# Serverless Computing

CS 40 | February 12, 2024

## Assignment 2 Due Tomorrow (2/13)

**GUEST LECTURE on Monday 2/26**
**by Maria Zhang, VP of Engineering, Google**
**MANDATORY ATTENDANCE (*email if conflict)**

***Guiding principle**: Make running web (or other) applications as independent as possible from the underlying infrastructure.*

*Containerization & Container Orchestration – Week 4*

***Guiding principle**: Make running web (or other) applications as independent as possible from the $\boxed{\text{underlying infrastructure.}}$*

Containers are still infrastructure!

*What if we could simply run code on the cloud without having to worry about infrastructure?*

## The Serverless Manifesto

1. Function are the unit of deployment and scaling.
2. No machines, VMs, or containers visible in the programming model.
3. Permanent storage lives elsewhere.
4. Scales per request; Users cannot over- or under-provision capacity.
5. Never pay for idle (no cold servers/containers or their costs).
6. Implicitly fault-tolerant because functions can run anywhere.
7. BYOC - Bring Your Own Code.
8. Metrics and logging are a universal right.

*David Potes & Ajay Nair – Building Complex Serverless Applications, 2016*

## Functions as a Service

- Deploy individual functions that scalably service requests to the cloud
    - Just write code in an application language (usually NodeJS); nothing else needed

- Runtime is only activated on an incoming request
    - Idle traffic means no compute capacity needs to be used
    - **No traffic → no compute cost!**

- Based on the idea of *lambda calculus*: a function on some inputs producing some outputs

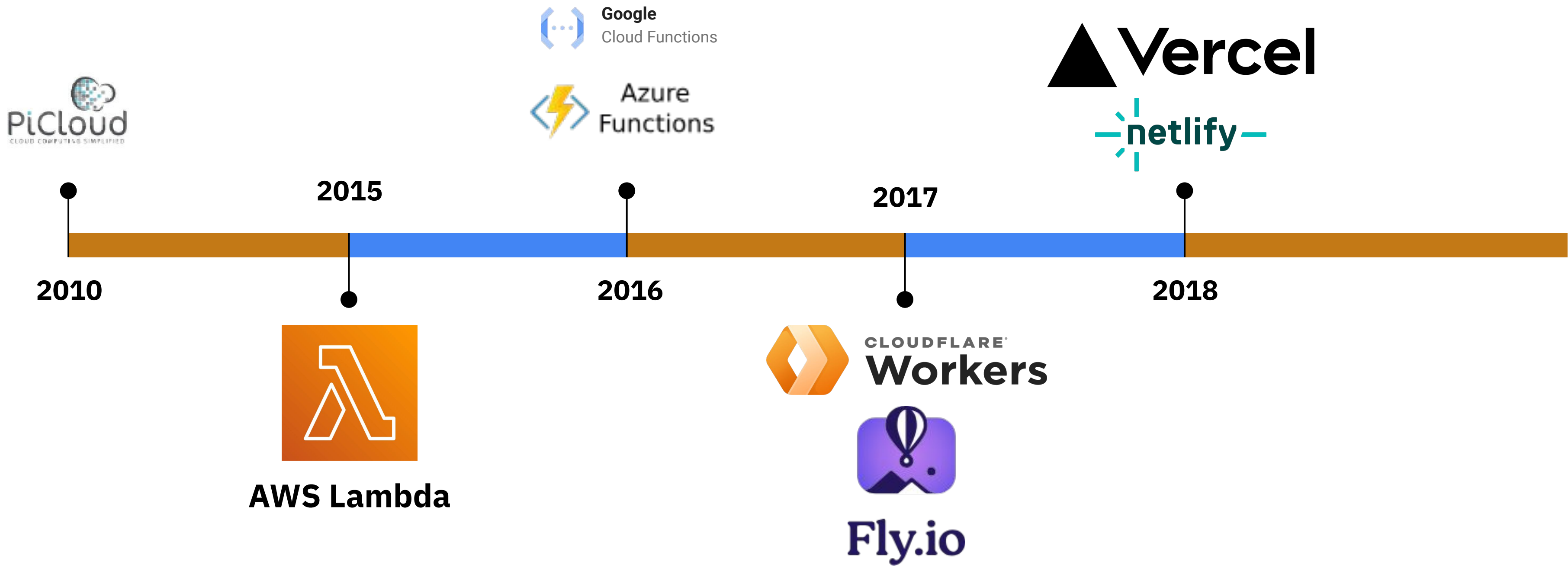
## FaaS Programming Model

- Functions need to be *stateless*: don’t depend on any context that lives in memory or storage
    - e.g. no user sessions; authentication needs to occur on *every* invocation
    - Idempotence is a useful property for failure retries

