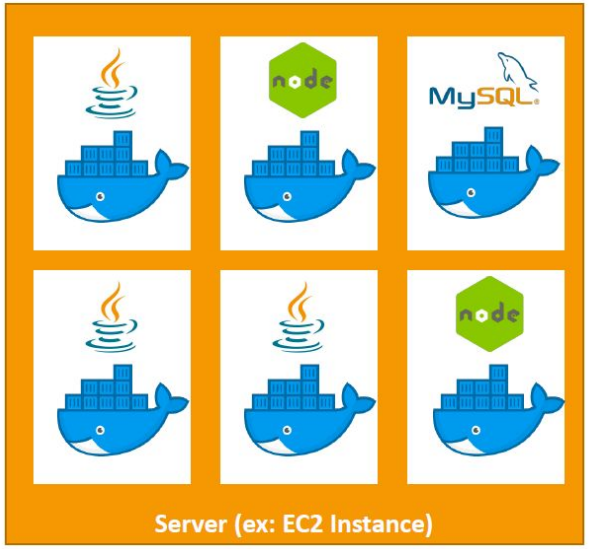
AWS ECS - Essentials

What is Docker

- Docker is a software development platform to deploy apps
- Apps are packaged in containers that can be run on any OS
- Apps run the same, regardless of where they're run
    - Any machine
    - No compatibility issues
    - Predictable behavior
    - Less work
    - Easier to maintain and deploy
    - Works with any language, any OS, any technology



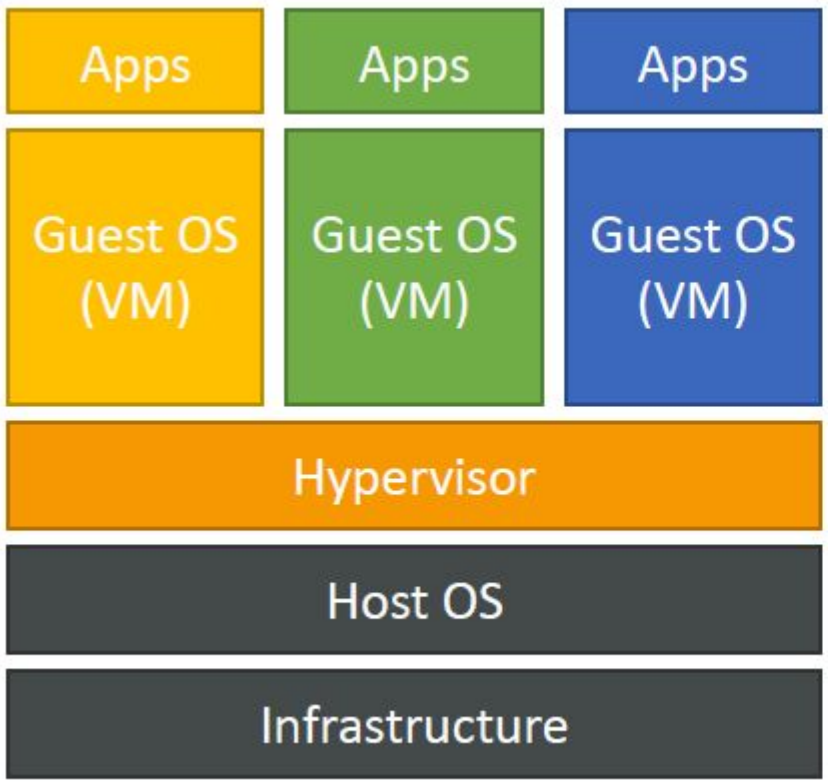
Docker on an OS

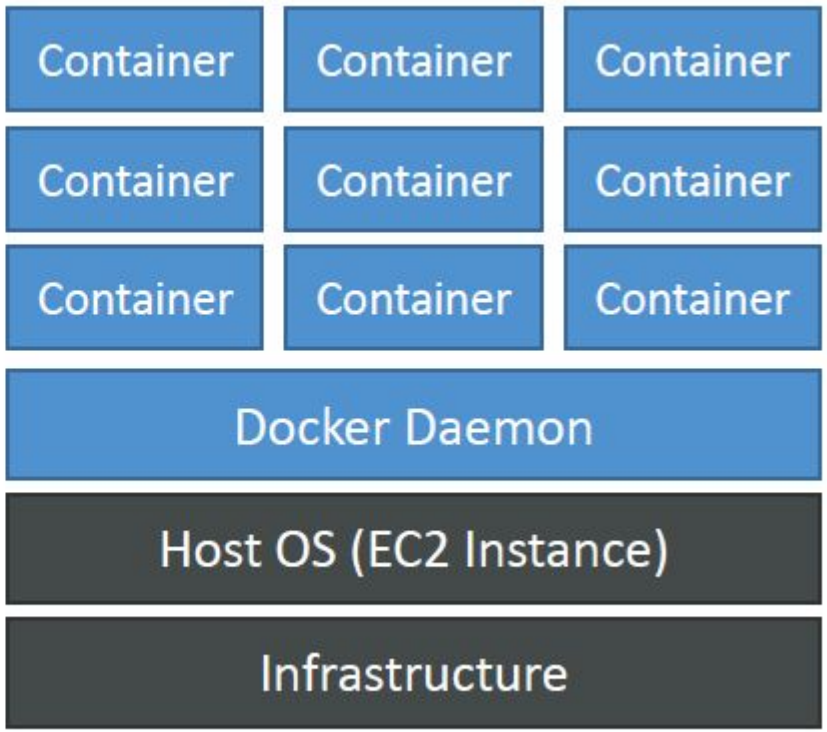
Where Docker images are stored?

- Docker images are stored in Docker Repositories
- Public: Docker Hub https://hub.docker.com/
    - Find base images for many technologies or OS:
    - Ubuntu
    - MySQL
    - NodeJS, Java…
- **Private: Amazon ECR (Elastic Container Registry)**

Docker versus Virtual Machines

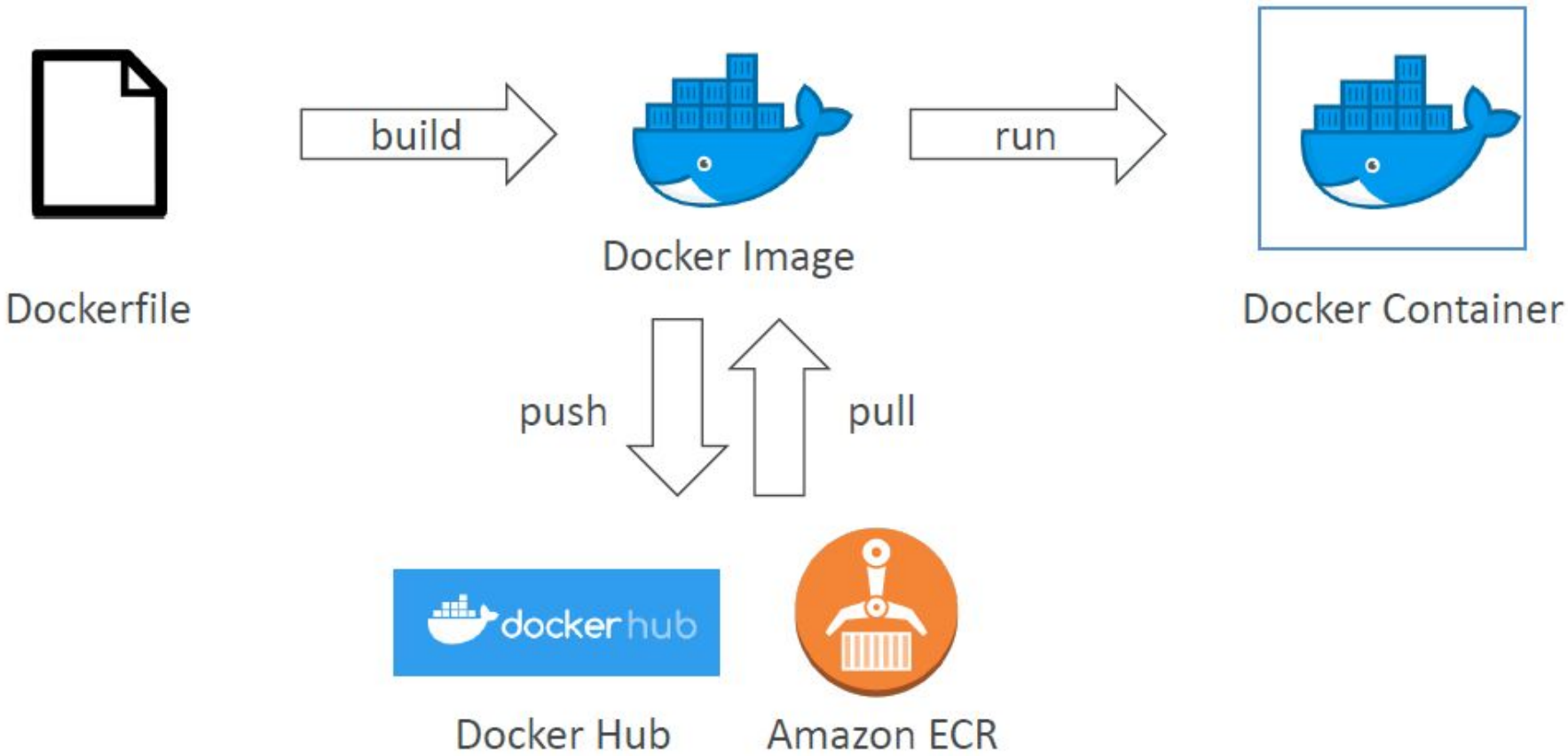
- Docker is ”sort of” a virtualization technology, but not exactly
- Resources are shared with the host => many containers on one server





Getting Started with Docker

- Download Docker at: https://www.docker.com/get-started

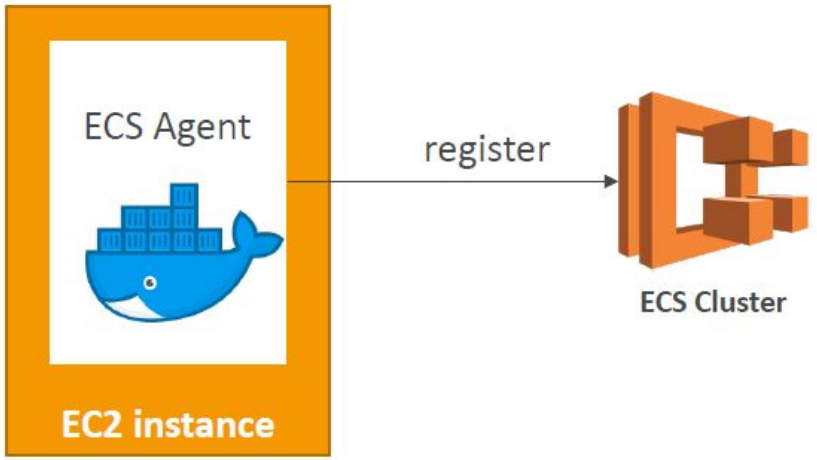


Docker Containers Management

- To manage containers, we need a container management platform
- Three choices:
  - ECS: Amazon's own platform
  - Fargate: Amazon's own Serverless platform
  - EKS: Amazon's managed Kubernetes (open source)

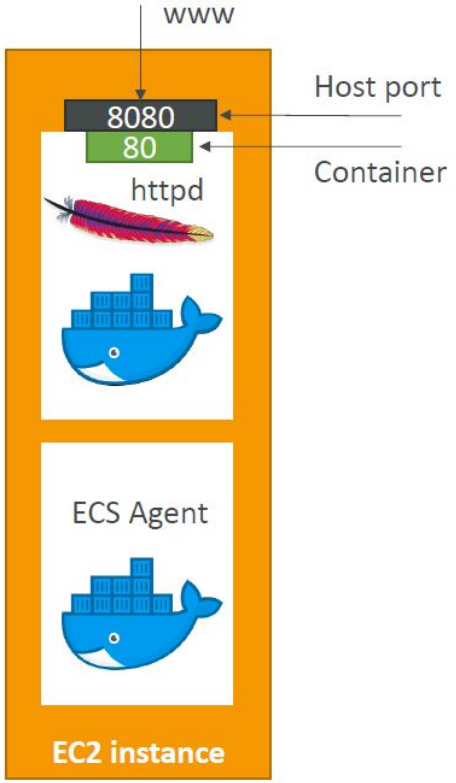
ECS Clusters Overview

- ECS Clusters are logical grouping of EC2 instances
- EC2 instances run the ECS agent (Docker container)
- The ECS agents registers the instance to the ECS cluster
- The EC2 instances run a special AMI, made specifically for ECS



ECS Task Definitions

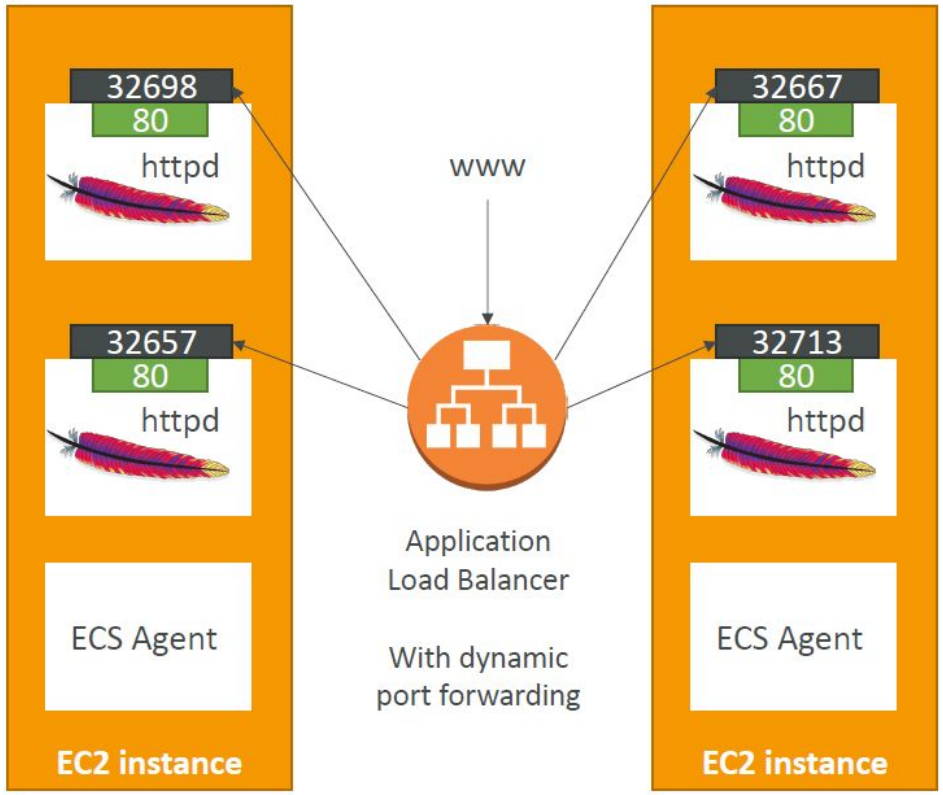
- Tasks definitions are metadata in JSON form to tell ECS how to run a Docker Container
- It contains crucial information around:
  - Image Name
  - Port Binding for Container and Host
  - Memory and CPU required
  - Environment variables
  - Networking information
  - IAM Role
  - Logging configuration (ex CloudWatch)



LAB 1: Create ECS and Task Definition to run httpd

ECS Service

- ECS Services help define how many tasks should run and how they should be run
- They ensure that the number of tasks desired is running across our fleet of EC2 instances.
- They can be linked to ELB / NLB / ALB if needed



