1. How would you configure the infrastructure so it grows automatically when more resources are needed?

In order to achieve high availability and autoscaling we can run the web application on AWS and can use services, features such as elastic load balancer (ELB) & auto scaling group. This kind of setup will share the load across all app instances running behind the load balancer and in the event of instance failure or high load automatically new instance will be launched by cloudwatch and start serving the request. Additionally we can use framework like Kubernetes, it will not only help us to manage & scale our web application but also can create more container in the event of high CPU, query per second metric as mentioned below.

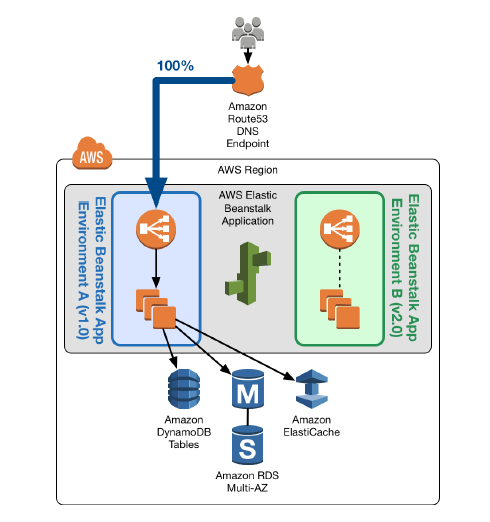
1. What will be the triggers to decide when the infrastructure has to add more resources?

To automatically trigger more instances in the AWS we can use cloudwatch alarm which will keep monitoring the resources such as CPU, Disk io, network throughput etc.. Also we can create our custom metric (memory, swap, disk usage etc.) which help take appropriate action in the event of any alarm generated or the resource utilization breach the threshold limit.

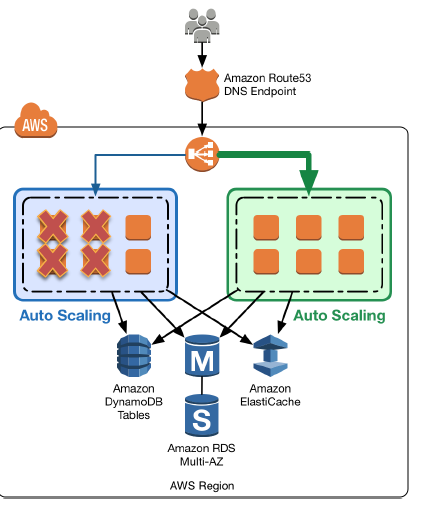
For autoscaling in Kubernetes we can use HorizontalPodAutoscaler controller which queries Heapster directly through the API server’s service proxy subresources and collect the metric which can be used to autoscaling the pods based on CPU, Memory (creating custom script) usage, QPS (query per second)metric. Replica controller help in maintaining the state of number of pods running in the cluster and in the event of failure of any pod it trigger the pod so that the desired replica set can be maintained.

1. How will be the deployment process for new releases of the app?

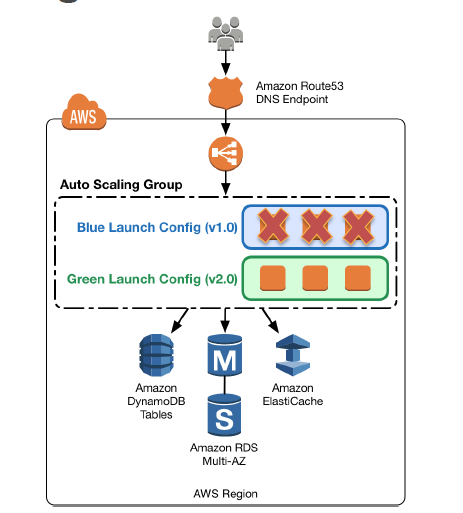
There are multiple ways we can perform the deployment of our new version of application. And some of the popular ones are blue green deployment and A/B testing. In AWS we can use elastic beanstalk for the deployment of new version of application and while doing such deployment there are chances that application will not be available to user for short duration of time but this can be avoid by deploy new version of code with old one and then swapping the CNAMEs of the two environment. This type of deployment know as classic DNS cutover where we uses route 53 (weighted DNS Switching) to transfer the small percentage let’s say 10 % of load to our new app version and help in monitoring the status of how app is behaving . And once we sure that the new app version is working as expected we can transfer whole traffic to new version.



