Building High Available Web Application in the Cloud

Evolving From a Static Web Page to Microservice-Oriented App on AWS

Bachelor-Thesis

zur Erlangung des akademischen Grades B.Sc.

Taras Sologub

2153832

Hochschule für Angewandte Wissenschaften Hamburg
Fakultät Design, Medien und Information
Department Medientechnik

Erstprüfer: Andreas Plaß

Zweitprüfer: Thorben Gülck

Hamburg, 21. 10. 2019

Contents

Abstract

This is a practical work for developing, managing, and deploying resilient, highly available web applications on Amazon Web Services. First, I will introduce some core concepts of AWS and continue with an in-depth overview of central services and frameworks. In the first practical chapter, I'll create a static web page, make a cost analysis, and set up manual deployment. The second one will migrate the previous approach to microservice-oriented architecture. Lastly, I will make an introduction to continuous integration and set up automatic deployments with an automation delivery pipeline.

Keywords: AWS, static website hosting, microservices, continuous integration.

Zusammenfassung

Dies ist eine praktische Arbeit für die Entwicklung, Verwaltung und Bereitstellung robuster, hochverfügbarer Webanwendungen auf Amazon Web Services. Zunächst werde ich einige Kernkonzepte von AWS erleutern und mit einem detaillierten Überblick über die zentrale Dienste und Frameworks fortfahren. Im ersten praktischen Kapitel erstelle ich eine statische Webseite, führe eine Kostenanalyse und richte eine manuelle Bereitstellung ein. Der zweite Kapitel wird den bisherigen Ansatz zur mikroservice-orientierten Architektur migrieren. Schließlich werde ich eine Einführung in die kontinuierliche Integration geben und automatische Bereitstellungen mit einer Automation Delivery Pipeline einrichten.

Schlüsselwörter: AWS, statisches Website-Hosting, Mikroservices, kontinuierliche Integration.

1 Introduction

"Invention requires two things: 1.The ability to try a lot of experiments, and 2.not having to live with the collateral damage of failed experiments"

- Andy Jassy, *CEO Amazon Web Services*

1.1 Motivation and Goals

Very few people realize that the concept of *Cloud Technology* is not an unattainable sphere of IT, but what many people use every day.

In the early days of the internet, if you wanted a server - you need to run it yourself. You had to buy one, set it up, power up, connect it to the internet, and keep it running 24/7. There was a lot of work involved just to manage a server. Later, people realized that they could massively outsource this - data centers came in a really big way. You would contact a data center, go to them, request a new server and they will set up a couple of racks for you to rent. There will be a slow time of a few days even weeks, to get these servers provisioned.

At first, they offered only the infrastructure support and provided servers with power, cooling, electricity. But later, people have requested to support the operating system, the application layer, so the data center companies started going deeper and deeper into it. With that said, there is still a significant upfront investment. You have to pay for those servers, sign-in the contracts. They are buying hardware for you, and want to make sure that you locked in for a couple of years. That is a relatively expensive way to access computing power.

Fast forward to 2006, Amazon comes out and launches its first cloud platform - Elastic Compute Cloud. The benefit of it was, you didn't have to worry about providing electricity, the internet, physical security - they managed all that for you. You are outsourcing a physical hosting of the machine to someone else, and that changed the world. You make an API call and have a virtual machine up and running within seconds. You don't need to think about it in terms that you've used to think about it - a physical computer. All you do now is give your code to a cloud provider and say, "Run, whenever the customer needs something." Providers deal with challenges of scaling that and operations. They provide you, as a customer, with guarantees for uptime and reliability, e.g., Service Level Agreement (SLA), so you can trust them to run your code. It means that you can focus on building software systems and don't have to think about employing staff to manage those computers.

Has the server run out of the space? Is the database full? Cloud is where the world is going.

If you are not using the Cloud, you can host your websites on your servers, in your own data centers, even on your PC. Of course, if you want a web site to be highly available, you have to build a whole bunch of redundancies. Make sure to have a redundant power supply, redundant networking. It is a lot easier and cheaper to put things in the Cloud, let somebody else handle that for you. Competition is driving industries. Companies become more cost-effective, more agile - that is driving a lot of businesses to invest in those technologies.

This is a practical work about building a highly available application on the AWS. This project will take us through a maturity process from developing a basic web-page to an extremely cheap scalable microservice web application with automatic deployments and delivery pipeline. Throughout the development, we will be using a significant majority of AWS services; some of them will be replaced in favor of other promising alternatives. The main concepts we will learn, one can apply to other cloud providers as well.

Therefore, the main goals of this project can be stated as :

1. Create a static web page with the EC2 Auto Scaling Group behind a load balancer.
2. Calculate the monthly cost for running the web page on the AWS.
3. Set up a deployment with Terraform scripts.
4. Redesign the web page with microservice-oriented architecture powering by Kubernetes under the hood. Register a domain name, persist the data in the database, and enable static content serving from the S3 bucket.
5. Deploy the web application on the Kubernetes cluster.
6. Automate the deployments with Jenkins delivery pipeline.

Chapter 2 gives a 10.000 feet overview of what Amazon Web Services is. Though this thesis is not about AWS theory, I want a reader to understand the whole picture and give an introduction to some of the core terminologies: Regions, Availability Zones, and pricing models. In the second part, I cover the central services and frameworks that we are going to be using throughout the project. Explain basic concepts and ideas, pros and cons, tradeoffs behind each of them.

Chapter 3 introduces the static web hosting implementation. I give guidance on how to develop a static web page on EC2 instances behind the load balancer and Auto Scaling Group. Next, we calculate the cost for running the web site 24/7 for the whole month and how to set up a deployment with Terraform configuration.

Chapter 4 introduces Microservices. Explains the downsides of a good-old monolith and why microservices became almost a defacto in software engineering. Next, we walk through the building blocks of our architecture, covering all the services, one by one. We retire Static Web Hosting implementation and build *Stock-of-the-Day*

microservice-oriented web application with new features by splitting the functionality into five separate services. Then, we determine a couple of key metrics and make a cost analysis for hosting this chapter in the Cloud. In the end, I explain different zero-downtime deployment models and bring our microservice app to the production on the Elastic Kubernetes Service cluster.

Chapter 5 emphasizes the importance of automated delivery and gives an introduction to continuous integration. We automate the deployments with Jenkins delivery pipeline in a step-by-step manner and show how to bring a single feature to the production via seven deployment stages. We take one of our services and construct an end-to-end delivery pipeline with Jenkins.

In chapter 6, I will discuss the results of my project and propose some directions for future work.

2 Amazon Web Services in a Nutshell

2.1 General Overview

Amazon Web Services is the subsidiary of Amazon that offers a broad set of global cloud-based products including storage, databases, compute, networking, analytics, security, developer tools, and a lot more. AWS is a way of establishing a worldwide data presence without the cost of building a cloud. As of 2019, it is a market-leading cloud platform.(Fig. 2.1) It is used almost by 80% of Fortune 500 Companies, to host their infrastructure.

Figure 2.1: Gartner: Magic Quadrant for Cloud Infrastructure as a Service

AWS has its global infrastructure around the world, any geographic location, where they have data centers - is called **Region**. In each region, there is a minimum of two data centers or clusters - **Availability Zones**(AZ). They are physically isolated from each other but have the same private network. And connected with fully redundant metro fiber which provides high bandwidth and low-latency networking. At the time of writing this thesis, there are in total of 21 geographical Regions and 66 Availability Zones.

For my project, I will assume a potential user base is located in Europe, which consists of 5 regions: Frankfurt, Paris, Stockholm, London, and Ireland. I will deploy my application in Frankfurt (eu-central-1) within its three AZs (eu-central-1a, eu-central-1b, eu-central-1c).

AWS Cloud Services are priced differently from traditional software and hardware - a utility-based model. As I already mentioned before, users do not need to invest in hardware infrastructure before they know the demand. You pay only for what you consume.

Amazon Services pricing provides four significant benefits. **Pay as you go** - you pay only for the services you need and only for as long as you need them. **Reduce costs** - it helps the clients to reduce significant upfront costs, such as facilities and hardware. **Optimized planning** - with the services model, users can plan and efficiently change capacity requirements.

AWS pricing model consists of three fundamental components: **compute** (you pay for compute and processing power by the hour), **storage** (you pay per GB) and **data transfer** (you pay per GB for the data out and there are no charges for incoming data).

2.2 Services and Frameworks

AWS offers more than 100+ services to its customers. To talk about all of them would be out of the scope of this thesis. To keep this short, I want to introduce services that we will be using heavily throughout this project. Talk about their strengths, use cases, pricing models, and gotchas to be aware of.

IAM. Identity and Access Management allows users to manage access to compute, database, storage, application services in the AWS Cloud. It uses access control concepts that we all are familiar with: **users** (services or people that are using AWS), **roles** (containers for permissions assigned to AWS service instances) and **groups** (containers for sets of users and their permissions). They can be applied to individual API calls, so we can specify permissions to control: which resources are available, ranging from virtual machines, database instances, and even the ability to filter database query results.

For example, we can determine which users have MFA access to Amazon EC2 and who can perform specific actions on those resources, such as launch EC2 instance. IAM can automatically rotate access keys on the VM instances, ensuring that only

trusted applications and users have appropriate access at any given time.

EC2. Elastic Compute Cloud. Virtual computers, virtual machines that are running in the cloud. Sometimes they are referred to as EC2 instances. They come in many types, corresponding to the capabilities of the virtual machine in CPU, RAM, speed, disk sizes, and network bandwidth. In terms of computing power there are different families of instances depending on the use case: video encoding/application streaming (p3 family), small web servers (t2/t3 family), machine learning/bitcoin mining (p2 family), etc.

In terms of pricing, there are three types of instances:

- On-Demand - is a way to pay for computing by the hour with no long-term commitment. It suits perfectly for the applications being developed on EC2 for the first time with an unpredictable workload that cannot be interrupted, for the users that want the lowest cost and flexibility.
- Reserved instances - allow you to get a significant discount on the compute in return for a commitment to pay for instance hours for a predefined time frame(1, 3 or 5 years.) Perfectly fit for use cases where workloads run all the time or where at least one component of the system is running all the time.
- Spot instances - is an option to get EC2 resources many times cheaper than on-demand prices. But you have to keep in mind that these instances can be terminated with little to no warning. They allow clients to bid on unused EC2 capacity and use them as long as their bid exceeds the current Spot Price. They are excellent for stateless, fault-tolerant and time-sensitive workloads.

CLB. Classic Load Balancer. AWS offers three types of load balancers - Classic Load Balancer (CLB), Application Load Balancer (ALB), and Network Load Balancer (NLB). Before introducing ALB in 2016, CLB was known as Elastic Load Balancer (ELB).

A load balancer distributes user requests to different EC2s in different AZs. It ensures that users are served only by healthy instances. Furthermore, it can automatically create and terminate instances depending on the current load. CLB is ideal for simple web balancing across multiple EC2s. ALB is more suited for applications needing advanced routing capabilities, container-based, microservices architecture. Classic LB does not support complex, rule-based routing and it can only route traffic to a single globally configured port on destination instances. Application LB can forward to ports that are configured on a per-instance basis, better support for routing to services on shared clusters with the dynamic port assignment.

CLB is better suited for AWS newcomers, i.e., if the client doesn't have complex load balancing needs. Even if the architecture is so simple - put a load balancer in front. That will get flexibility when upgrading since you won't have to change any DNS settings that will be slow to propagate. Both, CLB and ALB, can scale to very high throughput, but not instantly, i.e., they cannot handle sudden spikes in traffic

well. In those cases, best practice would be to "pre-warm" them by slowly transferring an increasing amount of traffic.

By using any of the LBs, you pay for the running time and traffic processed by a load balancer.

S3. Simple Storage Service. It is the standard cloud storage service for storing files[1] with unlimited capacity in the cloud. Objects are stored in so-called buckets. The bucket name is called a key and the bucket content - value. Amazon provides extremely high, eleven 9s[2] level of durability for the S3. By default - data is highly replicated within one region.

The creation and updating of individual objects are atomic. If you create a new object, you will be able to read it immediately - read-after-write consistency. If you update - you are only guaranteed eventual consistency.[3]

S3 has a static website hosting option. It enables us to configure HTTP index and error pages for redirecting HTTP traffic to public content into the bucket. It is one of the simplest ways to host static content or a fully static website. Another approach is to store user-generated files in S3.

S3 has a per GB pricing model. You pay for storage, data returned, data scanned, HTTP requests (PUT, COPY, POST, GET, SELECT, etc.), and data transfer out.

DynamoDB. Fully managed NoSQL database service suitable for document and key-value store models, focuses on speed, flexibility, and scalability. It is a proprietary AWS product, and it doesn't have any interface-compatible alternatives. The closest alternative is Cassandra.

In DynamoDB, all data is stored on SSDs, spread across three geographically distinct data centers (i.e., built-in high availability), perfect for read-heavy applications. It supports three data types - string, number, and binary. DynamoDB offers two different read consistency types:

- Eventual consistent reads - is the default. It means that the consistency is usually reached within 1 second. It gives the best performance, but there is a limitation - dirty reads.[4]
- Strong consistent reads - guarantees that all writes committed before the read taking place. It could result in the delay of your results being returned.

