Domain 2 (IAC & Configuration Management)

IAC

- Infrastructure as code
  - No resources are manually created, which is excellent for control
  - The code can be version controlled for example using git
  - Changes to the infrastructure are reviewed through code

- Cost
  - Each resources within the stack is stagged with an identifier so you can easily see how much a stack costs you
  - You can estimate the costs of your resources using the CloudFormation template
  - Savings strategy: In Dev, you could automation deletion of templates at 5 PM and recreated at 8 AM, safely

- Productivity
  - Ability to destroy and re-create an infrastructure on the cloud on the fly
  - Automated generation of Diagram for your templates!
  - Declarative programming (no need to figure out ordering and orchestration)

- Separation of concern: create many stacks for many apps, and many layers. Ex:
  - VPC stacks
  - Network stacks
  - App stacks

- Templates have to be uploaded in S3 and then referenced in CloudFormation
- To update a template, we can't edit previous ones. We have to re-upload a new version of the template to AWS
- Stacks are identified by a name
- Deleting a stack deletes every single artifact that was created by CloudFormation.

- Manual way:
  - Editing templates in the CloudFormation Designer
  - Using the console to input parameters, etc

- Automated way:
  - Editing templates in a YAML file
  - Using the AWS CLI (Command Line Interface) to deploy the templates
  - Recommended way when you fully want to automate your flow

Templates components (one course section for each):

1. **Resources: your AWS resources declared in the template (MANDATORY)**
2. Parameters: the dynamic inputs for your template
3. Mappings: the static variables for your template
4. Outputs: References to what has been created
5. Conditionals: List of conditions to perform resource creation
6. Metadata

Templates helpers:

1. References
2. Functions

Parameters Settings

Parameters can be controlled by all these settings:

- Type:
  - String
  - Number
  - CommaDelimitedList
  - List<Type>
  - AWS Parameter (to help catch invalid values – match against existing values in the AWS Account)
- Description
- Constraints
- ConstraintDescription (String)
- Min/MaxLength
- Min/MaxValue
- Defaults
- AllowedValues (array)
- AllowedPattern (regexp)
- NoEcho (Boolean)

How to Reference a Parameter

- The Fn::Ref function can be leveraged to reference parameters
- Parameters can be used anywhere in a template.
- The shorthand for this in YAML is !Ref

```yaml
DbSubnet1:
  Type: AWS::EC2::Subnet
  Properties:
    VpcId: !Ref MyVPC
```

!Ref : to refer Parameters as well as AWS Resources,

Concept: Pseudo Parameters

- AWS offers us pseudo parameters in any CloudFormation template.
- These can be used at any time and are enabled by default

Reference Value	Example Return Value
AWS::AccountId	1234567890
AWS::NotificationARNs	[arn:aws:sns:us-east-1:123456789012:MyTopic]
AWS::NoValue	Does not return a value.
AWS::Region	us-east-2
AWS::StackId	arn:aws:cloudformation:us-east-1:123456789012:stack/MyStack/1c2fa620-982a-11e3-aff7-50e2416294e0
AWS::StackName	MyStack

What are resources?

- Resources are the core of your CloudFormation template (MANDATORY)
- They represent the different AWS Components that will be created and configured
- Resources are declared and can reference each other

- AWS figures out creation, updates and deletes of resources for us
- There are over 224 types of resources (!)
- Resource types identifiers are of the form:

```
AWS::aws-product-name::data-type-name
```

https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-template-resource-type-ref.html

- Can I create a dynamic amount of resources?
  - ➢No, you can't. Everything in the CloudFormation template has to be declared. You can't perform code generation there

- Is every AWS Service supported?
  - ➢Almost. Only a select few niches are not there yet
  - ➢You can work around that using AWS Lambda Custom Resources

What are mappings?

- Mappings are fixed variables within your CloudFormation Template.
- They're very handy to differentiate between different environments (dev vs prod), regions (AWS regions), AMI types, etc
- All the values are hardcoded within the template
- Example:

```
Mappings:
  Mapping01:
    Key01:
      Name: Value01
    Key02:
      Name: Value02
    Key03:
      Name: Value03
```

```
RegionMap:
  us-east-1:
    "32": "ami-6411e20d"
    "64": "ami-7a11e213"
  us-west-1:
    "32": "ami-c9c7978c"
    "64": "ami-cfc7978a"
  eu-west-1:
    "32": "ami-37c2f643"
    "64": "ami-31c2f645"
```

Fn::FindInMap
Accessing Mapping Values

- We use **`Fn::FindInMap`** to return a named value from a specific key
- **`!FindInMap [ MapName, TopLevelKey, SecondLevelKey ]`**

ImageId: !FindInMap [RegionMap, !Ref “AWS::Region”, 32]

What are outputs?

- The Outputs section declares *optional* outputs values that we can import into other stacks (if you export them first)!
- You can also view the outputs in the AWS Console or in using the AWS CLI
- They're very useful for example if you define a network CloudFormation, and output the variables such as VPC ID and your Subnet IDs
- It's the best way to perform some collaboration cross stack, as you let expert handle their own part of the stack
- You can't delete a CloudFormation Stack if its outputs are being referenced by another CloudFormation stack

```yaml
Outputs:
  StackSSHSecurityGroup:
    Description: The SSH Security Group for our Company
    Value: !Ref MyCompanyWideSSHSecurityGroup
    Export:
      Name: SSHSecurityGroup
```

Cross Stack Reference

- We then create a second template that leverages that security group
- For this, we use the **`Fn::ImportValue`** function
- You can't delete the underlying stack until all the references are deleted too.

```yaml
Resources:
  MySecureInstance:
    Type: AWS::EC2::Instance
    Properties:
      AvailabilityZone: us-east-1a
      ImageId: ami-a4c7edb2
      InstanceType: t2.micro
      SecurityGroups:
        - !ImportValue SSHSecurityGroup
```

How to define a condition?

```yaml
Conditions:
  CreateProdResources: !Equals [ !Ref EnvType, prod ]
```

- The logical ID is for you to choose. It's how you name condition
- The intrinsic function (logical) can be any of the following:
  - `Fn::And`
  - `Fn::Equals`
  - `Fn::If`
  - `Fn::Not`
  - `Fn::Or`

- Conditions can be applied to resources / outputs / etc…

```yaml
Resources:
  MountPoint:
    Type: "AWS::EC2::VolumeAttachment"
    Condition: CreateProdResources
```

Must Know Intrisic Functions

- Ref
- Fn::GetAtt
- Fn::FindInMap
- Fn::ImportValue
- Fn::Join
- Fn::Sub
- Condition Functions (Fn::If, Fn::Not, Fn::Equals, etc…)

Fn::GetAtt

- Attributes are attached to any resources you create
- To know the attributes of your resources, the best place to look at is the documentation.
- For example: the AZ of an EC2 machine!

```yaml
Resources:
  EC2Instance:
    Type: "AWS::EC2::Instance"
    Properties:
      ImageId: ami-1234567
      InstanceType: t2.micro
```

```yaml
NewVolume:
  Type: "AWS::EC2::Volume"
  Condition: CreateProdResources
  Properties:
    Size: 100
    AvailabilityZone:
      !GetAtt EC2Instance.AvailabilityZone
```

Fn::Join

- Join values with a delimiter

```
!Join [ delimiter, [ comma-delimited list of values ] ]
```

- This creates "a:b:c"

```
!Join [ ":", [ a, b, c ] ]
```

Function Fn::Sub

- Fn::Sub, or !Sub as a shorthand, is used to substitute variables from a text. It's a very handy function that will allow you to fully customize your templates.
- For example, you can combine Fn::Sub with References or AWS Pseudo variables!
- String must contain ${VariableName} and will substitute them

```
!Sub
  - String
  - { Var1Name: Var1Value, Var2Name: Var2Value }
```

```
!Sub String
```

User Data in EC2 for CloudFormation

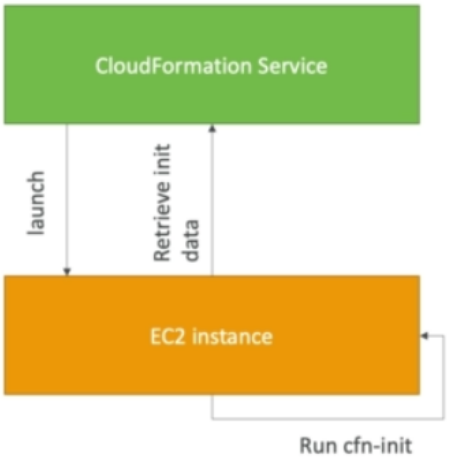
- We can have user data at EC2 instance launch through the console
- We can also include it in CloudFormation

- The important thing to pass is the entire script through the function Fn::Base64
- Good to know: user data script log is in /var/log/cloud-init-output.log

```
Resources:
  MyInstance:
    Type: AWS::EC2::Instance
    Properties:
      AvailabilityZone: us-east-1a
      ImageId: ami-009d6802948d06e52
      InstanceType: t2.micro
      KeyName: !Ref SSHKey
      SecurityGroups:
        - !Ref SSHSecurityGroup
      # we install our web server with user data
      UserData:
        Fn::Base64:
          #!/bin/bash -xe
          yum update -y
          yum install -y httpd
          systemctl start httpd
          systemctl enable httpd
          echo "Hello World from user data" > /var/www/html/index.html
```

cfn-init

- AWS::CloudFormation::Init must be in the Metadata of a resource
- With the cfn-init script, it helps make complex EC2 configurations readable
- The EC2 instance will query the CloudFormation service to get init data
- Logs go to /var/log/cfn-init.log



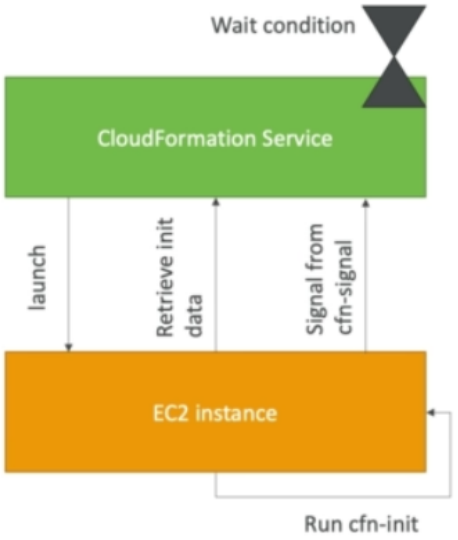

```yaml
Resources:
  MyInstance:
    Type: AWS::EC2::Instance
    Properties:
      AvailabilityZone: us-east-1a
      ImageId: ami-009d6802948d06e52
      InstanceType: t2.micro
      KeyName: !Ref SSHKey
      SecurityGroups:
        - !Ref SSHSecurityGroup
      # we install our web server with user data
      UserData:
        Fn::Base64:
```

```
          !Sub |
            #!/bin/bash -xe
            # Get the latest CloudFormation package
            yum update -y aws-cfn-bootstrap
            # Start cfn-init
            /opt/aws/bin/cfn-init -s ${AWS::StackId} -r MyInstance --region
${AWS::Region} || error_exit 'Failed to run cfn-init'
    Metadata:
      Comment: Install a simple Apache HTTP page
      AWS::CloudFormation::Init:
        config:
          packages:
            yum:
              httpd: []
          files:
            "/var/www/html/index.html":
              content: |
                <h1>Hello World from EC2 instance!</h1>
                <p>This was created using cfn-init</p>
              mode: '000644'
          commands:
            hello:
              command: "echo 'hello world'"
          services:
            sysvinit:
              httpd:
                enabled: 'true'
                ensureRunning: 'true'
```

Logs will be present in : /var/logs/cfn-init.log

cfn-signal & wait conditions

- We still don't know how to tell CloudFormation that the EC2 instance got properly configured after a **cfn-init**
- For this, we can use the **cfn-signal** script!
  - We run cfn-signal right after cfn-init
  - Tell CloudFormation service to keep on going or fail
- We need to define **WaitCondition**:
  - Block the template until it receives a signal from cfn-signal
  - We attach a **CreationPolicy** (also works on EC2, ASG)




```
UserData:
        Fn::Base64:
          !Sub |
            #!/bin/bash -xe
            # Get the latest CloudFormation package
            yum update -y aws-cfn-bootstrap
            # Start cfn-init
            /opt/aws/bin/cfn-init -s ${AWS::StackId} -r MyInstance --region
${AWS::Region}
            # Start cfn-signal to the wait condition
            /opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackId} --resource
SampleWaitCondition --region ${AWS::Region}
```

```
SampleWaitCondition:
    CreationPolicy:
      ResourceSignal:
        Timeout: PT2M
        Count: 1
    Type: AWS::CloudFormation::WaitCondition
```

Wait Condition Didn't Receive the Required Number of Signals from an Amazon EC2 Instance

- Ensure that the AMI you're using has the AWS CloudFormation helper scripts installed. If the AMI doesn't include the helper scripts, you can also download them to your instance.
- Verify that the cfn-init & cfn-signal command was successfully run on the instance. You can view logs, such as /var/log/cloud-init.log or /var/log/cfn-init.log, to help you debug the instance launch.
- You can retrieve the logs by logging in to your instance, but you must disable rollback on failure or else AWS CloudFormation deletes the instance after your stack fails to create.
- Verify that the instance has a connection to the Internet. If the instance is in a VPC, the instance should be able to connect to the Internet through a NAT device if it's is in a private subnet or through an Internet gateway if it's in a public subnet.