ECR

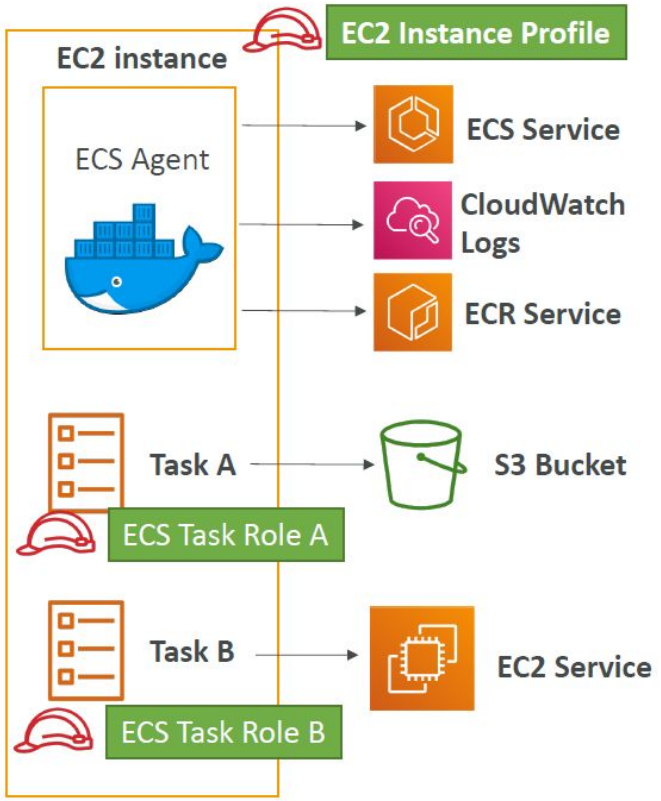
- So far we've been using Docker images from Docker Hub (public)
- ECR is a private Docker image repository
- Access is controlled through IAM (permission errors => policy)
- AWS CLI v1 login command (may be asked at the exam)
    - $(aws ecr get-login --no-include-email --region eu-west-1)
- AWS CLI v2 login command (newer, may also be asked at the exam - pipe)
    - aws ecr get-login-password --region eu-west-1 | docker login --username AWS –password-stdin 1234567890.dkr.ecr.eu-west-1.amazonaws.com
- Docker Push & Pull:
    - docker push 1234567890.dkr.ecr.eu-west-1.amazonaws.com/demo:latest
    - docker pull 1234567890.dkr.ecr.eu-west-1.amazonaws.com/demo:latest

Fargate

- When launching an ECS Cluster, we have to create our EC2 instances
- If we need to scale, we need to add EC2 instances
- So we manage infrastructure…
- With Fargate, it’s all Serverless!
- We don’t provision EC2 instances
- We just create task definitions, and AWS will run our containers for us
- To scale, just increase the task number. Simple! No more EC2

ECS IAM Roles Deep Dive

- EC2 Instance Profile:
  - Used by the ECS agent
  - Makes API calls to ECS service
  - Send container logs to CloudWatch Logs
  - Pull Docker image from ECR
- ECS Task Role:
  - Allow each task to have a specific role
  - Use different roles for the different ECS Services you run
  - Task Role is defined in the task definition



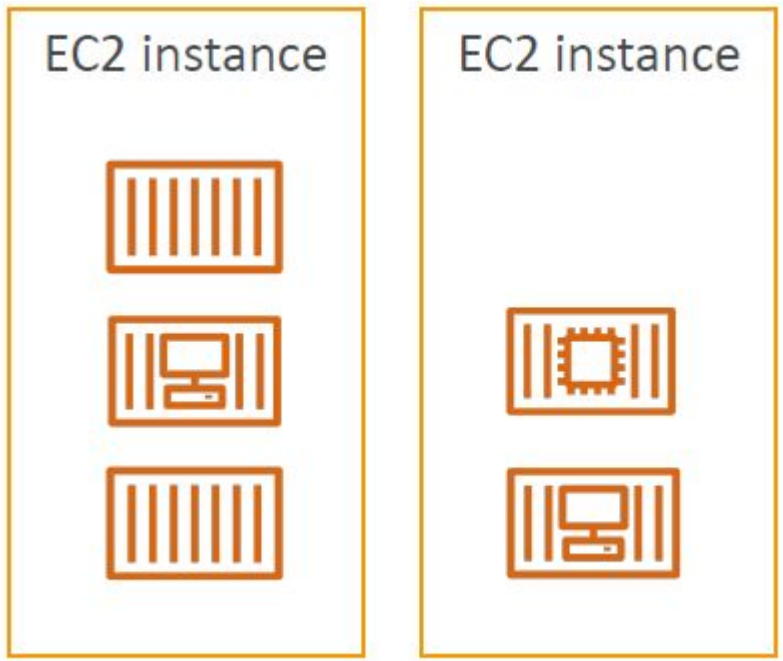
ECS Tasks Placement

- When a task of type EC2 is launched, ECS must determine where to place it, with the constraints of CPU, memory, and available port.
- Similarly, when a service scales in, ECS needs to determine which task to terminate.
- To assist with this, you can define a task placement strategy and task placement constraints
- **Note: this is only for ECS with EC2,not for Fargate**

ECS Task Placement Strategies

- Binpack
  - Place tasks based on the least available amount of CPU or memory
  - This minimizes the number of instances in use (cost savings)

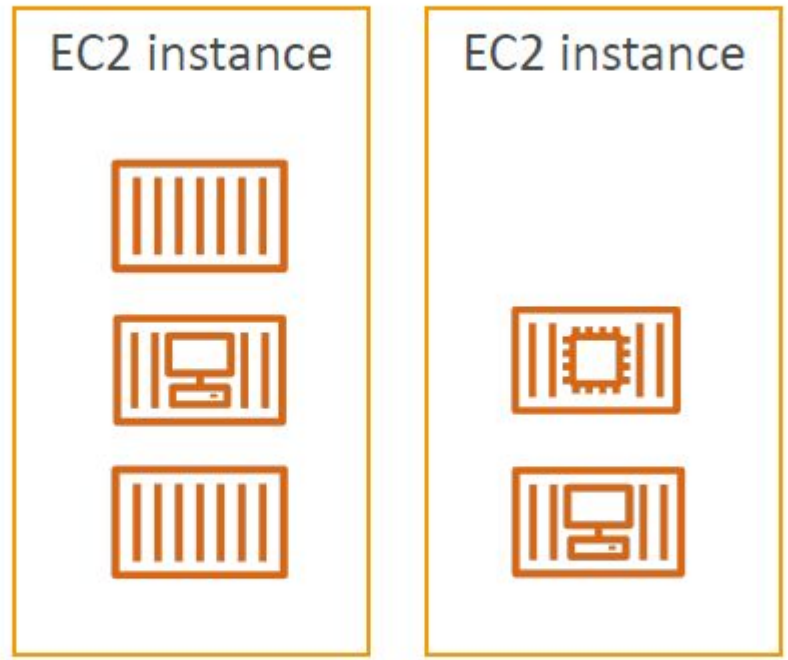
```
"placementStrategy": [
    {
        "field": "memory",
        "type": "binpack"
    }
]
```



Random

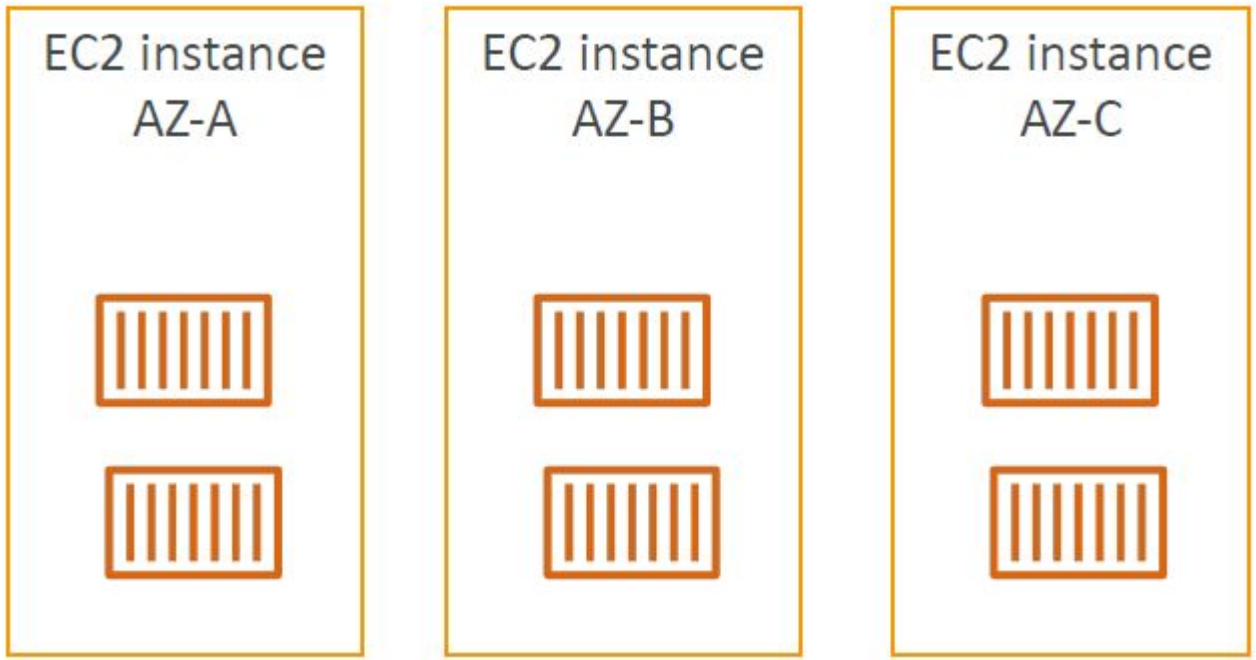
- Place the task randomly

```
"placementStrategy": [
    {
        "type": "random"
    }
]
```



- Spread
  - Place the task evenly based on the specified value
  - Example: instanceId, attribute:ecs.availability-zone

```
"placementStrategy": [
    {
        "field": "attribute:ecs.availability-zone",
        "type": "spread"
    }
]
```



ECS Task Placement Strategies

- You can mix them together

```
"placementStrategy": [
    {
        "field": "attribute:ecs.availability-zone",
        "type": "spread"
    },
    {
        "field": "instanceId",
        "type": "spread"
    }
]
```

```
"placementStrategy": [
    {
        "field": "attribute:ecs.availability-zone",
        "type": "spread"
    },
    {
        "field": "memory",
        "type": "binpack"
    }
]
```

ECS Task Placement Constraints

- distinctInstance: place each task on a different container instance

```
"placementConstraints": [
    {
        "type": "distinctInstance"
    }
]
```

- memberOf: places task on instances that satisfy an expression
  - Uses the Cluster Query Language (advanced)

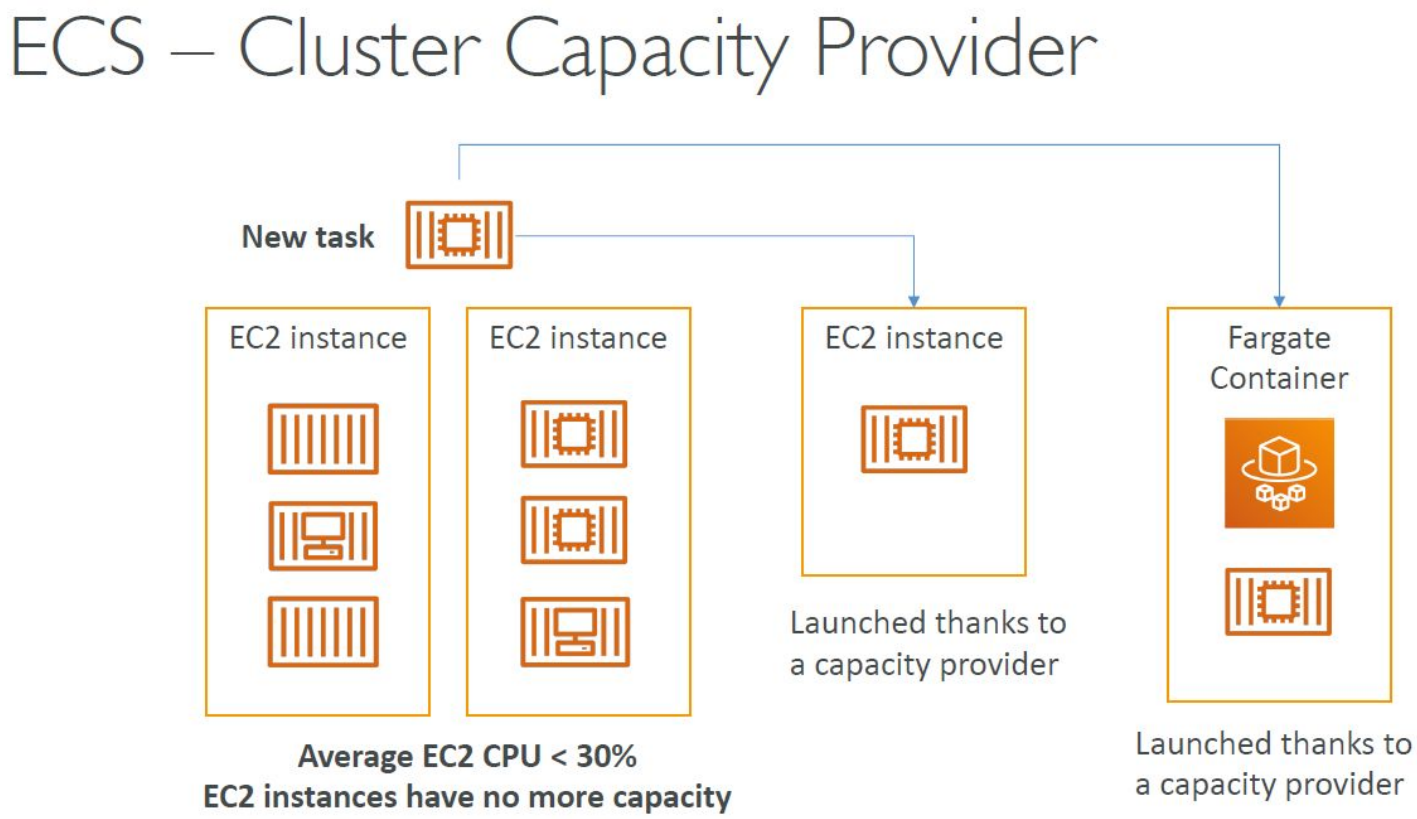
```
"placementConstraints": [
    {
        "expression": "attribute:ecs.instance-type =~ t2.*",
        "type": "memberOf"
    }
]
```

ECS – Service Auto Scaling

- CPU and RAM is tracked in CloudWatch at the ECS service level
- Target Tracking: target a specific average CloudWatch metric
- Step Scaling: scale based on CloudWatch alarms
- Scheduled Scaling: based on predictable changes
- ECS Service Scaling (task level) ≠ EC2 Auto Scaling (instance level)
- Fargate Auto Scaling is much easier to setup (because serverless)

ECS – Cluster Capacity Provider

- A Capacity Provider is used in association with a cluster to determine the infrastructure that a task runs on
- For ECS and Fargate users, the FARGATE and FARGATE_SPOT capacity providers are added automatically
- For Amazon ECS on EC2, you need to associate the capacity provider with an auto-scaling group
- When you run a task or a service, you define a capacity provider strategy, to prioritize in which provider to run.
- This allows the capacity provider to automatically provision infrastructure for you



ECS Summary + Exam Tips

- ECS is used to run Docker containers and has 3 flavors:
- ECS “Classic”: provision EC2 instances to run containers onto
- Fargate: ECS Serverless, no more EC2 to provision
- EKS: Managed Kubernetes by AWS

ECS Classic

- EC2 instances must be created
- We must configure the file **/etc/ecs/ecs.config with the cluster name**
- The EC2 instance must run an ECS agent
- EC2 instances can run multiple containers on the same type:
- You must not specify a host port (only container port)
- You should use an Application Load Balancer with the dynamic port mapping
- The EC2 instance security group must allow traffic from the ALB on all ports
- ECS tasks can have IAM Roles to execute actions against AWS
- Security groups operate at the instance level, not task level

ECR is used to store Docker Images

- ECR is tightly integrated with IAM
- AWS CLI v1 login command (may be asked at the exam)
- $(aws ecr get-login --no-include-email --region eu-west-1)
- “aws ecr get-login” generates a “docker login” command
- AWS CLI v2 login command (newer, may also be asked at the exam - pipe)
- aws ecr get-login-password --region eu-west-1 | docker login --username AWS --password-stdin 1234567890.dkr.ecr.eu-west-1.amazonaws.com
- Docker Push & Pull:
- docker push 1234567890.dkr.ecr.eu-west-1.amazonaws.com/demo:latest
- docker pull 1234567890.dkr.ecr.eu-west-1.amazonaws.com/demo:latest
- In case an EC2 instance (or you) cannot pull a Docker image, check IAM

Developer problems on AWS

- Managing infrastructure
- Deploying Code
- Configuring all the databases, load balancers, etc
- Scaling concerns
- Most web apps have the same architecture (ALB + ASG)
- All the developers want is for their code to run!
- Possibly, consistently across different applications and environments

AWS Elastic Beanstalk to rescue

- Elastic Beanstalk is a developer centric view of deploying an application on AWS
- It uses all the component's we've seen before:

EC2, ASG, ELB, RDS, etc…

- But it's all in one view that's easy to make sense of!
- We still have full control over the configuration
- Beanstalk is free but you pay for the underlying instances

Managed service

- Instance configuration / OS is handled by Beanstalk
- Deployment strategy is configurable but performed by Elastic Beanstalk

- Just the application code is the responsibility of the developer
- Three architecture models:
  - Single Instance deployment: good for dev
  - LB + ASG: great for production or pre-production web applications
  - ASG only: great for non-web apps in production (workers, etc..)

