Cloud Computing Architecture

High Availability Patterns

Image licensed under creative commons

SWINBURNE UNIVERSITY OF TECHNOLOGY

High Availability Patterns

This presentation:

- High Availability Factors
- Reliability vs Availability
- More high availability patterns

Images licensed under creative commons.

High Availability Factors

Fault Tolerance:

The **built-in redundancy** of an application's components and its **ability to remain operational**.

Week 3
Multi-AZ networks

Recoverability:

The process, policies, and procedures related to **restoring service** after a catastrophic event.

This week

Scalability:

The ability of an application to **accommodate growth** without changing design.

Week 7 & 8
Scaling

Reliability vs. Availability

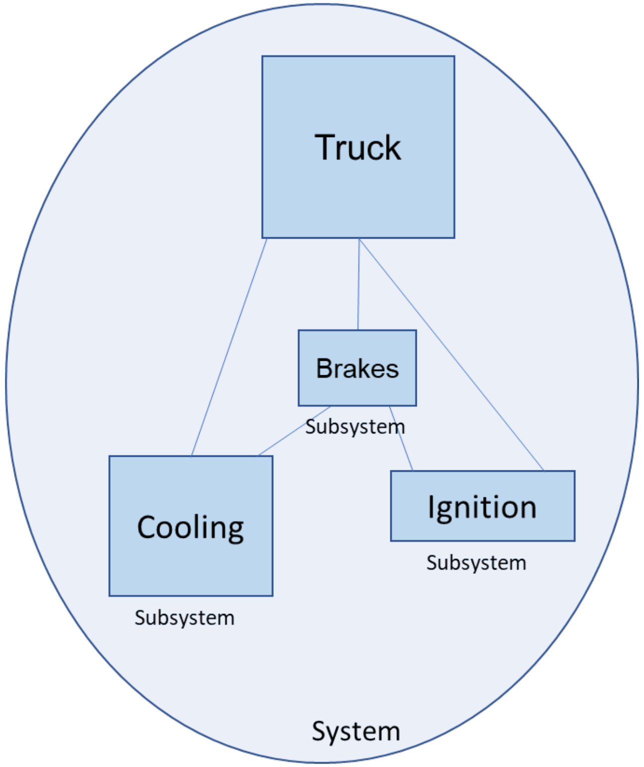
Reliability – A measure of **how long a resource/subsystem/system** performs its intended function.

Two common measures of reliability:

- **Mean Time Between Failure (MTBF)** – Total time in service/number of failures
- **Failure Rate** – Number of failures/total time in service

Availability – A measure of the **percentage of time** the resources are operating normally.

- A percentage of uptime (such as 99.9%) over a period of time (commonly a year).
- **Availability** – Normal Operation Time/Total Time
- **Common Shorthand** – Refers only to the number of 9s; for example, 5 nines is 99.999% available.