Other way we can deploy is by swapping autoscaling group running behind the elastic load balancer (ELB) and here we create our new autoscaling group running with new version of application code. Then perform some testing such connectivity with backend database and once we are ok with that register the new ASG with the ELB and detach the old autoscaling group.



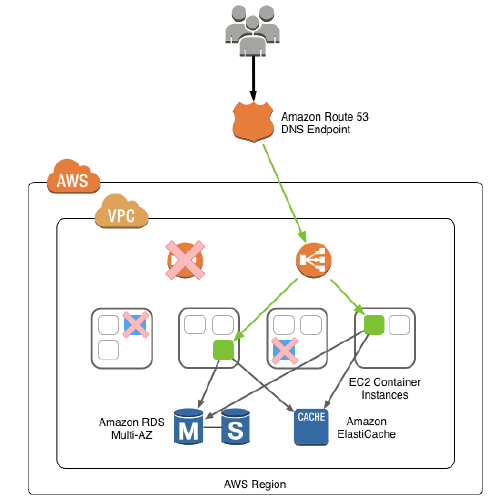
Third , we can swap launch config deployment process where we attach new launch config to the auto scaling group and grow its size to original size thus we have new instance running with new app and old instance will get terminated.



All this case can be applicable to the application running ECS container services.

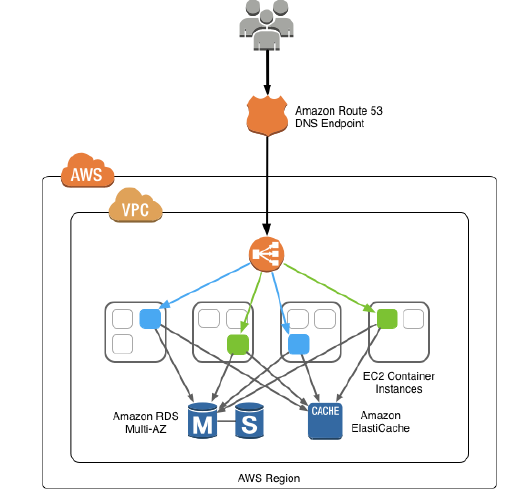
1. Blue/green ECS services, DNS update -

First create the new task definition based on new version of Docker image and new ELB. Then create the new green service with new task definition and once new environment ready update the route 53 alias record to direct traffic to new ELB endpoint and delete the old blue services resources.



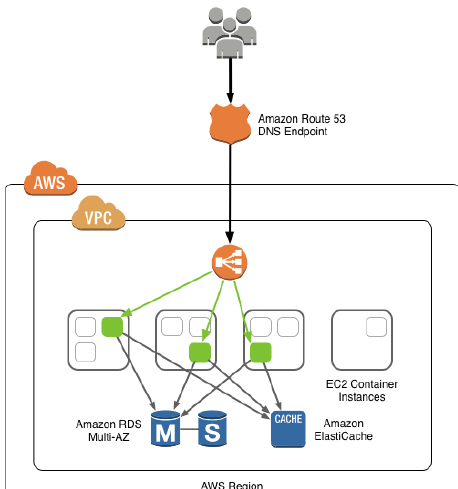
2. Blue/green ECS services, shared ELB

Create new task definition container new Docker version of app. After that create green service with new task definition and then map it to the existing ELB , scale the up green service by increment number of tasks. Now you have new version of app version running behind ELB and finally you can reduce task for old service to 0.

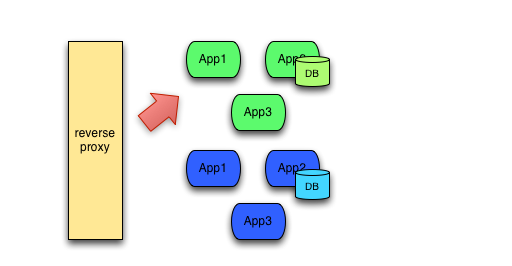


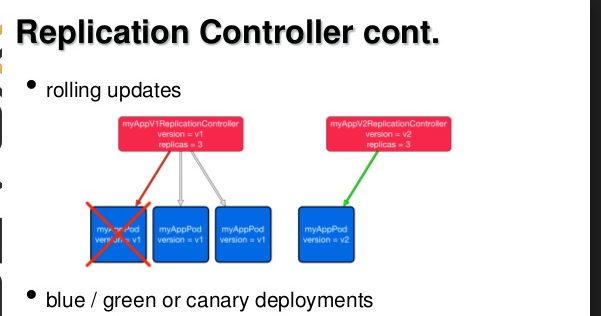
3. ECS update

Here we will update the only the task definition in the existing running service definition and automatically ECS will deploy the new task definition to container instance in the rollback fashion.

