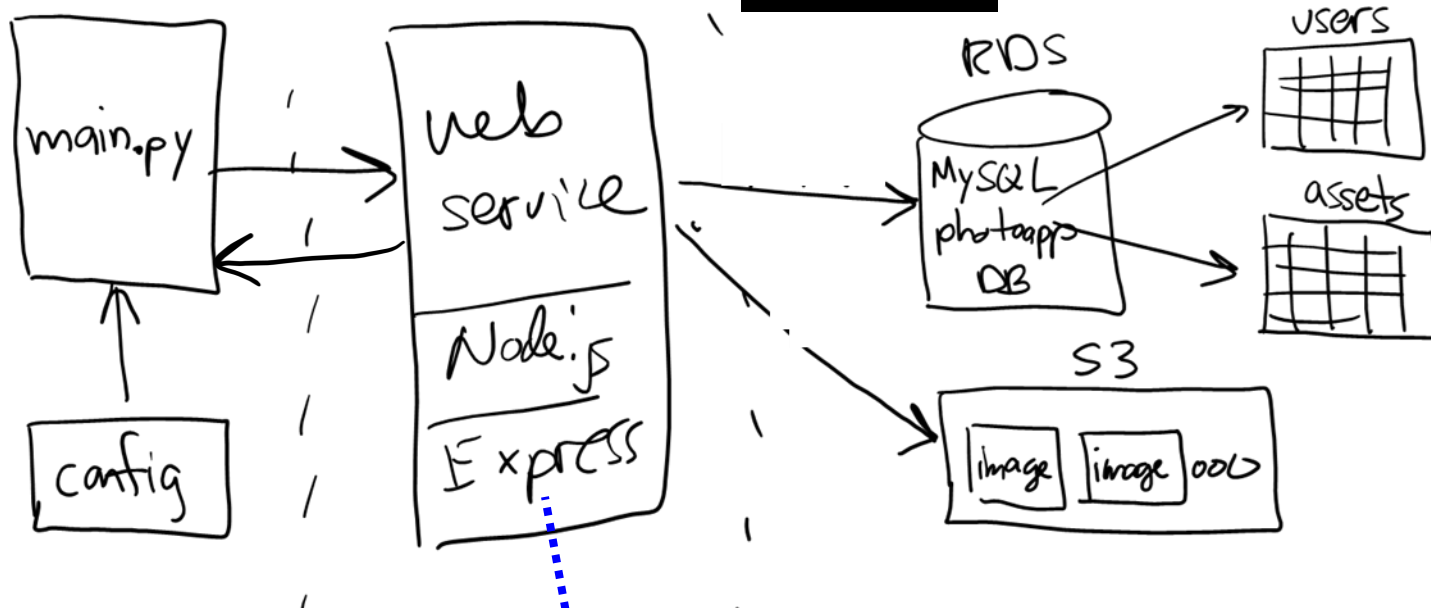
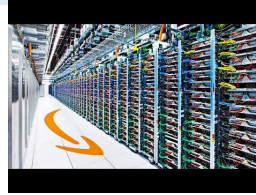