You can choose whether you want a guarantee to have the correct data or on occasions it might not have up-to-date versions. When you write something to DynamoDB, the changes have to be replicated to three geographically distinct datacenters on which DB is being stored.

[1] In AWS terminology, they are called objects.

[2] 99,999999999% durability rate - a mathematical calculation based on independent failure rates and levels of replication. If you store 1 million files in S3, statistically Amazon will lose one file every 659,000 years.

[3] The change will happen, but you don't know when. In practice it means "within seconds" but in rare cases - minutes or hours.

[4] You requested a read but did not receive the up-to-date version.

DynamoDB is charged using *provisioned throughput capacity*. When you create a DynamoDB table, you define the capacity you want to reserve for reads and writes.

Write throughput: AWS charges for every ten units of write capacity per hour. Ten units can handle 36,000 writes.

Read throughput: AWS charges for every fifty units of read capacity per hour. They can handle 360.000 eventually consistent reads or 180.000 strongly consistent reads per hour.

DynamoDB can be expensive for writes, but cheap for reads. There is also a storage cost per GB for the amount of data you use.

We decided to go with DynamoDB, instead of Relational Data Store, because it has built-in high availability and we don't need to manage replication. Besides that, if the application has a lot of reads and little writes, then one should be looking at DynamoDB rather than RDBMS.

Containers. A container is the running instance of an image. We start one or more container from a single image in the same way that we start a virtual machine from the VM template. Similar to virtual servers, containers provide a full server operating system. They use features of the Linux kernel or other operating systems, to partition different application environments under one OS.

Containers are even faster to provision than a VM, tens or hundreds of milliseconds. They are especially suitable for workloads that need to scale up and scale down quickly but don't usually use the full capacity of a single server. You can think of containers as a lightweight virtual machine. Yes, they start up and shut down faster, but you can a little less control over how much compute power and memory resources are available to a given container. Because of this, they considered as ephemeral, i.e., a container only exists while doing the actual work, whereas a virtual machine is often treated exactly like a real server and is always on. You can leave a container running all the time, and you can start and stop them as needed. They tend to be more suited to be an ephemeral style of managing your applications because they can be started and stopped so quickly. Docker, Kubernetes, and CoreOS are all container technologies.

Docker. Docker is an open platform for developing, running, and shipping applications. *„Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.“* (Docker Overview 2019).

It creates, orchestrates, and manages containers - the capability to package and run an application in an isolated environment. When most techies talk about Docker - they are referring to the Docker Engine.

Docker Engine - is the service that runs in the background on the host OS and controls everything required to interact and run docker containers.(Fig.2.2) Next, we have binaries and libraries; they get built into special packages - **docker images**.

The last piece is our applications. Each one has its image, isolated from other apps, and will be managed independently by the Docker. It helps to save a ton of disk space and other system resources, due to not needing to log around a bulky guest OS for each application that you run. There is also no requirement for virtualization with Docker since it runs directly on the host os.

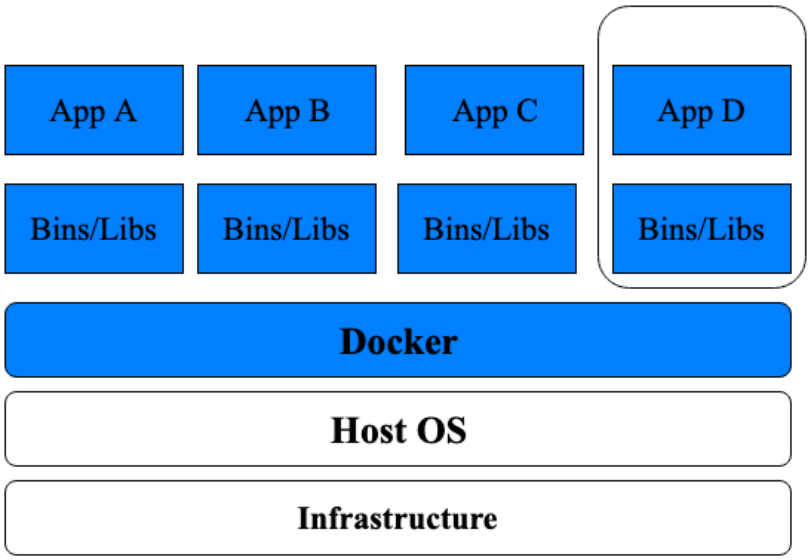

Figure 2.2: Running docker containers on a server

Docker helps developers to work in standardized environments using local containers that provide your application and services. They are great for continuous integration and continuous delivery (CI/CD) workflows. The following example helps us to understand better the benefits of Docker containers:

- *Your developers write code locally and share their work with their colleagues using Docker containers.*
- *They use Docker to push their applications into a test environment and execute automated and manual tests.*
- *When developers find bugs, they can fix them in the development environment and redeploy them to the test environment for testing and validation.*
- *When testing is complete, getting the fix to the customer is as simple as pushing the updated image to the production environment.*

(What can I use Docker for?).

Kubernetes.[5] K8s is an orchestration tool, scheduler, allowing users to run and manage container-based workloads.

[5]Is very often referred to as k8s.

„Container schedulers are based on the same principles as schedulers in general. The significant difference is that they are using containers as the deployment units. They are deploying services packaged as container images. They are trying to collocate them depending on desired memory and CPU specifications. They are making sure that the desired number of replicas is (almost) always running. All in all, they do what other schedulers do but with containers are the lowest and the only packaging unit. And that gives them a distinct advantage. They do not care what's inside. From scheduler's point of view, all containers are the same.“ (What Is A Container Scheduler? 2017).

Kubernetes has gathered significant implementation support from a lot of companies: Microsoft(AKS), Amazon (EKS), Google Cloud (GKE).

A scheduler consists of a master node that distributes the workloads to other nodes.(Fig.2.3) Deployment tools or developers send instructions to the master node to perform container deployments. Scheduler helps to ease the management of scaling, performing health checks and releases across multiple services, as well as utilizing underlying infrastructure efficiently.

From a 10.000 feet overview a Kubernetes container scheduler works as following:

- Software engineers write instructions in a declarative manner to specify which applications they want to run. Applications workloads may differ: stateful application, database, long-running service, stateless service, one-off job.
- Those instructions go to a master node.
- Master node executes them, distributing the workloads to a cluster of worker nodes.
- Each worker node pulls an image from a registry and runs a container from the image.

Kubernetes consists of a couple of components. The first building block of the k8s system is - *Kubernetes cluster services*. The fundamental promise behind Kubernetes is that we can enforce *Desired State Management*. In other words, I am going to feed cluster services a specific configuration, and it is up to the cluster services to go out and run that configuration in my infrastructure.

The second building block is a *worker*. It is a container host, and a unique thing about it is that a worker has a kublet process that is responsible for communicating with k8s cluster services. The desired state (deployment file) that we want to feed to the k8s exists in the configuration file.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: microservice-deployment
  labels:
```

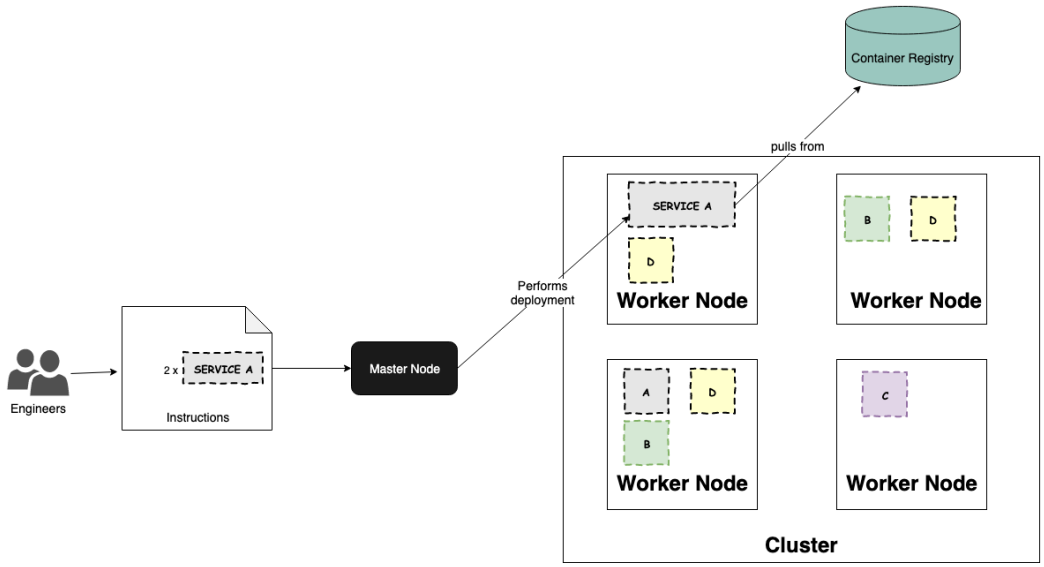

Figure 2.3: Kubernetes scheduler architecture

```
    app: microservice
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: microservice
    spec:
      containers:
      - name: microservice
        image: microservice:0.1
        ports:
        - containerPort: 80
```

The first piece in the deployment file is a Pod configuration. A Pod is the smallest deployment unit in Kubernetes in terms of the k8s object model, and it represents a single instance of a service. In a Pod, I can have multiple running containers. We can quickly scale our application horizontally. i.e., by adding or removing Pods - we scale up and down. To run that Pod, I need to specify container images. Besides that, I can specify how many containers I want to run in my Pod. We take this deployment file and feed it to the cluster services API. It is a cluster services job to figure out how to schedule my Pods in the environment and make sure that I always have the right amount of Pods running.

Another component that we need to understand is Kubernetes ReplicaSet. With ReplicaSet, we can maintain a stable set of replica Pods running at any given time.

It is very often used to guarantee the availability of a specified number of identical Pods. We use it when we want to specify the number of instances of one service to run in a Pod.

SQS. Simple Queue Service is a highly scalable message queuing service entirely managed by AWS. SQS provides the pull model, where producers push the messages to the queue, and consumers pull them off the queue. It ensures that the message being processed by a consumer will not be delivered to other consumers. But if the consumer does not delete it from the queue after processing, the message will become visible again to other consumers.

AWS has two different types of queues:

- Standart queue - guarantees **at least** one delivery and doesn't retain the order of delivery.
- FIFO queue - guarantees **only** one message delivery and the first in first out delivery order.

Unsuccessfully processed messages can be placed in dead-letter queues. These queues help to debug the application or messaging system. They let the developers isolate problematic messages and determine why their processing failed.

ElastiCache. ElastiCache is an in-memory cache service fully managed by AWS. It can be used to store temporary data in fast storage, to avoid repeating computation or reduce the number of requests to a microservice. ElastiCache supports two open-source cache implementations: Redis and Memcached. AWS takes care of running, optimizing, and patching the cache nodes, so the clients need to launch the cache cluster and configure the endpoint for the application. The rest will handle AWS for us.

For this project, we will be using Memcached, as it just a plain key-value store, and all the bells and whistles that Redis provides, we don't need. Moreover, the simplicity of Memcached allows it to be slightly faster than Redis.

Infrastructure as Code & Terraform. Back in the early days, most of the sys. administration tasks were done via scripts or even most of the time - manually. That brought a real mess when working in a team because nobody knew how the virtual machines were configured, what changes were made or how the managed infrastructure looked like. Self-written scripts didn't work either. They were better than manual work, but they brought a lot of ambiguity. There were no standard guidelines on how the script should look like. Thus, every script was unique and often required a lot of effort to understand. Moreover, as the systems became bigger, the scripts grew as well and became more complex.

These are the reasons why Infrastructure as Code became popular. Software engineers found out that it is convenient to keep all configuration scripts under version control. They have also realized that most of the configuration tasks are very similar: start a VM, create a firewall rule, install packages, start a service. Now, instead of writing a new implementation each time, we describe infrastructure using high-level configuration syntax built on the scripts.

Terraform is an open-source tool that allows defining infrastructure as a code by using simple, declarative language. Deploy and manage infrastructure across different cloud providers. Instead of clicking through a dozen pages in the AWS management console, here is a simple code that spins up a t2.micro instance in `eu-central-1` region.