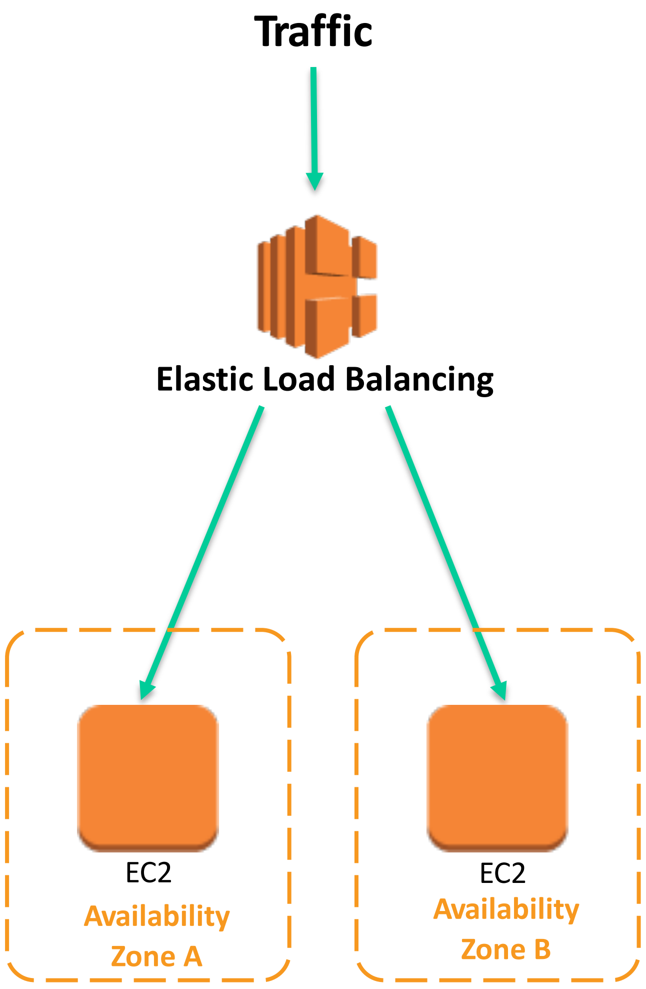
Multi-AZ Pattern

Pros:

- If an Availability Zone fails, the system is still available as a whole.

Implementation:

- Create an AMI for your instance.
- Spin up multiple instances using that AMI in multiple AZs.
- Create a load balancer in multiple AZs and attach the instances.
- Confirm instances are attached to load balancer and are in a healthy state.



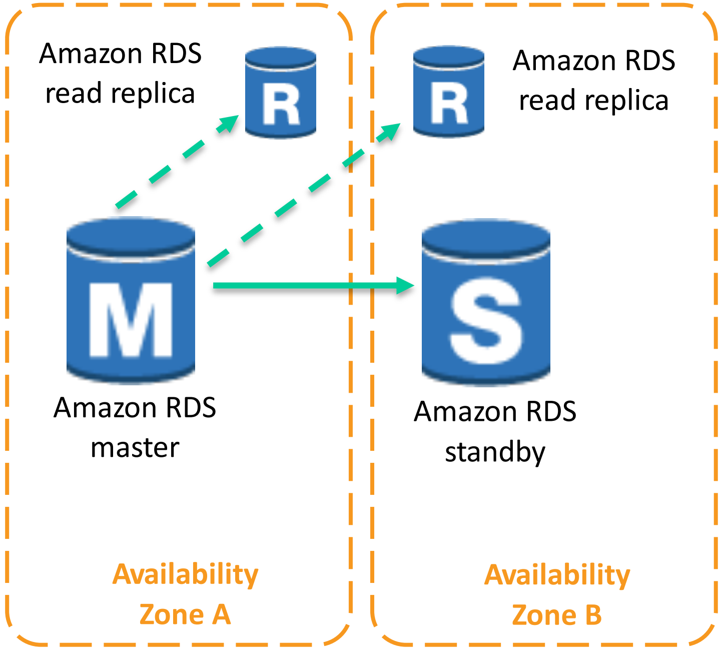
High-Availability Database Pattern

Pros:

- One connection string for master and slave with automatic failover.
- Maintenance does not bring down DB but causes failover.
- Read replicas take load off of master.

Implementation:

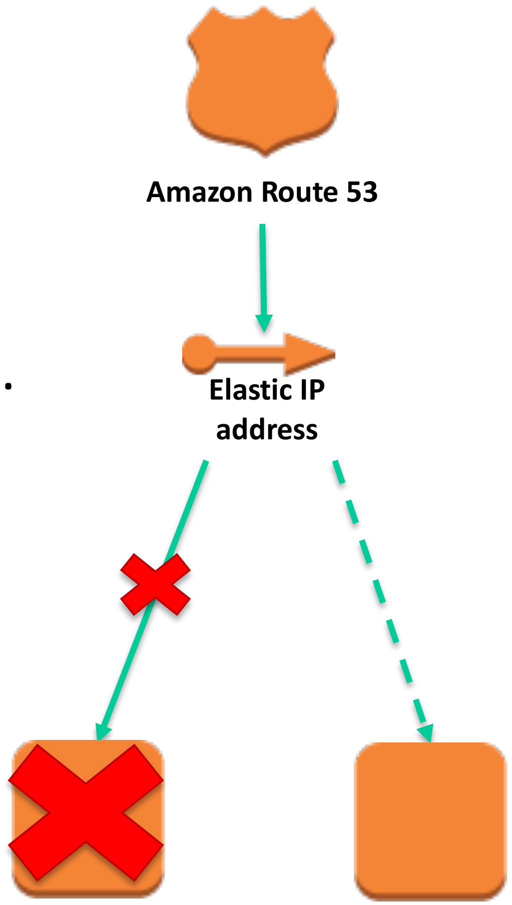
- Create an Amazon RDS instance (Aurora, MariaDB, MySQL, Oracle, PostgreSQL or SQL Server).
- Deploy in multiple Availability Zones.
- Create read replicas for each zone (Aurora, MariaDB, MySQL, or PostgreSQL).



Floating IP Address Pattern

Problem: Your instance fails or you need to upgrade it, so you need to push traffic to another instance with the same public IP address.

Solution: Use an Elastic IP address.



Pros:

- Since we are moving the Elastic IP address, DNS will not need to be updated.
- Fallback is as easy as moving the Elastic IP address back to the original instance.
- Elastic IP addresses can be moved across instances in different zones in the same region.

Implementation:

- Allocate the Elastic IP address for the EC2 instance.
- Upon failure or upgrade, launch a new EC2 instance.
- Disassociate the Elastic IP address from the EC2 instance and associate it to the new EC2 instance.

Floating Interface Pattern

Problem: When an instance fails or needs to be upgraded, traffic must be pushed to another instance with the same public and private IP addresses and the same network interface.

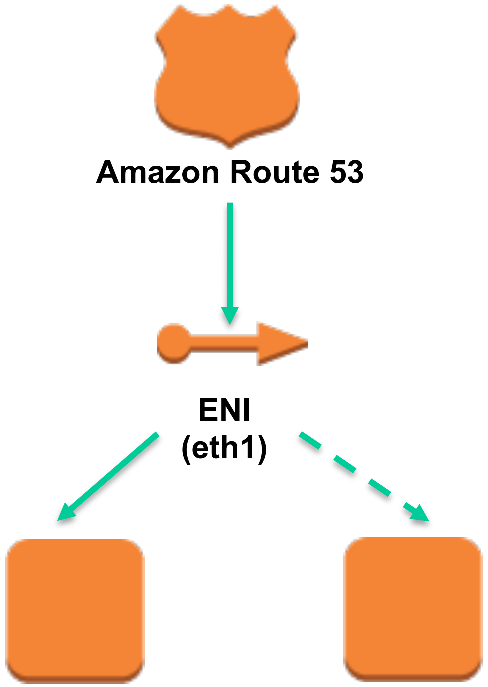
Solution: Deploy your application in VPC and use an elastic network interface (ENI) on eth1 that can be moved between instances.

Pros:

- DNS will not need to be updated.
- Easy rollback: move the ENI back to the original instance.
- Anything pointing to the public or private IP on the instance will not need to be updated.
- ENIs can be moved across instances in a subnet.

Implementation:

- Allocate the ENI for the instance.
- Upon failure or upgrade, launch a new instance.
- Detach the ENI from the instance and attach it to the new instance.



State-Sharing

Problem: You want your application to be **stateless** in order to better scale horizontally.

