Khoa Mai

Building High Availability Infrastructure in Cloud

Metropolia University of Applied Sciences

Bachelor of Engineering

Degree Programme

Bachelor’s Thesis

23 November 2017

Author Title Number of Pages Date	Khoa Mai Building High Availability Infrastructure in Cloud 34 pages 23 November 2017
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructors	Olli Hämäläinen, Senior Lecturer

Software development has always been a race. Companies want to release new products to users as soon as possible. Therefore, nowadays most software development companies implement Continuous Delivery and Infrastructure as Code practices to not only shorten the release process but also to ensure the quality of the software applications.

The main goal of the project was to build a high availability infrastructure in cloud for a microservices backend architecture. The study is based on the principles of Microservices architecture and Continuous Delivery and Infrastructure as Code software engineering practices. The process of creating a deployment pipeline was studied. Two types of deployment, EC2 instances and Containers, were studied to deploy the microservices.

Based on the results, Amazon Web Services was selected as cloud provider. Suitable tools and services were selected to build the deployment pipelines. For most of the use cases, the deployment of Containers was selected, but there were situations where EC2 instances was the most suitable choice.

Keywords	Cloud Computing, Microservices, Infrastructure as Code, Continuous Delivery, Containers, Kubernetes, AWS

Contents

List of Abbreviations

AWS	Amazon Web Services. Cloud computing services provider.
IaC	Infrastructure as Code. The process of managing and provisioning computer data centers through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools.
CI	Continuous Integration. Software development practice where members of a team integrate their work frequently, usually each person integrates at least daily – leading to multiple integrations per day.
CD	Continuous Delivery. The ability to get changes of all types – including new features, configuration changes, bug fixes and experiments – into production, or into the hands of users, safely and quickly in a sustainable way.
K8s	Kubernetes. Open-source system for automating deployment, scaling, and management of containerized applications.
EC2	Elastic Compute Cloud. A web service that provides secure, resizable compute capacity in the cloud.
AMI	Amazon machine images. A special type of virtual appliance that is used to create a virtual machine, which Amazon calls an “instance”, within the Amazon EC2.
HTTP	Hypertext transfer protocol. HTTP is a data access protocol currently run over TCP and is the basis of World-Wide Web.
JSON	JavaScript object notation. JSON is a lightweight data-interchange format.
YAML	YAML Ain't Markup Language. YAML is a human-friendly data serialization standard for all programming languages.

1. Introduction

The case company in this report wanted to build a product consisting of a cloud-hosted backend infrastructure and mobile software development kits (SDK). The mobile SDKs abstract the communication between mobile devices and the backend infrastructure, so that application programmers can quickly integrate the product's features into their applications in minutes. With the product, application developers only need to think about their application's core features. The cloud-hosted backend infrastructure automatically scales based on the usage, which gives the product a flexible pricing model and ability to reach out to small companies.

Since security is a top requirement, the first challenge is to have separated backend infrastructure for each customer. To fulfill that challenge, the company’s engineers had to automate all the steps needed to boot up the backend infrastructure without human interaction. Thus, cloud computing was selected to host the infrastructure instead of a traditional datacenter. Together with Infrastructure as Code (IaC), the company could boot up the infrastructure for a new customer in minutes. Amazon Web Services (AWS) was selected as the cloud service provider.

Another challenge is that the company wanted to shorten time-to-market. Instead of developing a full-fledged product, it was decided to prioritize features and to deliver minimum viable product first, then later adding features based on customer feedback. To be fast in delivery with a good quality product, Continuous Delivery (CD) was selected as the engineering approach from day one of development.

The author of this thesis is responsible for maintaining both the company’s and customers’ infrastructures. His tasks include automating development and deployment process, ensuring the availability and performance of systems, participating in the design and selection of technologies for backend microservices. Therefore, this report focuses on software engineering practices rather than the technical implementation of backend services.