```
provider "aws" {
  region = "eu-central-1"
}

resource "aws_instance" "stock-of-the-day" {
  ami           = "ami-00aa4671cbf840d82"
  instance_type = "t2.micro"
}
```

And a command for deployment in the AWS:

```
terraform init
terraform apply
```

Because of its power and simplicity, Terraform has become a key player in the Development-and-Operations world. Terraform allows developers to replace fragile parts of the infrastructure with reliable, automated foundation upon which they can build continuous integration, continuous delivery, automated testing and other tooling software (Docker, Chef, Puppet).

3 Static Web Hosting

3.1 Hosting with EC2 and Auto Scaling Group

At the very beginning, there is some amount of action that must be completed before moving on. That is - setting up the account and permissions with IAM.

The first step would be to create an account on the aws.amazon.com. That is going to be our root account, and it will have super-admin access. Afterward, we activate the MFA on the root account. Further using a root account is strongly discouraged; that is why we need another IAM user with granted admin access. Set up the billing alarm with CloudWatch and get email notifications if the costs exceed the expected amount. The final step is to configure the AWS CLI for a user using API credentials. The checkpoint for the IAM setup would be using the AWS CLI to interrogate information about the AWS account. With an `aws iam get-account-summary` command, we verify that everything has been set up correctly.

We will deploy a couple of EC2 VMs and host a simple static "Our Website is Coming Soon" web page. We'll take a snapshot one of our VM, delete the VM, and deploy a new one from that snapshot. Basically disk backup and disk restore operations. Next, we'll create an AMI[1] from the VM and put it in an Auto Scaling Group, so at least two machines are always running. We will make use of Classic Load Balancer and put it in front of our VMs and load balance between two availability zones (two EC2 instances in each AZ).(Fig. 3.1)

We start with defining the steps that must be performed on an already running EC2 instance to launch the webpage:

1. Install the httpd web server. (this will create two subfolders under `/var/www`)
2. Copy the index.html file from our S3 bucket into `/html` subfolder.
3. Copy the error.html file from the S3 bucket into `/error` subfolder.
4. Start the httpd server.

We can execute these steps manually by connecting via SSH to our instance. Or we create a script that will be executed directly after the instance creation and pass it to the configuration step as User data.(Fig. 3.2)

```
yum update -y
yum install httpd -y
```

[1] Amazon Machine Image - immutable images that are used to launch pre-configured EC2 instances.

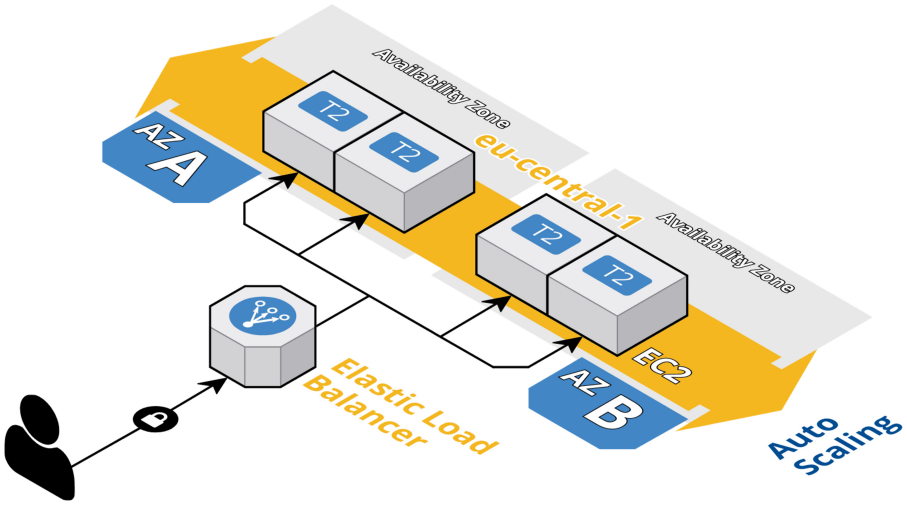

Figure 3.1: Static Web Hosting architecture

```
chkconfig httpd on
aws s3 cp s3://aws-the-right-way/.../index.html /var/www/html
aws s3 cp s3://aws-the-right-way/.../error.html /var/www/error
service httpd start
```

For steps 2 and 3 to work, we have to grant access to our instance that it can read and write data from the S3. We perform this by creating a new IAM Role `ec2-r-w-s3` and allowing EC2 instances to call AWS services on our behalf. Don't forget to attach this role to the configuration step.(Fig. 3.2) At this point, we have one, single EC2 up and running and by going to its public DNS, we can observe - *Website is Coming Soon.*

Next, to make our website more resilient and capable of handling different types of load - we create an Auto Scaling Group:

„Amazon EC2 Auto Scaling helps you ensure that you have the correct number of Amazon EC2 instances available to handle the load for your application. You create collections of EC2 instances, called Auto Scaling groups. You can specify the minimum number of instances in each Auto Scaling group, and Amazon EC2 Auto Scaling ensures that your group never goes below this size.“ (EC2 Auto Scaling User Guide 2019).

Paraphrasing that with simple words: when the AWS sees that the load on our servers increases rapidly, Amazon spins up new instances for us, based on the template we have provided to the Auto Scaling Configuration. To create the template we navigate to the AWS Console, in the EC2 Section/Instances select our running instance and click under Actions->Image->Create Image.

By that, we have created the AMI that we use in the Auto Scaling Group with two instances, one per AZ.(Fig. 3.3)

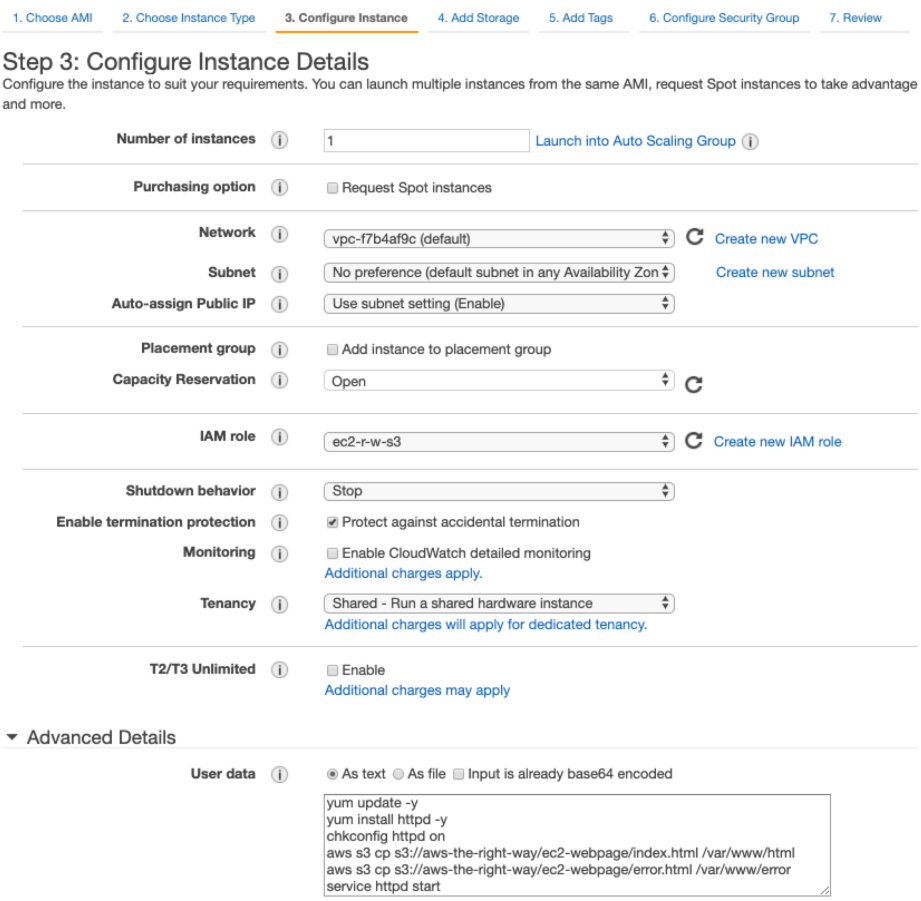

Figure 3.2: Configure user data and IAM role

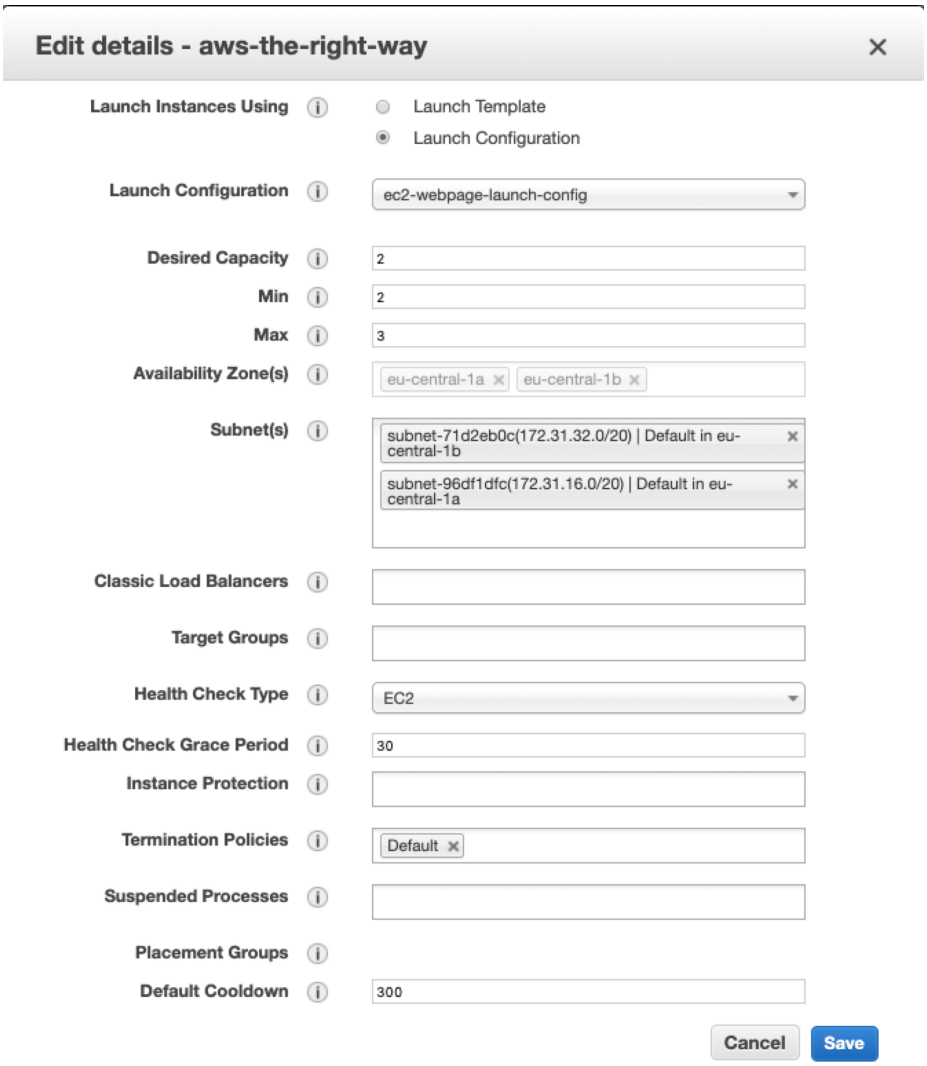

Figure 3.3: Auto Scaling configuration

The last thing that we need to do is to create a Classic Load Balancer and put our instances behind it. In case one instance goes down, LB will forward traffic to another.

3.2 Cost Analysis

Let us calculate the cost for running our static webpage 24/7 for the whole month.

The webpage is hosted on four t2.micro *On-Demand* instances in two AZ and one Auto Scaling Group. Different availability zones and Auto Scaling Group options come at no cost. Elastic Compute Cloud On-Demand instances allow us to pay by the hour or second rule (minimum of 60 seconds) with no long term commitment. Let's assume we are going to run our instances non-stop, i.e., 720 hours per month. One t2.micro cost $0.0134 per hour. If data transferred out from EC2 to the internet up to 1GB/month, Amazon will charge us $0 per GB. (to exceed this 1GB limit, in our scenario the webpage must be called more than seven million times per month).

For the Classic Load Balancer, we pay $0.03 per running hour and $0.008 per GB of data processed by CLB. We end up with:

$$4 * (720\text{h} * \$0,0134) + (720\text{h} * \$0,03) + \$0,008 = \ \$60,2/\text{month}$$

3.3 Deployment

An application is only useful if you can deploy it to the users. So let's do it with Terraform configuration.

The first step is typically to configure the provider and region we want use. Creating a `main.tf` and the following description will do the job.

```
provider "aws" {
  region = "eu-central-1"
}
```

Next, we have to define the launch configuration: which type of instances do we need, set up the IAM instance role, security groups.

```
resource "aws_launch_configuration" "webpage-launch-config" {
  name = "webpage_launch_config"
  image_id = "ami-07f0f6e10c9a18895"
  instance_type = "t2.micro"
  iam_instance_profile = "ec2-r-w-s3"
  security_groups = ["sg-0a9bd08ed359e165f"]
  user_data = "${data.template_file.user-data.rendered}"
  root_block_device {
```

```
    delete_on_termination = true
  }
}
```

Attach the instances to the Auto Scaling Group and where do we require them.

```
resource "aws_autoscaling_group" "webpage-autoscaling" {
  name = "webpage-autoscaling"
  max_size = 4
  min_size = 2
  availability_zones = [
    "eu-central-1b",
    "eu-central-1a"]
  launch_config = "${aws_launch_configuration.webpage-launch-config.name}"
  health_check_type = "EC2"
  health_check_grace_period = 30
  load_balancers = [
    "${aws_elb.webpage-elb.name}"]
}
```

After the instances are created, we install Docker on each of them and host static web application with the steps defined in the 3.2 section.

```
sudo su
yum install update -y
yum install docker -y
curl -L "https://github.com/docker/compose/.../docker-compose"
chmod +x /usr/local/bin/docker-compose
ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose
service docker start
usermod -aG docker ec2-user
yum install git -y
mkdir app1 && cd app1/
git clone https://github.com/TyLeRRR/aws-the-right-way
cd aws-the-right-way/static-webhosting/
docker-compose up -d
```

Define load balancer, ports, how to run health checks and an elastic IP to connect to the website.