Rollbacks on failures

- Stack Creation Fails: (CreateStack API)
  - Default: everything rolls back (gets deleted). We can look at the log OnFailure=ROLLBACK
  - Troubleshoot: Option to disable rollback and manually troubleshoot OnFailure=DO_NOTHING
  - Delete: get rid of the stack entirely, do not keep anything OnFailure=DELETE

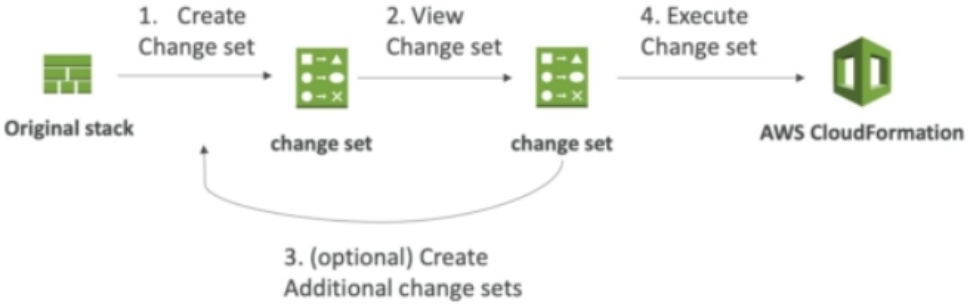
- Stack Update Fails: (UpdateStack API)
  - The stack automatically rolls back to the previous known working state
  - Ability to see in the log what happened and error messages

Nested stacks

- Nested stacks are stacks as part of other stacks
- They allow you to isolate repeated patterns / common components in separate stacks and call them from other stacks
- Example:
  - Load Balancer configuration that is re-used
  - Security Group that is re-used
- Nested stacks are considered best practice
- To update a nested stack, always update the parent (root stack)

ChangeSets

- When you update a stack, you need to know what changes before it happens for greater confidence
- ChangeSets won't say if the update will be successful



Retaining Data on Deletes

- You can put a DeletionPolicy on any resource to control what happens when the CloudFormation template is deleted
- DeletionPolicy=Retain:
  - Specify on resources to preserve / backup in case of CloudFormation deletes
  - To keep a resource, specify **Retain** (works for any resource / nested stack)
- DeletionPolicy=Snapshot:
  - EBS Volume, ElastiCache Cluster, ElastiCache ReplicationGroup
  - RDS DBInstance, RDS DBCluster, Redshift Cluster
- DeletePolicy=Delete (default behavior):
  - Note: for **AWS::RDS::DBCluster** resources, the default policy is Snapshot
  - Note: to delete an S3 bucket, you need to first empty the bucket of its content

Termination protection

Prevents the stack from being accidentally deleted. Once created, you can update this through stack actions.

- (•) Disabled
- ( ) Enabled

CloudFormation Custom Resources (Lambda)

- You can define a Custom Resource in CloudFormation to address any of these use cases:
- An AWS resource is yet not covered (new service for example)
- An On-Premise resource
- Emptying an S3 bucket before being deleted
- Fetch an AMI id
- Anything you want…!

CloudFormation Custom Resources (Lambda)

- The Lambda Function will get invoked only if there is a Create, Update or Delete event, not every time you run the template

```
{
   "RequestType" : "Create",
   "ResponseURL" : "http://pre-signed-S3-url-for-response",
   "StackId" : "arn:aws:cloudformation:us-west-2:123456789012:stack/stack-name/guid",
   "RequestId" : "unique id for this create request",
   "ResourceType" : "Custom::TestResource",
   "LogicalResourceId" : "MyTestResource",
   "ResourceProperties" : {
      "Name" : "Value",
      "List" : [ "1", "2", "3" ]
   }
}
```


```
{
   "RequestType" : "Create",
   "ResponseURL" : "http://pre-signed-S3-url-for-response",
```

```
    "StackId" :
"arn:aws:cloudformation:us-west-2:123456789012:stack/stack-name/guid",
    "RequestId" : "unique id for this create request",
    "ResourceType" : "Custom::TestResource",
    "LogicalResourceId" : "MyTestResource",
    "ResourceProperties" : {
       "Name" : "Value",
       "List" : [ "1", "2", "3" ]
    }
 }
```

```
---
AWSTemplateFormatVersion: '2010-09-09'

Resources:
  myBucketResource:
    Type: AWS::S3::Bucket

  LambdaUsedToCleanUp:
    Type: Custom::cleanupbucket
    Properties:
      ServiceToken: !ImportValue EmptyS3BucketLambda
      BucketName: !Ref myBucketResource
```

Service token :- Reference Lambda Arn

Stack Status	Description
CREATE_COMPLETE	Successful creation of one or more stacks.
CREATE_IN_PROGRESS	Ongoing creation of one or more stacks.
CREATE_FAILED	Unsuccessful creation of one or more stacks. View the stack events to see any associated error messages. Possible reasons for a failed creation include insufficient permissions to work with all resources in the stack, parameter values rejected by an AWS service, or a timeout during resource creation.

DELETE_COMPLETE	Successful deletion of one or more stacks. Deleted stacks are retained and viewable for 90 days.
DELETE_FAILED	Unsuccessful deletion of one or more stacks. Because the delete failed, you might have some resources that are still running; however, you can't work with or update the stack. Delete the stack again or view the stack events to see any associated error messages.
DELETE_IN_PROGRESS	Ongoing removal of one or more stacks.
REVIEW_IN_PROGRESS	Ongoing creation of one or more stacks with an expected StackId but without any templates or resources.

Important

A stack with this status code counts against the maximum possible number of stacks.

ROLLBACK_COMPLETE	Successful removal of one or more stacks after a failed stack creation or after an explicitly canceled stack creation. Any resources that were created during the create stack operation are deleted. This status exists only after a failed stack creation. It signifies that all operations from the partially created stack have been appropriately cleaned up. When in this state, only a delete operation can be performed.
ROLLBACK_FAILED	Unsuccessful removal of one or more stacks after a failed stack creation or after an explicitly canceled stack creation. Delete the stack or view the stack events to see any associated error messages.
ROLLBACK_IN_PROGRESS	Ongoing removal of one or more stacks after a failed stack creation or after an explicitly canceled stack creation.
UPDATE_COMPLETE	Successful update of one or more stacks.
UPDATE_COMPLETE_CLEANUP_IN_PROGRESS	Ongoing removal of old resources for one or more stacks after a successful stack update. For stack updates that require resources to be replaced, AWS CloudFormation creates the new resources first and then deletes the old resources to help reduce any interruptions with your stack. In this state, the stack has been updated and is usable, but AWS CloudFormation is still deleting the old resources.
UPDATE_FAILED	Unsuccessful update of one or more stacks. View the stack events to see any associated error messages.
UPDATE_IN_PROGRESS	Ongoing update of one or more stacks.
UPDATE_ROLLBACK_COMPLETE	Successful return of one or more stacks to a previous working state after a failed stack update.

UPDATE_ROLLBACK_COMPLETE_CLEANUP_IN_PROGRESS	Ongoing removal of new resources for one or more stacks after a failed stack update. In this state, the stack has been rolled back to its previous working state and is usable, but AWS CloudFormation is still deleting any new resources it created during the stack update.
UPDATE_ROLLBACK_FAILED	Unsuccessful return of one or more stacks to a previous working state after a failed stack update. When in this state, you can delete the stack or continue rollback. You might need to fix errors before your stack can return to a working state. Or, you can contact AWS Support to restore the stack to a usable state.
UPDATE_ROLLBACK_IN_PROGRESS	Ongoing return of one or more stacks to the previous working state after failed stack update.
IMPORT_IN_PROGRESS	The import operation is currently in progress.
IMPORT_COMPLETE	The import operation successfully completed for all resources in the stack that support resource import.
IMPORT_ROLLBACK_IN_PROGRESS	Import will roll back to the previous template configuration.
IMPORT_ROLLBACK_FAILED	The import rollback operation failed for at least one resource in the stack. Results will be available for the resources CloudFormation successfully imported.
IMPORT_ROLLBACK_COMPLETE	Import successfully rolled back to the previous template configuration.

Capability:

- CAPABILITY_IAM
- CAPABILITY_NAMED_IAM

Cfn-hup:

- run s as a daemon within ec2 instance to monitor the change in metadata for cfn-init.
- If metadata change detected (through changeset) cfn-hup daemon will automatically update the cfn-init metadata.
- Thus the ec2 instances are not getting replaced due to cfn-init change.

Stack Policy
When you create a stack, all update actions are allowed on all resources. By default, anyone with stack update permissions can update all of the resources in the stack. During an update, some resources might require an interruption or be completely replaced, resulting in new physical IDs or completely new storage. You can prevent stack resources from being unintentionally updated or deleted during a stack update by using a stack policy. A stack policy is a JSON document that defines the update actions that can be performed on designated resources.

After you set a stack policy, all of the resources in the stack are protected by default. To allow updates on specific resources, you specify an explicit Allow statement for those resources in your stack policy. You can define only one stack policy per stack, but, you can protect multiple resources within a single policy. A stack policy applies to all AWS CloudFormation users who attempt to update the stack. You can't associate different stack policies with different users

A stack policy applies only during stack updates. It doesn't provide access controls like an AWS Identity and Access Management (IAM) policy. Use a stack policy only as a fail-safe mechanism to prevent accidental updates to specific stack resources. To control access to AWS resources or actions, use IAM.

Lambda

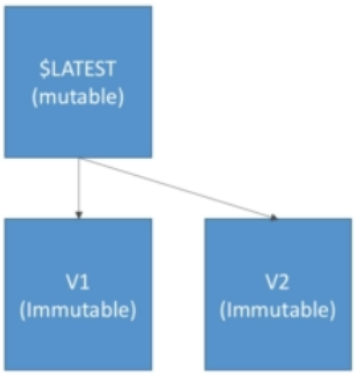
Features	
Default timeout	3 sec
Max timeout	15 min
Default Memory	128MB
Max Memory	3008 Mb (64 mb increments)
Debugging	Async Invocation - When you invoke your function asynchronously, AWS Lambda handles retries. Incoming events are placed in a queue before being sent to the function. If the function returns an error, Lambda retries up to two times. If the function is throttled, or Lambda returns an error, the event is kept in the queue for up to six hours. - When an event fails all attempts or stays in the asynchronous invocation queue for too long, Lambda discards it. - Provision to configure DLQ for async lambda invocation(Configure a dead-letter queue to send discarded events to an Amazon SQS queue or Amazon SNS topic.)