This document has 6 sections. It provides a short introduction to microservices architecture in section 2. After that, section 3 and 4 explain IaC and CD and how those two practices help in the deployment process for microservices architecture. Finally, section 5 demonstrates the deployment in AWS cloud platform. The examples in this document are simplified for demonstration purposes only because the source code of microservices and infrastructure definitions are confidential.

2. High Availability Infrastructure and Microservices

Nowadays, users expect online services to be fast, reliable and feature rich. Companies not only need to ensure the uptime of their applications but also to constantly add new features. It is hard to fulfill those needs with traditional monolith architecture, leading to the design of high availability infrastructure and microservices architecture.

2.1 High Availability Infrastructure

To determine the availability level of a system, we need to define availability and how to measure it. There are four elements of availability:

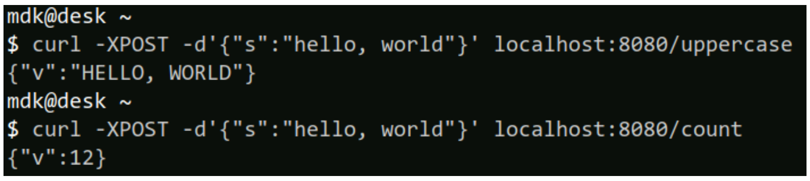
- Reliability: the ability to perform under stated conditions for a stated period of time.
- Recoverability: the ability to easily bypass and recover from a component failure
- Serviceability: the ability to perform effective problem determination, diagnosis, and repair.
- Manageability: the ability to create and maintain an environment that limits the negative impact people may have on the system. [4, 28.]

Then how can it be measured if the system is "high" in availability? Often, to determine the availability of a system, we measure the uptime of the system in a given month or year. A metric called nines is usually used to show the availability measurement of a system, that is how many "9" digits are there in the percentage of a system uptime. For example, a system achieves five nines (99.999% uptime) meaning it has a maximum of twenty-six seconds downtime in a month or five minutes and fifteen seconds downtime in a year.

A software application can also fail. In fact, one of the keys for designing high availability infrastructure is to always expect failures, but then it needs to be recovered fast. When problems occur, often there is a need to reboot the system. As the monolith system becomes bigger, the restarting time becomes longer which means longer downtime. Therefore, reliability and recoverability are hard to accomplish with monolith architecture. Monolith architecture relies on programming language features like function, class or modules packing system to de-couple components. Over time, the application becomes complex because of more functions, classes, modules added to the application, making it hard to reason about and to maintain. Thus, serviceability and manageability are hard to accomplish with monolith architecture. All in all, getting high availability with monolith architecture is challenging. Microservices architecture is designed to overcome that and to get more nines for the system.

2.2 Microservices

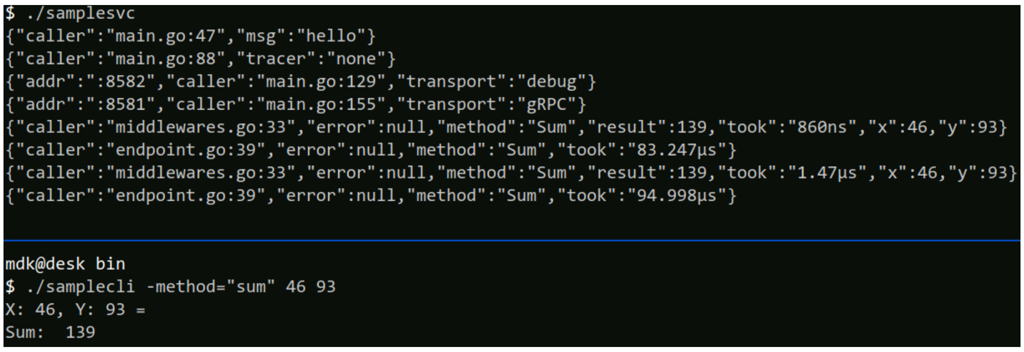
As stated by Sam Newman [5, 2], "microservices are small, autonomous services that work together". Autonomous service means it is a separated entity that can be deployed independently without changing anything else. Microservices communicate over network connectivity. For example, figure 1 shows a microservice that can be reached via HTTP requests.



```
mdk@desk ~
$ curl -XPOST -d'{"s":"hello, world"}' localhost:8080/uppercase
{"v":"HELLO, WORLD"}
mdk@desk ~
$ curl -XPOST -d'{"s":"hello, world"}' localhost:8080/count
{"v":12}
```

Figure 1. An example microservice that can uppercase and count characters of a string.