Monolithic multi-tier



- 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

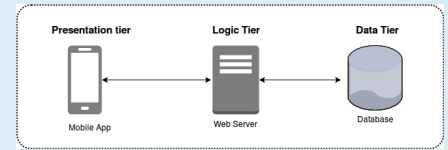


```
//  
// PhotoApp web service  
//  
app.get('/stats', (req, res) => {...});  
app.put('/user', (req, res) => {...});  
app.get('/users', (req, res) => {...});  
app.get('/assets', (req, res) => {...});  
app.get('/bucket', (req, res) => {...});  
app.get('/image/:assetid', (req, res) => {...}); 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

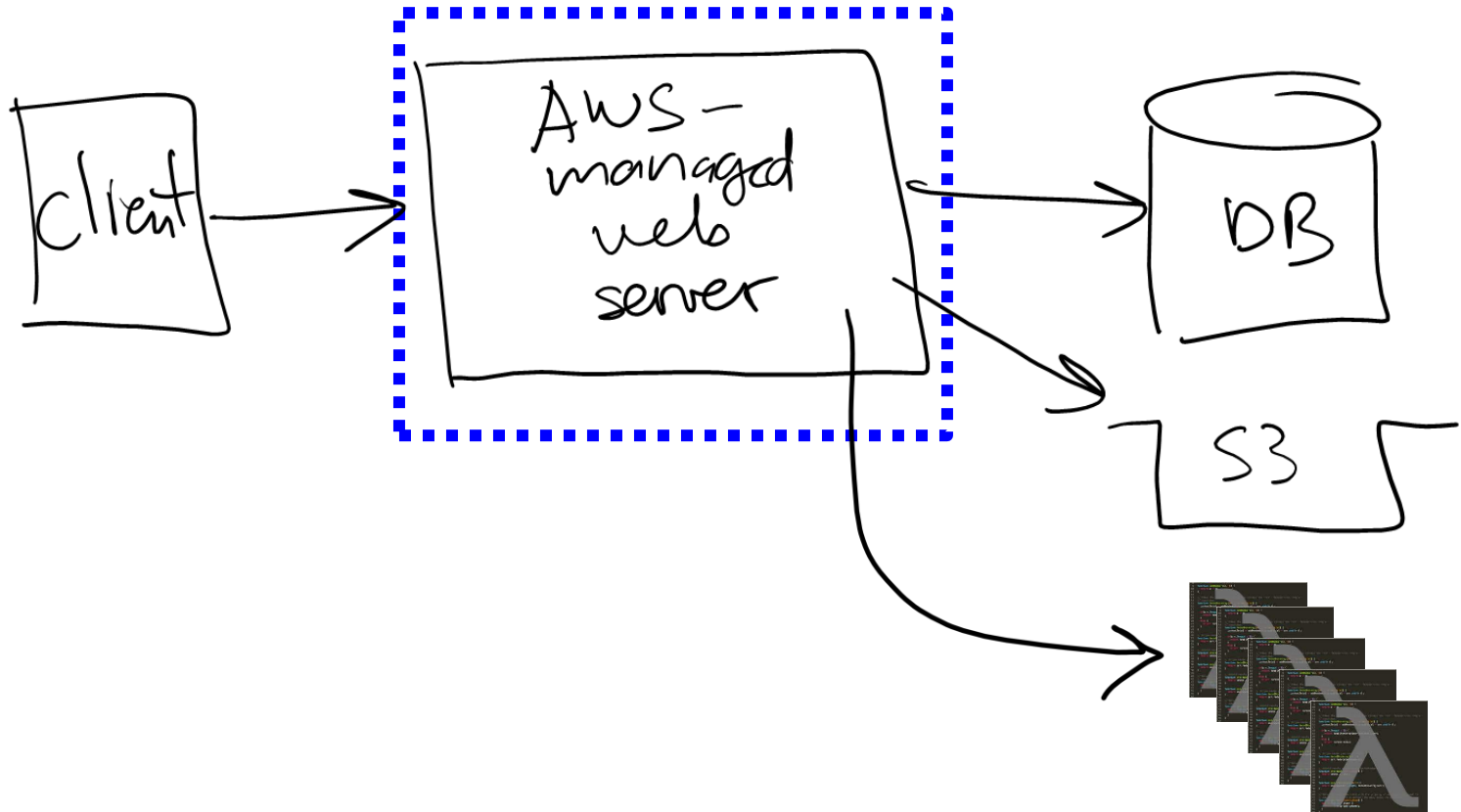
2. Event-driven

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

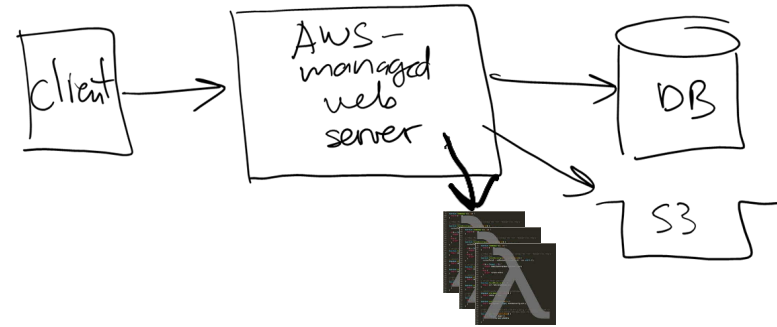
3. Serverless computing...

Serverless doesn't mean no server

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

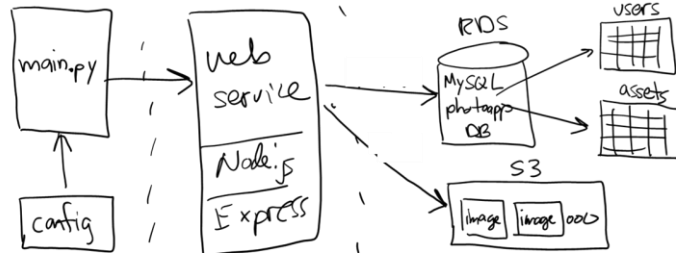


Why serverless?



• 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) => {...});
```