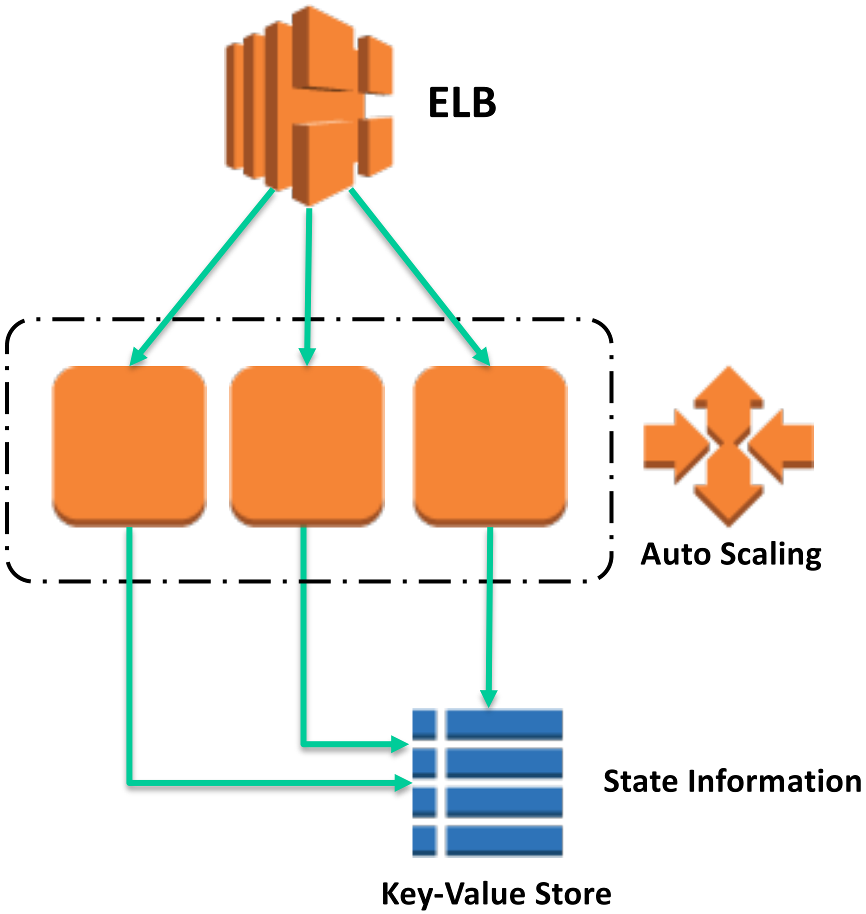
Solution: Move state off your server into a **key-value store.**

Pros:

- This lets you use the scale-out pattern without having to worry about inheritance or loss of state information.

Implementation:

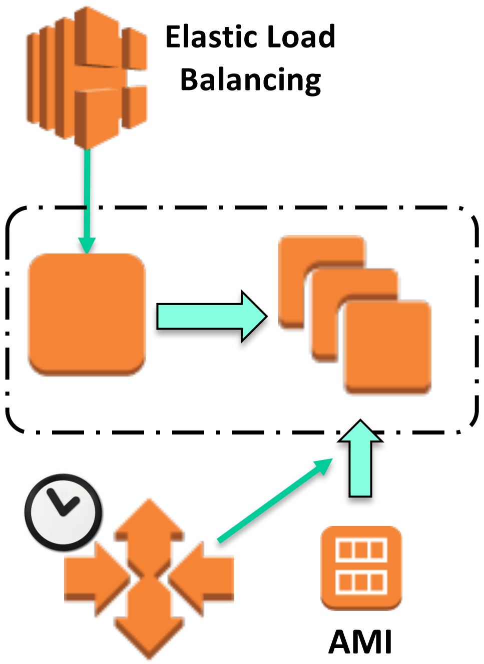
- Use Amazon ElastiCache and DynamoDB for data storage.
- Prepare a data store for storing the state information.
- Use, as a key in the data store, an ID that identifies the user (a session ID or user ID), and store the user information as a value.
- Store, reference, and update the state information in the data store, instead of storing it in the web/APP server.