Elastic Beanstalk has three components

- Application
- Application version: each deployment gets assigned a version
- Environment name (dev, test, prod…): free naming

• You deploy application versions to environments and can promote application versions to the next environment

• Rollback feature to previous application version

• Full control over lifecycle of environments

Elastic Beanstalk

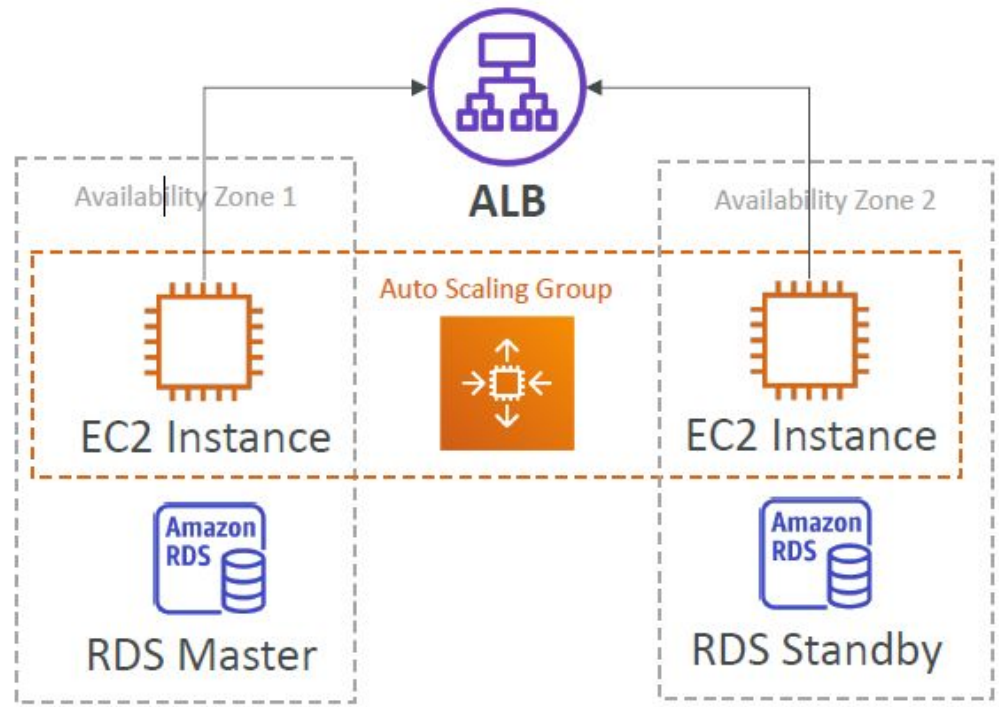
- Support for many platforms:
- Go
- Java SE
- Java with Tomcat
- .NET on Windows Server with IIS
- Node.js
- PHP
- Python
- Ruby
- Packer Builder
- Single Container Docker
- Multicontainer Docker
- Preconfigured Docker
- If not supported, you can write your custom platform (advanced)

Lab : EBS Hands-on

Elastic Beanstalk Deployment Modes

Single Instance
Great for dev

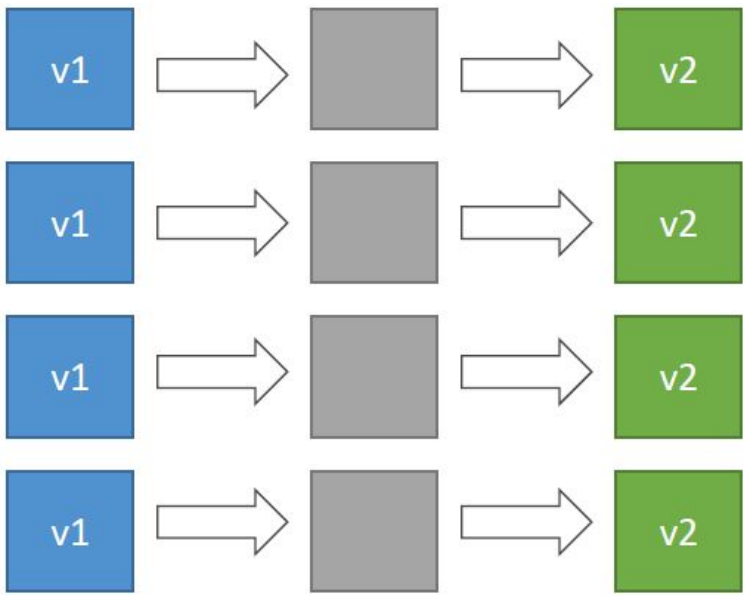
High Availability with Load Balancer
Great for prod



Elastic Beanstalk Deployment

❑ All at once

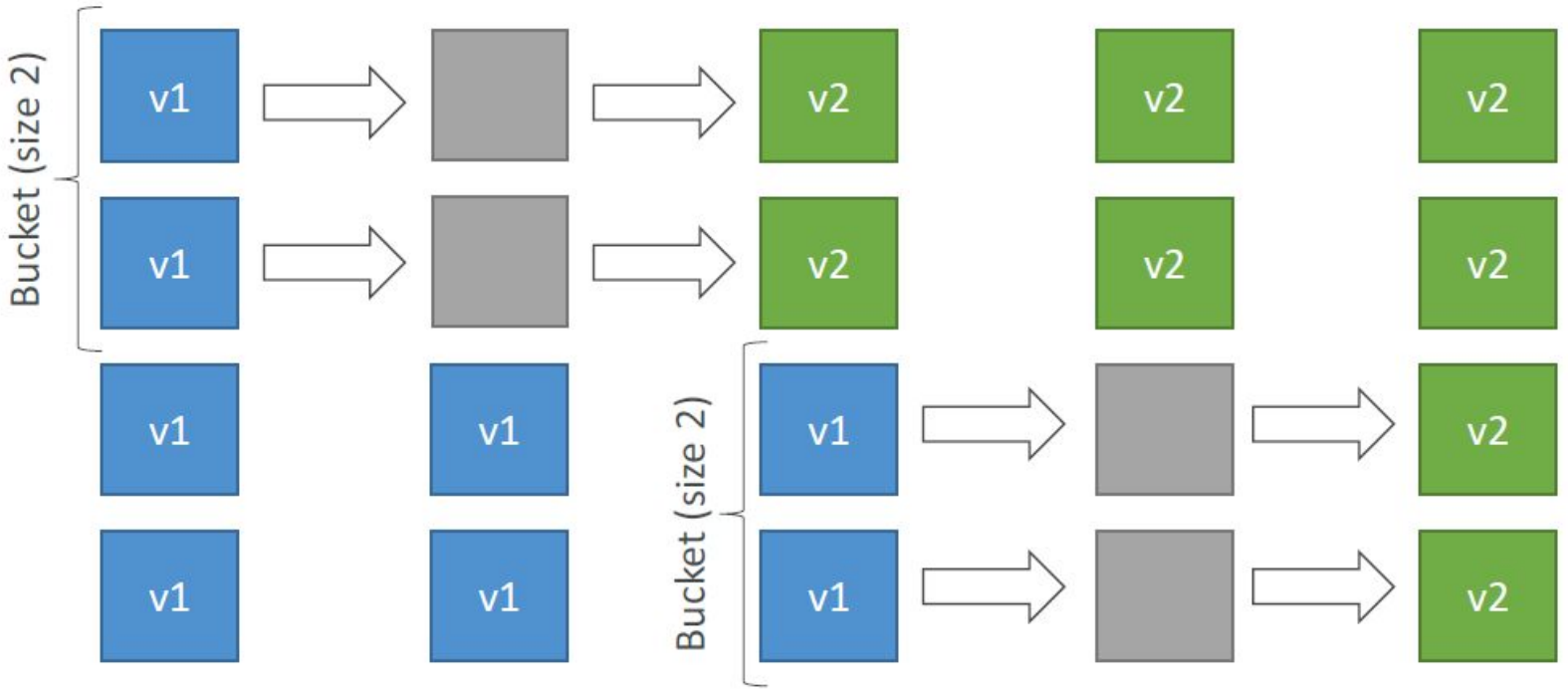
- Fastest deployment
- Application has downtime
- Great for quick iterations in development environment
- No additional cost



Elastic Beanstalk Deployment

❑ Rolling

- Application is running below capacity
- Can set the bucket size
- Application is running both versions simultaneously
- No additional cost
- Long deployment



Elastic Beanstalk Deployment

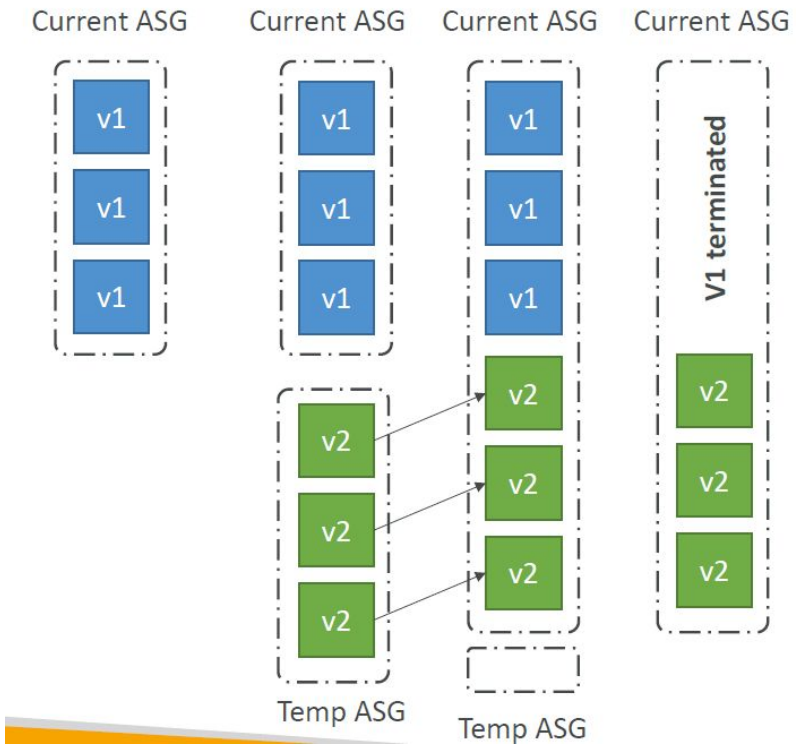
❑ Rolling with additional batches

- Application is running at capacity
- Can set the bucket size
- Application is running both versions simultaneously
- Small additional cost
- Additional batch is removed at the end of the deployment
- Longer deployment
- Good for prod

Elastic Beanstalk Deployment

❑ Immutable

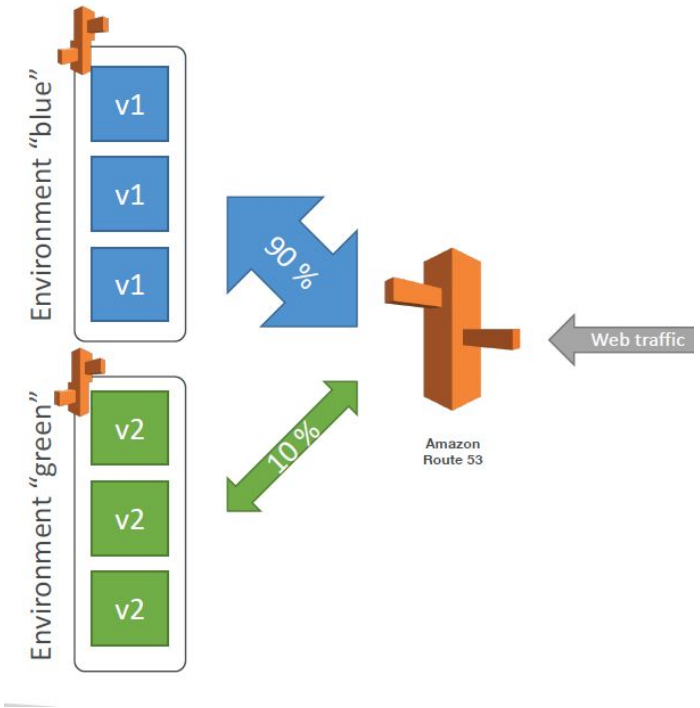
- Zero downtime
- New Code is deployed to new instances on a temporary ASG
- High cost, double capacity
- Longest deployment
- Quick rollback in case of failures (just terminate new ASG)
- Great for prod



Elastic Beanstalk Deployment

❑ Blue / Green

- Not a “direct feature” of Elastic Beanstalk
- Zero downtime and release facility
- Create a new “stage” environment and deploy v2 there
- The new environment (green) can be validated independently and roll back if issues
- Route 53 can be setup using weighted policies to redirect a little bit of traffic to the stage environment
- Using Beanstalk, “swap URLs” when done with the environment test



Elastic Beanstalk Deployment Summary from AWS Doc

- https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/usingfeatures.deploy-existing-version.html

Deployment methods

Method	Impact of failed deployment	Deploy time	Zero downtime	No DNS change	Rollback process	Code deployed to
All at once	Downtime	🕒	x	✓	Manual redeploy	Existing instances
Rolling	Single batch out of service; any successful batches before failure running new application version	🕒🕒†	✓	✓	Manual redeploy	Existing instances
Rolling with an additional batch	Minimal if first batch fails; otherwise, similar to **Rolling**	🕒🕒🕒†	✓	✓	Manual redeploy	New and existing instances
Immutable	Minimal	🕒🕒🕒🕒	✓	✓	Terminate new instances	New instances
Traffic splitting	Percentage of client traffic routed to new version temporarily impacted	🕒🕒🕒🕒††	✓	✓	Reroute traffic and terminate new instances	New instances
Blue/green	Minimal	🕒🕒🕒🕒	✓	x	Swap URL	New instances

Elastic Beanstalk Deployment Process

- Describe dependencies

(requirements.txt for Python, package.json for Node.js)

- Package code as zip, and describe dependencies
- Python: requirements.txt
- Node.js: package.json
- Console: upload zip file (creates new app version), and then deploy
- CLI: create new app version using CLI (uploads zip), and then deploy
- Elastic Beanstalk will deploy the zip on each EC2 instance, resolve dependencies and start the application

Beanstalk Lifecycle Policy

- Elastic Beanstalk can store at most 1000 application versions
- If you don’t remove old versions, you won’t be able to deploy anymore
- To phase out old application versions, use a lifecycle policy
- Based on time (old versions are removed)
- Based on space (when you have too many versions)
- Versions that are currently used won’t be deleted
- Option not to delete the source bundle in S3 to prevent data loss

Elastic Beanstalk Extensions

- A zip file containing our code must be deployed to Elastic Beanstalk
- All the parameters set in the UI can be configured with code using files
- Requirements:
    - in the .ebextensions/ directory in the root of source code
    - YAML / JSON format
    - .config extensions (example: logging.config)
    - Able to modify some default settings using: option_settings
    - Ability to add resources such as RDS, ElastiCache, DynamoDB, etc…
- Resources managed by .ebextensions get deleted if the environment goes away

Elastic Beanstalk Under the Hood

- Under the hood, Elastic Beanstalk relies on CloudFormation
- CloudFormation is used to provision other AWS services
- Use case: you can define CloudFormation resources in your .ebextensions to provision ElastiCache, an S3 bucket, anything you want!