As shown in figure 1, the microservice has two capabilities: uppercasing all characters of a string or counting the number of characters of a string. Figure 2 shows a microservice that can be reached via gRPC call.



```
$ ./samplesvc
{"caller":"main.go:47","msg":"hello"}
{"caller":"main.go:88","tracer":"none"}
{"addr":":8582","caller":"main.go:129","transport":"debug"}
{"addr":":8581","caller":"main.go:155","transport":"gRPC"}
{"caller":"middlewares.go:33","error":null,"method":"Sum","result":139,"took":"860ns","x":46,"y":93}
{"caller":"endpoint.go:39","error":null,"method":"Sum","took":"83.247µs"}
{"caller":"middlewares.go:33","error":null,"method":"Sum","result":139,"took":"1.47µs","x":46,"y":93}
{"caller":"endpoint.go:39","error":null,"method":"Sum","took":"94.998µs"}

mdk@desk bin
$ ./samplecli -method="sum" 46 93
X: 46, Y: 93 =
Sum:  139
```

Figure 2. An example microservice that can summarize two numbers and be remotely called using gRPC.

As shown in figure 2, the microservice can summarize two numbers.

To build a successful microservices system, there are a few principles to follow as shown in Figure 3.

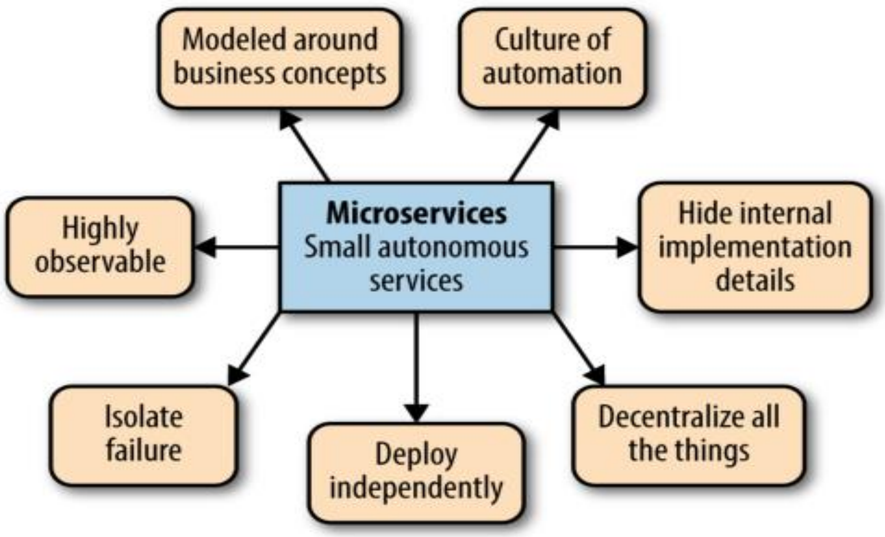

Modeled around business concepts
Culture of automation
Highly observable
Microservices Small autonomous services
Hide internal implementation details
Isolate failure
Deploy independently
Decentralize all the things

Figure 3. Principles of microservices. [5, 246]

As seen in Figure 3, 7 highly important principles are needed. They are explained in more detail below one by one.

Modeled around business concepts

Services in microservices architecture are divided and organized around business capability. It creates a system that models real-world business domains. Domain-driven design [6] can help you find stable, reusable boundaries.

Highly observable

Within microservices architecture, observing the status of a single machine or a server is not enough. Fake events that simulate real-user behavior should be injected to test if the system is working correctly, semantically. Logs and metric statistics should be aggregated to trace the problem down to the source.