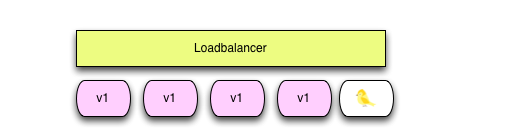


In Kubernetes deployment can be achieved using updating the image of running deployment or by update services with command patch, rolling update by mentioning the deployment version. We can use haproxy for the same by updating the configuration and pointing it to or adding the new environment.





Other method we can use is by using the Ansible Continuous Delivery and Rolling Upgrades. This method allows us to achieve new application deployment or upgrade with no downtime. Here we will use the Ansible playbook to deploy the new app version in incremental manner on each server rather than shifting whole traffic to new infrastructure which allow us to do testing for our code on few group of servers and once we are fully sure about it then we can deploy the across all nodes. Playbook contain list of hosts on which new app need to deploy and change (by defining serial parameter) will be implemented serial fashion rather than on all nodes. This methodology helps us to check if things are expected on new app version as expected. When we run the Ansible playbook it will first disable the node monitoring and then remove from load balancer after that new app deployed and all necessary health check defined in playbook will be performed to make sure it is working fine. Once it passes through all checks the node will bring under monitoring and then under the load balancer thus new app start serving the traffic. Keep monitor the new application server for some time and if looks ok then we can implement the same change across all the nodes. Here we apply changes to the running infrastructure and there is no need to create new duplicate infrastructure like in the case of blue-green deployment. In case if things not working as expected we can rollback to previous app version and we can trigger the rollback using Ansible playbook.



1. How do you monitor the new infrastructure if something stops working? In which situations can it happen?

AWS provide the cloudwatch service that help to monitor the instances resources and in case of high load, traffic it will trigger the alert which we can use to scale up/down the instances. Talking about the elastic load balancer health check so it keep monitoring the running application on port number defined in it so whenever elb does not receive any response from application instance it mark as unhealthy and will not send traffic to it . Such situation can be arise if application get crashed or high load on the instance and server not able to respond to the request etc.. We can also configure ELK stack & fluent to gather the data such as application, OS logs which we can use it for online analysis related to error, user request , traffic flow etc..

In Kubernetes we can use graphana, influxdb, cadvisor , heapster to generate the graph from metric generated at container and host level. Also we can add prometheus to gather detail statistics and send alert based on rules defined from infrastructure which help to monitor services, CPU, memory , storage load, network analysis and many more. During high load on the container , it stop responding and running service get crashed due to insufficient resources. We can configure Liveness/Readiness Probe Failure for monitor of our application running pod. In case our application is not responding /liveness probe failed Kubernetes will kill the container and create new one. If the readiness will fail that pod will not be available as service means no traffic will be sent to that pod until it become ready.

1. What kind of hardware is the optimal one? (In terms of HD type, RAM, ..) What is the most relevant for this kind of service?.

AWS manages its own hardware so no need to buy any hardware and based on your application you can choose instance type , database and other services from AWS. For Kubernetes we need at least 3 or 5 node cluster for etcd configuration and following is the standard specs . All these requirement applicable for cloud and bare metal infrastructure.

* A small cluster serves fewer than 100 clients, fewer than 200 of requests per second, and stores no more than 100MB of data. Example application workload: A 50-node Kubernetes cluster

PROVIDER TYPE VCPUS MEMORY (GB) MAX CONCURRENT IOPS DISK BANDWIDTH (MB/S)

AWS m4.large 2 8 3600 56.25

GCE n1-standard-1 + 50GB PD SSD 2 7.5 1500 25

* A medium cluster serves fewer than 500 clients, fewer than 1,000 of requests per second, and stores no more than 500MB of data. Example application workload: A 250-node Kubernetes cluster

PROVIDER TYPE VCPUS MEMORY (GB) MAX CONCURRENT IOPS DISK BANDWIDTH (MB/S)