Elastic Beanstalk Cloning

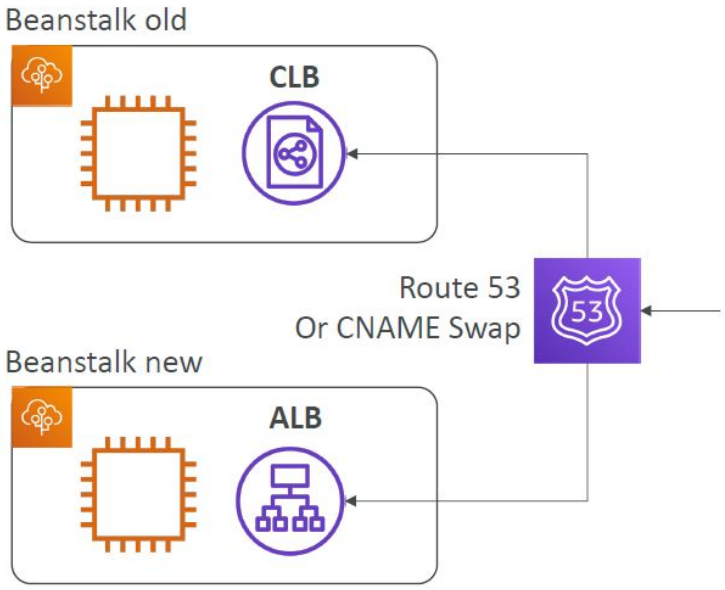
- Clone an environment with the exact same configuration
- Useful for deploying a “test” version of your application
- All resources and configuration are preserved:
    - Load Balancer type and configuration
    - RDS database type (but the data is not preserved)
    - Environment variables
- After cloning an environment, you can change settings

Elastic Beanstalk Migration: Load Balancer

• After creating an Elastic Beanstalk environment, you cannot change the Elastic Load Balancer type (only the configuration)

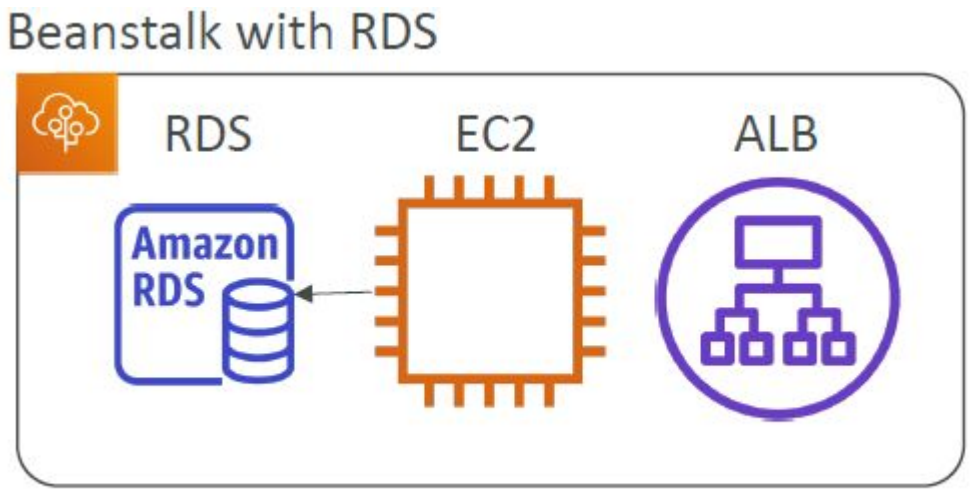
• To migrate:

1. create a new environment with the same configuration except LB (can't clone)
2. deploy your application onto the new environment
3. perform a CNAME swap or Route 53 update



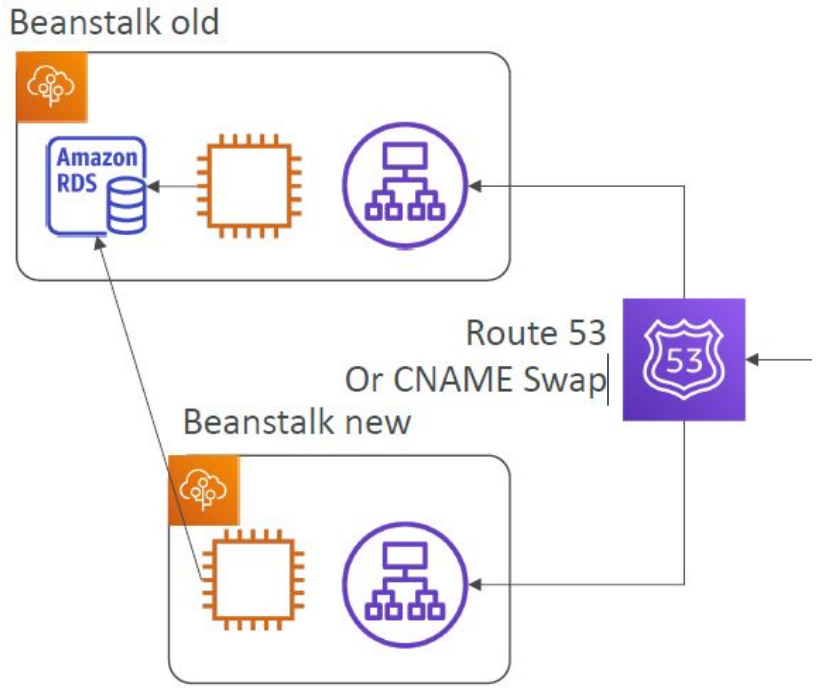
RDS with Elastic Beanstalk

- RDS can be provisioned with Beanstalk, which is great for dev / test
- This is not great for prod as the database lifecycle is tied to the Beanstalk environment lifecycle
- The best for prod is to separately create an RDS database and provide our EB application with the connection string



Elastic Beanstalk Migration: Decouple RDS

1. Create a snapshot of RDS DB (as a safeguard)
2. Go to the RDS console and protect the RDS database from deletion
3. Create a new Elastic Beanstalk environment, without RDS, point your application to existing RDS
4. perform a CNAME swap (blue/green) or Route 53 update, confirm working
5. Terminate the old environment (RDS won't be deleted)
6. Delete CloudFormation stack (in DELETE_FAILED state)



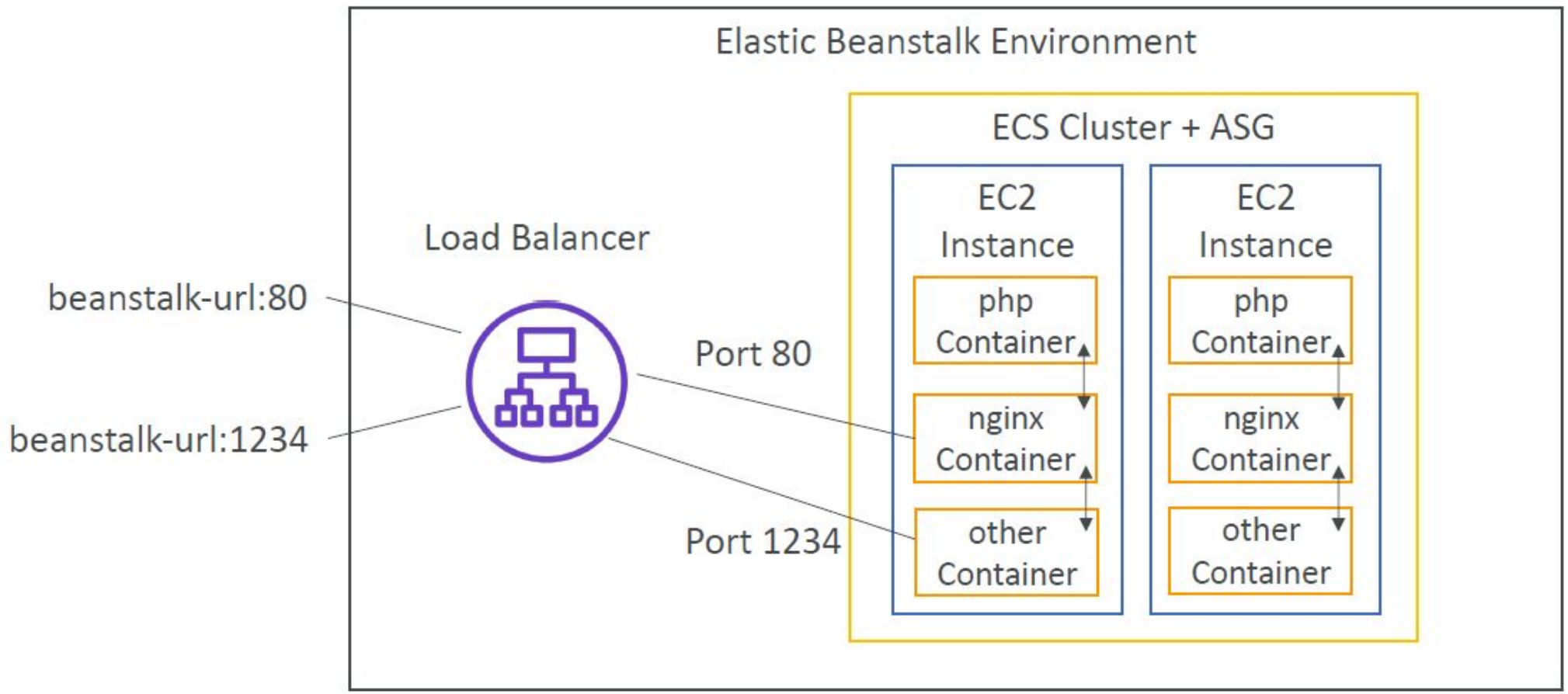
Elastic Beanstalk – Single Docker

- Run your application as a single docker container
- Either provide:
    - Dockerfile: Elastic Beanstalk will build and run the Docker container
    - Dockerrun.aws.json (v1): Describe where *already built* Docker image is
    - Image
    - Ports
    - Volumes
    - Logging
    - Etc...
- Beanstalk in Single Docker Container does not use ECS

Elastic Beanstalk – Multi Docker Container

- Multi Docker helps run multiple containers per EC2 instance in EB
- This will create for you:
- ECS Cluster
- EC2 instances, configured to use the ECS Cluster
- Load Balancer (in high availability mode)
- Task definitions and execution
- Requires a config Dockerrun.aws.json (v2) at the root of source code
- Dockerrun.aws.json is used to generate the ECS task definition
- Your Docker images must be pre-built and stored in ECR for example

Elastic Beanstalk + Multi Docker ECS

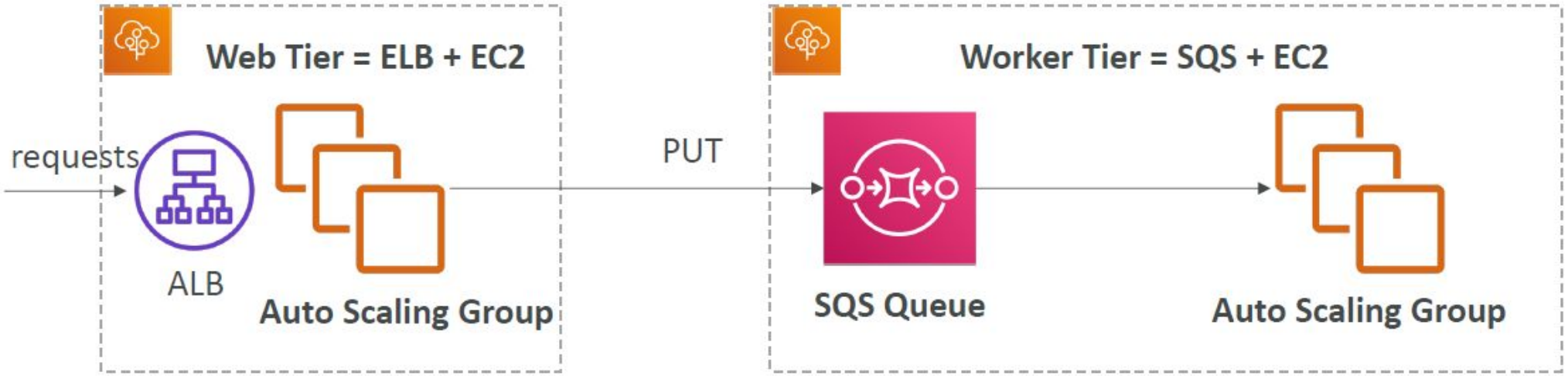


Elastic Beanstalk and HTTPS

- Beanstalk with HTTPS
  - Idea: Load the SSL certificate onto the Load Balancer
  - Can be done from the Console (EB console, load balancer configuration)
  - Can be done from the code: .ebextensions/securelistener-alb.config
  - SSL Certificate can be provisioned using ACM (AWS Certificate Manager) or CLI
  - Must configure a security group rule to allow incoming port 443 (HTTPS port)
- Beanstalk redirect HTTP to HTTPS
  - Configure your instances to redirect HTTP to HTTPS: https://github.com/awsdocs/elastic-beanstalk-samples/tree/master/configuration-files/awsprovided/security-configuration/https-redirect
  - OR configure the Application Load Balancer (ALB only) with a rule
  - Make sure health checks are not redirected (so they keep giving 200 OK)

Web Server vs Worker Environment

- If your application performs tasks that are long to complete, offload these tasks to a dedicated worker environment
- Decoupling your application into two tiers is common
- Example: processing a video, generating a zip file, etc
- You can define periodic tasks in a file cron.yaml



Elastic Beanstalk – Custom Platform (Advanced)

- Custom Platforms are very advanced, they allow to define from scratch:
  - The Operating System (OS)
  - Additional Software
  - Scripts that Beanstalk runs on these platforms
- Use case: app language is incompatible with Beanstalk & doesn't use Docker
- To create your own platform:
  - Define an AMI using Platform.yaml file
  - Build that platform using the Packer software (open source tool to create AMIs)
- Custom Platform vs Custom Image (AMI):
  - Custom Image is to tweak an existing Beanstalk Platform (Python, Node.js, Java…)
  - Custom Platform is to create an entirely new Beanstalk Platform

AWS Lambda : It’s a serverless world

Why AWS Lambda

- Virtual Servers in the Cloud
- Limited by RAM and CPU
- Continuously running
- Scaling means intervention to add / remove servers

Amazon EC2

- Virtual **functions** – no servers to manage!
- Limited by time - **short executions**
- Run **on-demand**
- **Scaling is automated**!

Amazon Lambda

Benefits of AWS Lambda

- Easy Pricing:
  - Pay per request and compute time
  - Free tier of 1,000,000 AWS Lambda requests and 400,000 GBs of compute time
- Integrated with the whole AWS suite of services
- Integrated with many programming languages
- Easy monitoring through AWS CloudWatch
- Easy to get more resources per functions (up to 3GB of RAM!)
- Increasing RAM will also improve CPU and network!

AWS Lambda language support

- Node.js (JavaScript)
- Python
- Java (Java 8 compatible)
- C# (.NET Core)
- Golang
- C# / Powershell
- Ruby
- Custom Runtime API (community supported, example Rust)
- Important: Docker is not for AWS Lambda, it’s for ECS / Fargate

AWS Lambda Integrations
Main ones

API Gateway

Kinesis

DynamoDB

S3

CloudFront

CloudWatch Events
EventBridge

CloudWatch Logs

SNS

SQS

Cognito

Example: Serverless Thumbnail creation

trigger

push

New thumbnail in S3

push

Metadata in DynamoDB

Image name
Image size
Creation date
etc…

New image in S3

AWS Lambda Function
Creates a Thumbnail

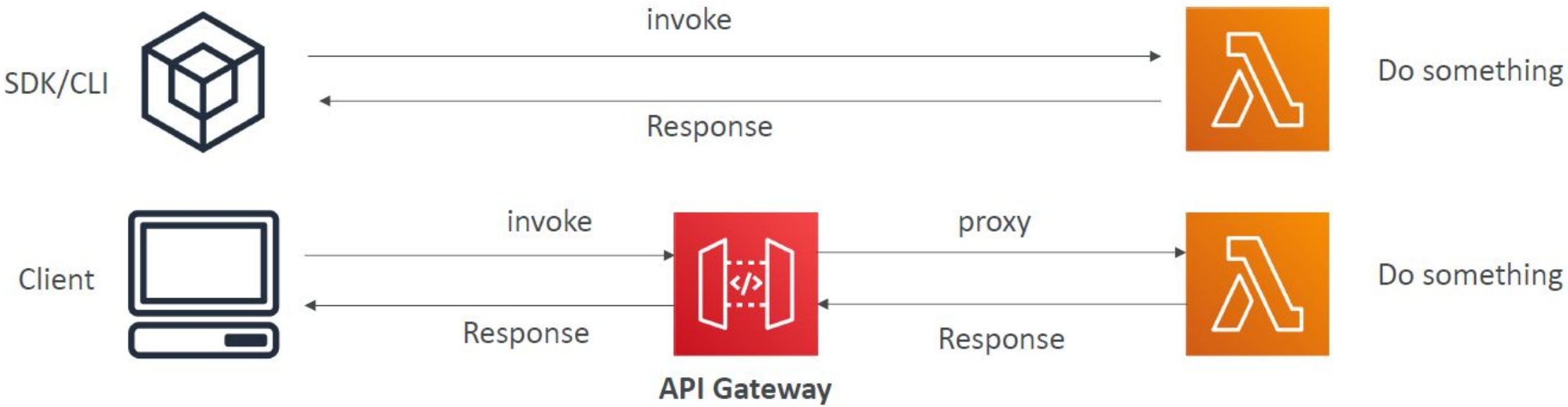
Example: Serverless CRON Job

AWS Lambda Pricing: example

- You can find overall pricing information here: https://aws.amazon.com/lambda/pricing/
- Pay per calls:
    - First 1,000,000 requests are free
    - $0.20 per 1 million requests thereafter ($0.0000002 per request)
- Pay per duration: (in increment of 100ms)
    - 400,000 GB-seconds of compute time per month if FREE
    - == 400,000 seconds if function is 1GB RAM
    - == 3,200,000 seconds if function is 128 MB RAM
- After that $1.00 for 600,000 GB-seconds
- It is usually very cheap to run AWS Lambda so it’s very popular