Isolate failure

Microservices are not reliable by default. Since microservices use the network to communicate, network timeout might happen. In some situation, make the right tradeoff decision between availability or consistency. Expect failure can happen anytime, anywhere and design the application so that it can tolerate the failure. Monitoring is a vital part when building microservices infrastructure.

Deploy independently

The decision when and why to update a microservice should be decided by the service. Each service should be hosted on a single computing environment so that deploying one service should not have impacts on other services. The delivery team gains confidence when doing deployment by testing as much as possible. Blue-green or canary release techniques should be considered to avoid errors in releasing.

Decentralize all the things

There is an observation named Conway's Law which appears to hold true:

> "Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure." Melvin Conway, 1967. [7]

And the inverse of Conway's Law is also valid and especially relevant to the microservices ecosystem: the organizational structure of a company is determined by the architecture of its product [8]. Because of Conway's Laws, siloed functional teams (separated team focusing only on one part of development like user experience team, backend server-side team) lead to siloed application architecture. A simple change in siloed functional teams can lead to a cross-team project taking time and budgetary approval [9]. To reduce the overhead of siloed functional team, cross-functional teams are introduced. A cross-functional team has the full range of skills: user experience, backend server-side and project management to make the decision and build the feature. For example, Netflix company, which heavily utilizes microservices architecture, has started to adopt cross-functional teams [10]. Conway's Law and the inverse of it hold true for a company like Netflix. Therefore, power should be pushed out of the center, organizationally and architecturally [11].

Hide internal implementation details

A service should evolve independently. A service should not know any implementation details of other services. In fact, each service can have completely different technology stack. The one certainty with technology is change. Select new programming languages, frameworks that best suit for building specific service.

Culture of automation

Human operators are one of the biggest sources of errors in any complex system [12]. There are lots of moving parts to deal with because of complexity in microservices architecture. Automated tests and automated deployment are essential to have a functioning microservices system. Many of systems being built with microservices are being built by a team with extensive experience of Continuous Delivery [9].

3. Continuous Integration and Continuous Delivery

Software application often is a product of collaboration with many members of a team. Continuous Integration (CI) and Continuous Delivery (CD) are software development practices that help team members communicate, detect problems and speed up delivering the application to users. The processes of delivering software to users can be summarized in four steps:

- Build: in this step, depending on the programming languages, either compile the source code (C/C++, Golang, ...) into libraries, executable binaries or package the source code (Python, JavaScript, …).

- Deploy: the result – artifact – of build step is shipped to an environment (server) that can run the artifact.

- Test: various kind of testing technique can be executed: unit test, integration test, performance test, end to end test, security test.

- Release: after tested that the changes meet all requirements in a development environment, deploy the artifact to the production environment where new changes will serve users.

CI is focused on automatically building and testing code, whereas, CD going one step further by (ideally) automatically delivering software to production environment or users. In practice, a team often tests in a staging environment, which try to mimic production environment as much as possible, to ensure code changes are always in a release-able state. After that, the product owner can decide when to deploy by just clicking a button and the steps of releasing should be automatically done. There are few reasons why the final approval is needed before releasing like making sure the graphics look good or application has good enough user experience with nice animation for example. Basically, the final approval is needed for steps that are hard to automate.

3.1 Deployment Pipeline

CD introduces a pattern called deployment pipeline to automate build, deploy, test and release effectively. Figure 4 illustrates the changes moving through the deployment pipeline.