	- Max no of retry : 2 - Max age of event : 6 hours - function's execution role requires permission to write to the queue or topic
Concurrency	functions' concurrency is the number of instances that serve requests at a given time. For an initial burst of traffic, your functions' cumulative concurrency in a Region can reach an initial level of between 500 and 3000, which varies per Region. When you allocate provisioned concurrency, your function is ready to serve a burst of incoming requests with very low latency. When all provisioned concurrency is in use, the function scales up normally to handle any additional requests. Unreserved account concurrency **1000**
Triggers	- API Gateway(Sync) - AWS Iot(Sync) - Alexa Skill(Sync) - Alexa Home(Sync) - Kafka - ALB - CloudFront - Cloudwatch Logs - EventBridge - Codecommit - Cognito Sync Trigger - DynamoDB - Kinesis - MQ - MSK - S3 - SNS - SQS
Environment Variables	An environment variable is a pair of strings that are stored in a function's version-specific configuration. Environment variables are not evaluated prior to the function invocation. Any value you define is considered a literal string and not expanded. Perform the variable evaluation in your function code You define environment variables on the unpublished version of your function. When you publish a version, the environment variables are locked for that version along with other version-specific configuration. The total size of all environment variables doesn't exceed 4 KB aws lambda update-function-configuration --function-name my-function \ --environment "Variables={BUCKET=my-bucket,KEY=file.txt}"

	To get current configuration: aws lambda get-function-configuration --function-name my-function Lambda encrypts environment variables with a key that it creates in your account (an AWS managed customer master key (CMK)). Use of this key is free. You can also choose to provide your own key for Lambda to use instead of the default key. Customer managed CMKs incur standard AWS KMS charges No AWS KMS permissions are required for your user or the function's execution role to use the default encryption key. To use a customer managed CMK, you need permission to use the key. Lambda uses your permissions to create a grant on the key. This allows Lambda to use it for encryption. ● kms:ListAliases – To view keys in the Lambda console. ● kms:CreateGrant, kms:Encrypt – To configure a customer managed CMK on a function. ● kms:Decrypt – To view and manage environment variables that are encrypted with a customer managed CMK. You can also encrypt environment variable values on the client side before sending them to Lambda, and decrypt them in your function code. This obscures secret values in the Lambda console and API output, even for users who have permission to use the key. In your code, you retrieve the encrypted value from the environment and decrypt it by using the AWS KMS API. Another option is to store passwords in AWS Secrets Manager secrets. You can reference the secret in your AWS CloudFormation templates to set passwords on databases. You can also set the value of an environment variable on the Lambda function.

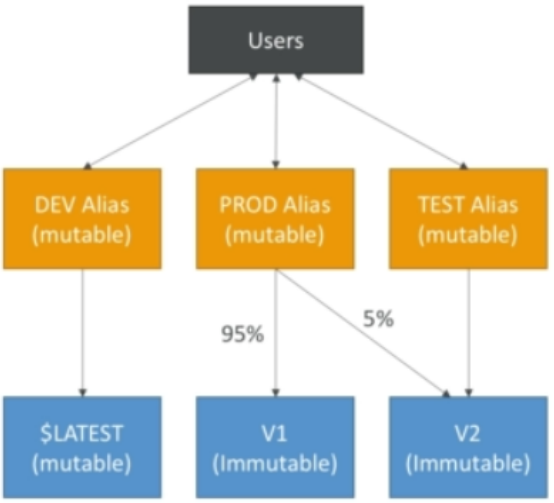
AWS Lambda Versions

- 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



SAM

Instrumentation:
sam init --runtime nodejs12.x
sam local invoke "HelloWorldFunction" -e events/event.json
sam local start-api

The AWS Serverless Application Model (AWS SAM) is an open-source framework

A **serverless application** is a combination of Lambda functions, event sources, and other resources that work together to perform tasks(is more than just a Lambda function—it can include additional resources such as APIs, databases, and event source mappings).

AWS SAM template specification. You use this specification to define your serverless application. It provides you with a simple and clean syntax to describe the functions, APIs,

permissions, configurations, and events that make up a serverless application. You use an AWS SAM template file to operate on a single, deployable, versioned entity that's your serverless application.

AWS SAM command line interface (AWS SAM CLI): provides commands that enable you to verify that AWS SAM template files are written according to the specification, invoke Lambda functions locally, step-through debug Lambda functions, package and deploy serverless applications to the AWS Cloud.

Advantages:

Single-deployment configuration. AWS SAM makes it easy to organize related components and resources, and operate on a single stack. You can use AWS SAM to share configuration (such as memory and timeouts) between resources, and deploy all related resources together as a single, versioned entity.
AWS SAM is an extension of AWS CloudFormation, you get the reliable deployment capabilities of AWS CloudFormation. You can define resources by using AWS CloudFormation in your AWS SAM template. Also, you can use the full suite of resources, intrinsic functions, and other template features that are available in AWS CloudFormation
Enforce best practices such as code reviews. Also, with a few lines of configuration, you can enable safe deployments through CodeDeploy, and can enable tracing by using AWS X-Ray
Locally build, test, and debug serverless applications that are defined by AWS SAM templates. The CLI provides a Lambda-like execution environment locally.
For authoring, testing, and debugging AWS SAM–based serverless applications, you can use the AWS Cloud9 IDE. To build a deployment pipeline for your serverless applications, you can use CodeBuild, CodeDeploy, and CodePipeline. You can also use AWS CodeStar to get started with a project structure, code repository, and a CI/CD pipeline that's automatically configured for you. To deploy your serverless application, you can use the Jenkins plugin. You can use the Stackery.io toolkit to build production-ready applications.

Step Functions

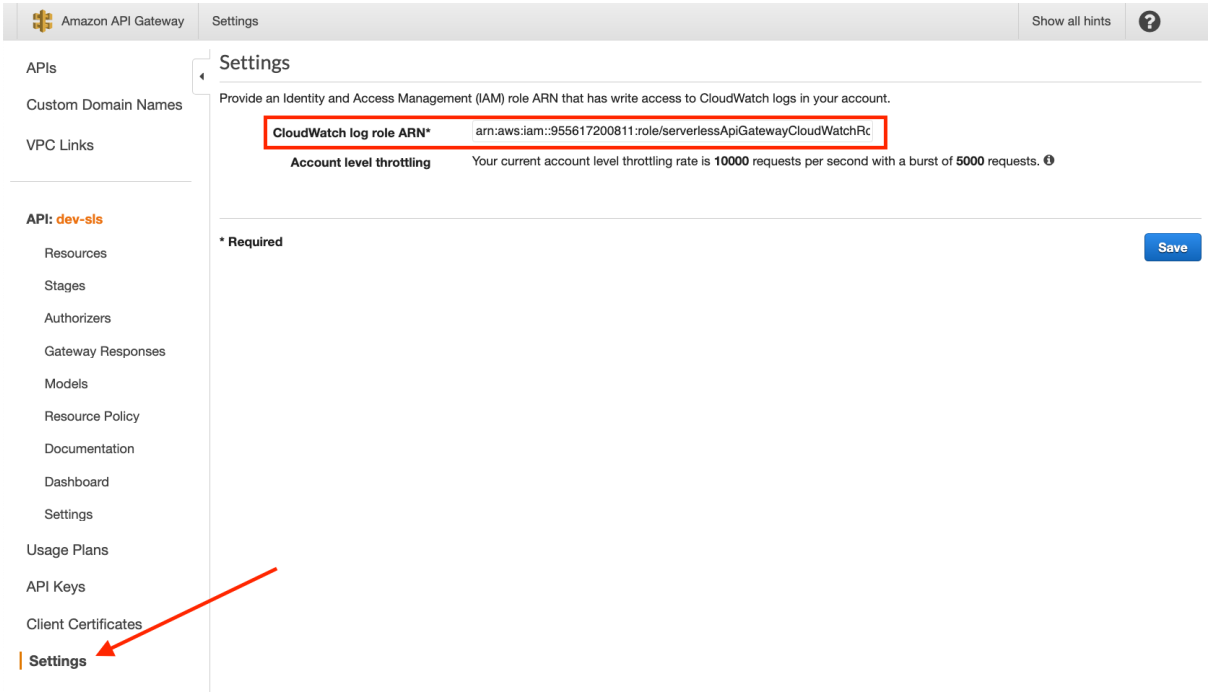
API Gateway

Access Logging

To allow your API Gateway to write to a CloudWatch Logs log group, you need to associate an IAM role that has permissions to write to CloudWatch Logs.

The key here is that **a single IAM role is configured for all API Gateway APIs in a region of your AWS account.** It's a singleton resource, rather than being an IAM role for each API Gateway API that you deploy.

In the API Gateway console, click on one of your deployed APIs. At the very bottom of the left-hand side, you should see a "Settings" option. Click that, and you will see the CloudWatch log role ARN for your API.



Again, while this appears to be in the context of your chosen API, it actually applies to all APIs in your current region. If you're configuring this via CloudFormation, you'll set it up as the AWS::ApiGateway::Account resource.

```
I deploy Service A, which has an API Gateway instance and configures the
AWS::ApiGateway::Account with an IAM role created in Service A's stack.

I deploy Service B, which also has an API Gateway instance and thus also
configures the AWS::ApiGateway::Account resource with an IAM role in
Service B's stack. This overrides the current setting for
AWS::ApiGateway::Account.

I remove Service B. This has the impact of deleting the IAM role while
still leaving its value in the AWS::ApiGateway::Account configuration.
```

```
However, because that role no longer exists, all of your APIs will stop
writing access logs to CloudWatch.
Additionally, even if you redeploy Service A, it won't update the value
in AWS::ApiGateway::Account because, from CloudFormation's view, it
doesn't look like that value has changed.
```

For SAM,

two options:

- Deploy a separate, standalone service that configures the IAM role and API Gateway Account resource in each region you use; or

- Add DeletionPolicy: Retain for the IAM role that is configured for your API Gateway Account. Even if the service is removed, the IAM role will persist and be able to write logs to CloudWatch.

Integration Type: Lambda

- By pointing to a Lambda Alias , it is possible to do canary/ab testing(applying weitage to different lambda versions)

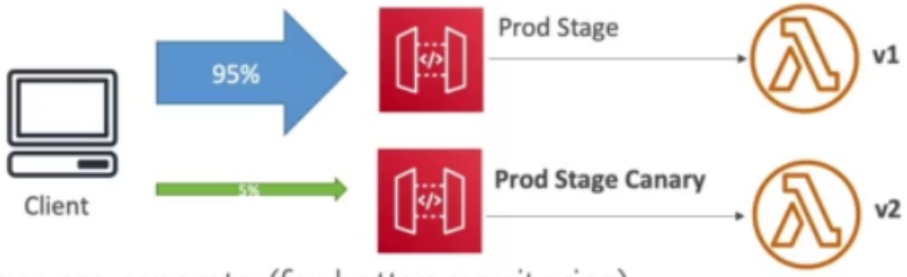
-

API Gateway – Stage Variables

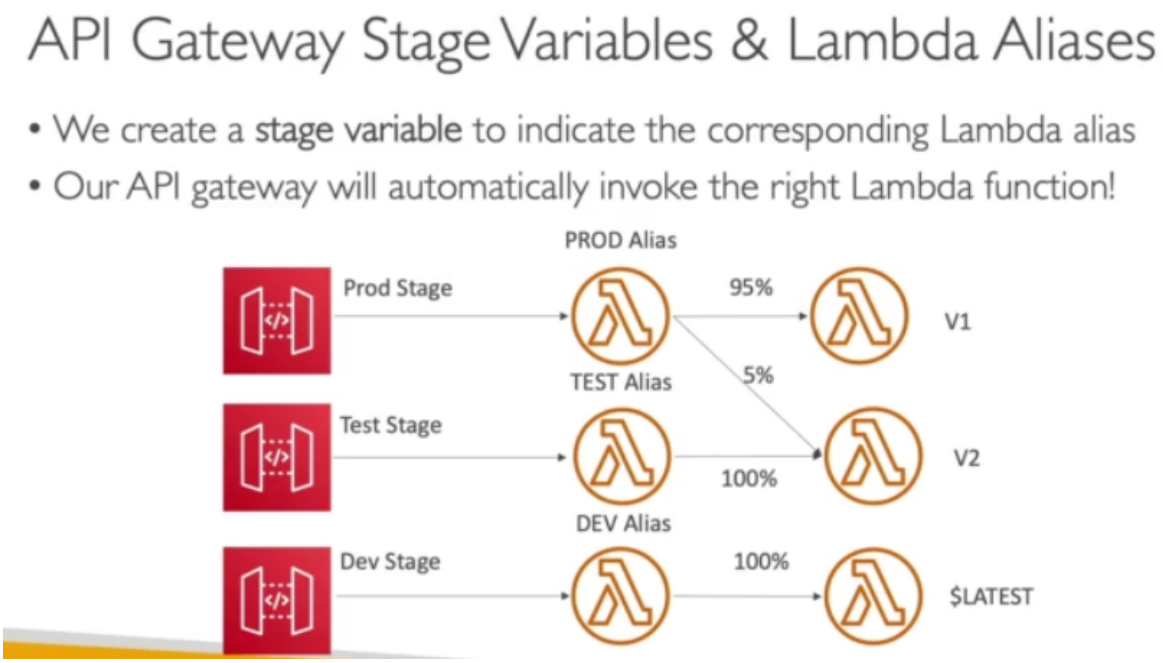
- Stage variables are like environment variables for API Gateway
- Use them to change often changing configuration values
- They can be used in:
  - Lambda function ARN
  - HTTP Endpoint
  - Parameter mapping templates
- Use cases:
  - Configure HTTP endpoints your stages talk to (dev, test, prod…)
  - Pass configuration parameters to AWS Lambda through mapping templates
- Stage variables are passed to the "context" object in AWS Lambda