```
resource "aws_elb" "webpage-elb" {
  name = "webpage-elb"
  availability_zones = ["eu-central-1b", "eu-central-1a"]
  listener {
```

```
    instance_port = 80
    instance_protocol = "http"
    lb_port = 80
    lb_protocol = "http"
  }

  health_check {
    healthy_threshold   = 10
    unhealthy_threshold = 4
    timeout             = 15
    target              = "HTTP:80/index.html"
    interval            = 30
  }
}

output "elb_ip" {
  value = aws_elb.webpage-elb.dns_name
  description="Endpoint to connect to our webpage."
}
```

The last step is to run `terraform apply` and connect to the CLB Elastic IP.

4 Microservice-Oriented Architecture

4.1 Monolith vs. Microservices

First, let's have a quick look at what was prior to microservices. In the early days, the whole application was built as a single unit. It consisted of three main parts: a user interface (usually HTML and Javascript), a database (usually some form of RDMS) and a server-side. The server handled and processed the HTTP requests, executed the domain logic, created and updated data in the database, and populated HTML views in the browser. This three-tier application is called **monolith**.

Let's imagine that we are starting a new house renting platform, like Airbnb. After some research and requirements gathering, we decide to create a new project with Spring Boot, Python, or PHP. This new application will look like something as following. (Fig. 4.1)

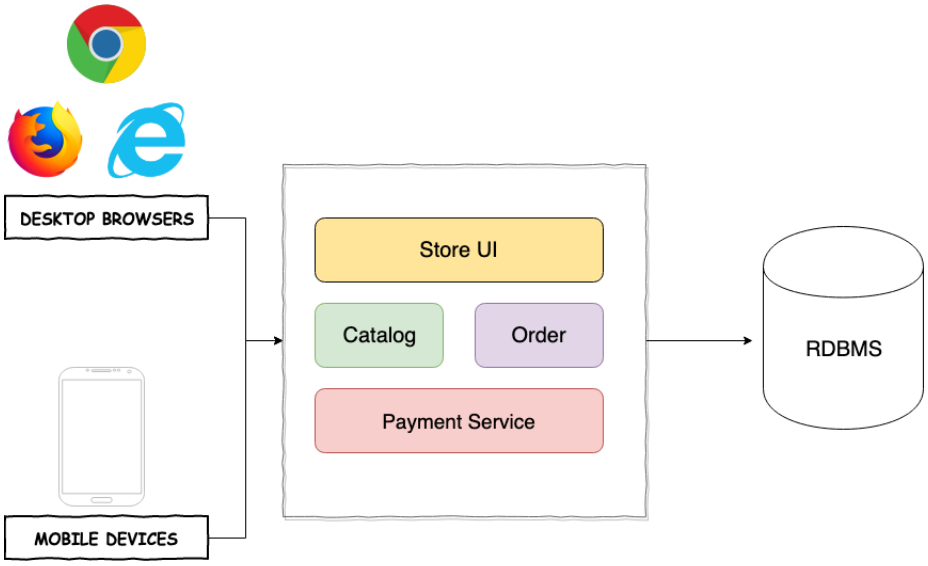

Figure 4.1: Monolithic architecture

In the center of the application is business logic, implemented with mini-modules that define services - surrounded by the adapters for connecting with the external world, e.g., REST API adapters (for connecting with userbase on mobile devices), database access adapters, etc.

Even having a logically modular architecture, the application is developed and

packaged as a monolith. The format depends on the programming language and frameworks. If this is a Java application - it will be packed as WAR file and deployed on an application server such as Apache Tomcat, Oracle Weblogic or Red Hat JBoss. Or, it can be packed as a self-contained executable - JAR file.

Applications developed in this style are widespread. They are relatively simple to create since our IDEs are mainly focused on building a single application. They are also simple to test, e.g., you can execute end-to-end testing just by launching the app and testing the UI with Katalon.[1] The transaction control and data integrity are straightforward. Since monoliths usually use a single database, they benefit a lot from standard transaction controls provided by the relational databases. They are also simple to deploy - copy the packaged app to a server. The application can be scaled by running multiple copies on multiple servers behind a load balancer. In the first year, it might work great.

Unfortunately, this simple approach has huge limitations. Successful applications tend to grow over time and eventually become enormous. Incoming new business requirements create new "features to implement" for developers, which results in adding many lines of code to the existing codebase. After a couple of years, a small, simple application grows into a monolith monster. It becomes too large for any single developer to understand fully. As a result, implementing new requirements and fixing bugs becomes painful and time-consuming. New engineers spend more time comprehending the existing codebase and less time adding product value. This will affect the development speed and release cycles drastically.

The larger the application - the longer the start-up time. If software engineers frequently have to restart the application server, then a large portion of their time will be spent waiting around.

Another problem with a monolithic application - continuous delivery. Nowadays, state of the art for any software product is to push changes into production a couple of times a day. That is very difficult to accomplish with a monolith since you have to redeploy the whole app to update any part of it. The long start-up times that I mentioned previously won't help either. Also, since the impact of a new feature in most cases is not very well understood, you likely have to do extensive testing manually. As a result, continuous delivery is nearly impossible to do.

For instance, that Airbnb example without microservices may run on one powerful server. In this case, the challenge is scaling the code, running the server: did we provision enough memory? What about when the vacation renting season comes around? Different modules might have conflicting resource requirements. One module might implement GPU-intensive video encoding logic and would ideally be deployed in Amazon EC2 Graphics Intensive instances (p3 family) with NVIDIA Teslas on board. Another module might be a machine learning training job, which is best suited for Amazon SageMaker instance types. In a monolith, these modules are deployed together, i.e., one single type of hardware for all modules. Moreover, the

[1] Automation testing software specialized for mobile and web testing.

amount of activity that the system performs may outgrow the capacity of original technology choices.

With the help of virtualization and containers, it is relatively easy nowadays to create lightweight applications that scale up and down quickly. That encouraged a pattern - **Microservices**. Structuring applications as fine-grained, loosely coupled, and independently deployable services - has become very popular among engineering teams. Instead of a large application running on a single server, you create a set of simple services that do one thing and do it well.

We might have one service that runs the store, another - for customer's catalog, one for inventory, one for payment processing - each operates independently. If our services are loosely coupled and with high cohesion, having one part crashed or getting overloaded, doesn't necessarily take down the rest of our system.

The following characteristics can be observed in each microservice:

- Each service has its datastore. This reduces coupling between services. The service can retrieve the data they don't own, only through the API that another service provides.
- Each microservice does one thing and does it well whether it is business-related capability or integration with third party services. The less each service knows about the inner implementation of others, the easier it is to make changes without forcing them to others.
- Each service can be deployed independently. This is the most important one. Otherwise, it still would be a monolith at the point of deployment.

Microservices are a natural mechanism for isolation failures. If we develop them independently, infrastructure or application failure may only affect one part of our system. If the image resizer service is unavailable, it won't be able to process an image and persist in the datastore. But the user will not get an error and continue browsing the web site. By using asynchronous communication, image service will catch up and resize the image later after recovery.

Splitting the application into multiple services helps not only isolate the failure but can multiply points of failure. We need to track what happens when a failure occurs to prevent cascades. This involves using circuit breakers, async. communication and timeouts appropriately.

There is no silver bullet for all. Microservices drastically increase the number of moving parts in a system. Identifying and scoping microservices requires in-depth domain knowledge. An incomplete understanding of the problem domain can lead to poor decisions and design choices. It is hard to identify the right contracts and boundaries between services and even more difficult and time-consuming to change it later. To make each microservice independent, services should expose an API contract - definition of request and response messages it expects to receive and respond with. Contracts tend to evolve over time, and it may become challenging to maintain backward compatibility for existing service clients.

4.2 Architecture

In this section, we develop the Stock-of-the-Day web application with Microservices. We retire that static website and deploy this part with Kubernetes. In the S3 bucket, we store uploaded static content such as images/logos/files. Instead of connecting directly to load balancer's public IP, we register a domain name, set Amazon Route 53 as the nameservers and use it for DNS routing. The business logic will be split into microservices, and each gets a single task to implement in his own narrow scope. Each service will expose a RESTful API and communicate with other parts of the system over HTTP.

Before we jump in, let me demonstrate what the application does. With the *Stock-of-the-Day*, I want to make smart financial investments available to everyone. I want to help users to keep their hands on the pulse of their favorite companies. Whether they decide to expand or reduce their portfolio, with the *Stock-of-the-Day* they can keep track of the latest changes on their favorite stocks.

On each page load, the random stock will be shown with a stock logo image on the left and the current price. Besides that, a user has an option to add a new stock to the watchlist.(Fig. 4.2)

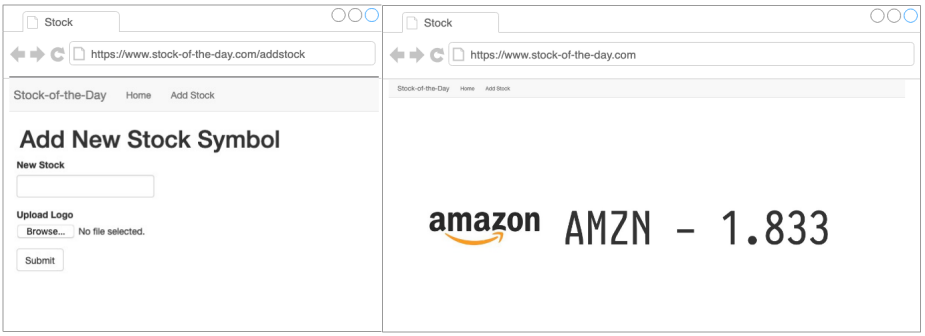

Figure 4.2: stock-of-the-day.com

That is a plain application with a couple of gotchas and limitations. The more we break down the architecture into smaller pieces, the more overall complexity increases. What if a user instead of a company logo uploads pictures with nudity or blocked content (adult content detection)? Based on which time series we show a stock price (daily, weekly, etc.)?

And I am not even talking about technical details: How to get a random item from the database without knowing the id beforehand? How to avoid deploying a broken version of the application to all users? There are a lot of pitfalls to be aware of.

First things first: let us take a look at the services that form the foundation of the app:

- Frontend service - the facade of the Stock-of-the-Day application. This service renders the web page and forwards all user requests to the backend.

- Stock-symbol service - responsible for retrieving a random stock from the database, as well as creating a new one per user request.
- Logo-resizer service - resizes the uploaded stock logos and stores them in S3.
- Queue-workers - update a stock item in the database with logo object URL taken from S3 so that we have all stock information stored in one place.
- Stock-price service - makes an external call to Alpha Vantage API and retrieves the current stock price based on the time series.

Resizing images, storing them in S3, and updating DB table with new image URLs - we do asynchronous via Amazon SQS. Stock-symbol service creates a new event, logo-resizer and queue-workers listen to those events.

4.2.1 Frontend Service

In the real world, nobody visits a website by connecting to its public IP directly. So let's go and register a domain name, luckily AWS provides this feature as well. After giving your Contact Details (name, phone, address, country, etc.), we have register - stock-of-the-day.com for a reasonably low price (more on that in the Cost Analysis section). The next time when somebody accesses the website, Amazon Route 53 will route the traffic to the nearest DNS server, which will hit then the stock-of-the-day frontend microservice.(Fig. 4.3)

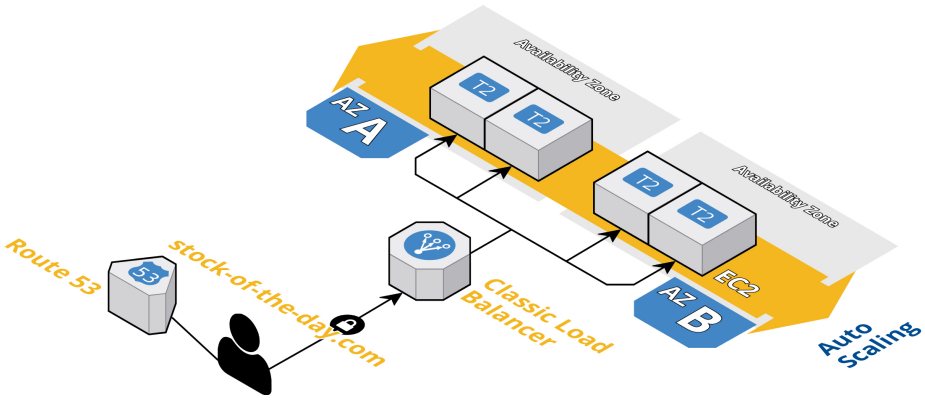

Figure 4.3: Frontend service

By decoupling the frontend into a separate service, we abstract the backend concerns from the consuming application. We make sure it doesn't know about any underlying microservices interaction. We delegate the requests to downstream services and transform their responses to the user with Flask templates.