- Functions should be *asynchronous* and terminate relatively quickly
    - Short-lived execution in response to events – not long-running complex jobs
    - Runtime timeout is usually on the order of minutes

- Functions should be low on compute resources
    - Cost is a function of compute resources and the amount of time those resources are used
    - AWS Lambda resource limit: 6 vCPU cores and 10 GB of RAM

# Timeline: Functions as a Service



## AWS Lambda

- Functions as *event handlers*: trigger on any observable change to cloud environment
    - e.g. HTTP request, file upload to S3 bucket, timer, etc.
    - Under the hood: *event* is a JSON formatted HTTP request

- On an event: AWS spins up function to respond
    - Behind the scenes infra managed for you, though built on containers
    - Under the hood: *response* is a JSON formatted HTTP response

## Example: AWS Lambda

```python
import json


def lambda_handler(event, context):
    print("Hello from Lambda!")
    return {
        "statusCode": 200,
        "body": json.dumps("Hello from Lambda!")
    }
```

## Lambda Runtime Infrastructure

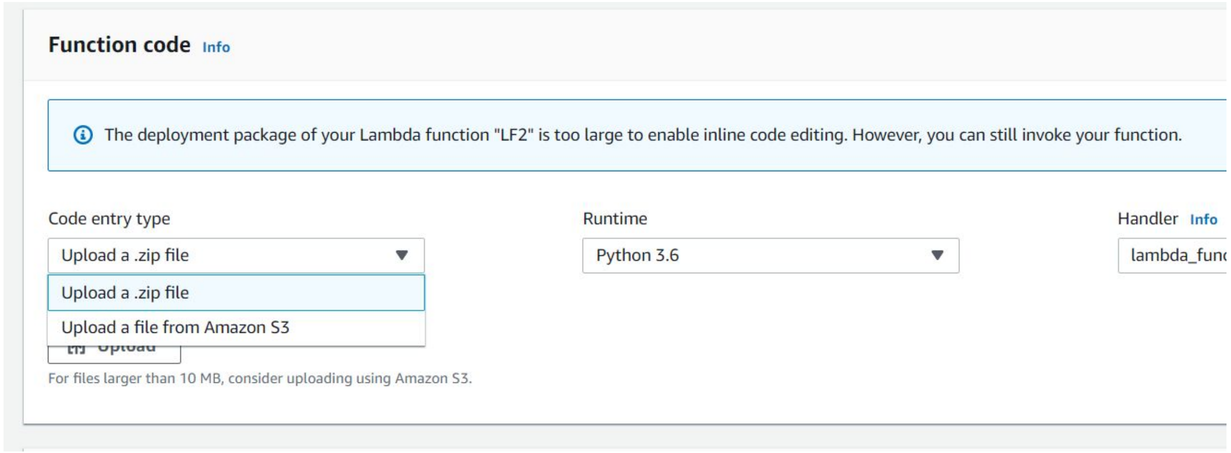
- Initially: NodeJS, Python, Ruby, Java, Go, C# supported (*managed runtime*)
    - Nov 2018: **Any language** supported, if you can implement a custom runtime
    - Dec 2020: **Bring your own container** to Lambda – allows you to interact more with FS too!

- Under the hood: EC2 instances with proprietary container orchestration

- **Lambda Runtime API**: Adapt events to HTTP requests processable by Lambda code

## Deploying Lambdas

- Originally: Deploy via uploading a ZIP file on AWS console or CLI
  - Later: Deploy ZIP file from S3 bucket, or from CDK



Function code Info

The deployment package of your Lambda function "LF2" is too large to enable inline code editing. However, you can still invoke your function.