API Gateway – Canary Deployment

- Possibility to enable canary deployments for any stage (usually prod)
- Choose the % of traffic the canary channel receives



- Metrics & Logs are separate (for better monitoring)
- Possibility to override stage variables for canary
- This is blue / green deployment with AWS Lambda & API Gateway

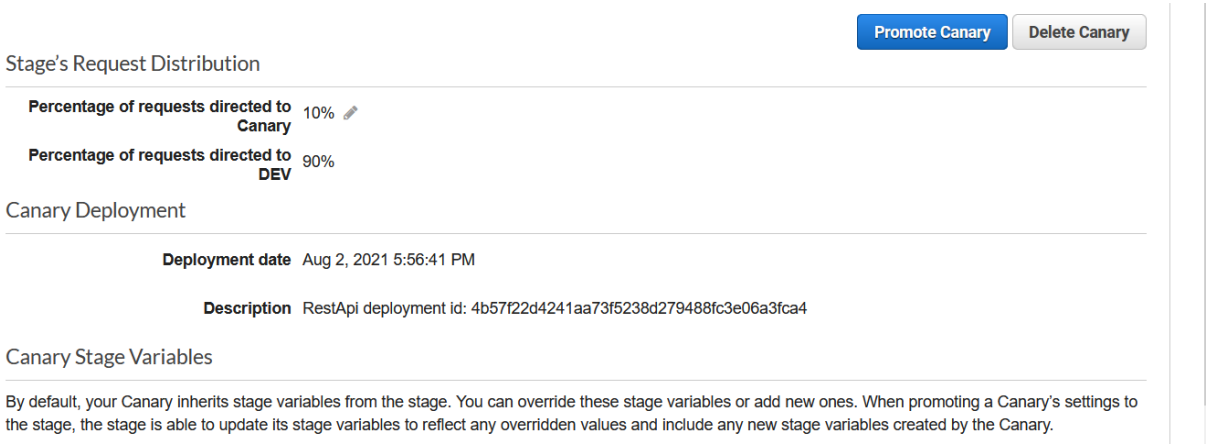


-

API Gateway Canary Deployment

A Canary is used to test new API deployments and/or changes to stage variables. A Canary can receive a percentage of requests going to your stage. In addition, API deployments will be made to the Canary first before being able to be promoted to the entire stage.

```
OrderAPI:
    Type: AWS::Serverless::Api
    Properties:
      Name: Order App Api
      StageName: !Ref Stage
      CacheClusterEnabled: true
      CacheClusterSize: "0.5"
      CanarySetting:
        PercentTraffic: 10
      Tags:
        Name:  Order App ApiGateway
```



API Gateway Throttle:

API Gateway Throttling levels
The current account level throttling rate is **10000** requests per second with a burst of **5000** requests.

By default, API Gateway limits the steady-state requests per second (rps) across all APIs within an AWS account, per Region. It also limits the burst (that is, the maximum bucket size) across all APIs within an AWS account, per Region. In API Gateway, the burst limit corresponds to the maximum number of concurrent request submissions that API Gateway can fulfill at any moment without returning 429 Too Many Requests error responses.

- If a caller submits 10,000 requests in a one-second period evenly (for example, 10 requests every millisecond), API Gateway processes all requests without dropping any.

- If the caller sends 10,000 requests in the first millisecond, API Gateway serves 5,000 of those requests and throttles the rest in the one-second period.

- If the caller submits 5,000 requests in the first millisecond and then evenly spreads another 5,000 requests through the remaining 999 milliseconds (for example, about 5 requests every millisecond), API Gateway processes all 10,000 requests in the one-second period without returning 429 Too Many Requests error responses.

- If the caller submits 5,000 requests in the first millisecond and waits until the 101st millisecond to submit another 5,000 requests, API Gateway processes 6,000 requests and throttles the rest in the one-second period. This is because at the rate of 10,000 rps, API Gateway has served 1,000 requests after the first 100 milliseconds and thus emptied the bucket by the same amount. Of the next spike of 5,000 requests, 1,000 fill the bucket and are queued to be processed. The other 4,000 exceed the bucket capacity and are discarded.

- If the caller submits 5,000 requests in the first millisecond, submits 1,000 requests at the 101st millisecond, and then evenly spreads another 4,000 requests through the remaining 899 milliseconds, API Gateway processes all 10,000 requests in the one-second period without throttling.

You can set the default method throttling to override the account-level request throttling limits for a specific stage or for individual methods in your API. The default method throttling limits are bounded by the account-level rate limits per Region, even if you set the default method throttling limits higher than the account-level limits.

In a usage plan, you can set a default per-method throttling limit for all methods at the API or stage level under **Create Usage Plan.**

API Gateway usage plan

A *usage plan* specifies who can access one or more deployed API stages and methods—and also how much and how fast they can access them. The plan uses API keys to identify API clients and meters access to the associated API stages for each key. It also lets you configure throttling limits and quota limits that are enforced on individual client API keys.

API keys are alphanumeric string values that you distribute to application developer customers to grant access to your API. You can use API keys together with usage plans or Lambda authorizers to control access to your APIs. API Gateway can generate API keys on your behalf, or you can import them from a CSV file. You can generate an API key in API Gateway, or import it into API Gateway from an external source.

An API key has a name and a value. (The terms "API key" and "API key value" are often used interchangeably.) The value is an alphanumeric string between 30 and 128 characters.

API key values must be unique. If you try to create two API keys with different names and the same value, API Gateway considers them to be the same API key.

An API key can be associated with more than one usage plan. A usage plan can be associated with more than one stage. However, a given API key can only be associated with one usage plan for each stage of your API.

A *throttling limit* is a request rate limit that is applied to each API key that you add to the usage plan. You can also set a default method-level throttling limit for an API or set throttling limits for individual API methods.

A *quota limit* is the maximum number of requests with a given API key that can be submitted within a specified time interval. You can configure individual API methods to require API key

authorization based on usage plan configuration. You can also use the get-usage CLI command or the usage:get REST API method to determine the usage for an API customer.

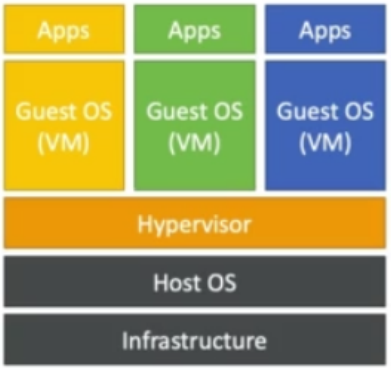
- Don't rely on API keys as your only means of authentication and authorization for your APIs. For one thing, if you have multiple APIs in a usage plan, a user with a valid API key for one API in that usage plan can access *all* APIs in that usage plan. Instead, use an IAM role, a Lambda authorizer, or an Amazon Cognito user pool.

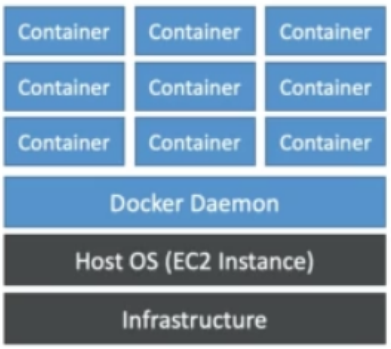
- If you're using a developer portal to publish your APIs, note that all APIs in a given usage plan are subscribable, even if you haven't made them visible to your customers.

ECS

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





- 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 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

Container instance IAM role

The **Amazon ECS container agent makes calls to the Amazon ECS API actions** on your behalf, **so container instances that run the agent require the ecsInstanceRole IAM policy and role** for the service to know that the agent belongs to you.

The trusted entities that can assume the role and the access conditions for the role.
Trust Policy

```
{
  "Version": "2008-10-17",
  "Statement": [
        {
        "Sid": "",
        "Effect": "Allow",
        "Principal": {
        "Service": "ec2.amazonaws.com"
        },
        "Action": "sts:AssumeRole"
        }
  ]
}
```

Permissions

```
{
      "Version": "2012-10-17",
      "Statement": [
      {
            "Effect": "Allow",
            "Action": [
                  "ec2:DescribeTags",
                  "ecs:CreateCluster",
                  "ecs:DeregisterContainerInstance",
                  "ecs:DiscoverPollEndpoint",
                  "ecs:Poll",
                  "ecs:RegisterContainerInstance",
                  "ecs:StartTelemetrySession",
                  "ecs:UpdateContainerInstancesState",
                  "ecs:Submit*",
```

```
                "ecr:GetAuthorizationToken",
                "ecr:BatchCheckLayerAvailability",
                "ecr:GetDownloadUrlForLayer",
                "ecr:BatchGetImage",
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": "*"
        }
        ]
}
```

AWS ECS Agent Docker Container

```
[ec2-user@ip-172-31-27-42 ~]$ docker ps
CONTAINER ID        IMAGE                            COMMAND             CREATED
        STATUS                  PORTS          NAMES
d2ffbd141081        amazon/amazon-ecs-agent:latest   "/agent"            7 minutes ago
        Up 7 minutes (healthy)                 ecs-agent
```

/etc/ecs/ecs.config

```
ECS_CLUSTER=demo-ec2-ecs-cluster
ECS_BACKEND_HOST=
```

Task Role

Optional IAM role that tasks can use to make API requests to authorized AWS services.

Trust Policy

```
{
  "Version": "2012-10-17",
  "Statement": [
        {
        "Sid": "",
        "Effect": "Allow",
        "Principal": {
        "Service": "ecs-tasks.amazonaws.com"
        },
        "Action": "sts:AssumeRole"
        }
  ]
}
```

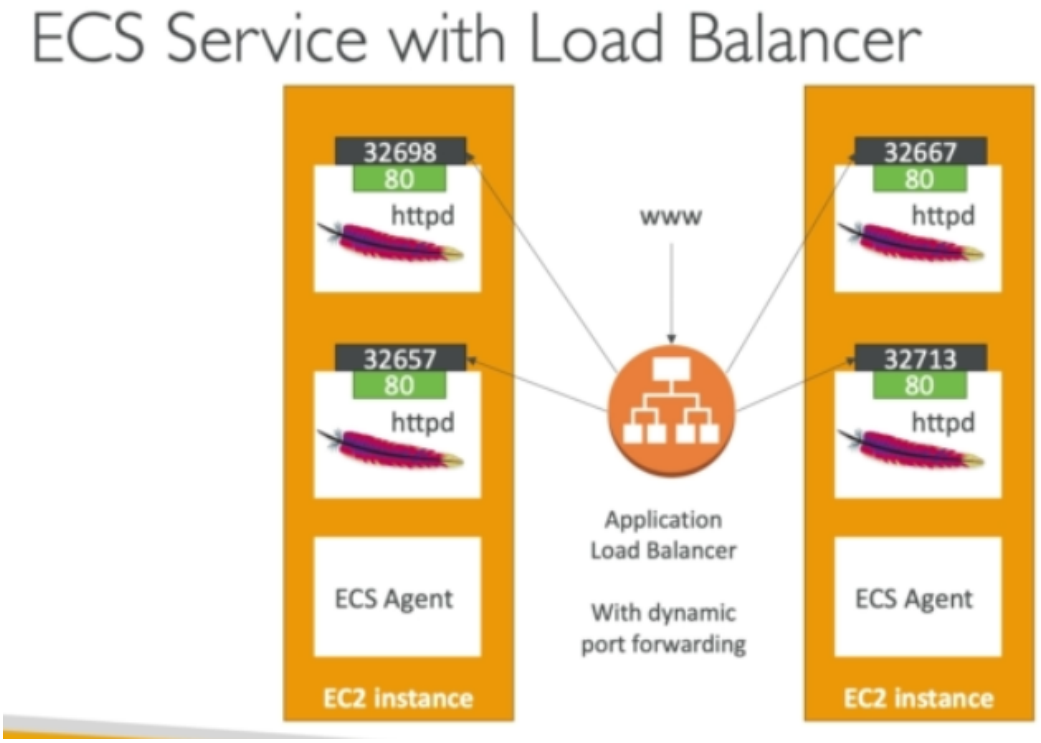
Task execution IAM role

This role is required by tasks to pull container images and publish container logs to Amazon CloudWatch on your behalf.
[
The role that authorizes Amazon ECS to pull private images and publish logs for your task. This takes the place of the EC2 Instance role when running tasks.
]

```
{
        "Version": "2012-10-17",
        "Statement": [
        {
        "Effect": "Allow",
        "Action": [
                "ecr:GetAuthorizationToken",
                "ecr:BatchCheckLayerAvailability",
                "ecr:GetDownloadUrlForLayer",
                "ecr:BatchGetImage",
                "logs:CreateLogStream",
                "logs:PutLogEvents"
        ],
        "Resource": "*"
        }
        ]
}
```

Trust Policy

```
{
  "Version": "2012-10-17",
  "Statement": [
        {
        "Sid": "",
        "Effect": "Allow",
        "Principal": {
        "Service": "ecs-tasks.amazonaws.com"
        },
        "Action": "sts:AssumeRole"
        }
  ]
}
```



Open the Amazon ECS console, and then set the host port to **0** for the task definition that you're creating or updating. Be sure to set the container port mappings for your application. **Important:** The **host** and **awsvpc** network modes do not support dynamic host port mapping.