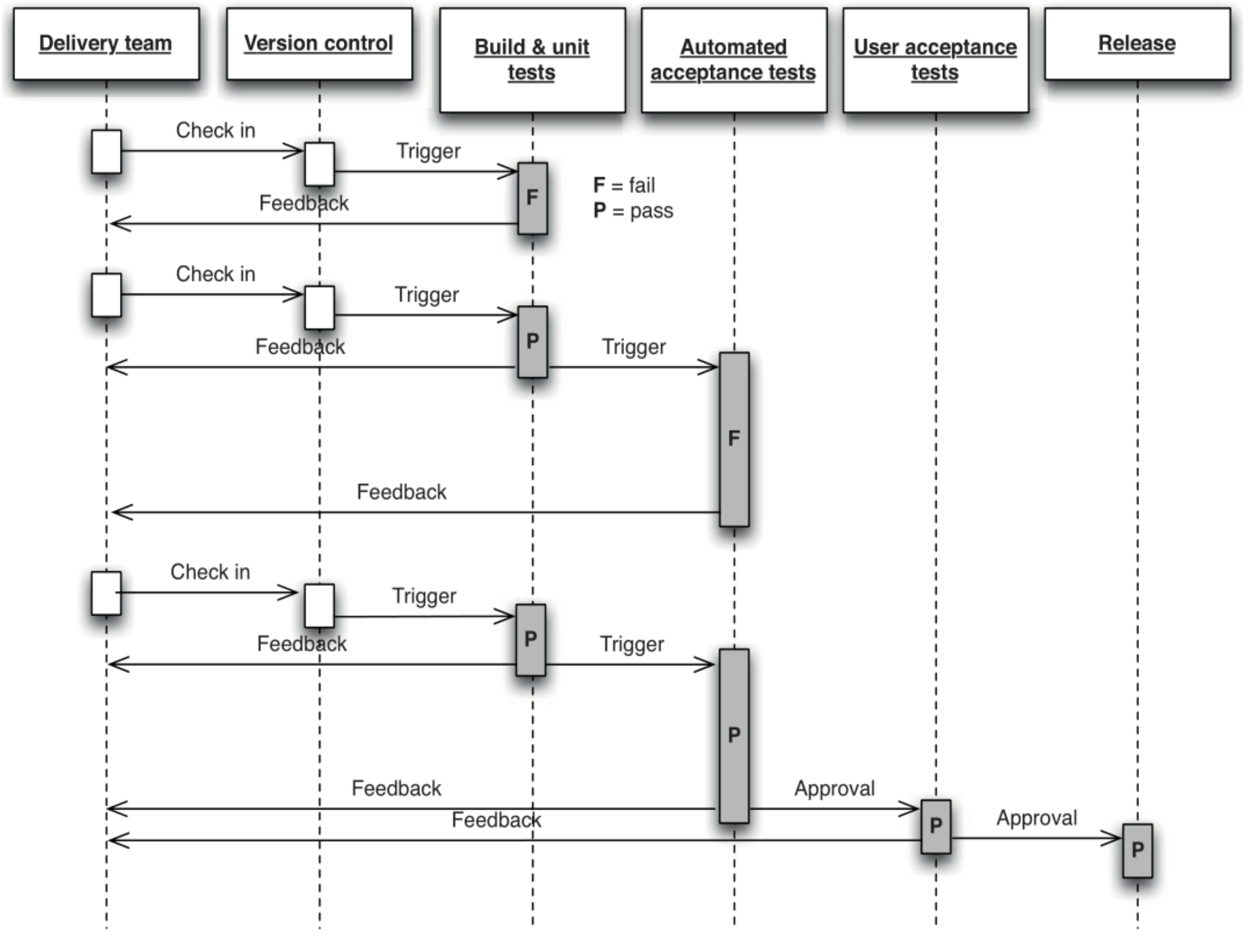


Figure 4. Changes moving through the deployment pipeline. [13, 109]

As showed in figure 4, all the steps from version control up to automated acceptance tests can be considered as parts of the CI process. The final two steps, user acceptance tests and release, can be considered as parts of the CD process. The main purposes of deployment pipeline to achieve are threefold:

- Make build, deploy, test, release process visible to everybody involved (developers, testers, project managers, or even customers).

- Identify problems as early as possible

- Enable teams to release software through a fully automated process.