Scheduled Scale-Out

Problem: An application's traffic does not scale organically, but has huge jumps at specific periods of the day or for an event.

Solution: Use **Scaling by Schedule** or **Scaling by Policy**.



Advantages:

- Scale in advance of a traffic spike you know will occur.

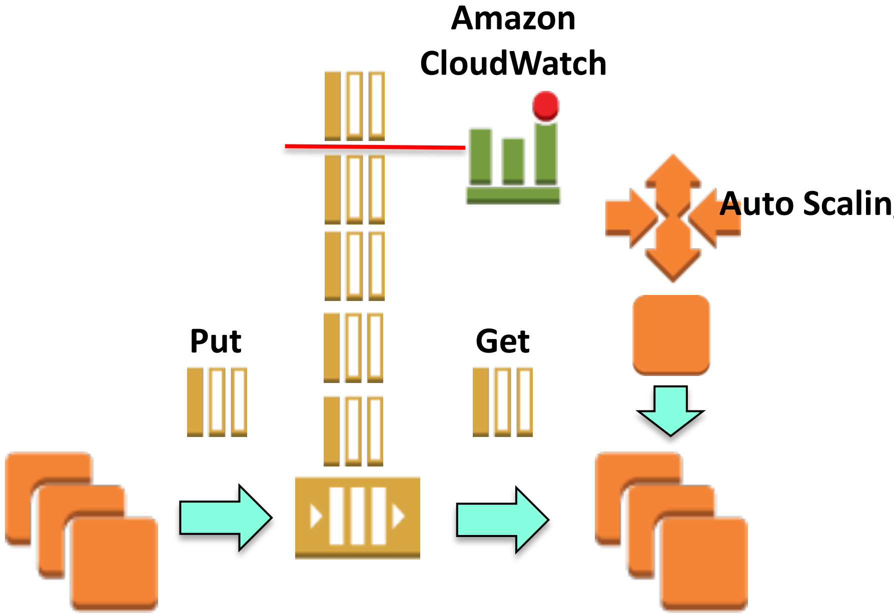
Implementation:

- Create a customized AMI.
- Create a Launch Config for your Auto Scaling group.
- Create an Auto Scaling group for your instances (behind a load balancer).
- Options:
  - Create Schedule Update to launch or terminate instances at a specified time.
  - Create Scale by Recurrence policy that will automatically scale your instances based upon cron.

Job Observer Pattern

Problem: You need to manage resources against the depth of your work queue.

Solution: Create an Auto Scaling group to scale compute resources based upon queue depth.



Pros:

- Compute scales with job size, providing efficiency and savings.
- Job can be completed in a shorter timeframe.
- Even if a work item fails to complete, process is resilient.

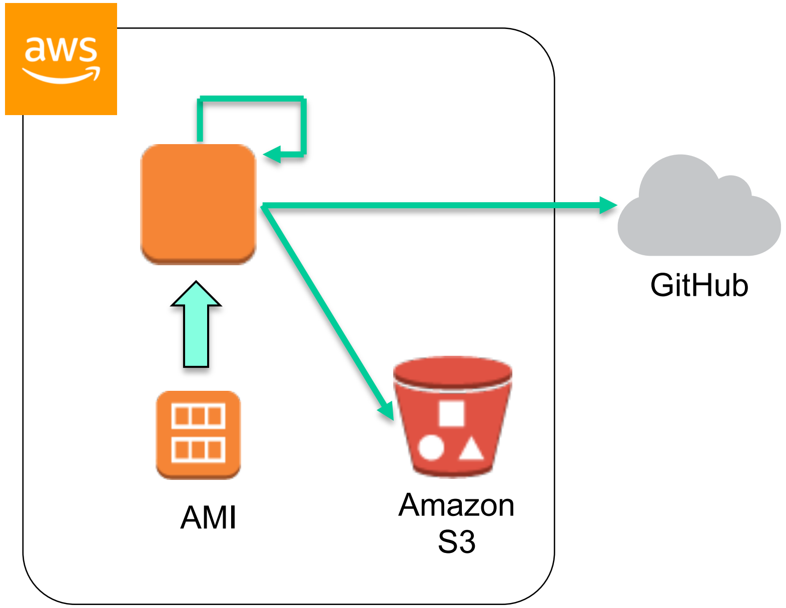
Implementation:

- Work items for batch job placed in Amazon SQS queue as messages.
- Auto Scaling group should be created to scale compute resources up or down based upon Amazon CloudWatch queue depth metric.
- Batch processing servers retrieve work items from Amazon SQS to complete job.

Bootstrap Instance

Problem: Code releases happen often. Creating a new AMI every time you have a release and managing these AMIs across multiple regions is difficult.

Solution: Develop a base AMI, and then bootstrap the instance during the boot process to install software, get updates, and install source code so that your AMI rarely or never changes.



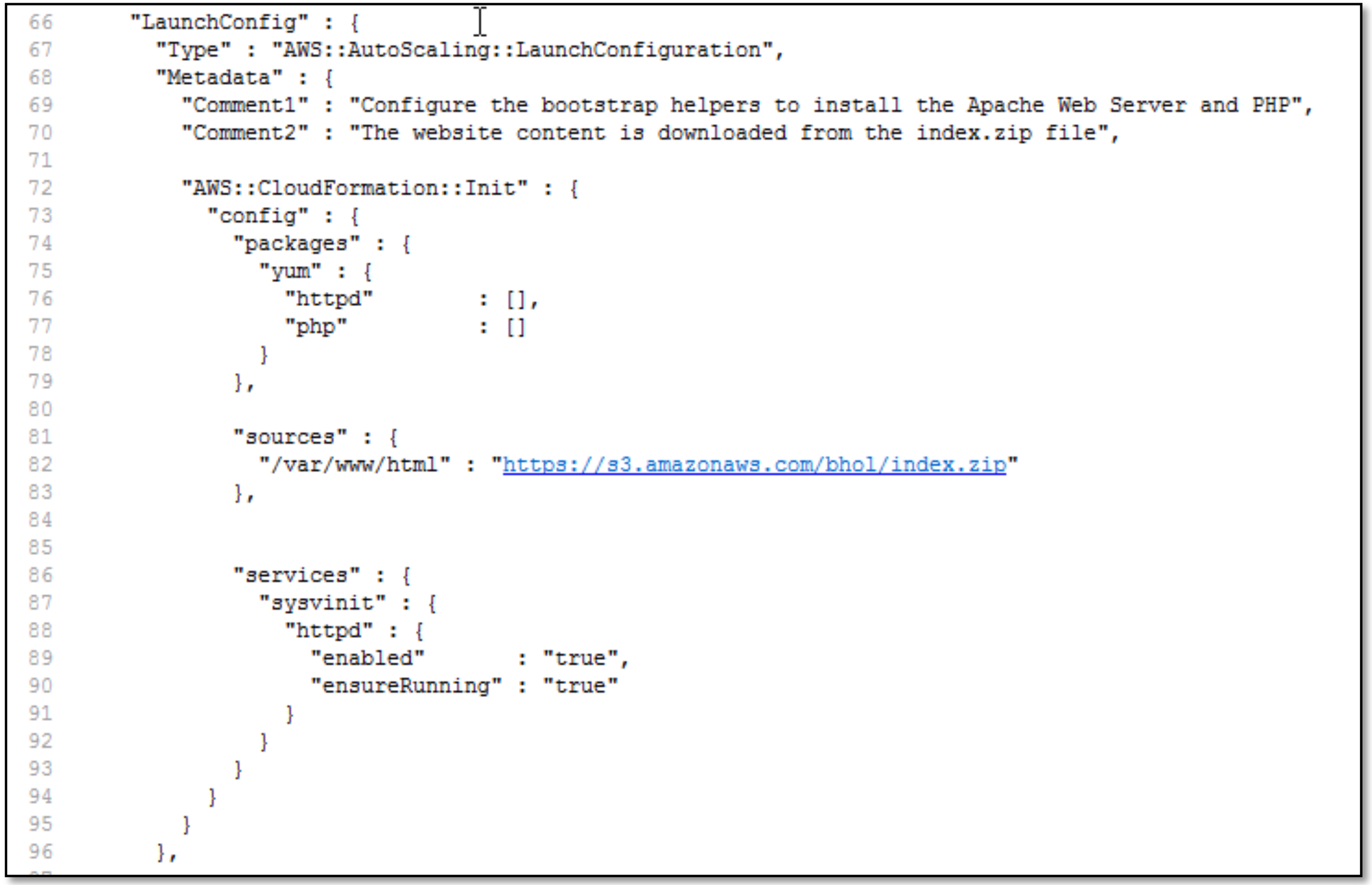
Pros:

- Do not need to update AMI regularly and move customized AMI between regions for each software release.

Implementation:

- Identify a base AMI to start from.
- Create a repository where your code is located.
- Identify all packages and configs that need to occur at launch of the instance.
- During launch, pass user data to your EC2 instances that will be executed to bootstrap your instance.

Bootstrap Instance: Example

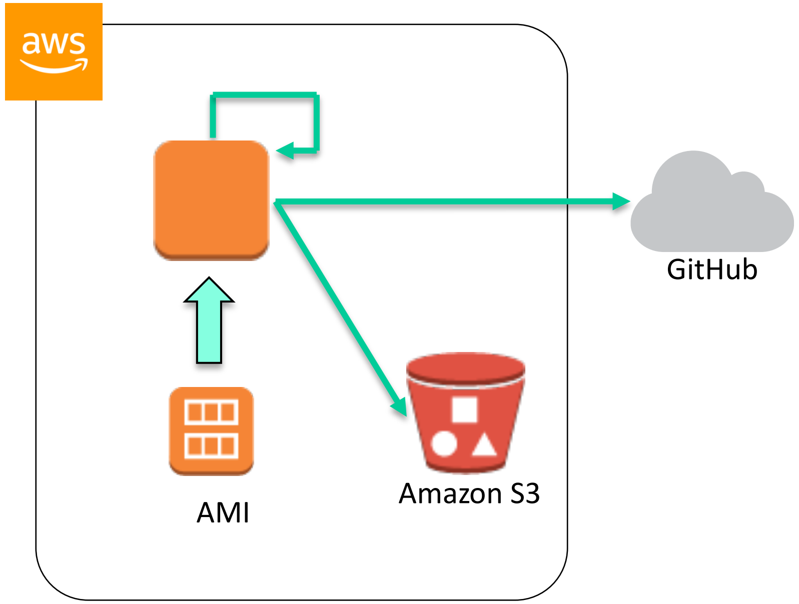


```
    "LaunchConfig" : {
      "Type" : "AWS::AutoScaling::LaunchConfiguration",
      "Metadata" : {
        "Comment1" : "Configure the bootstrap helpers to install the Apache Web Server and PHP",
        "Comment2" : "The website content is downloaded from the index.zip file",

        "AWS::CloudFormation::Init" : {
          "config" : {
            "packages" : {
              "yum" : {
                "httpd"        : [],
                "php"          : []
              }
            },

            "sources" : {
              "/var/www/html" : "https://s3.amazonaws.com/bhol/index.zip"
            },


            "services" : {
              "sysvinit" : {
                "httpd" : {
                  "enabled"       : "true",
                  "ensureRunning" : "true"
                }
              }
            }
          }
        }
      },
```


aws
GitHub
AMI
Amazon S3