Lambda – Synchronous Invocations

- Synchronous: CLI, SDK, API Gateway, Application Load Balancer
- Results is returned right away
- Error handling must happen client side (retries, exponential backoff, etc…)

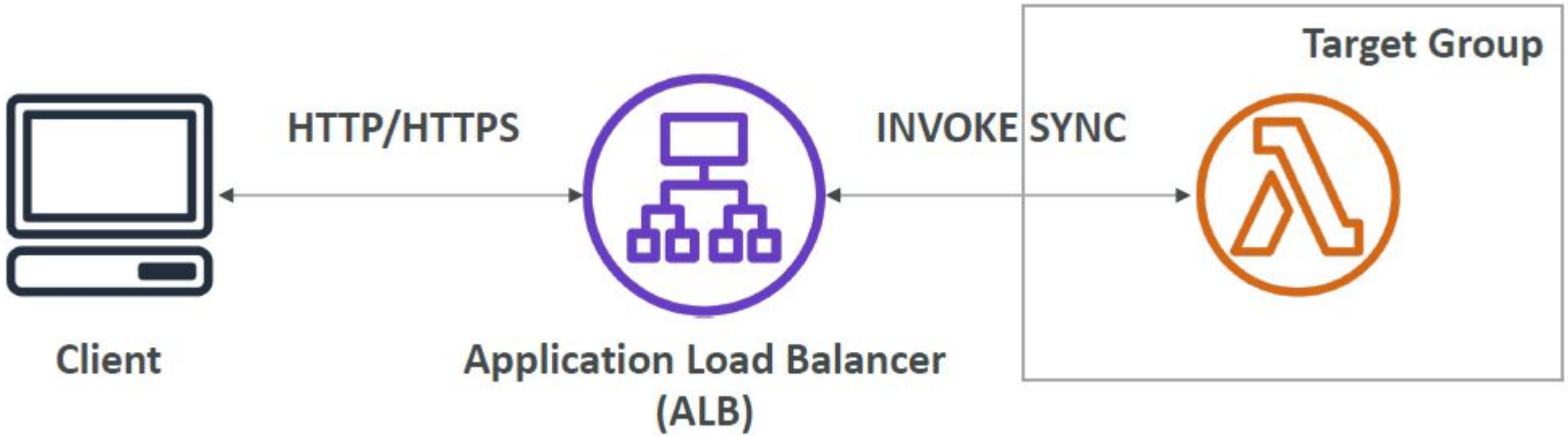


Lambda - Synchronous Invocations - Services

- User Invoked:
  - Elastic Load Balancing (Application Load Balancer)
  - Amazon API Gateway
  - Amazon CloudFront (Lambda@Edge)
  - Amazon S3 Batch
- Service Invoked:
  - Amazon Cognito
  - AWS Step Functions
- Other Services:
  - Amazon Lex
  - Amazon Alexa
  - Amazon Kinesis Data Firehose

Lambda Integration with ALB

- To expose a Lambda function as an HTTP(S) endpoint…
- You can use the Application Load Balancer (or an API Gateway)
- The Lambda function must be registered in a target group



ALB to Lambda: HTTP to JSON

Request Payload for Lambda Function

```
{
    "requestContext": {
        "elb": {
            "targetGroupArn": "arn:aws:elasticloadbalancing:us-east-2:12
            49e9d65c45c6791a"
        }
    },
    "httpMethod": "GET",
    "path": "/lambda",
    "queryStringParameters": {
        "query": "1234ABCD"
    },
    "headers": {
        "connection": "keep-alive",
        "host": "lambda-alb-123578498.us-east-2.elb.amazonaws.com",
        "user-agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWe
        Safari/537.36",
        "x-amzn-trace-id": "Root=1-5c536348-3d683b8b04734faae651f476",
        "x-forwarded-for": "72.12.164.125",
        "x-forwarded-port": "80",
        "x-forwarded-proto": "http",
    },
    "body": "",
    "isBase64Encoded": false
}
```

ELB information

HTTP Method & Path

Query String Parameters as Key/Value pairs

Headers as Key/Value pairs

Body (for POST, PUT...) & isBase64Encoded

Lambda to ALB conversions: JSON to HTTP

Response from the Lambda Function

```
{
    "statusCode": 200,
    "statusDescription": "200 OK",
    "headers": {
        "Content-Type": "text/html; charset=utf-8"
    },
    "body": "<h1>Hello world!</h1>",
    "isBase64Encoded": false
}
```

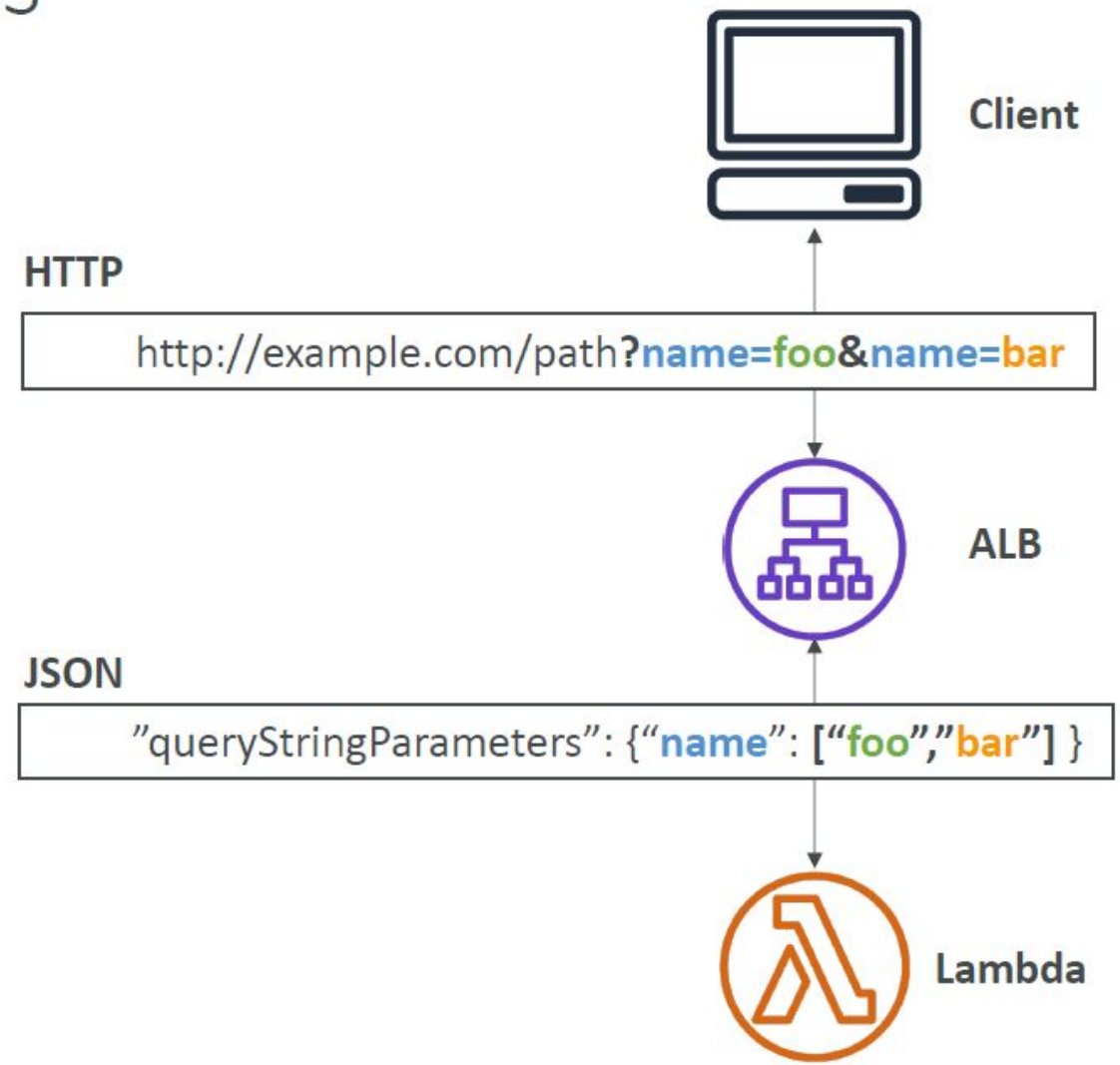
Status Code & Description

Headers as Key/Value pairs

Body & isBase64Encoded

ALB Multi-Header Values

- ALB can support multi header values (ALB setting)
- When you enable multi-value headers, **HTTP headers and query string parameters** that are sent with **multiple values** are shown **as arrays** within the AWS Lambda event and response objects.

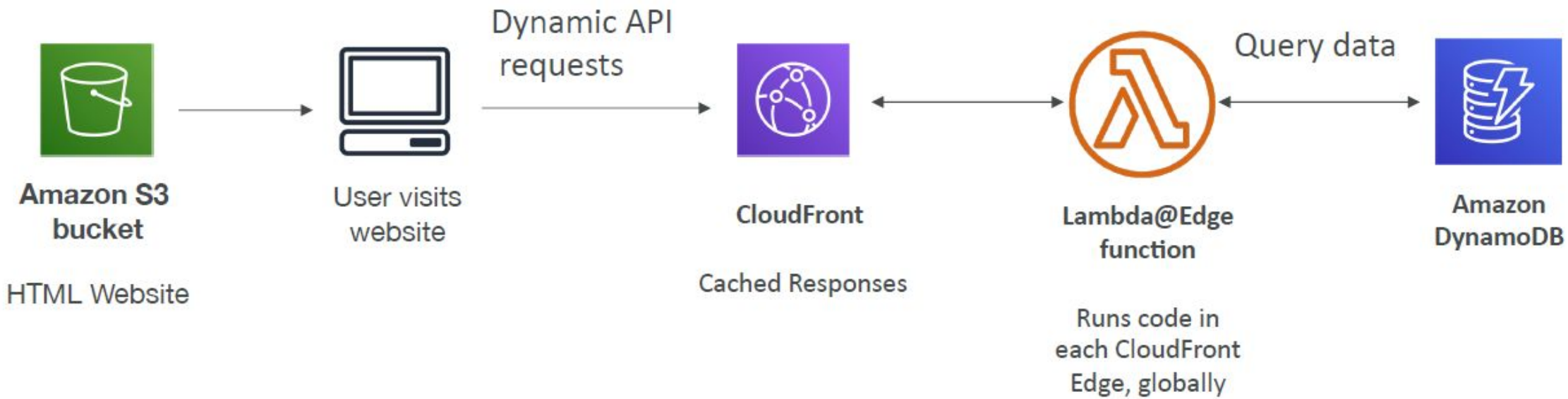


Lambda@Edge
- You have deployed a CDN using CloudFront
- What if you wanted to run a global AWS Lambda alongside?
- Or how to implement request filtering before reaching your application?
- For this, you can use Lambda@Edge: deploy Lambda functions alongside your CloudFront CDN
- Build more responsive applications
- You don’t manage servers, Lambda is deployed globally
- Customize the CDN content
- Pay only for what you use

Lambda@Edge
- You can use Lambda to change CloudFront requests and responses:
- After CloudFront receives a request from a viewer (viewer request)
- Before CloudFront forwards the request to the origin (origin request)
- After CloudFront receives the response from the origin (origin response)
- Before CloudFront forwards the response to the viewer (viewer response)

You can also generate responses to viewers without ever sending the request to the origin

Lambda@Edge: Global application

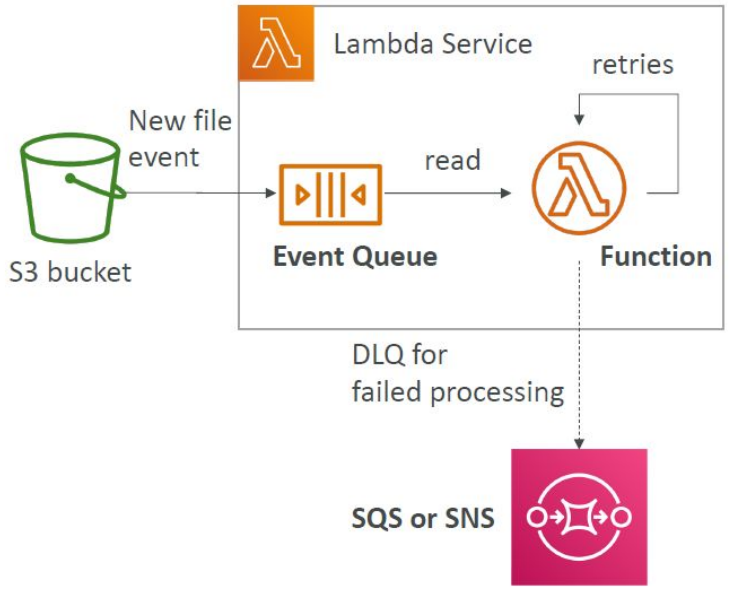


Lambda@Edge: Use Cases

- Website Security and Privacy
- Dynamic Web Application at the Edge
- Search Engine Optimization (SEO)
- Intelligently Route Across Origins and Data Centers
- Bot Mitigation at the Edge
- Real-time Image Transformation
- A/B Testing
- User Authentication and Authorization
- User Prioritization
- User Tracking and Analytics

Lambda – Asynchronous Invocations

- S3, SNS, CloudWatch Events…
- The events are placed in an Event Queue
- Lambda attempts to retry on errors
- 3 tries total
- 1 minute wait after 1st , then 2 minutes wait
- Make sure the processing is idempotent (in case of retries)
- If the function is retried, you will see duplicate logs entries in CloudWatch Logs
- Can define a DLQ (dead-letter queue) – SNS or SQS – for failed processing (need correct IAM permissions)
- Asynchronous invocations allow you to speed up the processing if you don't need to wait for the result (ex: you need 1000 files processed)



Lambda - Asynchronous Invocations - Services

- Amazon Simple Storage Service (S3)
- Amazon Simple Notification Service (SNS)
- Amazon CloudWatch Events / EventBridge
- AWS CodeCommit (CodeCommit Trigger: new branch, new tag, new push)
- AWS CodePipeline (invoke a Lambda function during the pipeline, Lambda must callback)

----- other -----

- Amazon CloudWatch Logs (log processing)
- Amazon Simple Email Service
- AWS CloudFormation
- AWS Config
- AWS IoT
- AWS IoT Events

CloudWatch Events / EventBridge

CRON or Rate EventBridge Rule

Trigger
Every 1 hour

AWS Lambda Function
Perform a task

CodePipeline EventBridge Rule

Trigger on
State Changes

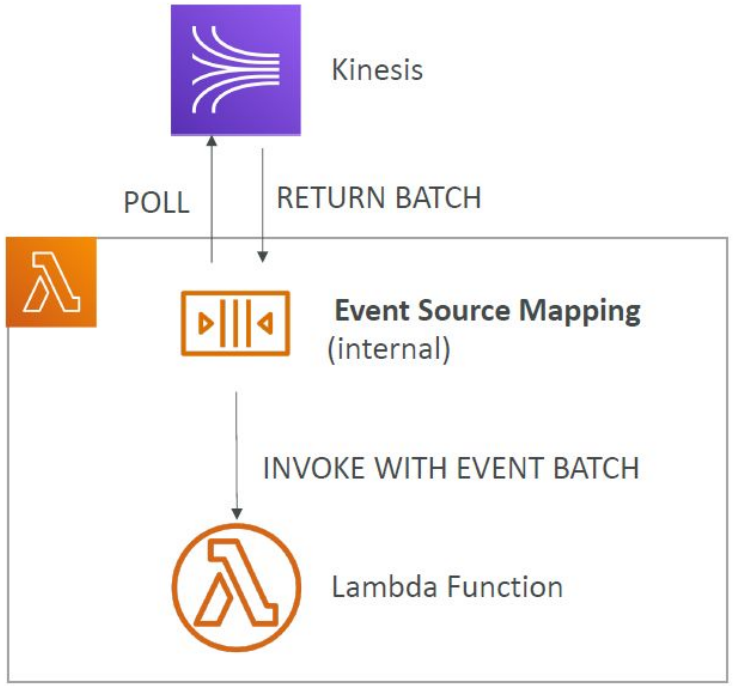
AWS Lambda Function
Perform a task

S3 Events Notifications

- S3:ObjectCreated, S3:ObjectRemoved, S3:ObjectRestore, S3:Replication…
- Object name filtering possible (*.jpg)
- Use case: generate thumbnails of images uploaded to S3
- S3 event notifications typically deliver events in seconds but can sometimes take a minute or longer
- If two writes are made to a single nonversioned object at the same time, it is possible that only a single event notification will be sent
- If you want to ensure that an event notification is sent for every successful write, you can enable versioning on your bucket.