Code entry type: Upload a .zip file

Runtime: Python 3.6

Handler Info: lambda_func

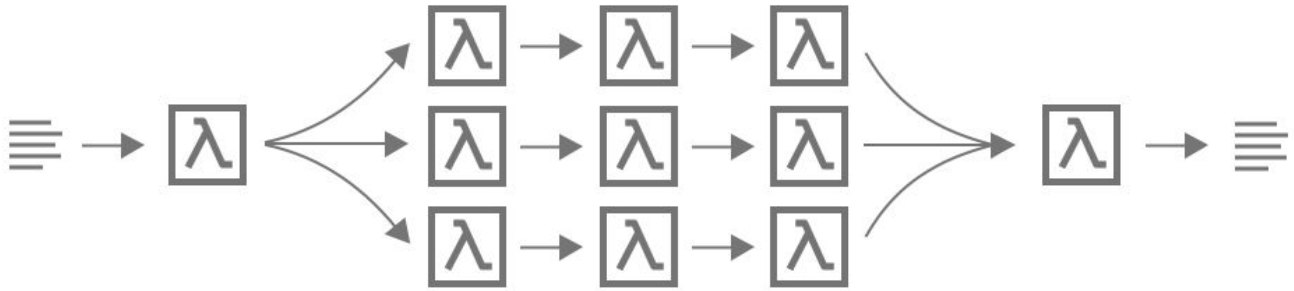
For files larger than 10 MB, consider uploading using Amazon S3.

- Dec 2020: Deploy from Docker image hosted on Elastic Container Registry
  - **Recommended option: makes release management easier**

## More Lambda features

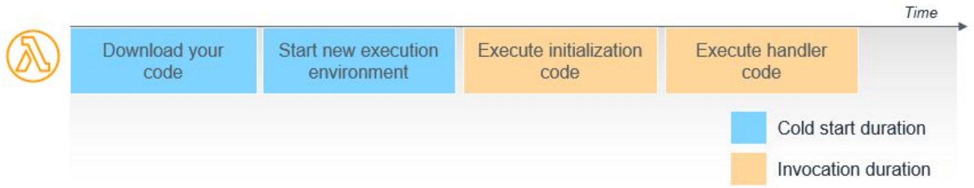
- Attach to a VPC
  - Allows controlling/connecting to EC2 instances, Aurora/RDS databases, etc.

- Step functions for more fine-grained Lambda orchestration



## Lambda Footguns

- *Cold Start Latency*: Cache miss for Lambda code requires code download and environment reprovisioning
  - This can take **hundreds of milliseconds** and is *worse* for compiled languages
  - Mitigation: **provisioned concurrency**: pay AWS to keep a running container, but this is costly



- *Container Reuse*: Execution environment is reused in entirety; filesystem may be “dirty” from previous invocation
  - This is an attempt to avoid cold start problem
  - Mitigation: clean up filesystem (including `/tmp`) at the beginning of a Lambda job

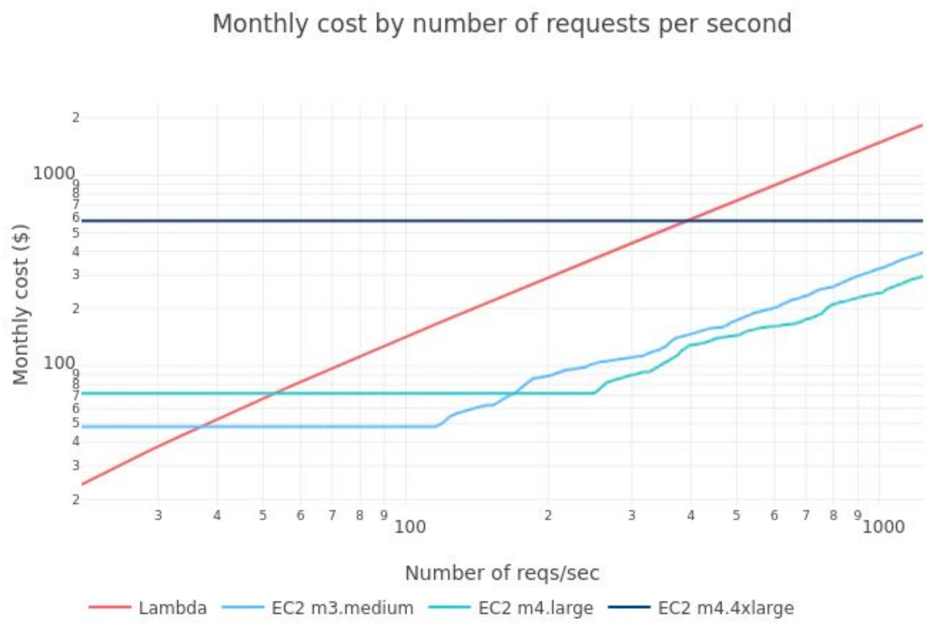
## Lambda Pricing Model

- Pricing is a function of execution duration, memory capacity, and requests
  - Lambda w/ 1.5 GB of memory costs $0.00000002/ms (i.e., $0.07/hr)
  - Request pricing: $0.20 per million requests (negligible)