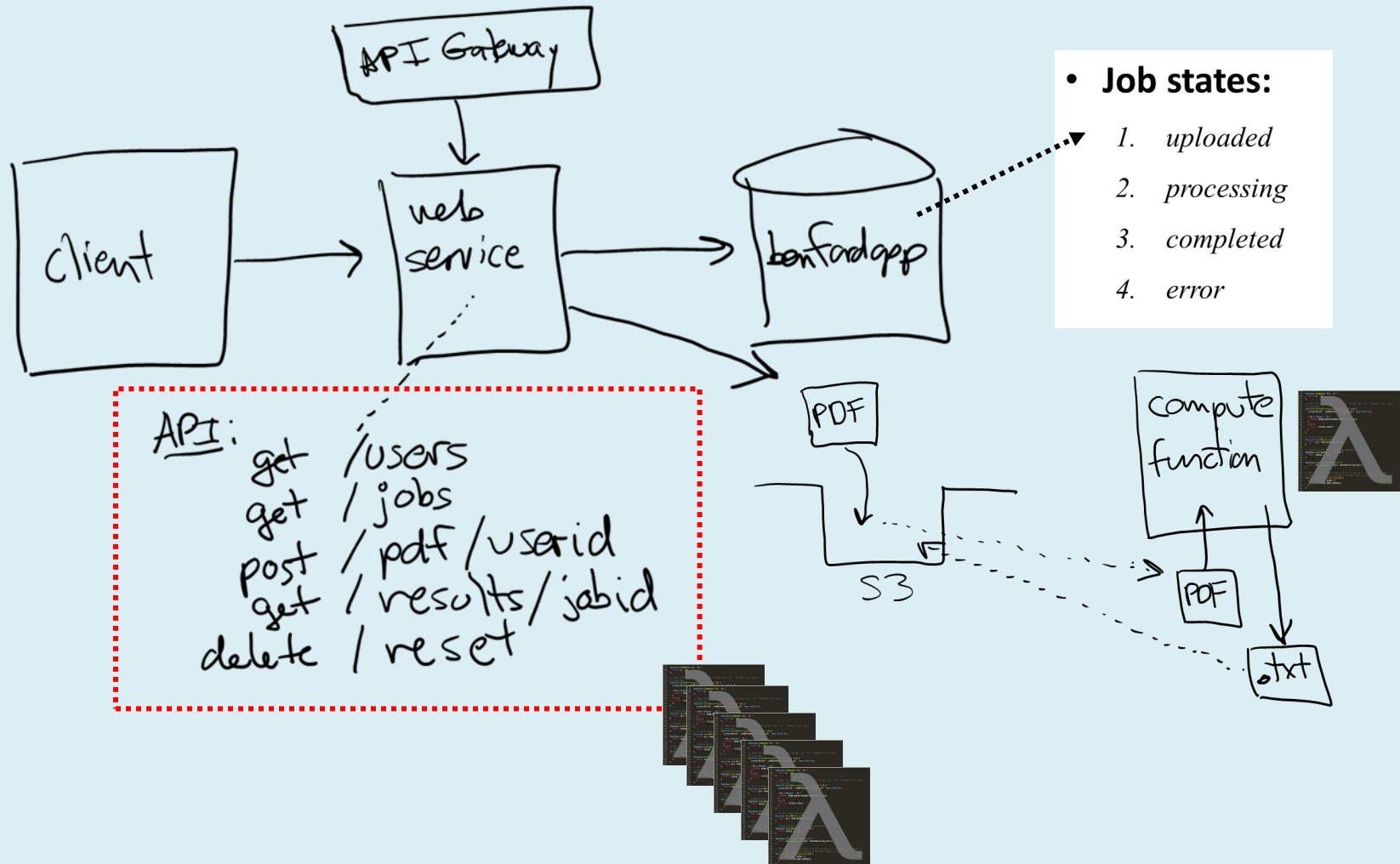
JavaScript



longer
latency ☹️

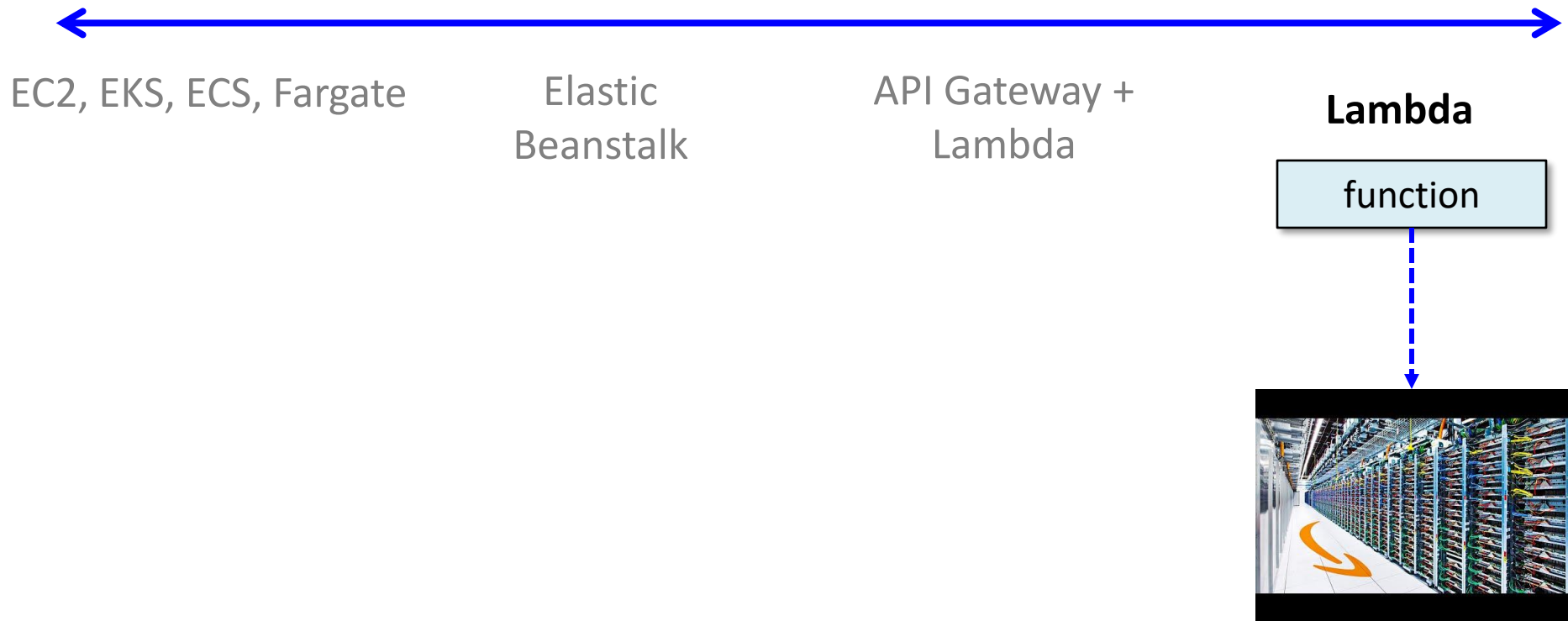
Example: Project 03

- *Serverless and event-driven...*



AWS lambda

- By far the simplest, least expensive way to compute