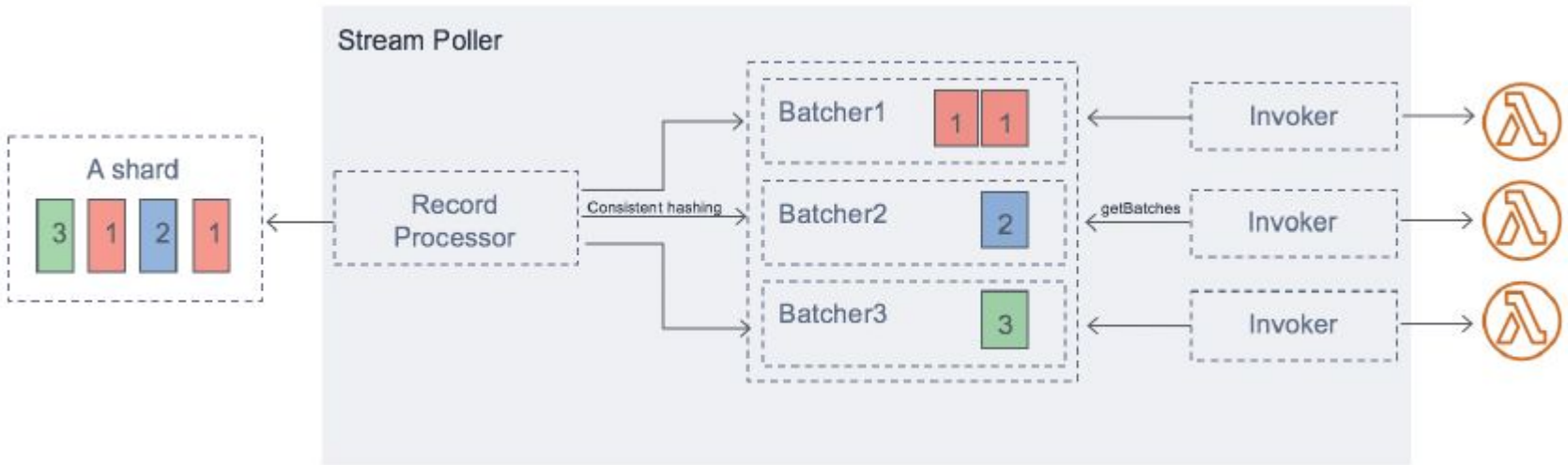
Lambda – Event Source Mapping

- Kinesis Data Streams
- SQS & SQS FIFO queue
- DynamoDB Streams
- Common denominator: records need to be polled from the source
- Your Lambda function is invoked synchronously



Streams & Lambda (Kinesis & DynamoDB)

- An event source mapping creates an iterator for each shard, processes items in order
- Start with new items, from the beginning or from timestamp
- Processed items aren't removed from the stream (other consumers can read them)
- Low traffic: use batch window to accumulate records before processing
- You can process multiple batches in parallel
- up to 10 batches per shard
- in-order processing is still guaranteed for each partition key,

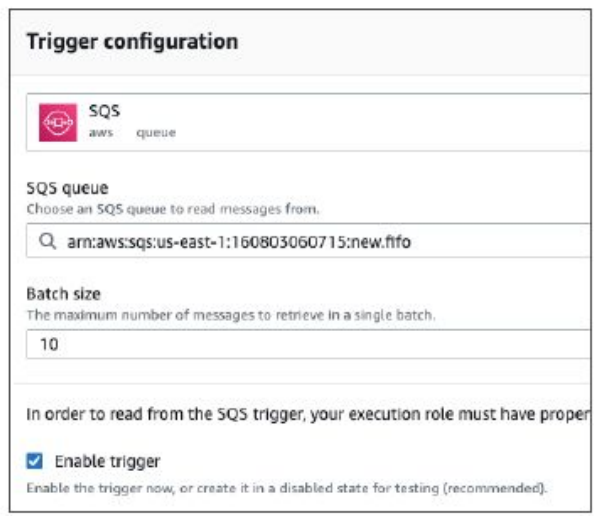


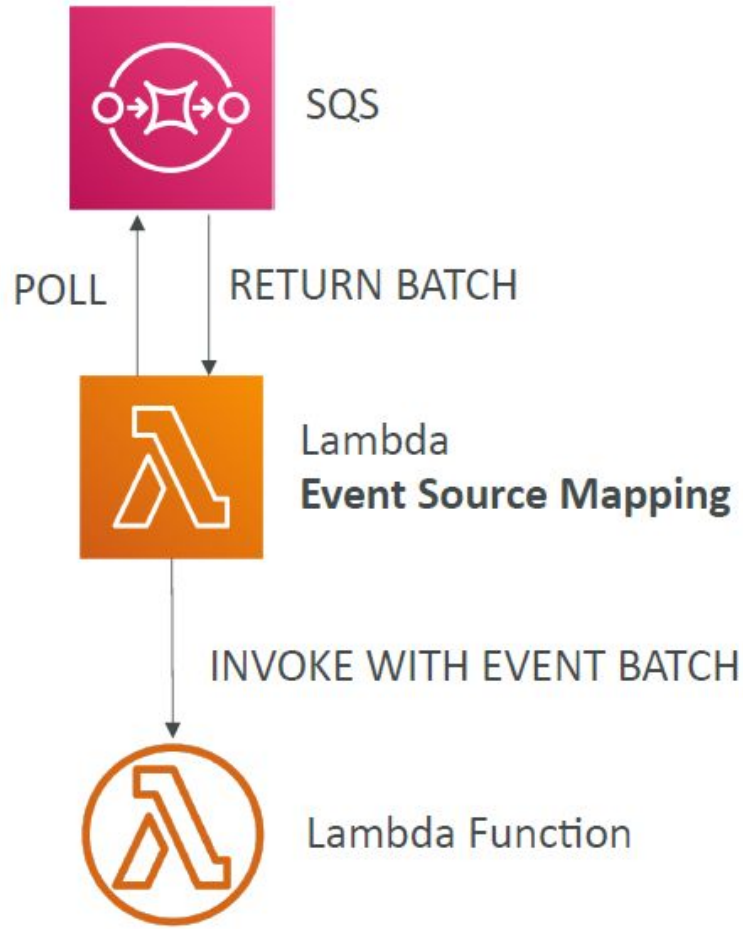
Streams & Lambda – Error Handling

- By default, if your function returns an error, the entire batch is reprocessed until the function succeeds, or the items in the batch expire.
- To ensure in-order processing, processing for the affected shard is paused until the error is resolved
- You can configure the event source mapping to:
    - discard old events
    - restrict the number of retries
    - split the batch on error (to work around Lambda timeout issues)
- Discarded events can go to a Destination

Lambda – Event Source Mapping
SQS & SQS FIFO

- Event Source Mapping will poll SQS (**Long Polling**)
- Specify **batch size** (1-10 messages)
- Recommended: Set the queue visibility timeout to 6x the timeout of your Lambda function
- To use a DLQ
  - set-up on the SQS queue, not Lambda (DLQ for Lambda is only for async invocations)
  - Or use a Lambda destination for failures





Queues & Lambda

- Lambda also supports in-order processing for FIFO (first-in, first-out) queues, **scaling up to the number of active message groups.**
- For standard queues, items aren't necessarily processed in order.
- Lambda scales up to process a standard queue as quickly as possible.

- When an error occurs, batches are returned to the queue as individual items and might be processed in a different grouping than the original batch.
- Occasionally, the event source mapping might receive the same item from the queue twice, even if no function error occurred.
- Lambda deletes items from the queue after they're processed successfully.
- You can configure the source queue to send items to a dead-letter queue if they can't be processed.

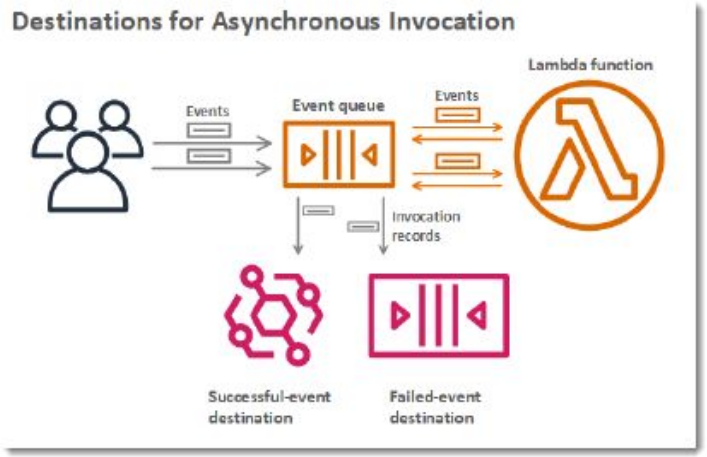
Lambda Event Mapper Scaling

- Kinesis Data Streams & DynamoDB Streams:
  - One Lambda invocation per stream shard
  - If you use parallelization, up to 10 batches processed per shard simultaneously
- SQS Standard:
  - Lambda adds 60 more instances per minute to scale up
  - Up to 1000 batches of messages processed simultaneously
- SQS FIFO:
  - Messages with the same GroupID will be processed in order
  - The Lambda function scales to the number of active message groups

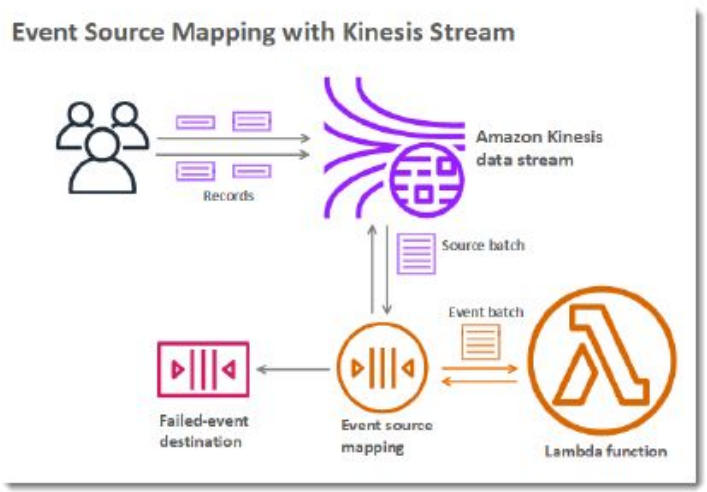
Lambda – Destinations

- Nov 2019: Can configure to send result to a **destination**
- **Asynchronous invocations** - can define destinations for successful and failed event:
  - Amazon SQS
  - Amazon SNS
  - AWS Lambda
  - Amazon EventBridge bus
- *Note: AWS recommends you use destinations instead of DLQ now (but both can be used at the same time)*

- **Event Source mapping:** for discarded event batches
  - Amazon SQS
  - Amazon SNS
- *Note: you can send events to a DLQ directly from SQS*



https://docs.aws.amazon.com/lambda/latest/dg/invocation-async.html



https://docs.aws.amazon.com/lambda/latest/dg/invocation-eventsourcemapping.html

Lambda Execution Role (IAM Role)

- Grants the Lambda function permissions to AWS services / resources
- Sample managed policies for Lambda:
  - AWSLambdaBasicExecutionRole – Upload logs to CloudWatch.
  - AWSLambdaKinesisExecutionRole – Read from Kinesis
  - AWSLambdaDynamoDBExecutionRole – Read from DynamoDB Streams
  - AWSLambdaSQSQueueExecutionRole – Read from SQS
  - AWSLambdaVPCAccessExecutionRole – Deploy Lambda function in VPC
  - AWSXRayDaemonWriteAccess – Upload trace data to X-Ray.

- **When you use an event source mapping to invoke your function, Lambda uses the execution role to read event data.**
- Best practice: create one Lambda Execution Role per function

Lambda Resource Based Policies

- Use resource-based policies to give other accounts and AWS services permission to use your Lambda resources
- Similar to S3 bucket policies for S3 bucket
- An IAM principal can access Lambda:
  - if the IAM policy attached to the principal authorizes it (e.g. user access)
  - OR if the resource-based policy authorizes (e.g. service access)

- When an AWS service like Amazon S3 calls your Lambda function, the resource-based policy gives it access.

Lambda Environment Variables

- Environment variable = key / value pair in "String" form
- Adjust the function behavior without updating code
- The environment variables are available to your code
- Lambda Service adds its own system environment variables as well

- Helpful to store secrets (encrypted by KMS)
- Secrets can be encrypted by the Lambda service key, or your own CMK

Lambda Logging & Monitoring

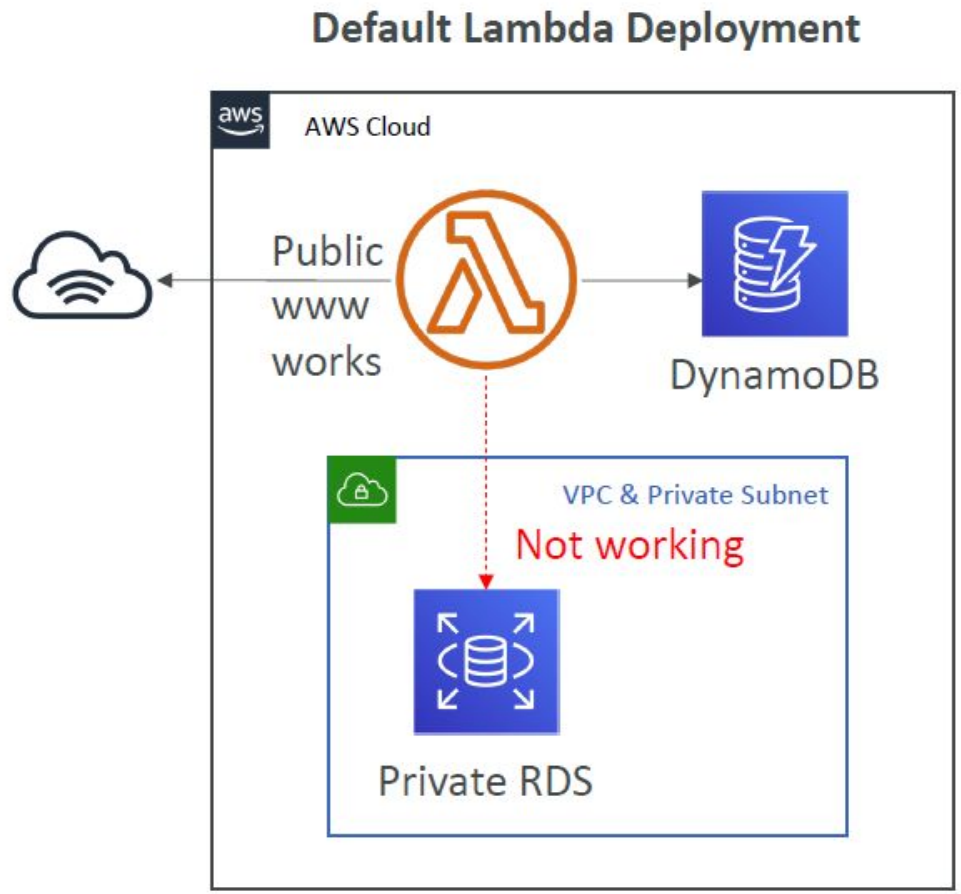
- CloudWatch Logs:
  - AWS Lambda execution logs are stored in AWS CloudWatch Logs
  - Make sure your AWS Lambda function has an execution role with an IAM policy that authorizes writes to CloudWatch Logs
- CloudWatch Metrics:
  - AWS Lambda metrics are displayed in AWS CloudWatch Metrics
  - Invocations, Durations, Concurrent Executions
  - Error count, Success Rates, Throttles
  - Async Delivery Failures
  - Iterator Age (Kinesis & DynamoDB Streams)

Lambda Tracing with X-Ray

- Enable in Lambda configuration (**Active Tracing**)
- Runs the X-Ray daemon for you
- Use AWS X-Ray SDK in Code
- Ensure Lambda Function has a correct IAM Execution Role
  - The managed policy is called AWSXRayDaemonWriteAccess
- Environment variables to communicate with X-Ray
  - **_X_AMZN_TRACE_ID**: contains the tracing header
  - **AWS_XRAY_CONTEXT_MISSING**: by default, LOG_ERROR
  - **AWS_XRAY_DAEMON_ADDRESS**: the X-Ray Daemon IP_ADDRESS:PORT