Figure 4 is greatly simplified, but the automated acceptance tests often involve many steps: configure the staging environments, run smoke tests (cover most of the high-level functions of software but not in depth), run performance tests (ensure latency, startup time, stability, capacity within requirement). Because of many moving parts, all the steps involved in the deployment pipeline need to be automated. It would be risky whenever releasing the software, an engineer needs to manually type in lots of commands.

Since almost all steps are automated, CD encourages a team to make code changes or commits small and frequently, often few times in a working day. Because of small code changes, reasoning and communicating between team members become easier. Small code commits also good for maintaining because it is easier to locate the lines of code that introduce bugs.

Not only the deployment pipeline needs to be automated, but it also needs to run fast. The shorter runtime, the better. Ideal runtime for a deployment pipeline should be within ten minutes. If the deployment pipeline run too slow, it discourages team members to make code changes. Instead, they will avoid waiting for the pipeline by making big commits. Code commits size might be enforced by using code review and making sure all developers follow set of rules while reviewing. However, when deployment pipeline run slow, a company is wasting employee working time.

A deployment pipeline is also a collaboration tool for management. A project manager can keep track of current state of development and delivery process in detail. Security policy or performance specifications should be part of deployment pipeline to ensure customer's requirements. Documentation and reports produced by deployment pipeline can be used as proof to show the customer.

3.2 Releasing Strategy

The last step in a deployment pipeline is rolling out new changes to users. There are two releasing strategies often used to minimize risks of downtime: blue-green and canary deployment.

3.2.1 Blue-green Deployment

In blue-green deployment, there are two production environments as identical as possible. One environment is live and serving users, another one is idle. The network router is the switch that determines which environment is live. When the releasing process is triggered by deployment pipeline, new changes will be deployed into the idle environment. Final testing, often synthetic test, will then be proceeded. After the synthetic test passed, flip environment color by switching the network router. If problems occurred after the flip, roll back to the previous known-good-version is extremely easy, just switch the router again. In practice, the idle environment is only kept running for short period of time (few hours, few days) then destroyed. Figure 5 describes the process of blue-green deployment.

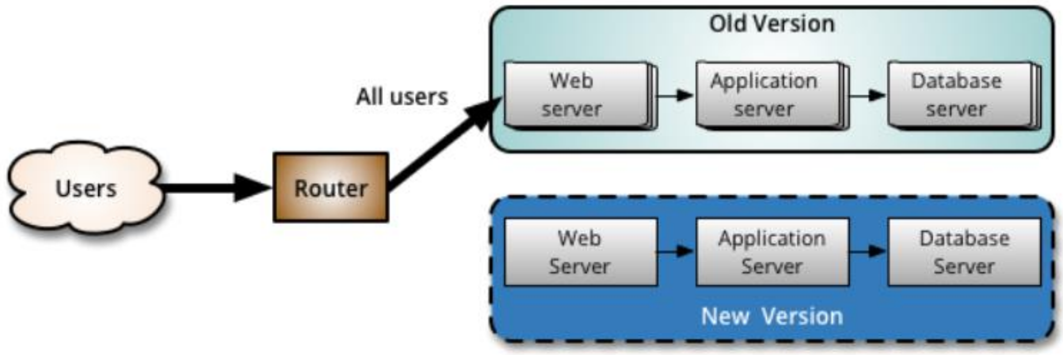


Figure 5. Blue-green deployment. [18]

As shown in figure 5, in blue-green deployment, the network router switches all connections from the old environment to the new environment.

3.2.2 Canary Deployment

Canary deployment has the same fundamental process as blue-green deployment. Instead of only one environment which is live on a blue-green deployment, in a canary release, both environments are live during the release process. However, one environment, the one that is deployed with new changes, serves a smaller amount of traffic compared to the environment. The network router acts as a load balancer and slowly increases traffic to the newly deployed environment. Monitoring and testing are continuously executed while traffic is re-distributed. When all traffic is re-routed, the old environment can be destroyed. If problems occurred, simply route back all traffic to the old environment. The advantage of canary deployment over blue-green deployment is reducing impacts. In case problems occur, because only a small volume of traffic serving the new version, the number of users experiencing the defects is smaller compared to blue-green deployment. Figure 6 describes the process of canary deployment.