Add a rule to allow inbound traffic from your load balancer to your container instances. The security group and network access control list (network ACL) must allow traffic from the load balancer to the instances over the ephemeral port range.

You can add a load balancer only during the creation of the service. After service creation, you can't change the target group's Amazon Resource Name (ARN), container name, or the container port specified in the service definition. You can't add, remove, or change the load balancer configuration of an existing service. If you update the service task definition, then the container name and container port specified at service creation must remain in the task definition.

Service
“A service lets you specify how many copies of your task definition to run and maintain in a cluster. You can optionally use an Elastic Load Balancing load balancer to distribute incoming traffic to containers in your service. Amazon ECS maintains that number of tasks and coordinates task scheduling with the load balancer. You can also optionally use Service Auto Scaling to adjust the number of tasks in your service.”

ECR

- ECR is a private Docker image repository
- Access is controlled through IAM (permission errors => policy)
- You need to run some commands to push pull:
  - $(aws ecr get-login --no-include-email --region eu-west-1)
  - docker push 1234567890.dkr.ecr.eu-west-1.amazonaws.com/demo:latest
  - docker pull 1234567890.dkr.ecr.eu-west-1.amazonaws.com/demo:latest

aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 803138993991.dkr.ecr.us-east-1.amazonaws.com

docker build -t generateorderreceipt .

docker tag generateorderreceipt:latest 803138993991.dkr.ecr.us-east-1.amazonaws.com/generateorderreceipt:latest

docker push 803138993991.dkr.ecr.us-east-1.amazonaws.com/generateorderreceipt: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…

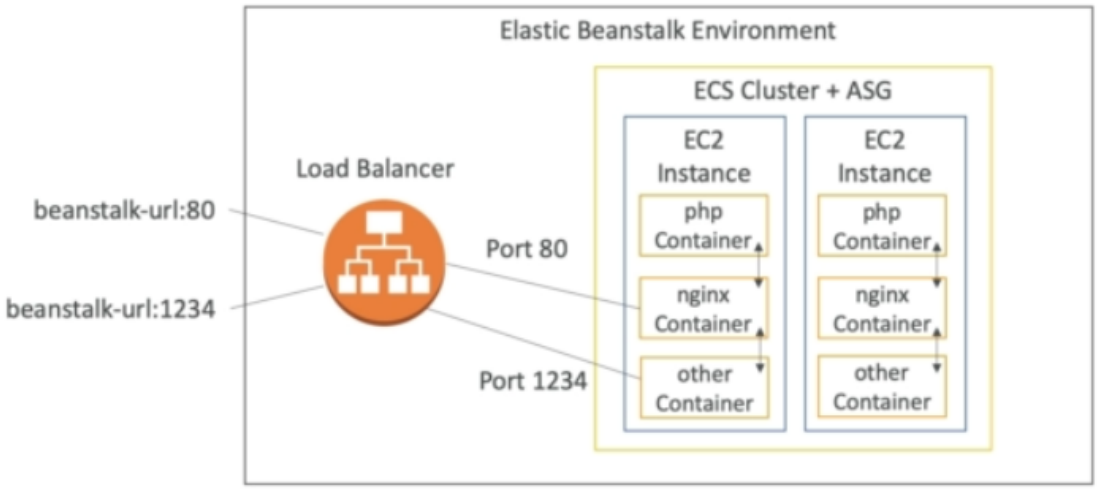
- With Fargate, it's all Serverless!
- We don't provision EC2 instances
- We just create task definitions, and AWS will run our containers for us
- To scale, just increase the task number. Simple! No more EC2 ☺

Elastic Beanstalk + ECS

- You can run Elastic Beanstalk in Single & Multi Docker Container mode
- 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 file **Dockerrun.aws.json** at the root of source code
- Your Docker images must be pre-built and stored in ECR for example

Host port mappings are not valid when the network mode for a task definition is host or awsvpc. To specify different host and container port mappings, choose the Bridge network mode.

Elastic Beanstalk + ECS



Beanstalk does not create services. It creates tasks.

ECS Auto Scaling

- Ec2 autoscaling
- Task auto scaling

Step Scaling

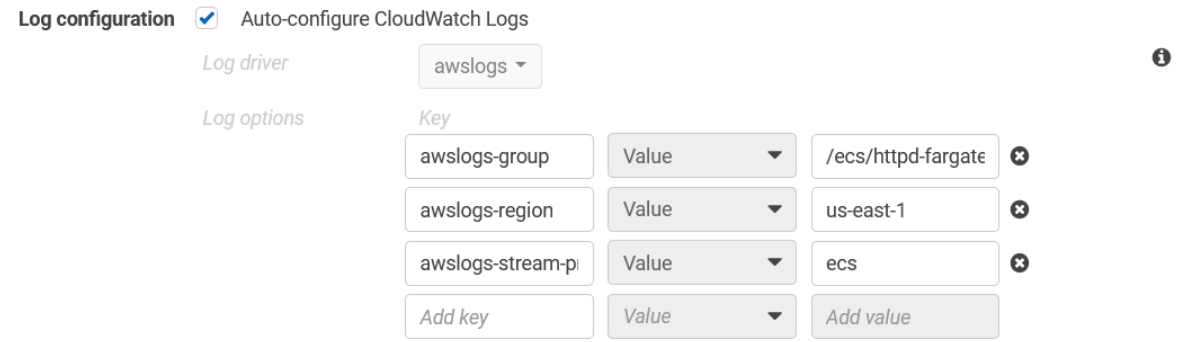
Target Tracking Scaling policy

Auto Scaling done at Service Level.

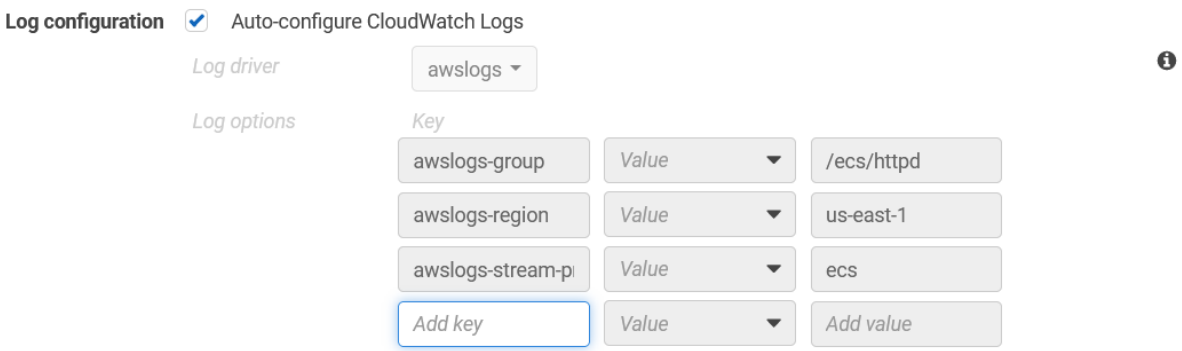
Elastic Beanstalk with ECS automatically manages auto scaling with EC2 & Tasks

Cloudwatch Logs

Fargate

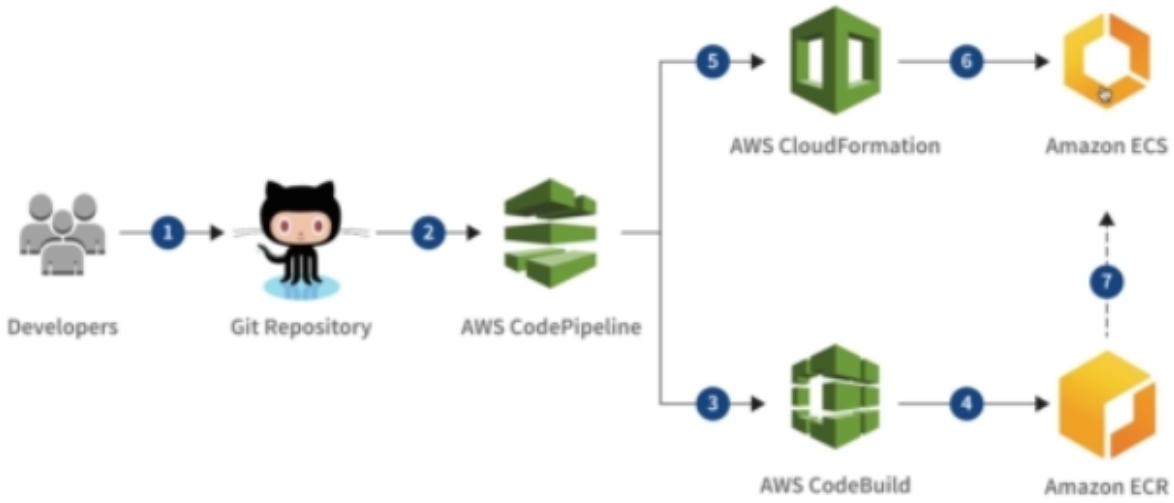


EC2

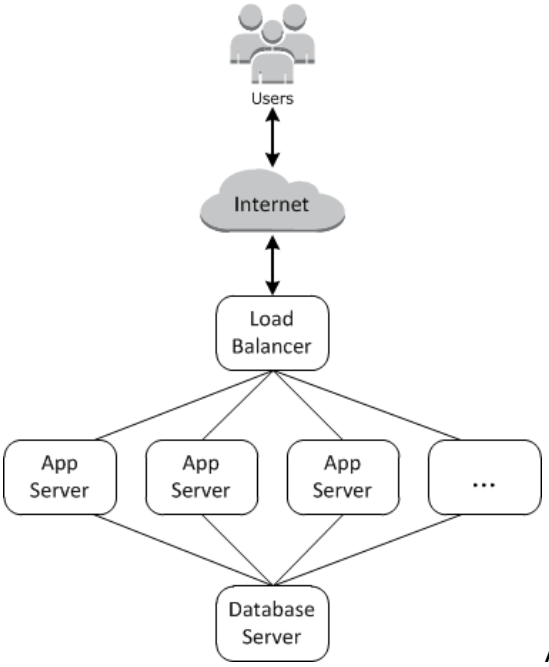


To log ec2 instances logs to cloudwatch.
https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/EC2NewInstanceCWL.html

CloudWatch Container Insights is a monitoring and troubleshooting solution for containerized applications and microservices. It collects, aggregates, and summarizes compute utilization such as CPU, memory, disk, and network; and diagnostic information such as container restart failures to help you isolate issues with your clusters and resolve them quickly.