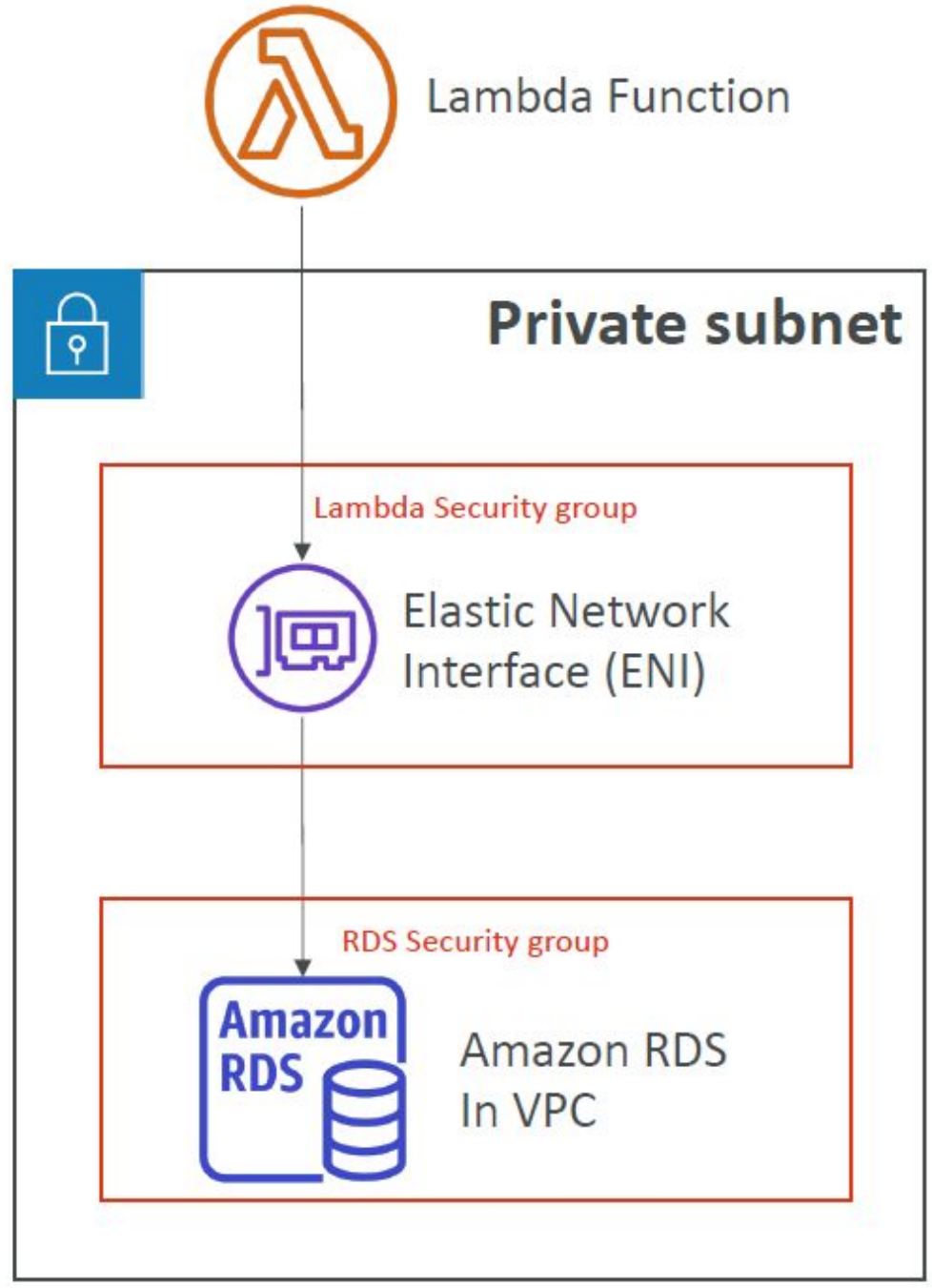
Lambda by default

- By default, your Lambda function is launched outside your own VPC (in an AWS-owned VPC)
- Therefore it cannot access resources in your VPC (RDS, ElastiCache, internal ELB…)



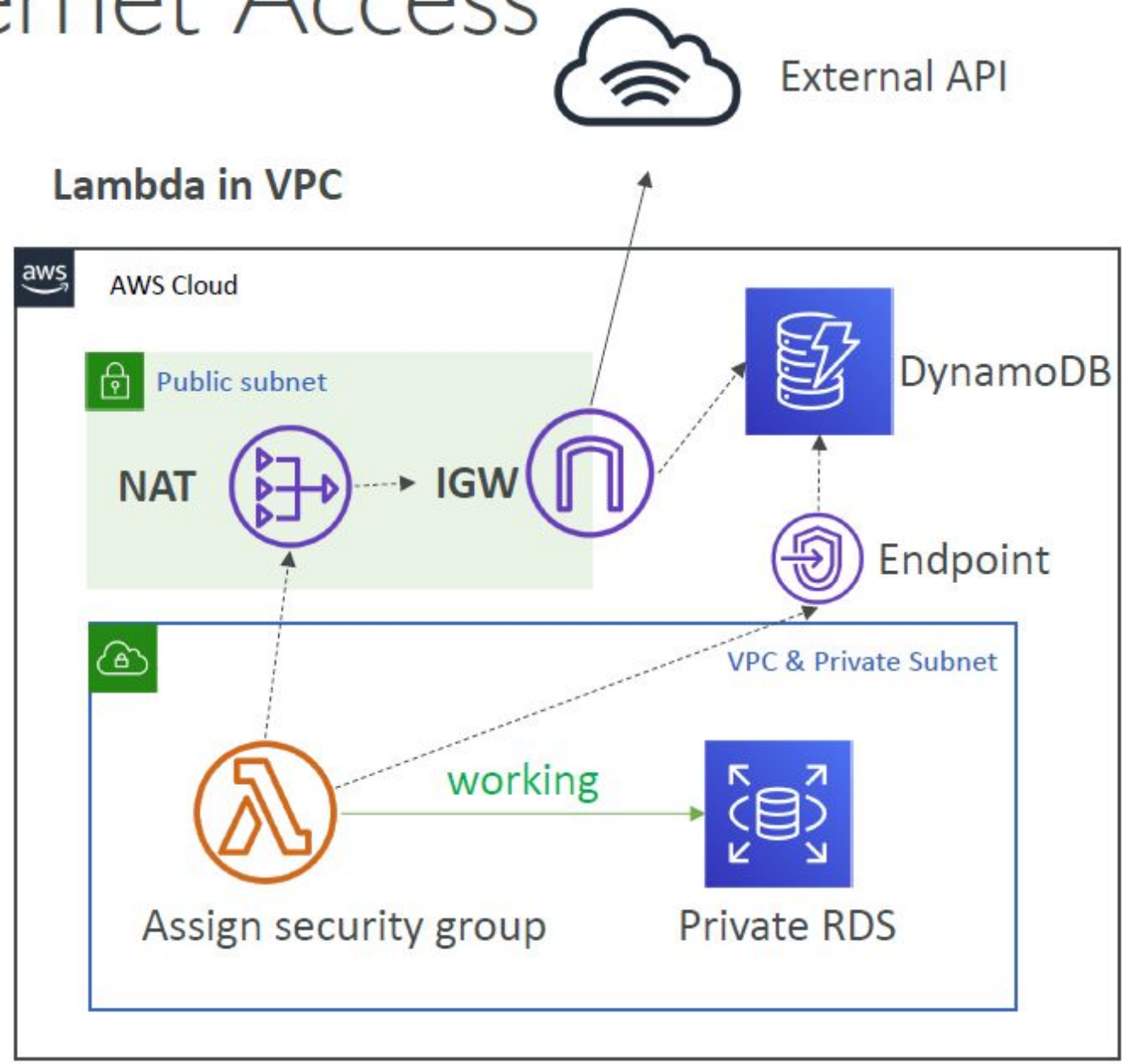
Lambda in VPC

- You must define the VPC ID, the Subnets and the Security Groups
- Lambda will create an ENI (Elastic Network Interface) in your subnets
- AWSLambdaVPCAccessExecutionRole



Lambda in VPC – Internet Access

- A Lambda function in your VPC does not have internet access
- **Deploying a Lambda function in a public subnet does not give it internet access or a public IP**
- Deploying a Lambda function in a private subnet gives it internet access if you have a **NAT Gateway / Instance**
- You can use **VPC endpoints** to privately access AWS services without a NAT



Lambda Function Configuration

- RAM:
  - From 128MB to 3,008MB in 64MB increments
  - The more RAM you add, the more vCPU credits you get
  - At 1,792 MB, a function has the equivalent of one full vCPU
  - After 1,792 MB, you get more than one CPU, and need to use multi-threading in your code to benefit from it
- If your application is CPU-bound (computation heavy), increase RAM

- Timeout: default 3 seconds, maximum is 900 seconds (15 minutes)

Lambda Execution Context

- The execution context is a temporary runtime environment that initializes any external dependencies of your lambda code
- Great for database connections, HTTP clients, SDK clients…
- The execution context is maintained for some time in anticipation of another Lambda function invocation
- The next function invocation can "re-use" the context to execution time and save time in initializing connections objects
- The execution context includes the */tmp* directory

Initialize outside the handler

BAD!

```python
import os

def get_user_handler(event, context):

    DB_URL = os.getenv("DB_URL")
    db_client = db.connect(DB_URL)
    user = db_client.get(user_id = event["user_id"])

    return user
```

The DB connection is established
At every function invocation

GOOD!

```python
import os

DB_URL = os.getenv("DB_URL")
db_client = db.connect(DB_URL)

def get_user_handler(event, context):

    user = db_client.get(user_id = event["user_id"])

    return user
```

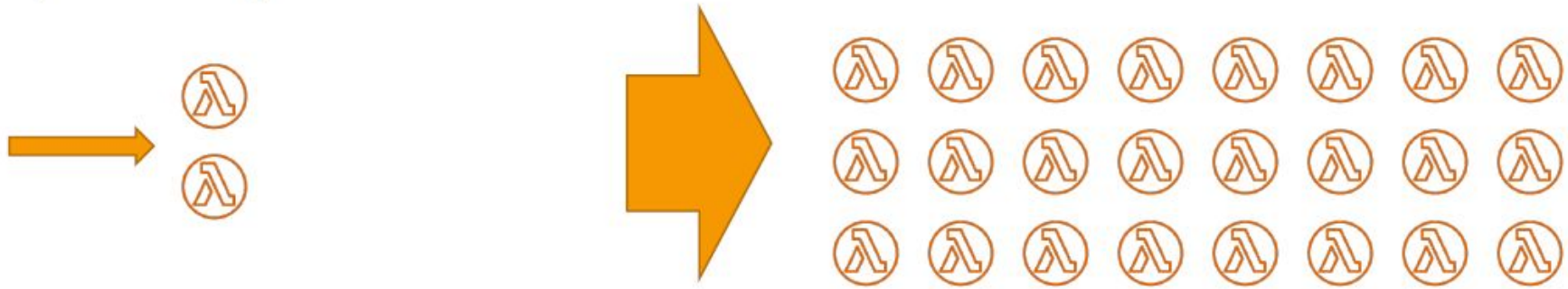
The DB connection is established once
And re-used across invocations

Lambda Functions /tmp space

- If your Lambda function needs to download a big file to work…
- If your Lambda function needs disk space to perform operations…
- You can use the /tmp directory
- Max size is 512MB
- The directory content remains when the execution context is frozen, providing transient cache that can be used for multiple invocations
(helpful to checkpoint your work)
- For permanent persistence of object (non temporary), use S3

Lambda Concurrency and Throttling

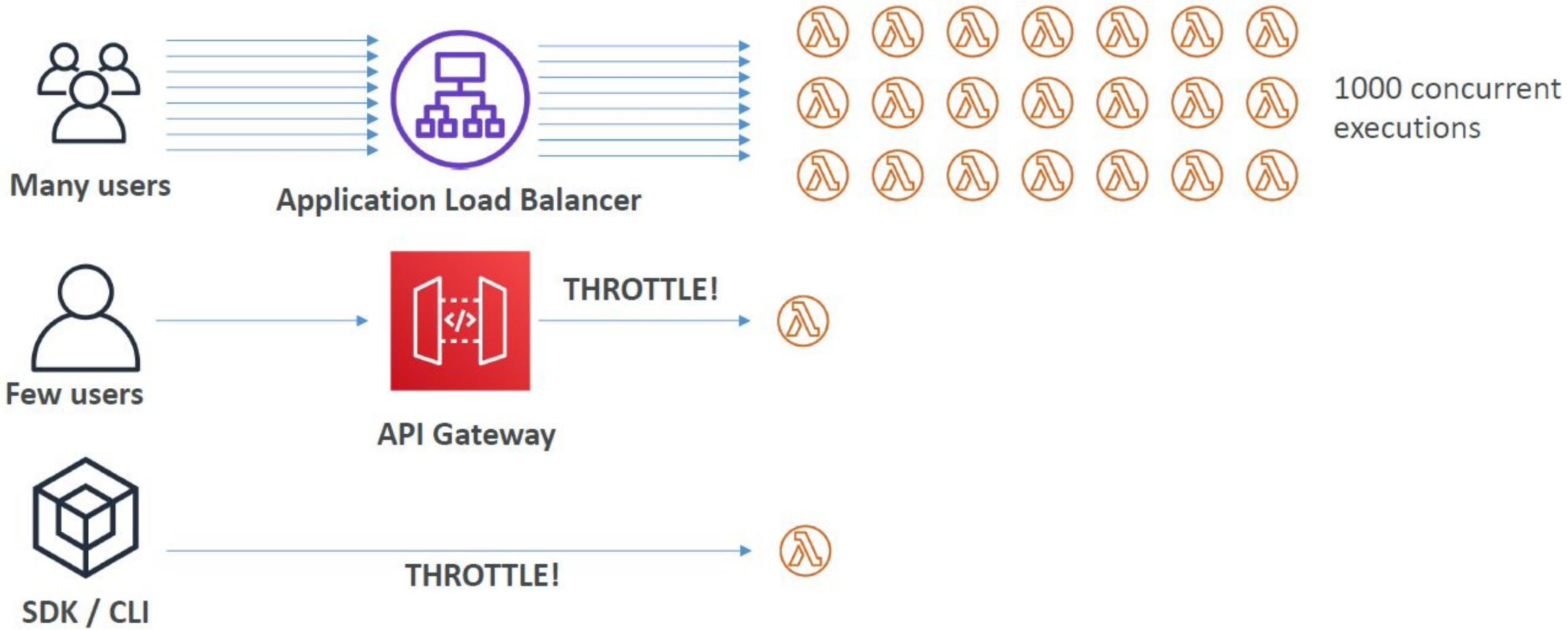
- Concurrency limit: up to 1000 concurrent executions



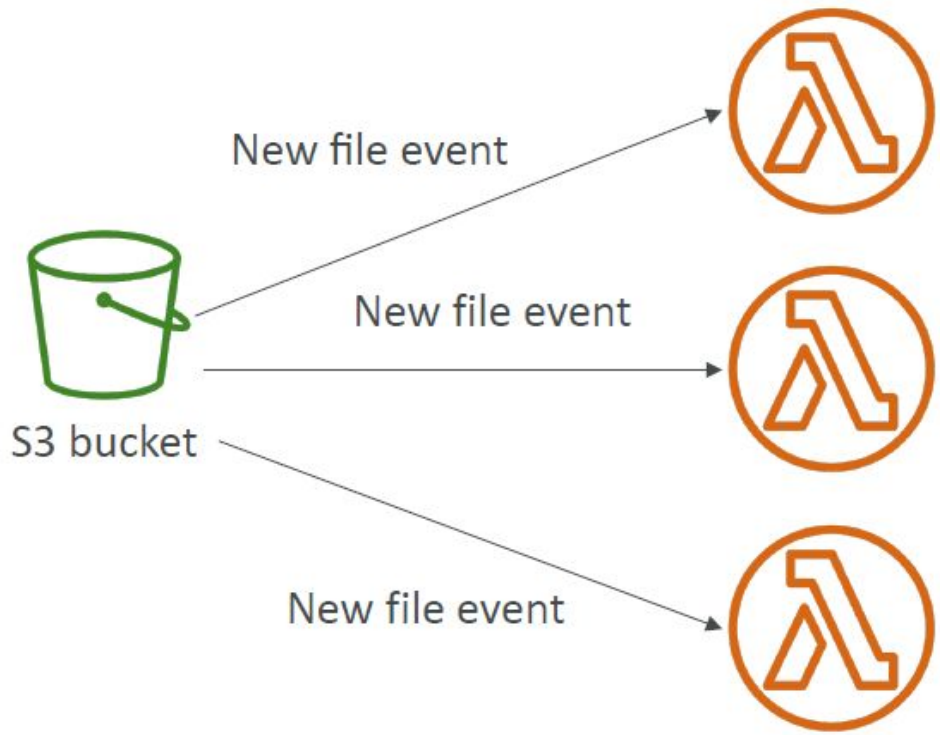
- Can set a "**reserved concurrency**" at the function level (=limit)
- Each invocation over the concurrency limit will trigger a "Throttle"
- Throttle behavior:
  - If synchronous invocation => return ThrottleError - 429
  - If asynchronous invocation => retry automatically and then go to DLQ
- If you need a higher limit, open a support ticket

Lambda Concurrency Issue

- If you don't reserve (=limit) concurrency, the following can happen:



Concurrency and Asynchronous Invocations



- If the function doesn't have enough concurrency available to process all events, additional requests are throttled.
- For throttling errors (429) and system errors (500-series), Lambda returns the event to the queue and attempts to run the function again for up to 6 hours.
- The retry interval increases exponentially from 1 second after the first attempt to a maximum of 5 minutes.