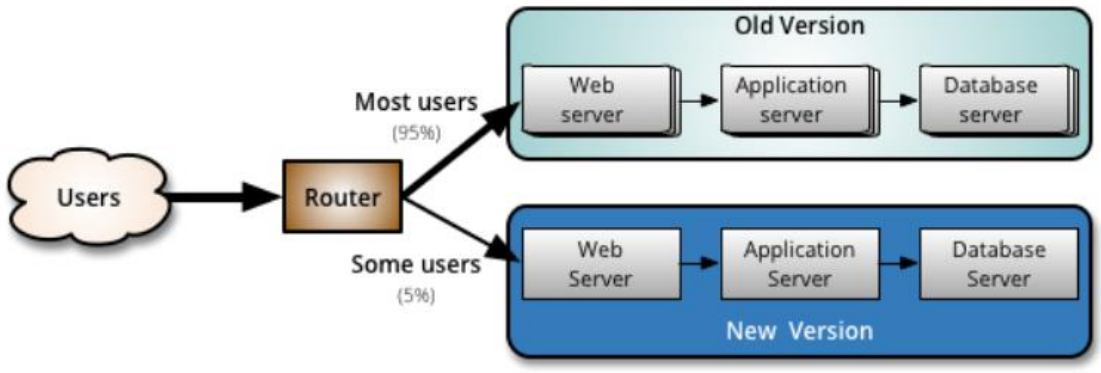


Figure 6. Canary deployment. [18]

As shown in figure 6, in canary deployment, the network router switches some connections first then gradually switches the rest if no problems occurred.