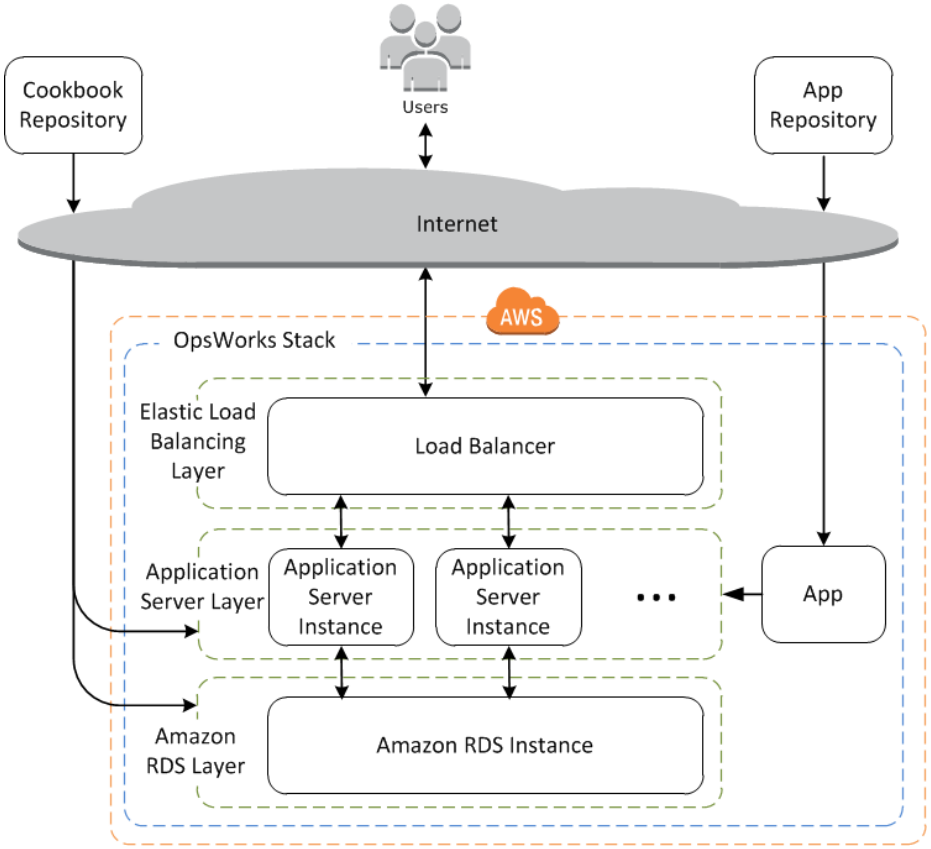


OpsWorks



AWS OpsWorks Stacks provides a simple and flexible way to create and manage stacks and applications.

Here's how a basic application server stack might look with AWS OpsWorks Stacks. It consists of a group of application servers running behind an Elastic Load Balancing load balancer, with a backend Amazon RDS database server.



The *stack* is the core AWS OpsWorks Stacks component. It is basically a container for AWS resources—Amazon EC2 instances, Amazon RDS database instances, and so on—that have a common purpose and should be logically managed together. The stack helps you manage these resources as a group and also defines some default configuration settings, such as the instances' operating system and AWS region. If you want to isolate some stack components from direct user interaction, you can run the stack in a VPC.

You define the stack's constituents by adding one or more *layers*. A layer represents a set of Amazon EC2 instances that serve a particular purpose, such as serving applications or hosting a database server.

You can customize or extend layers by modifying packages' default configurations, adding Chef recipes to perform tasks such as installing additional packages, and more.

For all stacks, AWS OpsWorks Stacks includes *service layers*, which represent the following AWS services.

- Amazon Relational Database Service
- Elastic Load Balancing
- Amazon Elastic Container Service

Layers give you complete control over which packages are installed, how they are configured, how applications are deployed, and more.

Layers depend on Chef recipes to handle tasks such as installing packages on instances, deploying apps, running scripts, and so on. One of the key AWS OpsWorks Stacks features is a set of *lifecycle events*—***Setup, Configure, Deploy, Undeploy, and Shutdown***—which automatically run a specified set of recipes at the appropriate time on each instance.

Each layer can have a set of recipes assigned to each lifecycle event, which handle a variety of tasks for that event and layer. For example, after an instance that belongs to a web server layer finishes booting, AWS OpsWorks Stacks does the following.

An *instance* represents a single computing resource, such as an Amazon EC2 instance. It defines the resource's basic configuration, such as operating system and size. Other configuration settings, such as Elastic IP addresses or Amazon EBS volumes, are defined by the instance's layers. The layer's recipes complete the configuration by performing tasks such as installing and configuring packages and deploying apps.

You can use AWS OpsWorks Stacks to create instances and add them to a layer. When you start the instance, AWS OpsWorks Stacks launches an Amazon EC2 instance using the configuration settings specified by the instance and its layer. After the Amazon EC2 instance has finished booting, AWS OpsWorks Stacks installs an agent that handles communication between the instance and the service and runs the appropriate recipes in response to lifecycle events.

You store applications and related files in a repository, such as an Amazon S3 bucket/git. **Each application is represented by an *app***, which specifies the application type and contains the information that is needed to deploy the application from the repository to your instances, such as the repository URL and password. When you deploy an app, AWS OpsWorks Stacks triggers a Deploy event, which runs the Deploy recipes on the stack's instances.

Lifecycle events

Setup, Deploy, UnDeploy, ShutDown → Instance level
Configure → all instances

Each layer has a set of five lifecycle events, each of which has an associated set of recipes that are specific to the layer. When an event occurs on a layer's instance, AWS OpsWorks Stacks automatically runs the appropriate set of recipes. To provide a custom response to these events, implement custom recipes and assign them to the appropriate events for each layer. AWS OpsWorks Stacks runs those recipes after the event's built-in recipes.

Setup

This event occurs after a started instance has finished booting. You can also manually trigger the Setup event by using the Setup stack command. AWS OpsWorks Stacks runs recipes that set the instance up according to its layer.

A **Setup** event takes an instance out of service. Because an instance is not in the **Online** state when the **Setup** lifecycle event runs, instances on which you run **Setup** events are removed from a load balancer.

Configure

This event occurs ***on all of the stack's instances*** when one of the following occurs:

- An instance enters or leaves the online state.
- You associate an Elastic IP address with an instance or disassociate one from an instance.
- You attach an Elastic Load Balancing load balancer to a layer, or detach one from a layer.

For example, suppose that your stack has instances A, B, and C, and you start a new instance, D. After D has finished running its setup recipes, AWS OpsWorks Stacks triggers the Configure event on A, B, C, and D. If you subsequently stop A, AWS OpsWorks Stacks triggers the Configure event on B, C, and D. AWS OpsWorks Stacks responds to the Configure event by running each layer's Configure recipes, which update the instances' configuration to reflect the current set of online instances

Deploy

This event occurs when you run a **Deploy** command, typically to deploy an application to a set of application server instances. The instances run recipes that deploy the application and any related files from its repository to the layer's instances

Setup includes Deploy; it runs the Deploy recipes after setup is complete.

Undeploy

This event occurs when you delete an app or run an Undeploy command to remove an app from a set of application server instances. The specified instances run recipes to remove all application versions and perform any required cleanup.

Shutdown

This event occurs after you direct AWS OpsWorks Stacks to shut an instance down but before the associated Amazon EC2 instance is actually terminated. AWS OpsWorks Stacks runs recipes to perform cleanup tasks such as shutting down services.

If you have attached an Elastic Load Balancing load balancer to the layer and enabled support for connection draining, AWS OpsWorks Stacks waits until connection draining is complete before triggering the Shutdown event.

After triggering a Shutdown event, AWS OpsWorks Stacks allows Shutdown recipes a specified amount of time to perform their tasks, and then stops or terminates the Amazon EC2 instance. The default Shutdown timeout value is 120 seconds.

After a started instance has finished booting, the remaining startup sequence is as follows:

1. AWS OpsWorks Stacks runs the instance's built-in Setup recipes, followed by any custom Setup recipes.

2. AWS OpsWorks Stacks runs the instance's built-in Deploy recipes, followed by any custom Deploy recipes.

   The instance is now online.

3. AWS OpsWorks Stacks triggers a Configure event on all instances in the stack, including the newly started instance.

   AWS OpsWorks Stacks runs the instances' built-in Configure recipes, followed by any custom Configure recipes.

Auto Healing

Every instance has an AWS OpsWorks Stacks agent that communicates regularly with the service. AWS OpsWorks Stacks uses that communication to monitor instance health. If an agent does not communicate with the service for more than approximately five minutes, AWS OpsWorks Stacks considers the instance to have failed.

If a layer has auto healing enabled—the default setting—AWS OpsWorks Stacks automatically replaces the layer's failed instances as follows:

Instance store-backed instance

1. Stops the Amazon EC2 instance, and verifies that it has shut down.

2. Deletes the data on the root volume.

3. Creates a new Amazon EC2 instance with the same host name, configuration, and layer membership.

4. Reattaches any Amazon EBS volumes, including volumes that were attached after the old instance was originally started.

5. Assigns a new public and private IP Address.

6. If the old instance was associated with an Elastic IP address, associates the new instance with the same IP address.

Amazon EBS-backed instance

1. Stops the Amazon EC2 instance, and verifies that it has stopped.

2. Starts the EC2 instance.

After the auto-healed instance is back online, AWS OpsWorks Stacks triggers a Configure lifecycle event on all of the stack's instances.

If you specify an Amazon EBS volume for a layer's instances, AWS OpsWorks Stacks creates a new volume and attaches it to each instance when the instance is started.

When AWS OpsWorks Stacks auto heals one of a layer's instances, it handles volumes in the following way:

- If the volume was attached to the instance when the instance failed, the volume and its data are saved, and AWS OpsWorks Stacks attaches it to the new instance.

- If the volume was not attached to the instance when the instance failed, AWS OpsWorks Stacks creates a new, empty volume with the configuration specified by the layer, and attaches that volume to the new instance.

Auto healing is enabled by default for all layers, but you can edit the layer's General Settings to disable it.

If you have auto healing enabled, be sure to do the following:

- Use only the AWS OpsWorks Stacks console, CLI, or API to stop instances.

  If you stop an instance in any other way, such as using the Amazon EC2 console, AWS OpsWorks Stacks treats the instance as failed, and auto heals it.

- Use Amazon EBS volumes to store any data that you don't want to lose if the instance is auto healed.

  Auto healing stops the old Amazon EC2 instance, which destroys any data that is not stored on an Amazon EBS volume. Amazon EBS volumes are reattached to the new instance, which preserves any stored data.

When you create a stack, you specify an IAM role, usually called a service role, that grants AWS OpsWorks Stacks the appropriate permissions.

The service role must have this trust relationship for AWS OpsWorks Stacks to act on your behalf.
Standard IAM Role:

```
{
    "Statement": [
        {
```

```
            "Action": [
                "ec2:*",
                "iam:PassRole",
                "cloudwatch:GetMetricStatistics",
                "cloudwatch:DescribeAlarms",
                "ecs:*",
                "elasticloadbalancing:*",
                "rds:*"
            ],
            "Effect": "Allow",
            "Resource": [
                "*"
            ]
        }
    ]
}
```

Trust Policy

```
{
  "Version": "2012-10-17",
  "Statement": [
        {
        "Effect": "Allow",
        "Principal": {
        "Service": "opsworks.amazonaws.com"
        },
        "Action": "sts:AssumeRole"
        }
  ]
}
```

Elastic Beanstalk

Eb cli commands

commands:

abort	Cancels an environment update or deployment.
appversion	Listing and managing application versions
clone	Clones an environment.
codesource	Configures the code source for the EB CLI to use by default.
config	***Modify an environment's configuration. Use subcommands to manage saved configurations.***
console	Opens the environment in the AWS Elastic Beanstalk Management Console.
create	Creates a new environment.
deploy	Deploy your source code to the environment.

```
  events        Recent events.
  health        Shows detailed environment health.
  init    Initializes your directory with the EB CLI. Creates the application.
  labs          Extra experimental commands.
  list    Lists all environments.
  local         Runs commands on your local machine.
  logs          Gets recent logs.
  open          Opens the application URL in a browser.
  platform      Commands for managing platforms.
  printenv      Shows the environment variables.
  restore       Restores a terminated environment.
  scale         Changes the number of running instances.
  setenv        Sets environment variables.
  ssh           Opens the SSH client to connect to an instance.
  status        Gets environment information and status.
  swap          Swaps two environment CNAMEs with each other.
  tags          Allows adding, deleting, updating, and listing of environment tags.
  terminate     Terminates the environment.
  upgrade       Updates the environment to the most recent platform version.
  use           Sets default environment.

C:\workspace\projects\personal\aws-devops\aws-devops\elasticbeanstalk\helloworld>eb
health --help
usage: eb health <environment_name> [options ...]

Shows detailed environment health.

positional arguments:
  environment_name   environment name

optional arguments:
  -h, --help            show this help message and exit
  --debug               toggle debug output
  --quiet         suppress all output
  -v, --verbose         toggle verbose output
  --profile PROFILE     use a specific profile from your credential file
  -r REGION, --region REGION
                        use a specific region
  --no-verify-ssl       don't verify AWS SSL certificates
  --refresh             refresh
  --mono                no color
  --view {split,status,request,cpu}
```