AWS m4.xlarge 4 16 6000 93.75

GCE n1-standard-4 + 150GB PD SSD 4 15 4500 75

1. What type of backups do you recommend? How would you implement them?

On AWS take the EBS volume backup also schedule the backup for database as per the requirement. So we can create the custom script to take the EBS backup and keep testing the restore when everything is working fine. We can also use the Glacier service to keep backup for longer period of time .Keep copy of application & database out of cloud to have another copy of data for disaster recovery. There is third party software available to take the backup from cloud which can be one of the options. In Kubernetes we can configure the script to take the configuration & data directory which is persistent across the pods and put in other secured place such as tape drive, external connected storage etc..

1. How would you confirm that the new setup is supporting the amount of traffic it is intended to support?

Perform the test and before make it as production setup.

Following is the detail about test need application need to pass through:

a. Performance Test: Do performance which make sure that the application is stable, perform well, scalability and/or show the throughput of web application.

b. Capacity Test: Perform capacity test is a test to determine how many users your application can handle before either performance or stability becomes unacceptable. By knowing the number of users the application can handle “successfully”, you will have better visibility into events that might push your site beyond its limitations. This is a way to avoid potential problems in the future.

c. Load Test: A load test consists of applying load to an application and measuring the results. The load may or may not be at the high end of application capacity. These tests can help determine normal performance metrics. By using iterative testing, we can determine whether new code has helped or hurt performance.

d. Stress Test: Then perform stress test that pushes an application beyond normal load conditions. When you push your application to the extreme, you will see which components fail first. Making these components more robust, or efficient, will help determine new thresholds.

e. Soak Test: A soak test is a long-running test that is used to determine application performance and/or stability over time. An application may work well for an hour or two, and then start to experience issues. These tests are especially useful when trying to track down memory leaks or corruption.

f. Component Test: Testing a discrete component of your application requires a component test. This might include a search function, a file upload, a chat feature, an email function, or a 3rd-party component like a shopping cart.

e: Security Test: Security check help in determining any bug in our security of application code or web content is vulnerable to external threats.

Practical challenge: Create environment for Rails app



1. To deploy the package I have created the Ansible playbook and in the hosts file we can define myapp.daliaresearch.com and add entry in /etc/hosts file for remote host ip address.
2. Also I have used the simple shell script to execute the heroku deployment task which exist under director setup/files/ called install.sh
3. I done testing of this playbook on Centos 7 :

[heroku@chef heroku]$ cat /etc/redhat-release

CentOS Linux release 7.2.1511 (Core)

[heroku@chef heroku]$ uname -a

Linux chef.server.com 3.10.0-327.36.3.el7.x86\_64 #1 SMP Mon Oct 24 16:09:20 UTC 2016 x86\_64 x86\_64 x86\_64 GNU/Linux

[heroku@chef heroku]$

1. Ruby version I have installed on my virtual machine is 2.0.0. So change variable ruby version in file under the path setup/vars/myvars.yml according to your version.

[heroku@chef heroku]$ ruby -v

ruby 2.0.0p598 (2014-11-13) [x86\_64-linux]

[heroku@chef heroku]$

5) To create the heroku environment simply run below command after login into heroku directory attached here:

ansible-playbook -i hosts site.yml

6) To run heroku simply login as heroku user after ansibe playbook complete and run heroku local

[heroku@chef heroku]$ heroku local

[WARN] No ENV file found

8:36:49 AM web.1 | [5462] Puma starting in cluster mode...

8:36:49 AM web.1 | [5462] \* Version 2.11.1 (ruby 2.0.0-p598), codename: Intrepid Squirrel

8:36:49 AM web.1 | [5462] \* Min threads: 5, max threads: 5

8:36:49 AM web.1 | [5462] \* Environment: development

8:36:49 AM web.1 | [5462] \* Process workers: 2

8:36:49 AM web.1 | [5462] \* Preloading application

8:36:51 AM web.1 | [5462] \* Listening on tcp://0.0.0.0:5000

8:36:51 AM web.1 | [5462] Use Ctrl-C to stop

8:36:52 AM web.1 | [5462] - Worker 1 (pid: 5467) booted, phase: 0

8:36:52 AM web.1 | [5462] - Worker 0 (pid: 5465) booted, phase: 0