- Example: 1M image uploads per month through 1-second processing lambda
  - Cost: $20.00 for processing + $0.20 for requests = $20.20 per month

*Lambda lets you allocate infrastructure cost in terms of requests and responses, rather than always-on compute capacity.*

# Lambda vs EC2: Cost Comparison



*BBVA – Economics of Serverless (2018)*
*Breakeven at 10-40 reqs/s*

## When to use Lambdas vs ECS?

*Recall ECS Fargate is also bring-your-own-container*

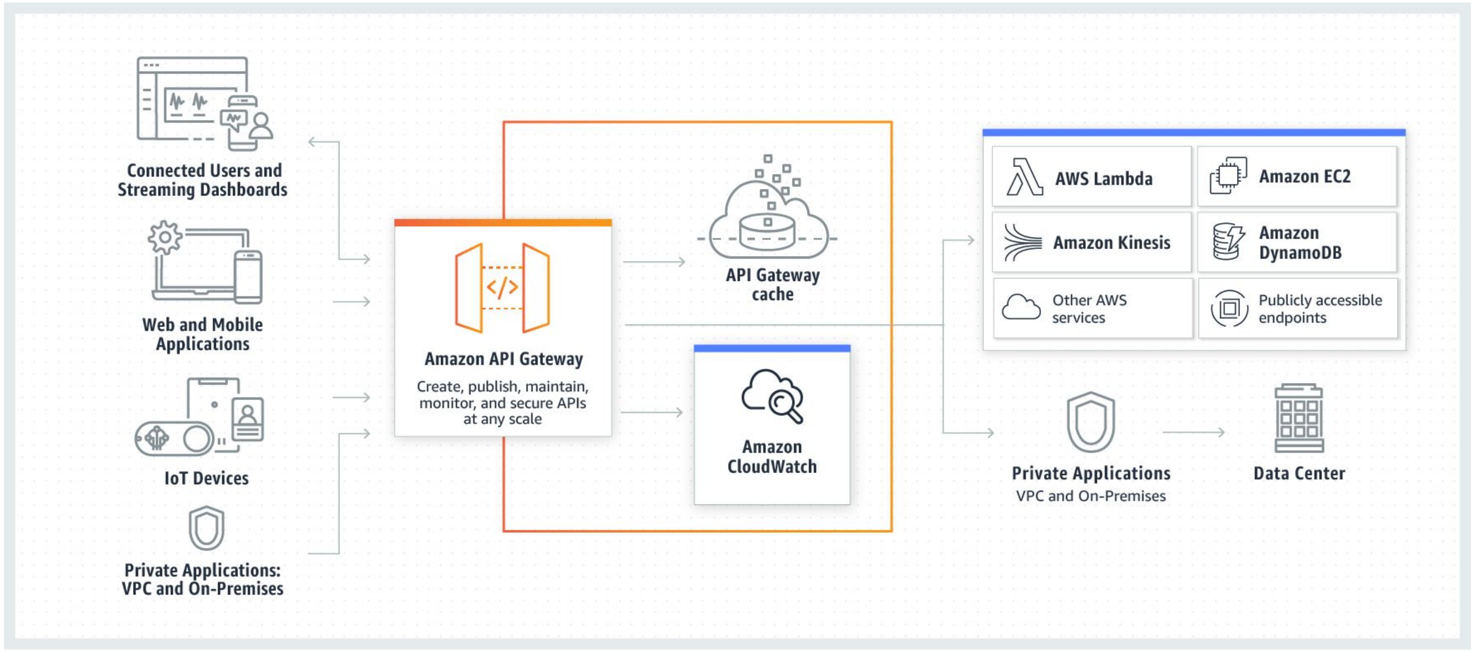
- Lambdas are better for **short-lived, low-resource, self-contained tasks**, where demand is **inconsistent** and may be zero at times
    - e.g. database and S3 periodic cleanup; some processing pipelines
    - **Don't write your entire backend as a Lambda!** Beyond resource limitations, it also forces certain paradigms surrounding user sessions and state.