Saved Configuration

The non default configuration set generated by eb cli config.

```
eb config save dev-env --cfg initial-configuration
```

```
# 2) configurations

```
# this backs up the current dev environment configuration
eb config save dev-env --cfg initial-configuration

# this sets an environment variable on the environment
eb setenv ENABLE_COOL_NEW_FEATURE=true

# save our config from the current state of our environment
eb config save dev-env --cfg prod
```

make changes to `.elasticbeanstalk/saved_configs/prod.cfg.yml`
- add an environment variable
- add auto scaling rules
```
  AWSEBAutoScalingScaleUpPolicy.aws:autoscaling:trigger:
    UpperBreachScaleIncrement: '2'
  AWSEBCloudwatchAlarmLow.aws:autoscaling:trigger:
    LowerThreshold: '20'
    MeasureName: CPUUtilization
    Unit: Percent
  AWSEBCloudwatchAlarmHigh.aws:autoscaling:trigger:
    UpperThreshold: '50'
```

then update the saved prod configuration
```
eb config put prod
```

Update current environments from saved configurations
```
eb config dev-env --cfg prod
```
```

During environment creation, configuration options are applied from multiple sources with the following precedence, from highest to lowest:

- **Settings applied directly to the environment** – Settings specified during a create environment or update environment operation on the Elastic Beanstalk API by any client, including the Elastic Beanstalk console, EB CLI, AWS CLI, and SDKs. The Elastic Beanstalk console and EB CLI also apply recommended values for some options that apply at this level unless overridden.

- **Saved Configurations** – Settings for any options that are not applied directly to the environment are loaded from a saved configuration, if specified.

- **Configuration Files (.ebextensions)** – Settings for any options that are not applied directly to the environment, and also not specified in a saved configuration, are loaded from configuration files in the .ebextensions folder at the root of the application source bundle.

  Configuration files are executed in alphabetical order. For example, .ebextensions/01run.config is executed before .ebextensions/02do.config.

- **Default Values** – If a configuration option has a default value, it only applies when the option is not set at any of the above levels.

If the same configuration option is defined in more than one location, the setting with the highest precedence is applied. When a setting is applied from a saved configuration or settings applied directly to the environment, the setting is stored as part of the environment's configuration.

Elastic Beanstalk in the background uses cloudformation for resource creation.

ssh onto the instance and run:
/opt/elasticbeanstalk/bin/get-config environment
/opt/elasticbeanstalk/bin/get-config environment -k NOTIFICATION_TOPIC
/opt/elasticbeanstalk/bin/get-config environment -k DYNAMODB_TABLE
/opt/elasticbeanstalk/bin/get-config optionsettings

```
wseb-e-kpgpppyazg-stack-NotificationTopic-1WFJOQJ0V3ZD9"}[root@ip-172-31
-24-28 ec2-user]# /opt/elasticbeanstalk/bin/get-config environment | jq
{
  "AWS_REGION": "us-east-1",
  "DYNAMODB_TABLE":
"awseb-e-kpgpppyazg-stack-DynamoDBTable-1A84FRVORMJW1",
  "ENABLE_COOL_NEW_FEATURE": "true",
  "NOTIFICATION_TOPIC":
"arn:aws:sns:us-east-1:803138993991:awseb-e-kpgpppyazg-stack-Notificatio
nTopic-1WFJOQJ0V3ZD9"
}
```

Commands

You can use the commands key to execute commands on the EC2 instance. The commands run before the application and web server are set up and the application version file is extracted.

```
commands:
  create_hello_world_file:
        command: touch hello-world.txt
        cwd: /home/ec2-user
```

These commands are run early in the deployment process, before the web server has been set up, and before your application code has been unpacked:

> The commands are processed in alphabetical order by name, and they run before the application and web server are set up and the application version file is extracted.[1]

By default, the commands run in the root user's home folder. This and various other pieces of EB's behavior can be changed via options (working directory, whether to continue on error, environment variables to pass to commands, etc.) that can be passed along with the command.

Container Commands

Container commands run after the application and web server have been set up and the application version archive has been extracted, but before the application version is deployed**.**

By default, these commands run in the staging folder, so that any changes you make to the current folder will persist once your application is deployed (the path will change though, so be careful about relative links!).

Container commands support all the same options as (non-container) commands, but they also support a "leader_only" option:

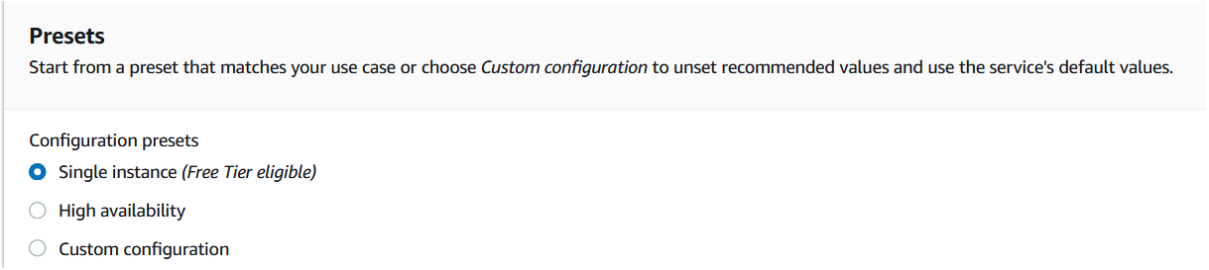
You can use leader_only to only run the command on a single instance, or configure a test to only run the command when a test command evaluates to true. Leader-only container commands are only executed during environment creation and deployments**, while other commands and server customization operations are performed every time an instance is provisioned or updated.**

You can use the container_commands key to execute commands that affect your application source code. Container commands run after the application and web server have been set up and the application version archive has been extracted, but before the application version is deployed.

```
container_commands:
  modify_index_html:
```

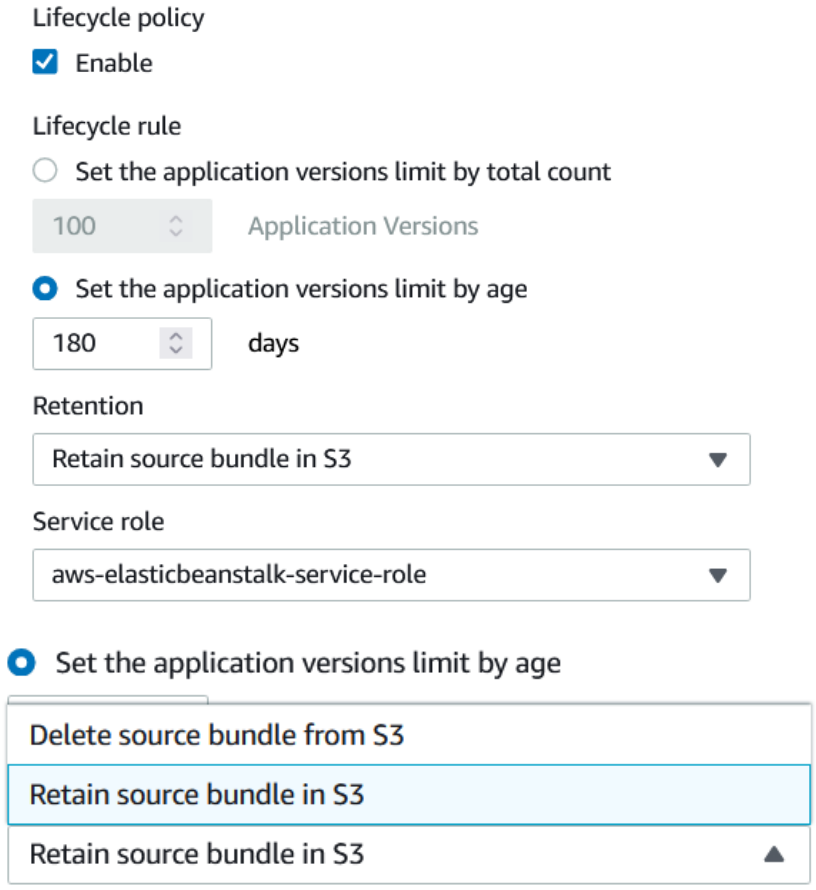
```
    command: 'echo " - modified content" >> index.html'

database_migration:
    command: 'echo "do a database migration"'
    # You can use leader_only to only run the command on a single instance
    leader_only: true
```

Important Notes:

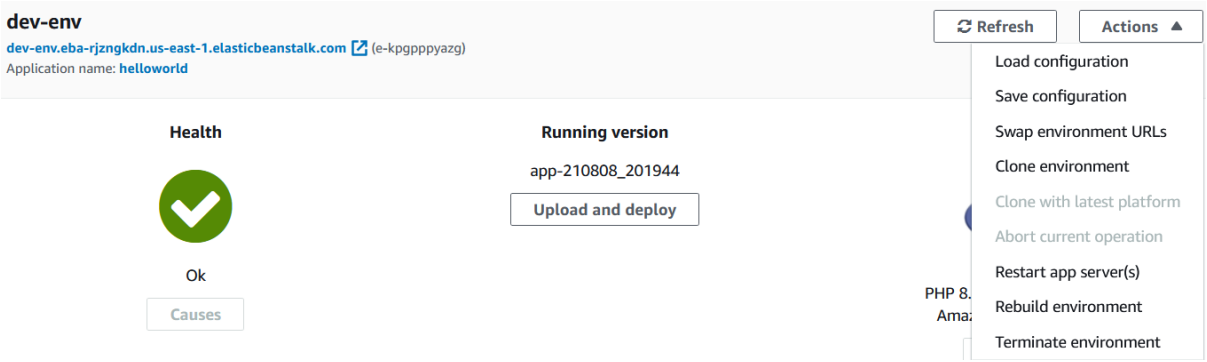


Application version lifecycle settings

Configure a lifecycle policy to limit the number of application versions to retain for future deployments. This policy will not delete application versions that are currently deployed or are in the process of being created.

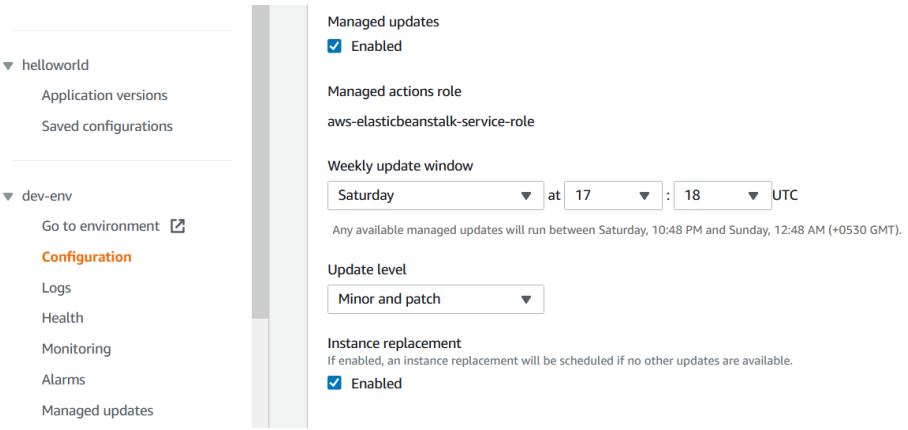


Rebuilding the environment may take several minutes, during which your application will not be available and **any attached Amazon RDS DB instance will be deleted**. If you want to save your data, create a snapshot before rebuilding your environment.



Managed platform updates

Enable managed platform updates to apply platform updates automatically during a weekly maintenance window that you choose. Your application stays available during the update process.