```
{% extends "base.html" %}

{% block app_content %}
```

```
        <link rel="stylesheet" href="{{ url_for('static') }}">
        <p id="stockSymbol" class="container">
                <img src="{{ stock.logo_url }}">
                {{ stock.name }}
                {{ stock.price }}
        </p>
{% endblock %}
```

Templates help to separate the business logic and presentation layer. Otherwise, if one day we decided to alter the layout of the website, we would have to update the HTML in every view function, which is not an option when the application grows and scales.

To spread the load evenly across our machines, we put a load balancer in front - the same way as we did in the Static Web Hosting implementation. Our frontend is served from four EC2 instances in two AZs, and if one of the machines goes down, we'd like to stop forwarding traffic to those instances.

The behavior and state of a microservice should be observable. At any time, we should be able to determine whether the service is healthy and whether it's processing a workload in the way we expect. That can be done with load balancer health checks. We create the `ping` method in the frontend service, which simply returns HTTP status code 200. A load balancer will call `/api/ping` with 30 seconds interval. If there is no response for two consecutive calls, each 5 seconds long, LB will mark that instance as unhealthy and stop forwarding traffic there. The same goes for considering unhealthy instances as healthy, but this time, it must be five consecutive calls. For services to communicate, they need somehow be able to discover each other - **service discovery**. Load balancers are suited for that as well.

Except for serving the frontend, the service is also responsible for sending user requests to the backend. One such example would be when the user creates a new stock. The frontend service will pass the user-given stock name and logo to the stock-symbol service. For uploading images, we have a unique endpoint. It gives us more power over handling the logos. We do not want to store them locally on the server, but directly upload them into S3 bucket; therefore, we want some general handling over them.

As soon a user selects an image, the frontend service starts uploading them to `/api/image` on the background. That approach is very user-friendly because a user typically first choose an image and then proceeds with filling out other forms. When he hits the *submit stock* button, the image is already uploaded. He doesn't wait and see the uploading screen. The only drawback of this approach is when the user somewhere in the middle of the process. He changes his mind and closes the tab, but meanwhile, you have already uploaded the image. That means we have uploaded the logo and assigned it nowhere.

There are two ways to handle that:

1. As it may happen so rarely, we don't even bother with such cases. S3 provides

unlimited storage, and generally, the price is very low.

2. Create a time-based job scheduler that will scan and delete unassigned images older than one month.

4.2.2 Stock-Symbol Service

Stock-symbol service is the central block of the system. It is responsible for CRUDing the information from the database, uploading the logo-images to SQS and retrieving the latest stock price from stock-price service. We upload images asynchronously via Amazon SQS and retrieve the stock-price synchronously.

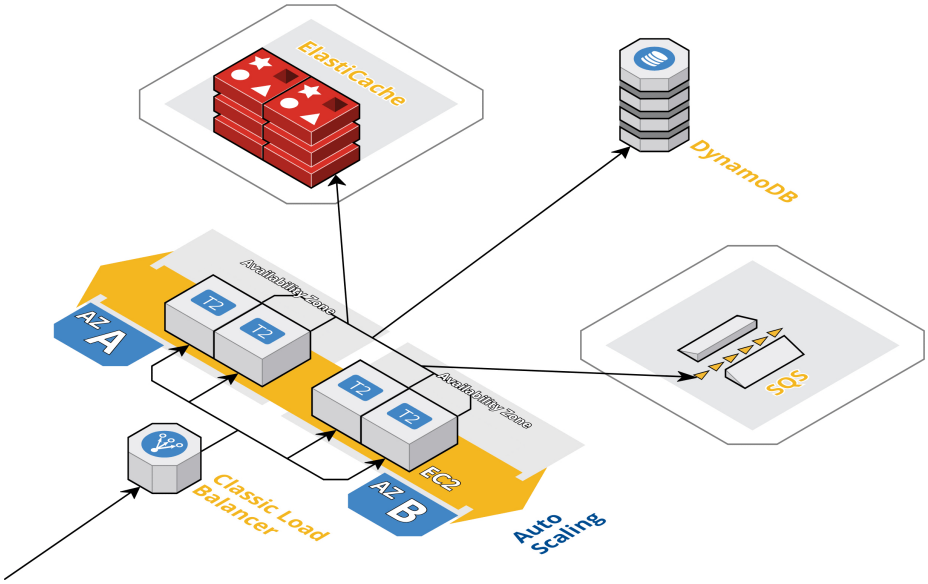

Figure 4.4: Stock-symbol service

Communication is a fundamental element of a microservice application. Synchronous messages are well-suited in scenarios where an action's results - or acknowledgment of success or failure - are required before proceeding with another task. **Synchronous messages** have limitations:

1. Tighter coupling between services, as services, must be aware of their collaborators.
2. Block code execution while waiting for responses. In the process- or thread-based server model, this can exhaust capacity and trigger cascading failures.
3. Overuse of synch. messages can build deep dependency chains, which increases the overall fragility of a call path.

For retrieving the current stock price, we use synchronous calls, i.e., get a response from stock-price service and then respond back to the frontend service, so-called

request-response pattern. Dependency chains of multiple microservices will achieve the most useful capabilities in an application. When one microservice fails, how does that impact its collaborators and ultimately, the application's end customers?

If failure is inevitable, we need to design and build our services to maximize availability, correct operation, and rapid recovery when failure does occur. When a request to stock-price service fails, we need some fallback option to make the application more resilient. Caching is one of these options.

The stock-symbol service caches the stock price for up to 24 hours in the dedicated AWS ElastiCache. This solution would both improve the performance and provide contingency in the event of a temporary outage. It reduces the need to query the stock-price service as well. As the stock prices don't change drastically on day to day basis, we consider this the best fallback option. When the call fails, we show the price from the day before.

An **asynchronous** style of messaging is more flexible. By announcing events, you make it easy to extend the system for handling new requirements, because services no longer need to know their downstream consumers. New services can consume existing events without changing the implementation. This style enables more fluid evolution and creates loose coupling between services.

That does come at a cost: asynchronous interactions are more challenging to implement correctly because overall system behavior is no longer explicitly encoded into linear sequences. Asynchronous messaging typically requires a communication broker, an independent system component that receives events and distributes them to event consumers.

For passing the images to SQS and processing them later by logo-resizer service, we are using Amazon SQS broker and "job queue" - async. communication pattern. In this pattern, workers take jobs from a queue and execute them.(Fig. 4.5)

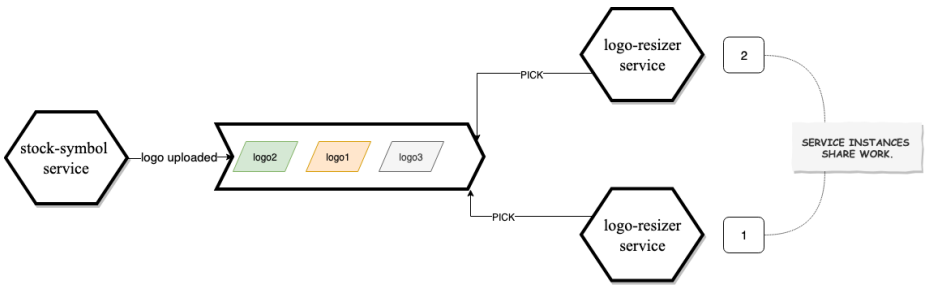

Figure 4.5: Job queue pattern

A job should only be processed once, regardless of how many worker instances we operate. This pattern is also known as the winner takes all. This approach is handy where: 1. A 1:1 relationship exists between an event and a work to be done in response to that event. 2. A task that needs to be done is time-consuming, so it should be performed out-of-band from the triggering event.

For persisting stocks in DynamoDB, we are using the universally unique identifier (UUID)[2] for `stock_id`. The only downside is that we can't order by ID to get the insert order. Hence arises another problem - retrieving a random stock on each page load. DynamoDB doesn't provide such functionality out-of-the-box, so we have to implement it on our own. Generating a random number (n) and then iterate the collection and return the n^{th} item is out of the discussion. That would be not only the $O(n)$ time complexity but would make this particular part of the system an enormous bottleneck as the system evolves. We have implemented another approach that executes in a constant time - $O(1)$.

```
@app.route('/api/stocks', methods=['GET'])
def get_stock():
        response = table.scan(
                ExclusiveStartKey={"stock_id": str(uuid.uuid1())},
                Limit=1
        )
        stock = response['Items'][0]
        return jsonify({'stock': stock})
```

We generate another random UUID and retrieve a stock by that id. It doesn't need to exist in the DB table. For any given key, DynamoDB knows where it "should" exist. When we query with a random one, DynamoDB starts at that location, moves to the next item, and returns it. In this particular case, ElastiCache won't help in speeding up the retrieval, as it will cache miss for each GET operation.

4.2.3 Logo-Resizer Service

With logo-resizer service, we rescale user-uploaded stock images and store them in the S3 bucket. With each page load, they will be loaded directly from the S3 so that we can improve the response time.(Fig. 4.6)

After the stock-symbol service puts the image logo in a queue:

1. Logo-resizer service pulls the message with an image from the SQS.
2. Retrieves the named file.
3. Resizes the image.
4. Writes the image to the S3 bucket.
5. Writes a *Task Complete* message back to the queue.
6. Puts the logo URL into another queue for queue-workers.

[2]While each generated UUID is not guaranteed to be unique, the total number of unique keys (2^{128}) is so large that the probability of the same number being generated twice is very small. For example, consider the observable universe, which contains about 5×10^{22} stars; every star could then have 6.8×10^{15} universally unique UUIDs.

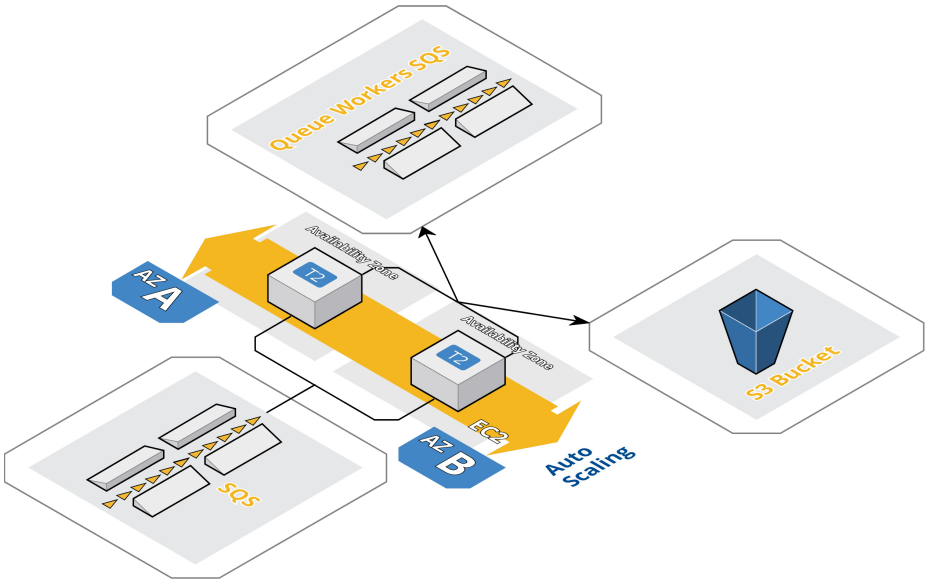

Figure 4.6: Logo-Resizer service

7. Deletes the original task message.
8. Checks for more messages in the queue.

```
def push_to_sqs(stocklogo_s3_url, stock_name):
  response = sqs.send_message(
    QueueUrl=queue_url,
    DelaySeconds=10,
    MessageAttributes={
      'StockName': {
        'DataType': 'String',
        'StringValue': stock_name
      },
      'LogoUrl': {
        'DataType': 'String',
        'StringValue': stocklogo_s3_url
      }
    },
    MessageBody=(
      'Information about resized logo. S3 object URL.'
    )
  )
  print(response['MessageId'])
```

Two instances of the logo-resizer service are listening to the SQS events with the long polling approach.

SQS long polling is a method for retrieving messages from the Amazon SQS queue. The traditional short polling method returns immediately, even if the polling queue is empty, while long polling doesn't return a response until a message arrives in the SQS. This method helps it easy and inexpensive to retrieve events from the queue as soon as they are available. When we receive a message from the queue and start processing it, sometimes we may find the visibility timeout for the SQS is insufficient to process and delete that message fully. To give ourselves more time for processing, we extend its visibility timeout by using `ChangeMessageVisibility` API. SQS will restart the timeout period using the new value.

We resize the logo and store in memory as BytesIO without writing to disk. To make it accessible as an image file for everyone, we have explicitly changed its content-type and access control lists to the public read. Otherwise, it would remain as binary/octet-stream in the bucket, and the content of the image would be hidden without downloading it beforehand. That would have made serving static content from S3 bucket impossible.