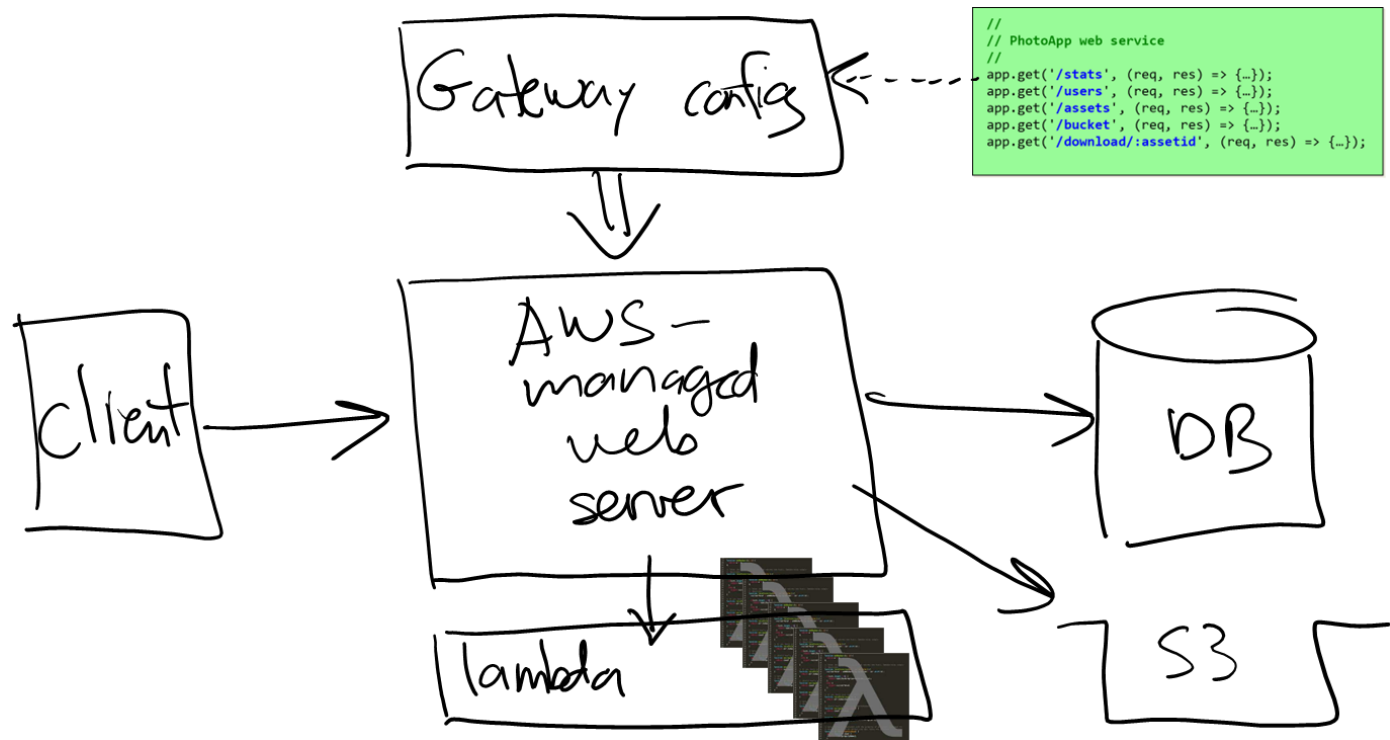
Serverless computing with API Gateway

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

EC2, EKS, ECS, Fargate

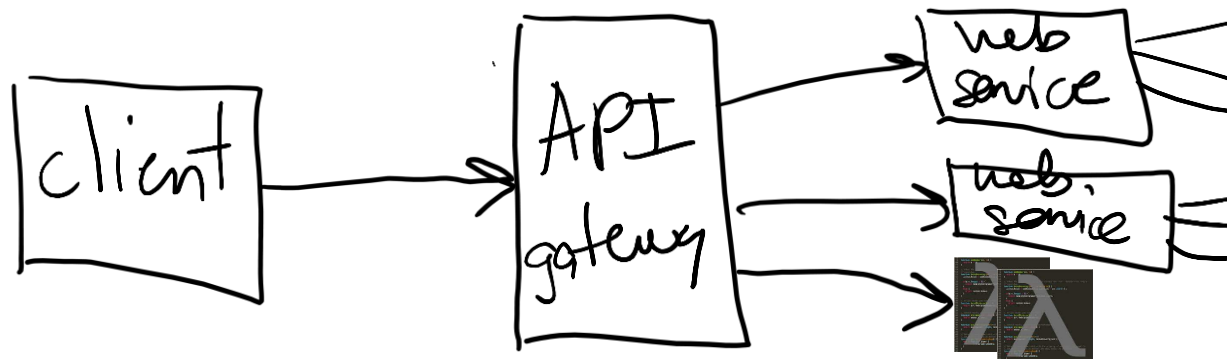