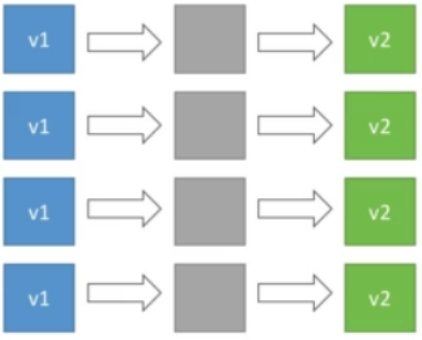


Deployment Modes

- **All at once (deploy all in one go)** – fastest, but instances aren't available to serve traffic for a bit (downtime)
- **Rolling:** update a few instances at a time (bucket), and then move onto the next bucket once the first bucket is healthy
- **Rolling with additional batches:** like rolling, but spins up new instances to move the batch (so that the old application is still available)
- **Immutable:** spins up new instances in a new ASG, deploys version to these instances, and then swaps all the instances when everything is healthy

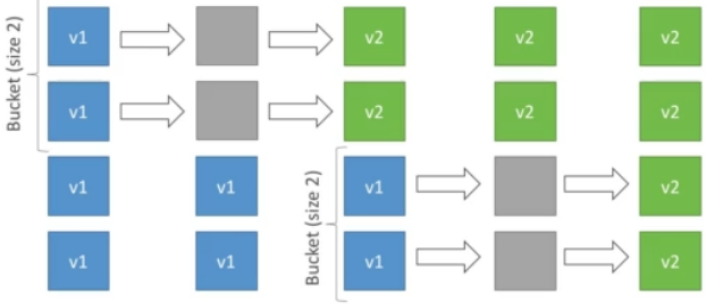
Elastic Beanstalk Deployment All at once

- 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

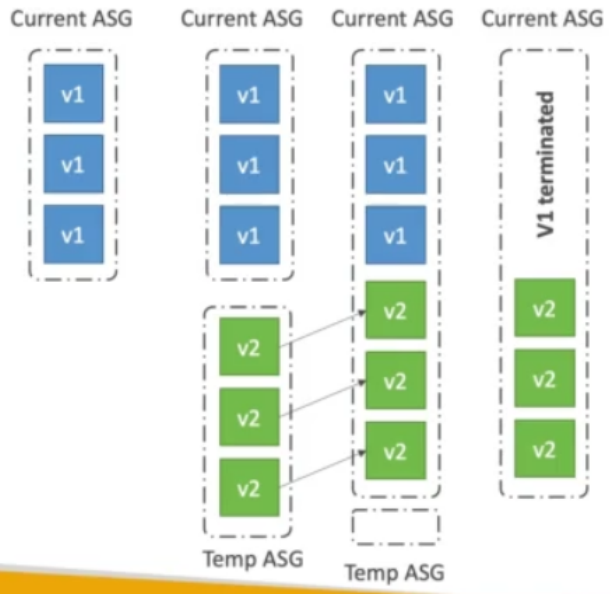


Elastic Beanstalk Deployment Rolling with additional batches

- 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

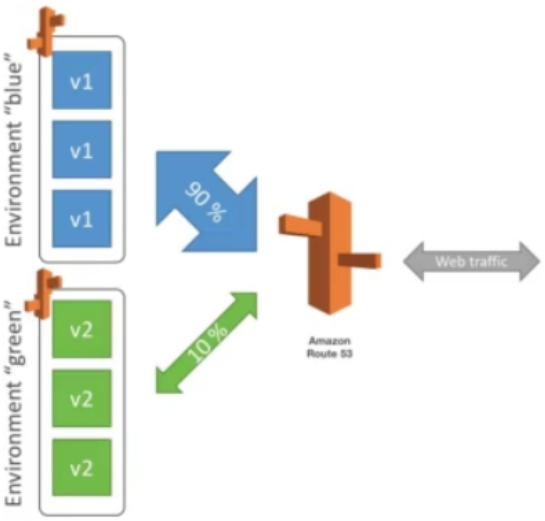
- Zero downtime
- New Code is deployed to new instances on a temporary ASG
- High cost, double capacity
- Longest deployment
- Quick rollback in case of failures (just terminate new ASG)
- Great for prod



In the temp asg, first beanstalk will launch single instance to validate it works, after it will launch the remaining.

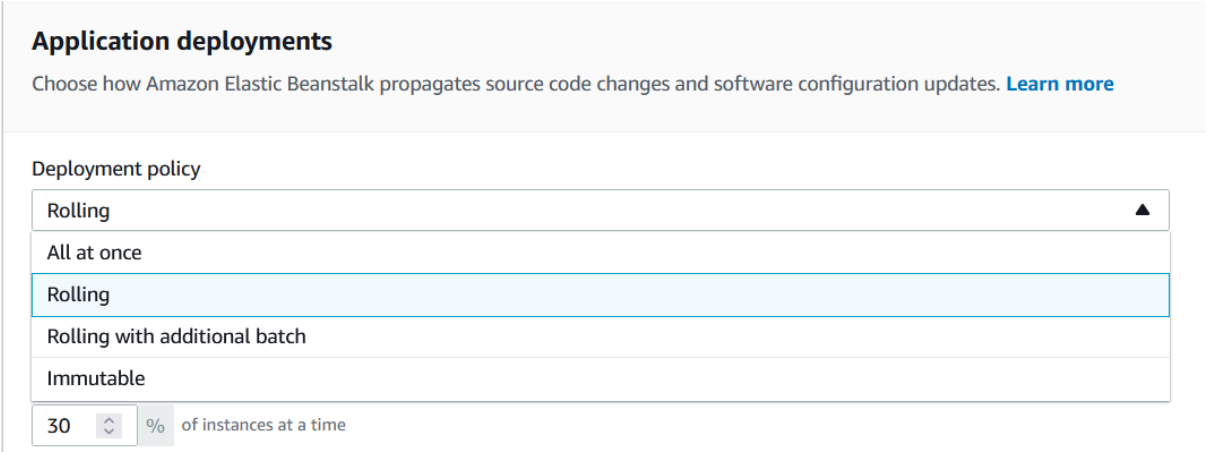
Elastic Beanstalk Deployment Blue / Green

- 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



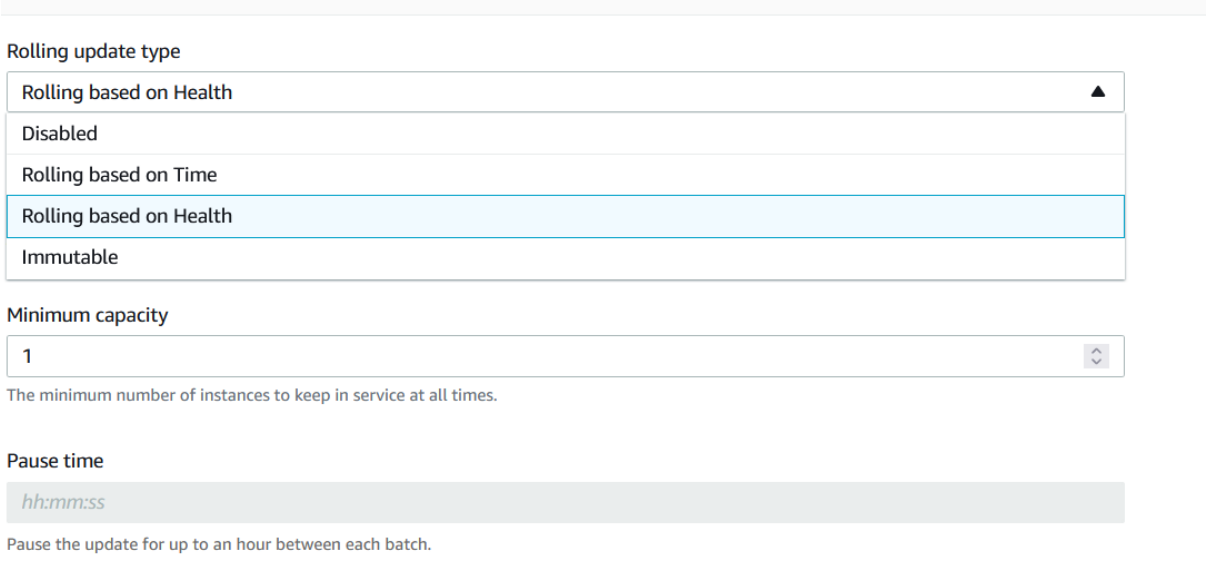
Deployment Methods

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 prior to failure running new application version	†	✓	✓	Manual Redeploy	Existing instances
Rolling with additional batch	Minimal if first batch fails, otherwise, similar to **Rolling**	†	✓	✓	Manual Redeploy	New and existing instances
Immutable	Minimal		✓	✓	Terminate New Instances	New instances
Blue/green	Minimal		✓	x	Swap URL	New instances



Configuration updates

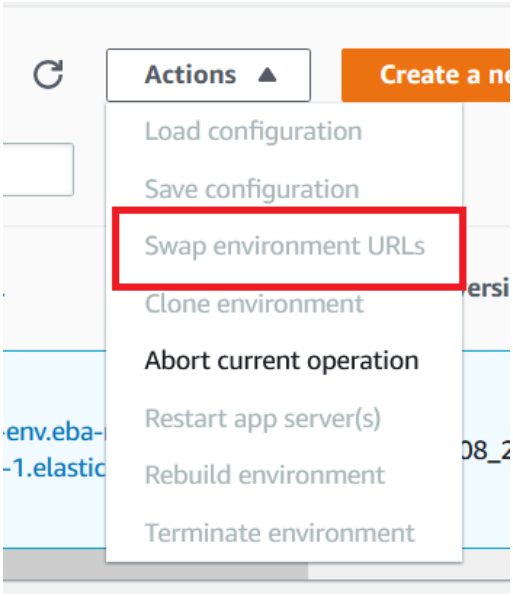
Changes to virtual machine settings and VPC configuration trigger rolling updates to replace the instances in your environment without downtime.



Blue/Green Deployment

```
Create new environments from saved configurations

```

```
# you can create environments from configurations

eb create prod-env --cfg prod
```



Select environment tier

Amazon Elastic Beanstalk has two types of environment tiers to support different types of web applications. Web servers are standard applications that listen for and then process HTTP requests, typically over port 80. Workers are specialized applications that have a background processing task that listens for messages on an Amazon SQS queue. Worker applications post those messages to your application by using HTTP.

Cron.yaml → to define scheduled tasks

Multi Docker Containers

A Dockerrun.aws.json file is an Elastic Beanstalk–specific JSON file that describes how to deploy a set of Docker containers as an Elastic Beanstalk application. You can use a Dockerrun.aws.json file for a multicontainer Docker environment.

A Dockerrun.aws.json file can be used on its own or zipped up with additional source code in a single archive. Source code that is archived with a Dockerrun.aws.json is deployed to Amazon EC2 container instances and accessible in the /var/app/current/ directory. Use the volumes section of the config to provide file volumes for the Docker containers running on the host instance. Use the mountPoints section of the embedded container definitions to map these volumes to mount points that applications on the Docker containers can use.

```
{
  "AWSEBDockerrunVersion": 2,
  "volumes": [
    {
      "name": "php-app",
      "host": {
        "sourcePath": "/var/app/current/php-app"
      }
    },
    {
      "name": "nginx-proxy-conf",
      "host": {
        "sourcePath": "/var/app/current/proxy/conf.d"
      }
    }
  ],
  "containerDefinitions": [
    {
      "name": "php-app",
      "image": "php:fpm",
      "environment": [
        {
          "name": "Container",
          "value": "PHP"
        }
      ],
      "essential": true,
      "memory": 128,
      "mountPoints": [
        {
          "sourceVolume": "php-app",
          "containerPath": "/var/www/html",
          "readOnly": true
        }
      ]
    },
    {
      "name": "nginx-proxy",
      "image": "nginx",
      "essential": true,
      "memory": 128,
      "portMappings": [
        {
          "hostPort": 80,
          "containerPort": 80
        }
```

```
      ],
      "links": [
        "php-app"
      ],
      "mountPoints": [
        {
          "sourceVolume": "php-app",
          "containerPath": "/var/www/html",
          "readOnly": true
        },
        {
          "sourceVolume": "nginx-proxy-conf",
          "containerPath": "/etc/nginx/conf.d",
          "readOnly": true
        },
        {
          "sourceVolume": "awseb-logs-nginx-proxy",
          "containerPath": "/var/log/nginx"
        }
      ]
    }
  ]
}
```

Add the information about the Amazon S3 bucket that contains the authentication file in the authentication parameter of the Dockerrun.aws.json file. Make sure that the authentication parameter contains a valid Amazon S3 bucket and key. The Amazon S3 bucket must be hosted in the same region as the environment that is using it. Elastic Beanstalk will not download files from Amazon S3 buckets hosted in other regions.