- ECS is better for **longer (continuous) tasks** that need to touch other AWS resources, and those that need more **compute resources**
    - e.g. Web backends and APIs, containerized ML tasks

*Lambdas should be considered part of infra "glue code" rather than application logic*

# AWS API Gateway

*By default, Lambdas are not invokable from the public internet!*
*API Gateway turns HTTP requests into Lambda event triggers.*


Connected Users and Streaming Dashboards
Web and Mobile Applications
IoT Devices
Private Applications: VPC and On-Premises
Amazon API Gateway
Create, publish, maintain, monitor, and secure APIs at any scale
API Gateway cache
Amazon CloudWatch
AWS Lambda
Amazon EC2
Amazon Kinesis
Amazon DynamoDB
Other AWS services
Publicly accessible endpoints
Private Applications
VPC and On-Premises
Data Center

## Demo: Lambda with API Gateway

## FaaS on PaaS

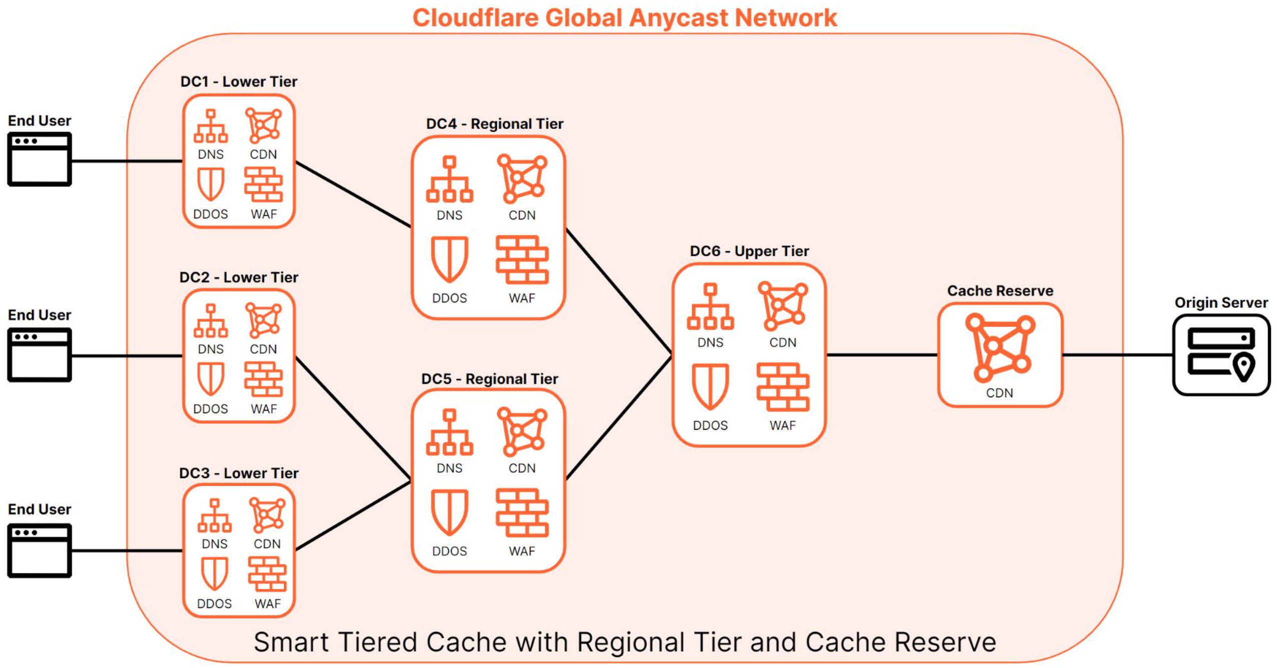
- Many Javascript frameworks offer API routing for backend-like functionality
  - e.g. Next.js, Remix, SvelteKit, React Server Components

- FaaS gives you a way to deploy these minimal backends at low cost
  - By definition, don't need API functions to be constantly listening