```
@app.route('/api/upload_image', methods=['POST'])
def upload_to_s3():
  bucket = 'aws-the-right-way'
  logo_name = uuid.uuid4()
  key = "logos/%s_logo.jpeg" % logo_name
  to_save = request.files['image']

  args = {
    'ContentType': 'image/jpeg',
    'ACL': 'public-read'
  }
  filepath = resize(to_save, str(logo_name))
  stock_name = json.loads(request.form['name'])['name']
  s3.upload_file(Filename=filepath, Bucket=bucket, Key=key, ExtraArgs=args)
  response = push_to_sqs(logo_url=bucket_url + key, name=stock_name)

  return response.status_code
```

4.2.4 Queue-Workers

After the image is resized and stored in S3, the corresponding stock item must be updated in DynamoDB with the logo URL from the S3. The next time users request the info about that stock, all the data can be loaded from the S3 with one single `GET` call.(Fig. 4.7)

Logo-resizer service creates a new message containing the S3 URL and passes the message to the SQS. We have two worker instances that are listening to SQS with long polling. After a message arrives in a queue, a worker updates the `logo_url`

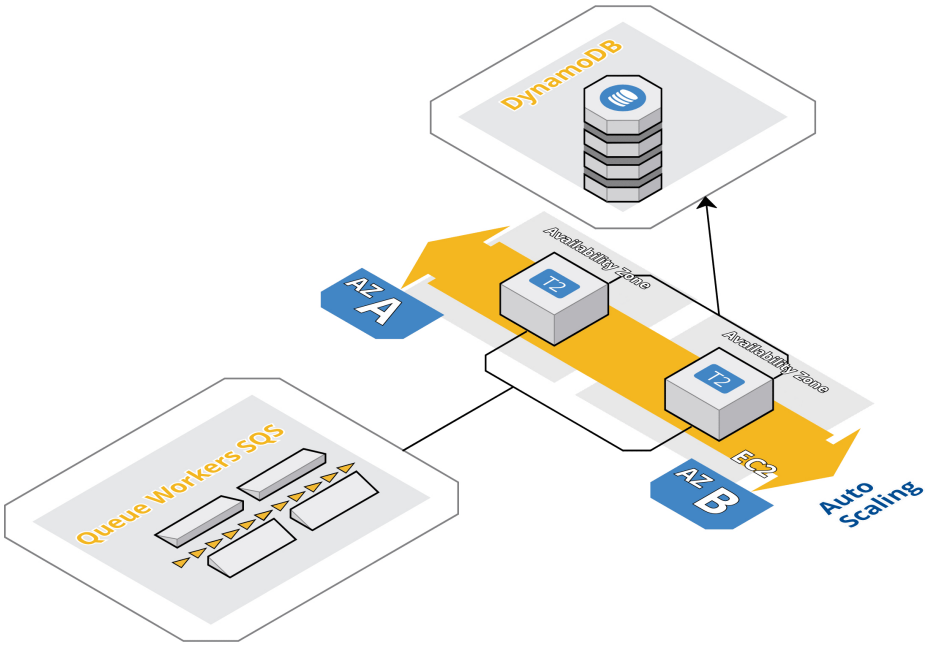

Figure 4.7: Queue workers

attribute in DynamoDB by `stock_id`.

```
def update_dynamo_item(stock_name, logo_url):
  stock_id = get_id(stock_name)
  table.update_item(
    Key={
      'stock_id': stock_id
    },
    UpdateExpression="set logo_url = :l",
    ExpressionAttributeValues={
      ':l': logo_url
    },
    ReturnValues="UPDATED_NEW"
  )
```

4.2.5 Stock-Price Service

After a user goes to `stock-of-the-day.com`, the stock-symbol service will request the price by calling the stock-price service. We are using the public Alpha Vantage API to retrieve the latest price on a stock request.(Fig. 4.8)

The API will return daily time series (date, open, high, low, close, volume) of the global equity specified.

```
{
```

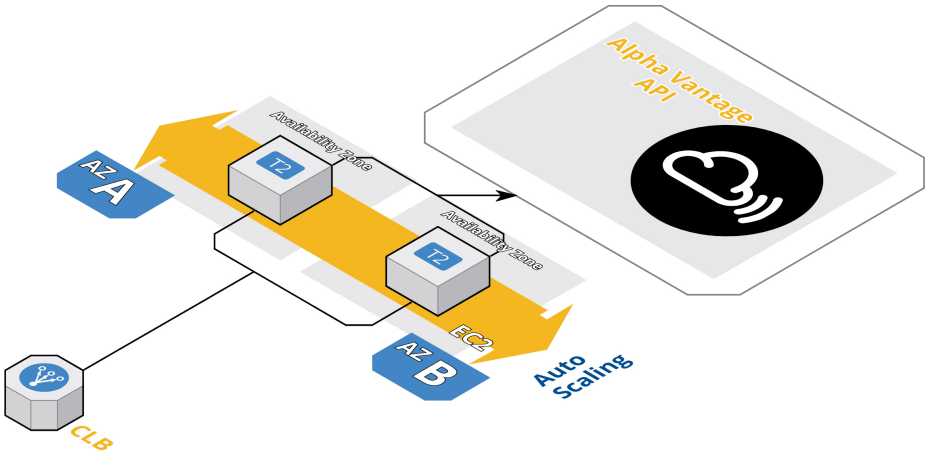

Figure 4.8: Stock-Price service


```
  "Meta Data": {
    "1. Information": "Daily Prices and Volumes",
    "2. Symbol": "MSFT",
    "3. Last Refreshed": "2019-08-30",
    "4. Output Size": "Compact",
    "5. Time Zone": "US/Eastern"
  },
  "Time Series (Daily)": {
    "2019-08-30": {
      "1. open": "134.8800",
      "2. high": "139.1000",
      "3. low": "136.2800",
      "4. close": "137.8600",
      "5. volume": "21877723"
  }
}
```


As the stock prices are updated daily and we can request not only daily time series but weekly and monthly, we decided not to use any persistent storage for this service. In case the Alpha Vantage request fails, or the entire service goes down, we will return HTTP 503. Stock-symbol service will respond to the frontend with the price for the previous day, loaded from the cache.

4.3 Cost Analysis

Let us calculate the cost for running our microservices implementation 24/7 for the whole month. To make the analysis for this chapter plausible, we make the following assumptions:

- We have 10.000 daily active users.
- Each user views ten random stocks and adds a new one per day.
- $10.000 \times 10 \times 30\text{days} = 3$ million **read requests** per month.
- $10.000 \times 1 \times 30\text{days} = 300.000$ **write requests** per month.
- A typical logo image (not-resized) is 1MB.
- A typical resized logo image is 500KB.
- HTTP Header is 1KB[3].
- Request/response content - 1KB. (images are served directly from S3 bucket)
- On each *show new random stock* operation, application consumes 0,002MB.
- On each *add new stock* operation, application consumes 1,002MB.

Route53 & DNS. For owning the domain name *stock-of-the-day.com*, we pay 1$ per month.

Elastic Container Registry. The application consists of 5 microservices. Each service builds a docker image for each new release with 300MB per image. Let us assume we work on a be-weekly sprint basis and release new versions twice per month. AWS ECR pricing for storage is 0,1$ per GB-month. Pushing data to ECR is free, but for pulling Amazon charges 0,09$ per GB.

$$300\text{MB} * 5\text{images} * 2\text{releases} = 3\text{GB}$$

$$3 * \$0,1 + 3 * \$0,09 = \$0,57/\text{month}$$

Elastic Kubernetes Service. For using EKS, Amazon charges 0,2$ per hour for 1 EKS cluster.

$$\$0,2 * 720\text{h} = \$144/\text{month}$$

DynamoDB. Cost calculation for DynamoDB can be split into:

- Read&Write requests - AWS charges 1,525$ per 1 million writes and 0,305$ per 1 million reads.
- Data storage - 25GB of storage is at no cost.
- Backups - 0,2448$ per GB-month.
- Data transfer - transfer-in is 0$, transfer-out is 0,09$ per GB.

In DynamoDB Strings are encoded with UTF-8, and each character uses 1 to 4 bytes. If we assume that each item in the table is 2KB, this would be enough to fit 500 characters in the worse case. Five hundred characters are sufficient to fit-in the UUID, name, and logo URL.

[3]https://dev.chromium.org/spdy/spdy-whitepaper

```
{
"stock_id": "09577d74-b5c6-11e9-8398-c4b301d8bec3",
"name": "APPL",
"logo_url": "bucket.s3.region.amazonaws.com/logos/stock_id_logo.jpeg"
}
```

If the table contained information about stocks of all trading companies in the world[4], our DynamoDB table would be 400MB in size.

$$\$1,525 + \$0,305 * 3 + \$0,24448 + \$0,09 = \ \$2,77/\text{month}$$

Frontend service. The service is fronted by a classic load balancer. AWS charges 0,03$ per running hour and 0,008$ per GB of data processed by CLB. Frontend service runs on four t2.micro instances, each costs 0,0134$ per hour with transfer-in 0,09$ per GB and transfer-out 0,02$ per GB.

In traffic measurements it is: 3.000.000reads $\times$ 0,002MB = 6GB. 300.000writes $\times$ 1,002MB = 300GB. For simplicity of calculation, let us assume that traffic is evenly spread across all four instances.

$$\$0,03*720\text{h}+\$0,008*6+\$0,008*300+4*\$0,0134*720\text{h}+\$0,09*300+\$0,02*6 = \$89,76$$

Stock-symbol service. The same as frontend service, stock-symbol runs on four t2.micro instances faced by CLB. ElastiCache cluster consists of two cache.t2.small nodes with 0,038$ per hour. There is no charge for data transfer between Amazon EC2 and Amazon ElastiCache withing the same AZ. SQS costs 0,4$ for 1 million requests per month.

$$\$89,76 + \$0,038 * 2 * 720\text{h} + \$0,4 = \$144,88/\text{month}$$

Stock-price service. Service is faced by CLB and runs on two t2.micro with 0,0134$ per instance-hour. The traffic on this service is very low, won't exceed 1GB in and out. Applies the same 0,09$ for traffic-in and 0,02$ for traffic-out.

$$\$0,0134 * 2 * 720\text{h} + \$0,09 + \$0,02 = \$19,406/\text{month}$$

Logo-resizer service. Logo-resizer operates on two t2.micro instaces with 0,0134$ per hour each. Serves an SQS with 0,4$ per 1 million requests and writes data to the S3 bucket. We store resized images in *S3 Standard Storage*. Let us round up the number of public trading companies to 200.000. With that said, in the worse case, we have to store 200.000 logo images. As we already have mentioned, a resized logo is 500KB. This results in 200.000 $\times$ 500KB = 100GB. S3 charges 0,0245$ per GB for storage. Data returned by S3 - 0,0008$ per GB.

$$\$0,0134 * 2 * 720\text{h} + \$0,4 + \$0,0245 * 100 + \$0,0008 * 100 = \$22,22/\text{month}$$

[4]https://www.quora.com/How-many-publicly-traded-companies-are-in-the-world

Queue workers. Workers run on two t2.micro instances. It results in additional:

$$\$0,0134 * 2 * 720\text{h} = \$19,296/\text{month}$$

The **final calculation** for running our infrastructure for the whole month 24/7 is:

$$\$1+\$0,57+\$144+\$2,77+\$89,76+\$144,88+\$19,406+\$22,22+\$19,296 = \ \$444 \text{ per month}$$

The price can be reduced up to 40% by hosting some of the services on the half amount of instances and migrating from *On Demand* to *Reserved* instances.

4.4 Deployment

In a real application, deployment is an ongoing process. We can't pause or stop the app, deploy a new version, and then resume. We need to able to roll out new releases without downtime. There are three patterns for zero-downtime deployments:

- Rolling deployments - we are progressively taking down old instances while spinning up the instances with a new version, ensuring that we maintain a minimum percentage of capacity during deployments.
- Canary deployments - we add a single instance with a new version to test the reliability before continuing with a full rollout. The canary pattern gives an additional layer of safety beyond a normal rolling deploy.
- Blue-green deployments - we create a parallel group of services running the new version (green group), progressively shift user requests from the old version (blue group) to the green one. This works very well in scenarios where service clients are susceptible to error rates and can't accept the risk of an unhealthy canary.

Before we continue, I'd like to introduce Kubernetes deployments briefly. In Kubernetes Deployment object is a high-level abstraction for orchestrating new replica sets. Each time we update the object, the scheduler will orchestrate a rolling update of instances in a replica set, ensuring they are deployed seamlessly. For canary deployments, we deploy a single instance of a microservice to ensure that a new version is stable when it faces production load. This instance runs along with existing production instances. Our canary release, for stock-symbol service, consists of the following steps:

1. We release a single instance of stock-symbol service with the `2.0` version, alongside the `1.0` version.
2. We route 30% of the traffic to the new instance.
3. We monitor the health checks and error rates on the new instance.
4. If it considered healthy, we route 100% of traffic and delete one instance with `1.0` version. If not, we remove the canary instance, halting the release.

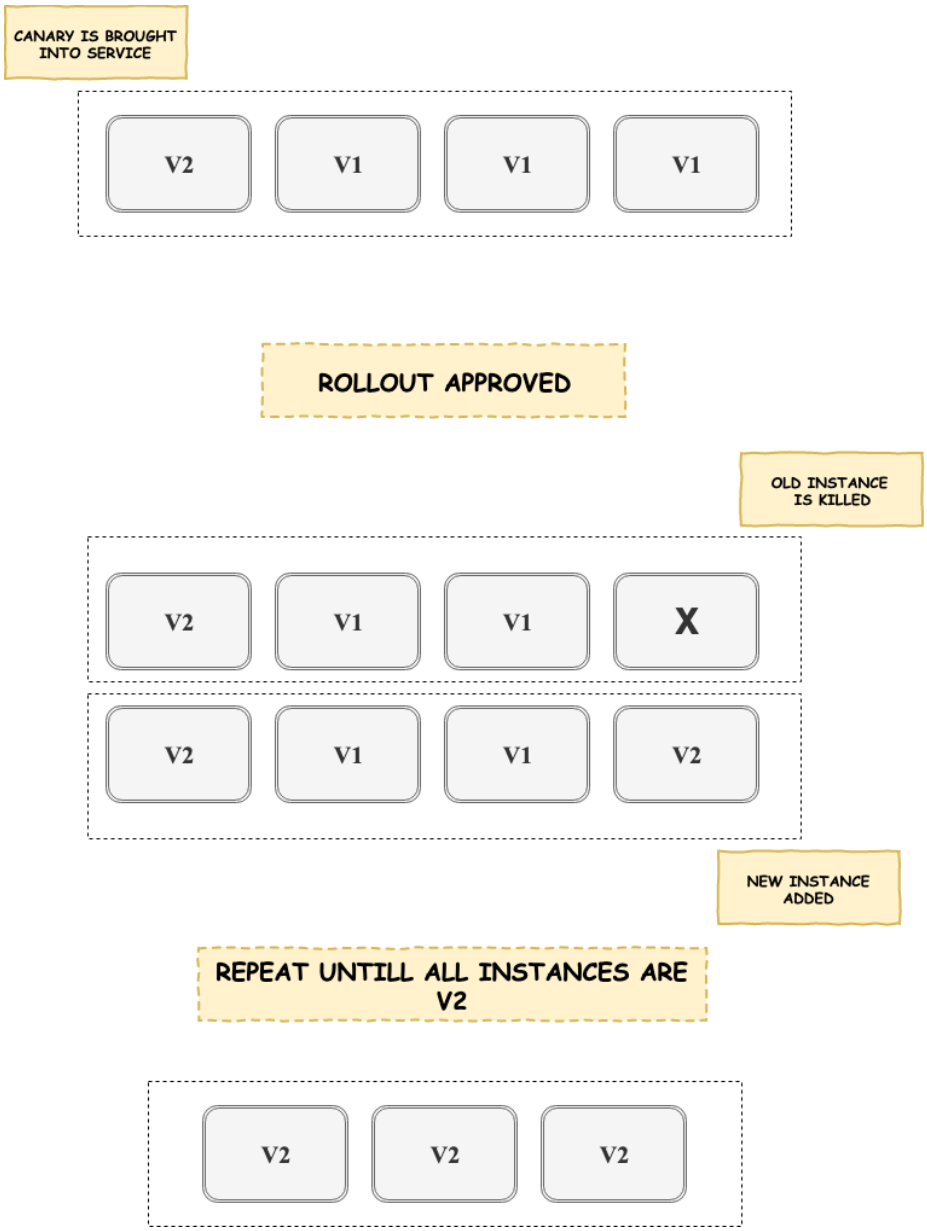

Figure 4.9: Stages of canary deployment

On k8s, we use labels to identify a canary pod. To deploy a canary pod, we set `track: canary` label. The load balancer will route 30% of the traffic to that pod.

The first thing that we need to do is to take our code, get it running on EC2s, and make it accessible for the outside world. Downloading the code and compiling it on the running machine would make our releases slow and unpredictable. This would result in different commits running in production, i.e., service instances would run incompatible versions of the code. Without any explicit packaging or versioning, there would be no easy way to roll our code forward or back.

Service artifacts(Fig. 4.10) are the way to make the deployments predictable, fast, and automated. It is an immutable and deterministic package for the service that contains: compiled code, libraries, binary dependencies. A perfect microservice deployment artifact would allow to package a specific version of the compiled code, specify any binary dependencies, and provide an operational abstraction for starting and stopping the service. It should be possible to run the same artifact locally and in production. There are different types of service artifacts: OS packages, VM images.

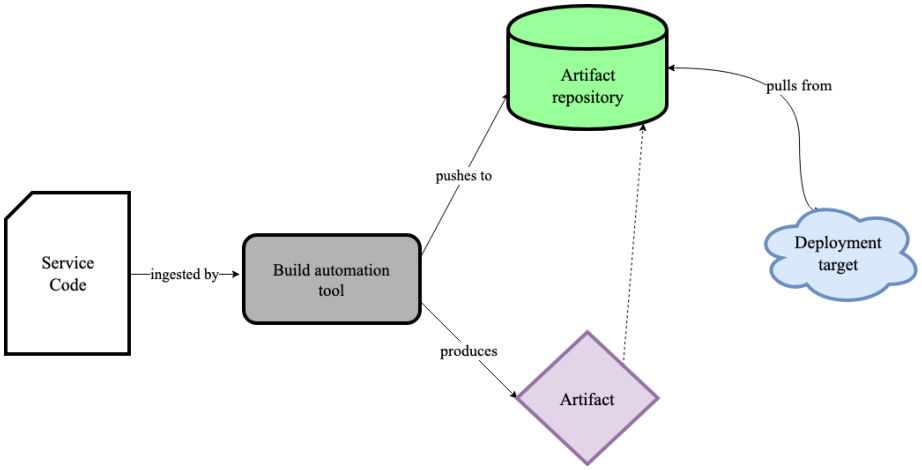

Figure 4.10: Storing artifacts into repository

All our services in this implementation are deployed by using Docker container images. This helps to avoid the overhead of virtualizing the guest operating system and the disk on each VM. We need a Dockerfile:

```
FROM alpine:3.10

RUN mkdir /app

COPY . /app

WORKDIR /app
```

```
RUN echo "**** install Python ****" && \
apk add --no-cache python3 && \
if [ ! -e /usr/bin/python ]; then ln -sf python3 /usr/bin/python ; fi && \
\
echo "**** install pip ****" && \
python3 -m ensurepip && \
rm -r /usr/lib/python*/ensurepip && \
pip3 install --no-cache --upgrade pip setuptools wheel && \
if [ ! -e /usr/bin/pip ]; then ln -s pip3 /usr/bin/pip ; fi

RUN \
apk add bash \
&& apk add bash-completion \
&& pip3 install --trusted-host pypi.python.org -r requirements.txt

CMD ["python3", "app/app.py"]

EXPOSE 5000
```

We build a docker image and push it to the Amazon Elastic Container Registry (ECR).

```
docker build . -t stock-symbol-service:2.0
docker push $AWS_ECR_URI:2.0
```

We tag this version as `2.0`, though in practice it is not the best approach to apply numeric versioning to the services. Tagging with commit ID works better. After we've pushed our image, we have to prepare the *Desired State Management* YAML file for *k8s cluster services*.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: stock-symbol-service-canary
  labels:
    customer: taras.sologub
spec:
  replicas: 1 #create 1 canary
  template:
    metadata:
      labels:
        app: stock-symbol-service
        tier: backend
        track: canary  #mark with canary label
```

```
    spec:
      containers:
        image: ecr_url/stock-symbol-service:2.0 #deploy 2.0 version
       -name: stock-symbol-service
        ports:
          - containerPort: 5000
        livenessProbe:
          httpGet:
            path: /api/ping
            port: 5000
          initialDelaySeconds: 10
          timeoutSeconds: 15
        readinessProbe:
          httpGet:
            path: /api/ping
            port: 5000
          initialDelaySeconds: 10
          timeoutSeconds: 15
imagePullSecrets:
  name: my-ecr-credentials
```

After reading this YAML file, Kubernetes will create a new replica set containing a single canary Pod with version `2.0`. Along with production instances with old version, now we have the `2.0` canary instance. At this stage, we might run some tests or monitor the output of the canary, to ensure that everything is working as expected.

Let's assume that this canary is healthy and we can safely roll out the new version. We have to update the YAML file with the following:

```
...
metadata:
  name: stock-symbol-service
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 30%
      maxSurge: 30%
    ...
      labels:
      track: stable
    ...
```

This configuration will create a new replica set, starting instances one by one while removing them from the original set.

5 Continuous Integration

5.1 Introduction

Reliably and rapidly releasing new features to production is crucial for maintaining an application successfully. Investing in consistent and robust deployment infrastructure and tooling will go a long way toward making the success of any project. We can achieve stable releases by applying the principles of continuous delivery.

Continuous delivery is a software approach in which developers produce software in short cycles, ensuring that they can reliably release to production at any time. It is all about building a pipeline that automatically takes the latest code from commit to production.(Fig. 5.1) Each step in the pipeline provides live feedback to the development team on the correctness of their system.

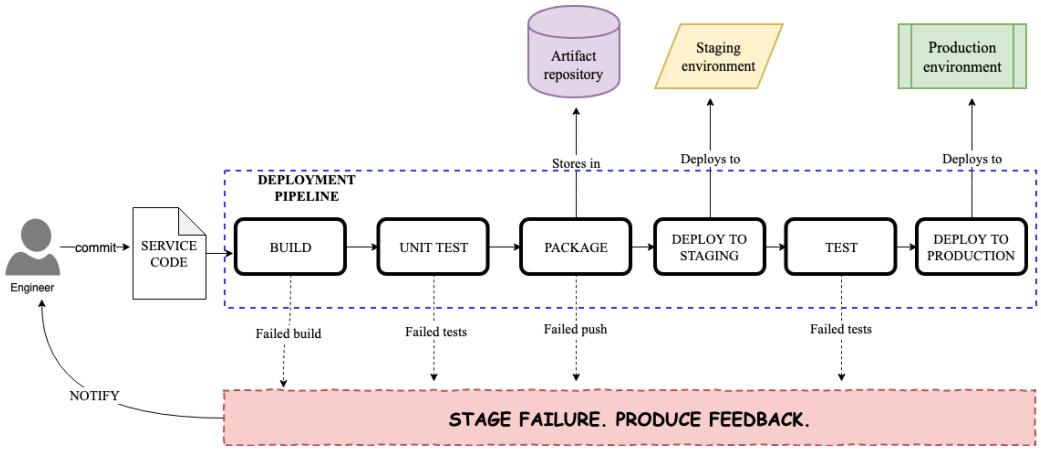

Figure 5.1: Deployment pipeline

1. An engineer commits some code to a repository.
2. The build automation server builds the code.
3. If the build is successful, the automation server runs unit tests.
4. If the tests are green, the automation server stores the service artifact to the artifact registry.
5. The automation server deploys code to a staging environment, where developers can test the service against other services.

6. If step No.5 successful, the automation server will deploy the code to the production.

Each step provides instant feedback to engineers, whether their code was correct or not. If step No.4 fails, they'll receive a list of failed test cases. Building such a deployment pipeline enables developers to work safely and at a faster pace as they iteratively develop an application.

Imagine the following scenario: the team is developing and deploying hundreds of independent services on their own schedule, without cross-team coordination or collaboration. Human errors cause the most issues in production; any critical change to a service might have a wide-ranging impact on the performance of the whole application. Implementing such a pipeline will make the deployment process highly transparent in case something goes wrong. Regardless of underlying technology or language, every service we deploy should be able to follow a similar process.

5.2 Configuring Delivery Pipeline with Jenkins

In the previous chapters, we ran Terraform and Kubernetes YAML files manually. Now we use Jenkins - a build automation tool that helps to connect: building containers, publishing artifacts to ECR, and deploying code - into a single executable unit.

Unfortunately, there is no one-fits-all tool, and even with Jenkins, we use a combination of multiple frameworks.(Fig.5.2)

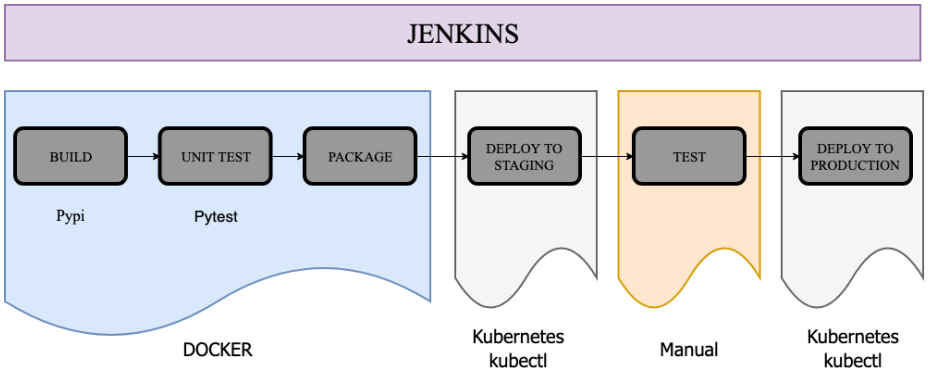

Figure 5.2: Jenkins pipeline frameworks

Jenkins consists of a master node and any number of worker nodes. Build automation job executes script across these workers and performs deployment activities. Jenkins job operates within a workspace - a local copy of code repository. For building the deployment pipeline, we are going to use a feature - Scripted Pipeline. This will enable using general-purpose domain-specific language (DSL) written in Groovy. In DSL are defined the common methods for programming the build jobs. Jenkins

executes jobs by running a pipeline script defined in a Jenkinsfile. Let's create a Jenkinsfile in the root of our stock-symbol service.

```
stage("Build Info") {
  node {
    def commit = checkout scm
    echo "Latest commit sha --> ${commit.GIT_COMMIT_SHA}"
  }
}
```

This script will check out a code repository to Jenkins workspace and print out the latest commit SHA. Let us verify the Jenkins installation:

1. Go to EC2 IP where Jenkins is installed.
2. Navigate to the Create New Job page.
3. Enter `stock-symbol-service` name, as a job type select *Multibranch Pipeline*.
4. Configure a job (Fig.5.3) The build tool will scan the service repository for new commits every minute.