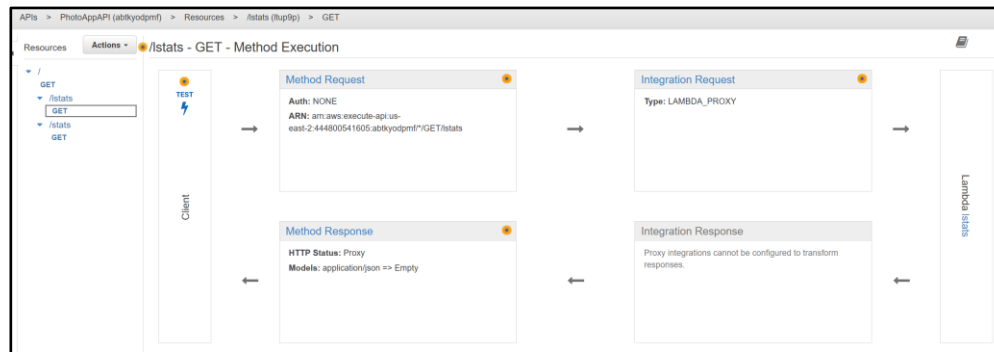
Elastic Beanstalk

**API Gateway +
Lambda**



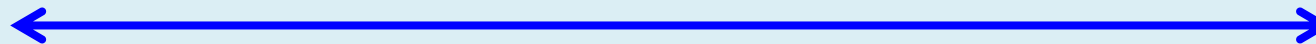
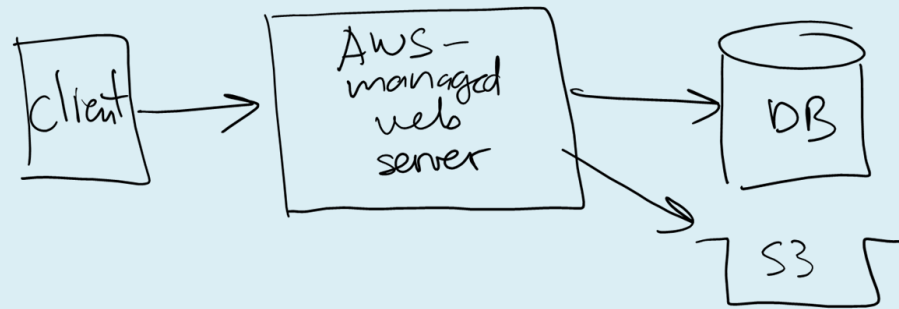
API Gateway

- **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...*



Need faster response (lower latency)?

- *Replace lambda with faster technology (more \$)...*



EC2, EKS, ECS, Fargate

Elastic
Beanstalk

...

Lambda

That's it, thank you!