- Given hosting frontend + backend, deploying on PaaS platforms can be cheaper than IaaS
  - e.g. Vercel, Netlify: 100 GB/mo free, 1 TB/mo for $20

# Functions on the Edge

*Recall:* ***Content Delivery Network*** *(CDN) – distribute web hosting close to users*



## Functions on the Edge

- **Motivation**: Take advantage of CDN points of presence to run serverless functions close to users
    - Improves latency for functions callable from the open web

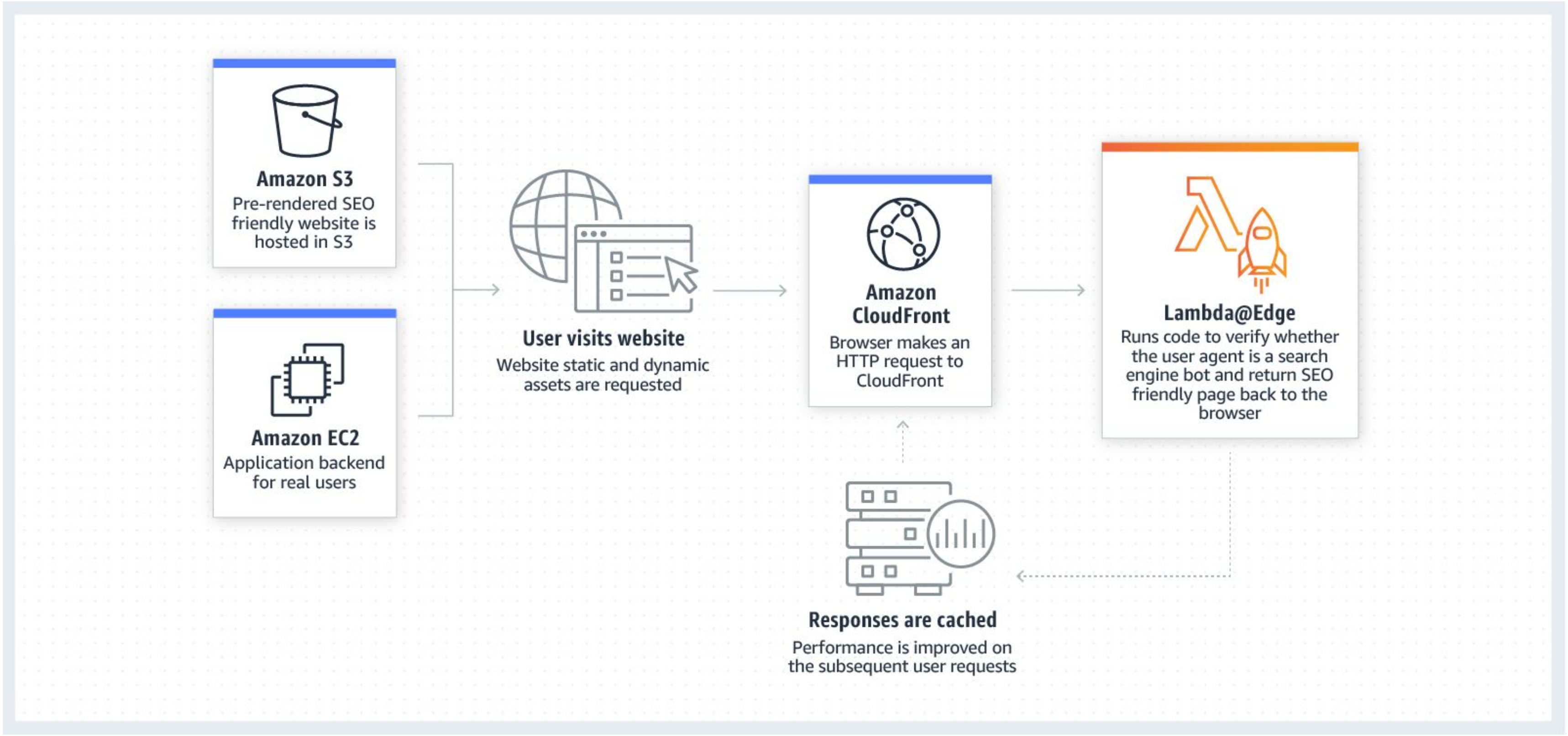
- e.g. *CS40 Provisioner* is run on Cloudflare Pages Functions