Cold Starts & Provisioned Concurrency

- Cold Start:
  - New instance => code is loaded and code outside the handler run (init)
  - If the init is large (code, dependencies, SDK…) this process can take some time.
  - First request served by new instances has higher latency than the rest
- Provisioned Concurrency:
  - Concurrency is allocated before the function is invoked (in advance)
  - So the cold start never happens and all invocations have low latency
  - Application Auto Scaling can manage concurrency (schedule or target utilization)

Lambda Function Dependencies

- If your Lambda function depends on external libraries:
  for example AWS X-Ray SDK, Database Clients, etc…
- You need to install the packages alongside your code and zip it together
  - For Node.js, use npm & "node_modules" directory
  - For Python, use pip --target options
  - For Java, include the relevant .jar files
- Upload the zip straight to Lambda if less than 50MB, else to S3 first
- Native libraries work: they need to be compiled on Amazon Linux
- AWS SDK comes by default with every Lambda function

Lambda and CloudFormation – inline

```
AWSTemplateFormatVersion: '2010-09-09'
Description: Lambda function inline
Resources:
  primer:
    Type: AWS::Lambda::Function
    Properties:
      Runtime: python3.x
      Role: arn:aws:iam::123456789012:role/lambda-role
      Handler: index.handler
      Code:
        ZipFile: |
          import os

          DB_URL = os.getenv("DB_URL")
          db_client = db.connect(DB_URL)
          def handler(event, context):
              user = db_client.get(user_id = event["user_id"])
              return user
```

- Inline functions are very simple
- Use the **Code.ZipFile** property
- You cannot include function dependencies with inline functions

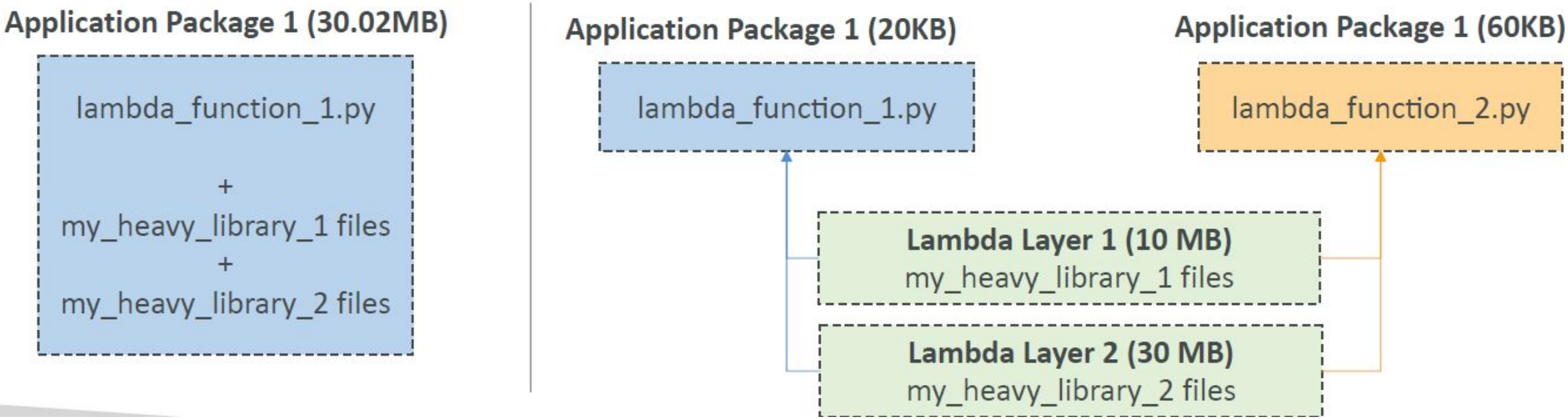
Lambda and CloudFormation – through S3

```yaml
AWSTemplateFormatVersion: '2010-09-09'
Description: Lambda from S3
Resources:
  Function:
    Type: AWS::Lambda::Function
    Properties:
      Handler: index.handler
      Role: arn:aws:iam::123456789012:role/lambda-role
      Code:
        S3Bucket: my-bucket
        S3Key: function.zip
        S3ObjectVersion: String
      Runtime: nodejs12.x
```

- You must store the Lambda zip in S3
- You must refer the S3 zip location in the CloudFormation code
  - S3Bucket
  - S3Key: full path to zip
  - S3ObjectVersion: if versioned bucket
- If you update the code in S3, but don't update S3Bucket, S3Key or S3ObjectVersion, CloudFormation won't update your function

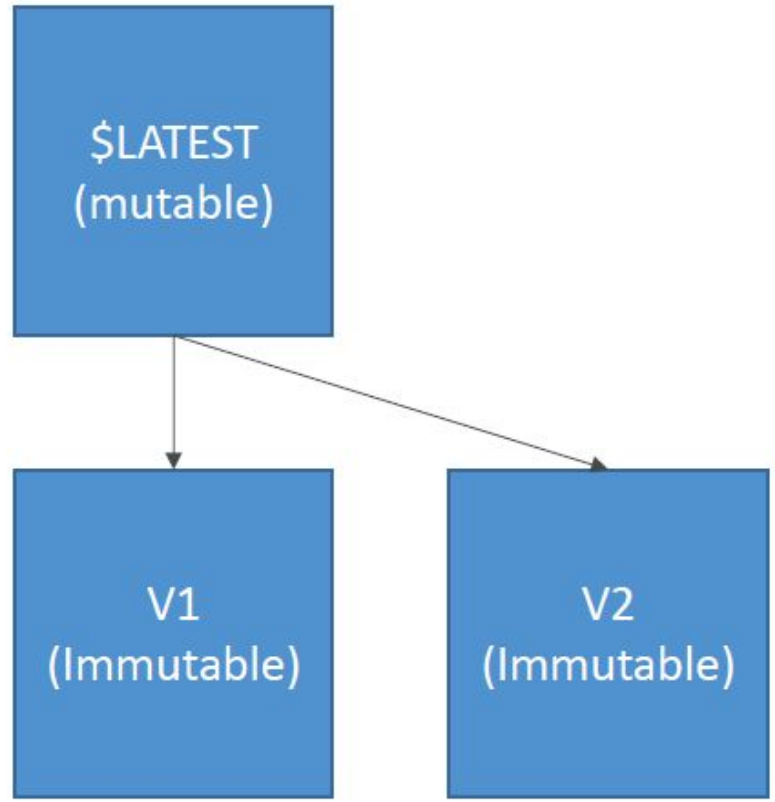
Lambda Layers

- Custom Runtimes
  - Ex: C++ https://github.com/awslabs/aws-lambda-cpp
  - Ex: Rust https://github.com/awslabs/aws-lambda-rust-runtime
- Externalize Dependencies to re-use them:



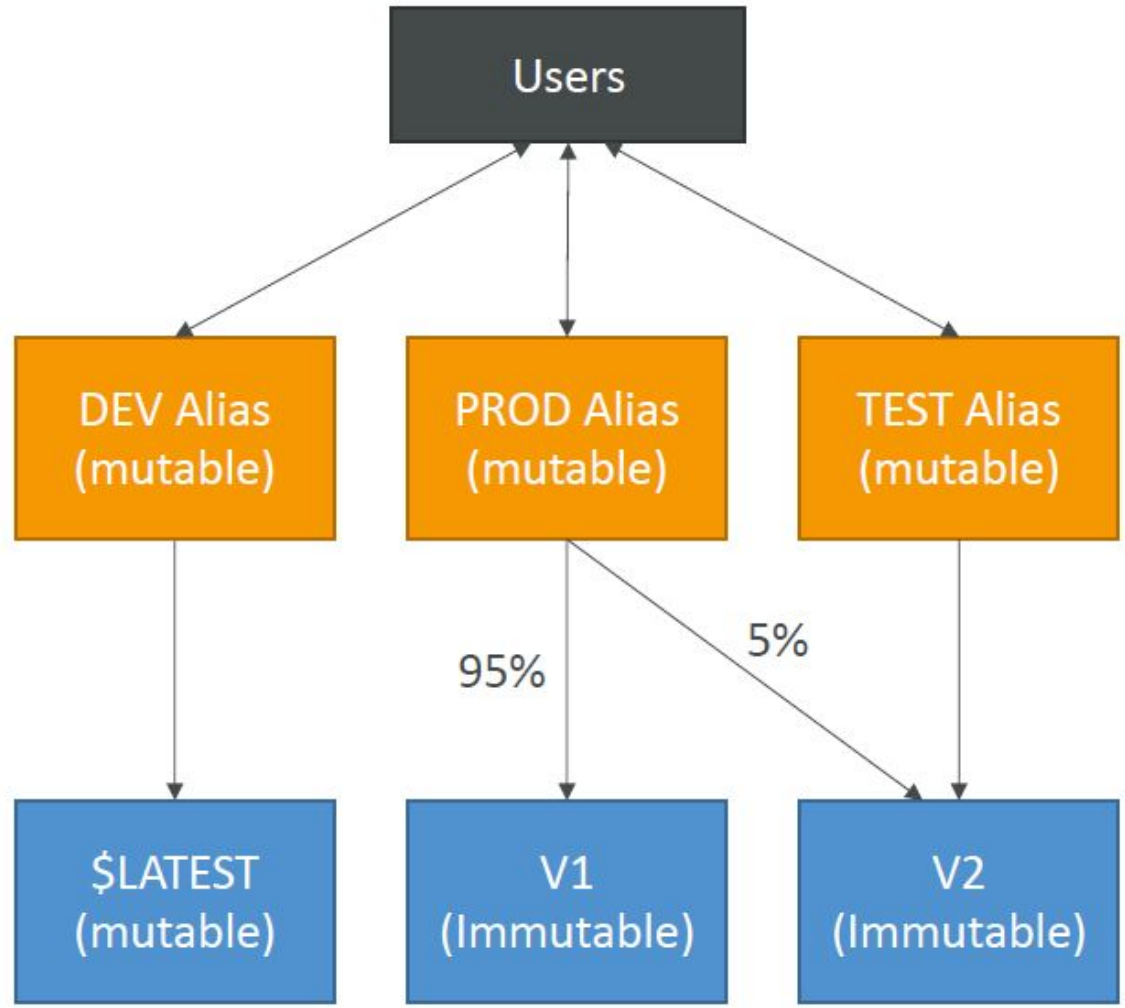
AWS Lambda Versions

- When you work on a Lambda function, we work on **$LATEST**
- When we're ready to publish a Lambda function, we create a version
- Versions are immutable
- Versions have increasing version numbers
- Versions get their own ARN (Amazon Resource Name)
- Version = code + configuration (nothing can be changed - immutable)
- Each version of the lambda function can be accessed



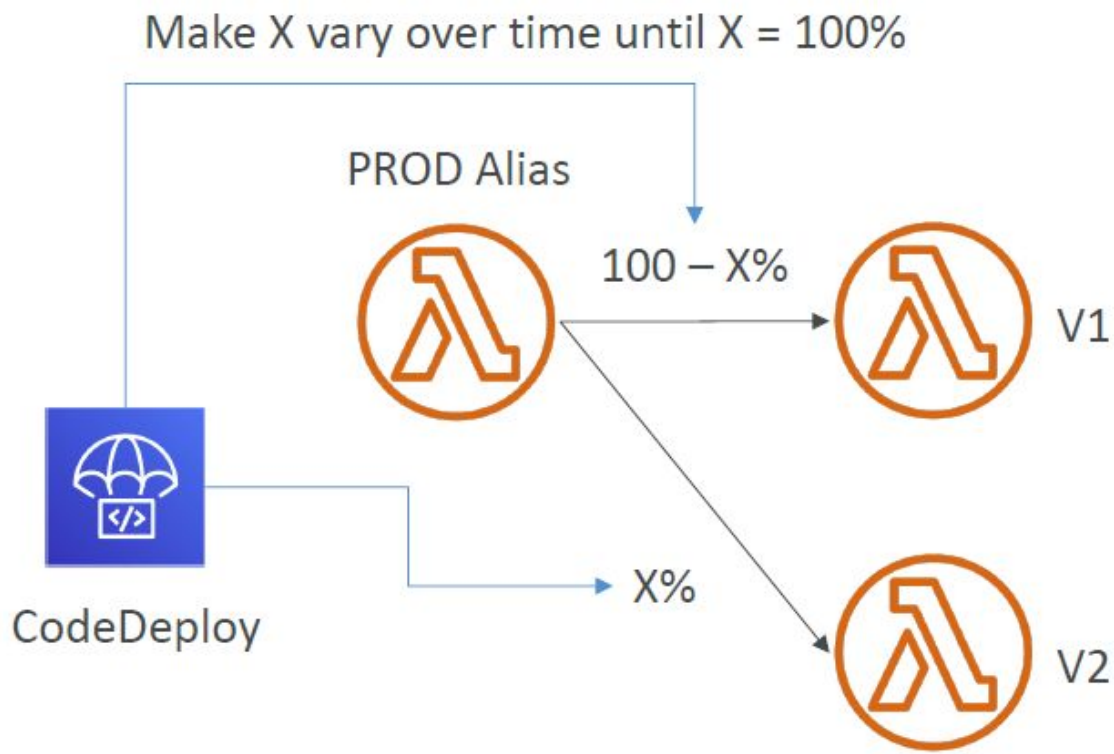
AWS Lambda Aliases

- Aliases are "pointers" to Lambda function versions
- We can define a "dev", "test", "prod" aliases and have them point at different lambda versions
- Aliases are mutable
- Aliases enable Blue / Green deployment by assigning weights to lambda functions
- Aliases enable stable configuration of our event triggers / destinations
- Aliases have their own ARNs
- **Aliases cannot reference aliases**



Lambda & CodeDeploy

- **CodeDeploy** can help you automate traffic shift for Lambda aliases
- Feature is integrated within the SAM framework
- **Linear:** grow traffic every N minutes until 100%
  - Linear10PercentEvery3Minutes
  - Linear10PercentEvery10Minutes
- **Canary:** try X percent then 100%
  - Canary10Percent5Minutes
  - Canary10Percent30Minutes
- **AllAtOnce:** immediate
- Can create Pre & Post Traffic hooks to check the health of the Lambda function



AWS Lambda Limits to Know - per region

- Execution:
  - Memory allocation: 128 MB – 3008 MB (64 MB increments)
  - Maximum execution time: 900 seconds (15 minutes)
  - Environment variables (4 KB)
  - Disk capacity in the "function container" (in /tmp): 512 MB
  - Concurrency executions: 1000 (can be increased)
- Deployment:
  - Lambda function deployment size (compressed .zip): 50 MB
  - Size of uncompressed deployment (code + dependencies): 250 MB
  - Can use the /tmp directory to load other files at startup
  - Size of environment variables: 4 KB

AWS Lambda Best Practices

- Perform heavy-duty work outside of your function handler
  - Connect to databases outside of your function handler
  - Initialize the AWS SDK outside of your function handler
  - Pull in dependencies or datasets outside of your function handler
- Use environment variables for:
  - Database Connection Strings, S3 bucket, etc… don't put these values in your code
  - Passwords, sensitive values… they can be encrypted using KMS
- Minimize your deployment package size to its runtime necessities.
  - Break down the function if need be
  - Remember the AWS Lambda limits
  - Use Layers where necessary
- Avoid using recursive code, never have a Lambda function call itself