After saving the configuration, Jenkins will trigger the first job and generate a unique build for each branch.

Next, we are going to use Docker to build and package images with Jenkins. We modify the Jenkinsfile with:

```
/*use Pod template to run the job*/
def withPod(body) {
  podTemplate(label: 'pod', serviceAccount: 'jenkins', containers: [

    containerTemplate(name: 'docker', image: 'stock-symbol', command: 'cat',
    ttyEnabled: true),

    containerTemplate(name: 'kubectl', image: 'stock-symbol',
    command: 'cat', ttyEnabled: true)
   ],

  volumes: [
    hostPathVolume(mountPath: '/var/run/docker.sock',
     hostPath: '/var/run/docker.sock'),
  ]
) { body() }
}

withPod {
```

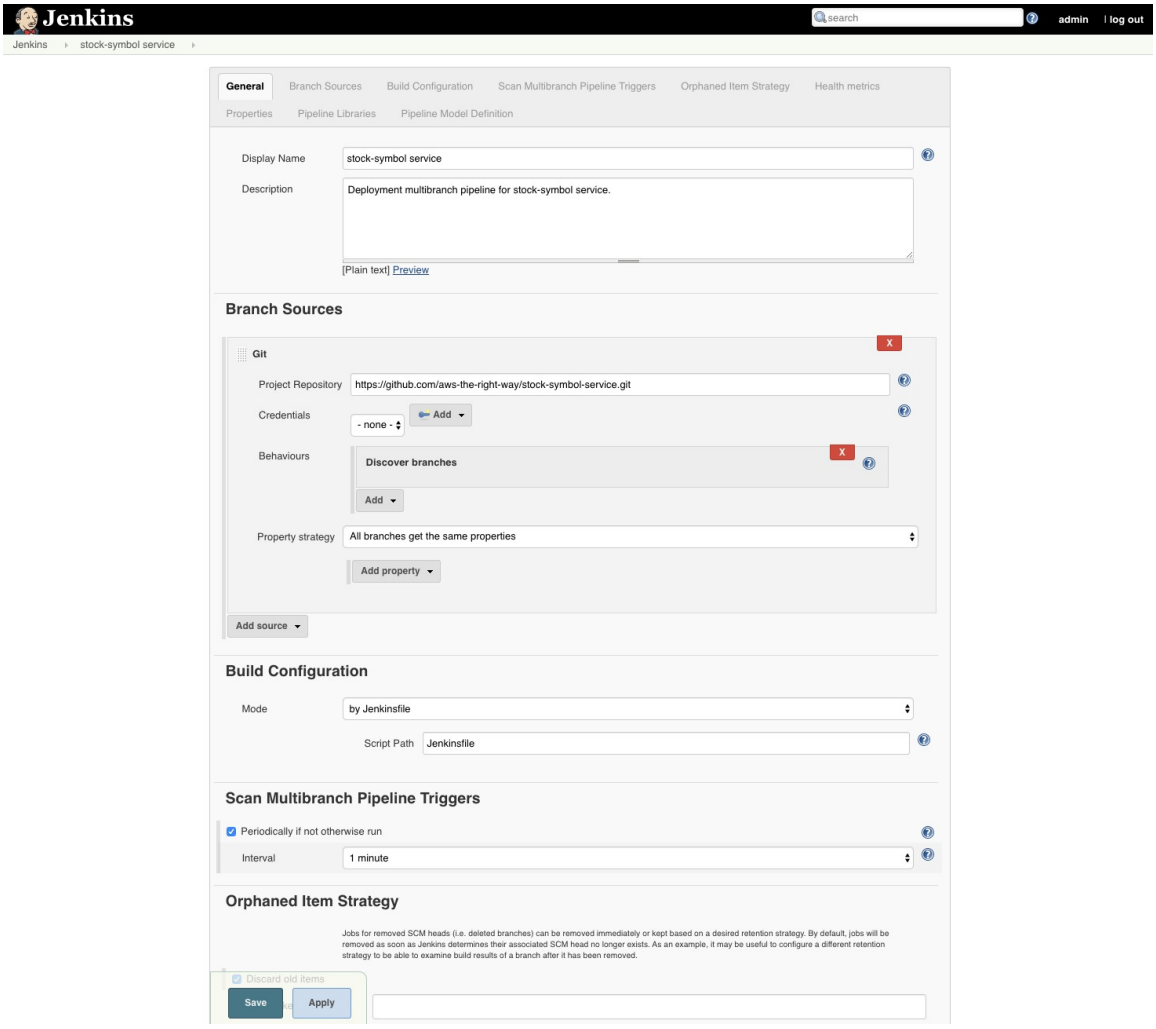

Figure 5.3: Stock-symbol service configuration screen

```
  node('pod') { /*request an instance of Pod template*/
    def tag = "${env.BRANCH_NAME}.${env.BUILD_NUMBER}"
    def service = "stock-symbol:${tag}"

    checkout scm /*check out the latest code from git*/

    container('docker') { /*enter Docker container*/
      stage('Build') { /*start new pipeline stage*/
        sh("docker build -t ${service} .") /*build Docker image*/
      }
    }
}
```

This definition builds a stock-symbol service and tags the resulting Docker container with the build number:

1. We define a Pod template for the build. Jenkins will create Kubernetes Pods for a build-worker, based on the template.
2. Within that Pod, we check out the latest commit from Git.
3. We start a new pipeline stage - Build.
4. Withing that stage, we connect to the Docker container and run the `docker` command to build a stock-symbol service image.

Now, we have to publish the image to the container registry. As we are using a private registry, we configure Docker credentials in Jenkins. This can be done by going to Credentials -> System -> Global Credentials -> Add Credentials. By adding the following code to the Jenkinsfile and running the job - we publish the image to the ECR registry.

```
def tagToDeploy = "[ecr-url]/${service}"

stage('Publish') {
  withDockerRegistry(registry: [credentialsId:'aws-ecr']) {
    sh("docker tag ${service} ${tagToDeploy}")
    sh("docker push ${tagToDeploy}")
  }
}
```

At this point, we have tested our service in complete isolation without interacting with other service's upstream or downstream collaborators. We could deploy directly to production and hope for the best, but the smarter choice would be to deploy first to a staging environment. Run additional manual and automated tests against the real collaborators and only then roll out to production.

With Kubernetes namespaces, we can logically separate stage and production environments. Let's edit the YAML file we've been using previously and replace `image: ecr_url/stock-symbol-service:2.0` with `image: BUILD_TAG`. We replace this placeholder in our pipeline with the version we're deploying. Before rolling out, let's create separate namespaces for different environments.

```
kubectl create namespace staging
kubectl create namespace production
```

Now, let's add a deploy stage to our pipeline, as follows:

```
stage('Deploy Stage') {
  /*use sed to replace BUILD_TAG with the Docker image name*/
  sh("sed -i.bak 's#BUILD_TAG#${tagToDeploy}#' ./deploy/staging/*.yml") ,

  container('kubectl') {
    /*apply all config. files in deploy/staging to cluster,
    using staging namespace*/
    sh("kubectl --namespace=staging apply -f deploy/staging/")
  }
}
```

After committing the change and running the build, this time, Jenkins will trigger Kubernetes deploy.

The final step in the pipeline would be production rollout. If all previous actions were successful, we continue with:

1. The pipeline should wait for human approval to proceedtu.
2. After approval, we release a canary first and validate on little traffic that a new build is stable.
3. The pipeline can continue with deploying the remaining instances to production.

In continuous delivery, we don't want necessarily push every commit to production. For No.1, we add an approval step to the Jenkinsfile:

```
stage('Release approval') {
  input message: "Release ${tagToDeploy} to production?"
}
```

That will trigger the dialog box with two options: **Proceed** and **Abort**. In the real world, the human approval step can be replaced with an automated decision. We could write code that would monitor the key metrics, such as error rate on the canary instance.

After selecting "Proceed", we deploy the canary first:

```
stage('Canary deployment') {
  deploy.toKubernetes(tagToDeploy, 'canary', 'stock-symbol-canary')

  try {
    /*manual action is required*/
    input message: "Release ${tagToDeploy} to production?"
    } catch (Exception e) {
      deploy.rollback('stock-symbol-canary')
    }
}
```

and afterward, release to production.

```
stage('Deploy to production') {
  deploy.toKubernetes(tagToDeploy, 'production', 'stock-symbol')
}
```

The complete delivery pipeline in Jenkins looks like the following:

Figure 5.4: Stock-symbol service deployment pipeline

These techniques help developers to build a reliable and consistent foundation for delivering rapidly and safely to production.

6 Conclusions and Future Work

The aim of this work was to introduce web application development in the AWS on practical examples. With this project, I wanted to show how it relatively cheap and agile is to build real-world examples without being a professional in all 100+ services.

The first example is based on static content hosting, which is suited for *one-page applications*. I didn't want to rush in but start simple. We built this implementation around the central services with the highest profit - EC2, load balancers, and Auto Scaling Groups.

In the second example, I have concentrated more on a real-world scenario. Microservices are the current industry trend. Every new application is developed by separating business requirements into individual blocks. Even the companies with a monolith, make significant investments into breaking the monolith down to microservices. For my scenario, I have retired the *Web site is coming soon* page and separated new features into six microservices.

Besides those examples, we have observed how second approach enabled efficient continuous integration.

In the last section, I have automated the manual deployments with Jenkins's deployment pipeline.

By outsourcing physical infrastructure concerns to a cloud provider, developers can focus more on building applications. Let AWS manage Auto Scaling, Kubernetes Control Panel, DynamoDB and S3 replications for us.

Current *Stock-of-the-Day* implementation can be enhanced and widen in several ways:

- Setup AWS Key Management Service (KMS) to IAM so we can centrally manage symmetric encrypting keys.
- Instead of using standard VPC, create a custom one with multiple subnets. Place VMs in each subnet and make them talk to each other using only private IPs.
- Setup encryption on EC2 disk volumes and encrypt the AMI.
- Add an adult or suggestive content detection on uploaded image logos with Sightengine.
- Extend the deployment to another region and implement latency-based routing.
- Build a monitoring system for collecting metrics and observing the behavior of individual services

- Set up a logging infrastructure.
- Retire all our VMs and implement Serverless approach with AWS Lambdas. It will notably reduce costs.

List of Figures

Bibliography

About AWS: *Global Infrastructures Regions and AZs* https://aws.amazon.com/about-aws/global-infrastructure/regions_az

Amazon Simple Storage Service: *Overview* https://aws.amazon.com/s3

AWS Documentation: *What Is Amazon DynamoDB?* https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html

AWS Documentation: *What Is Amazon EC2 Auto Scaling?* https://docs.aws.amazon.com/autoscaling/ec2/userguide/what-is-amazon-ec2-auto-scaling.html

AWS Documentation: *What Is Amazon ElastiCache for Redis?* https://docs.aws.amazon.com/AmazonElastiCache/latest/red-ug/WhatIs.html

AWS Documentation: *What Is Amazon Simple Queue Service?* https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/welcome.html

AWS Documentation: *What Is Elastic Compute Cloud?* https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html

AWS Documentation: *What Is Elastic Load Balancing?* https://docs.aws.amazon.com/elasticloadbalancing/latest/userguide/what-is-load-balancing.html

AWS Documentation: *What Is Identity and Access Management?* https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html

AWS Pricing: *How Does AWS Pricing Work?* https://aws.amazon.com/pricing

Continuous Delivery: *The Deployment Pipeline* https://continuousdelivery.com/implementing/patterns/#the-deployment-pipeline

Continuous Integration: *Overview* https://martinfowler.com/articles/continuousIntegration.html

Docker Documentation: *Docker Overview* https://docs.docker.com/engine/docker-overview

Introduction to Terraform: *Introduction to Terraform* https://www.terraform.io/intro/index.html

Jenkins Documentation: *What is a Jenkins Pipeline?* https://jenkins.io/doc/pipeline/tour/hello-world

Microservices: *A Definition of This New Architectural Term* https://martinfowler.com/articles/microservices.html

What can I use Docker for?: *Docker Overview* https://docs.docker.com/engine/docker-overview/#what-can-i-use-docker-for

What Is A Container Scheduler?: *Scheduler Overview* https://technologyconversations.com/2017/12/14/what-is-a-container-scheduler

Ich versichere, die vorliegende Arbeit selbstständig ohne fremde Hilfe verfasst und keine anderen Quellen und Hilfsmittel als die angegebenen benutzt zu haben. Die aus anderen Werken wörtlich entnommenen Stellen oder dem Sinn nach entlehnten Passagen sind durch Quellenangaben eindeutig kenntlich gemacht.

Hamburg, 21.10.2019 Taras Sologub