Lambda@Edge

CLOUDFLARE Workers

Fly.io

# Example: Edge Functions


Amazon S3
Pre-rendered SEO friendly website is hosted in S3
Amazon EC2
Application backend for real users
User visits website
Website static and dynamic assets are requested
Amazon CloudFront
Browser makes an HTTP request to CloudFront
Lambda@Edge
Runs code to verify whether the user agent is a search engine bot and return SEO friendly page back to the browser
Responses are cached
Performance is improved on the subsequent user requests

*If hosting serverless functions at the edge decreases latency, why don’t we do so for all serverless functions?*

## Caveats of Edge Functions

- **Runtime support**: Lambda@Edge supports only NodeJS and Python; Cloudflare Workers supports only a small subset of NodeJS

- **Resource limits**
  - Lambda@Edge functions can only last up to 30 seconds of execution, and request and response sizes can only be up to 1MB
  - Cloudflare Workers can only use up to 128 MB memory

- **Feature limits**: Lambda@Edge functions can't communicate with VPC resources, and don't support environment variables or observability tracing

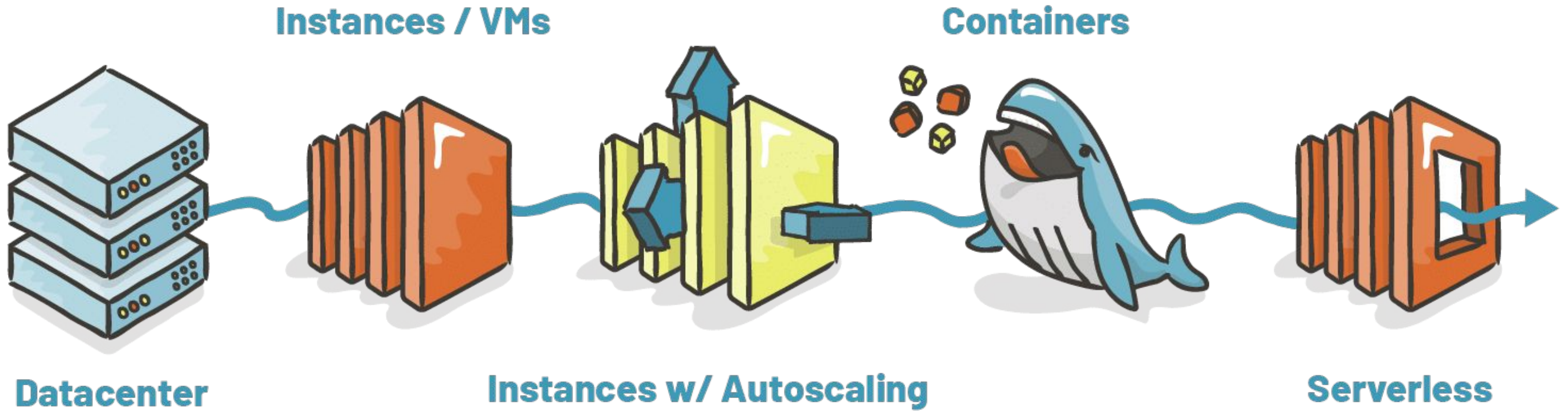
# Serverless Beyond FaaS

*Recall: We want to abstract away hardware and base software infrastructure details. AWS offers a number of such serverless managed services.*

| Compute | Storage | Application |
|---|---|---|
| Lambda* | S3* | SQS |
| Fargate* | DynamoDB | API Gateway* |
| | Aurora Serverless* | SNS |
| | ElastiCache Serverless* | Cognito |

** used by CS40*

## Serverless as a Spectrum



*Duckbill Group – A Simple, Yet Effective Cost Optimization Framework, 2023*

## Takeaway

- We can make *any* workload more serverless by using more managed services
    - Doesn't matter if the workload is FaaS-appropriate or not
    - Recall FaaS isn't the only type of serverless technology!

- Serverless is a **sane, low-cost, low-overhead default** for new projects
    - Specific requirements may dictate more involved solutions, but this usually doesn't occur at greenfield stage
    - Migrate from serverless paradigm only when it becomes necessary and can spare engineering time to manage the infrastructure in a more involved way

## Next Lecture: Cloud AI/ML & HPC (2/14)