4. Infrastructure as Code

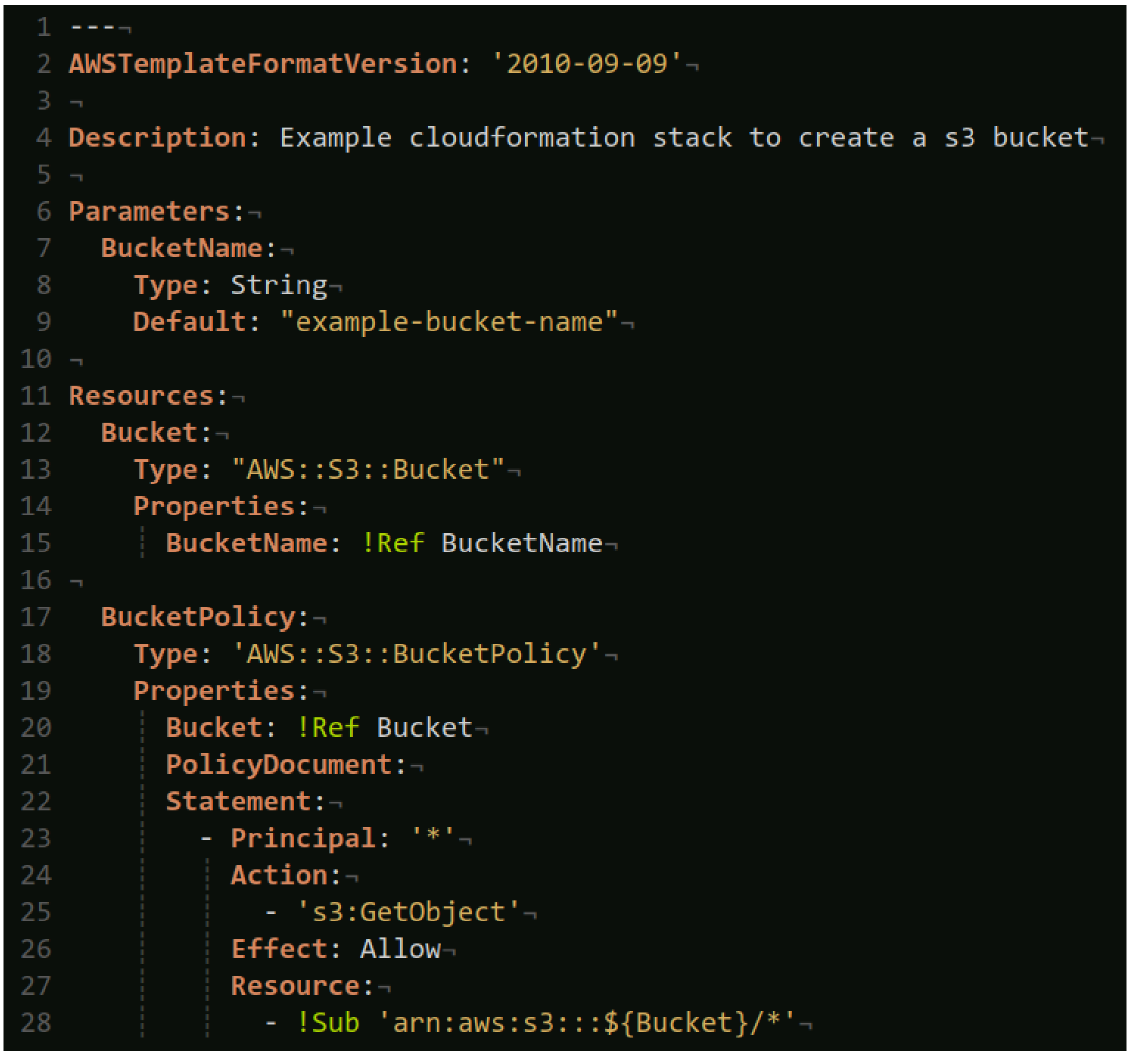
When building microservices backend infrastructure with Continuous Delivery, each microservice will have its own deployment pipeline. For a big scale system, the backend infrastructure might have hundreds of microservices. Each individual microservice has its own specific requirements and configurations. Those microservices also evolve over time, requiring repeatable, consistent routines for creating, changing, managing. With cloud computing platform like Amazon Web Services, all resources, and steps to provision can be described in code. Creating a server, adding storages, making network connections are done with HTTP requests. Infrastructure as Code (IaC) is an approach to infrastructure automation based on practices from software development. [14, 5]. There are two important sets of tools in IaC: infrastructure definition tool and server configuration tool.

4.1 Infrastructure Definition Tool

Infrastructure definition tool helps to provision and to manage barebone infrastructure resources such as storages, servers, and networking. AWS provides software development kit (SDK) to create and manage resources. The SDKs are available in multiple languages: Python, Java, Golang, JavaScript. However, there are tools like Cloudformation, Terraform, which enable us to describe resources in a declarative way. We only need to define resources and their states. Cloudformation or Terraform will check and change AWS resources to match the defined definitions. In this study, Cloudformation is used as infrastructure definition tool.

Cloudformation

AWS Cloudformation template can be written using JSON or YAML format. Both JSON and YAML are data serialization standards. YAML is designed to be easier for human, therefore YAML format will be used to write Cloudformation template. Figure 7 shows a simple Cloudformation template that provisions an S3 bucket (AWS S3 is a service to store and share files):



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

Description: Example cloudformation stack to create a s3 bucket

Parameters:
  BucketName:
    Type: String
    Default: "example-bucket-name"

Resources:
  Bucket:
    Type: "AWS::S3::Bucket"
    Properties:
      BucketName: !Ref BucketName

  BucketPolicy:
    Type: 'AWS::S3::BucketPolicy'
    Properties:
      Bucket: !Ref Bucket
      PolicyDocument:
      Statement:
        - Principal: '*'
          Action:
            - 's3:GetObject'
          Effect: Allow
          Resource:
            - !Sub 'arn:aws:s3:::${Bucket}/*'
```

Figure 7. Cloudformation template to create S3 bucket

In figure 7, an S3 bucket is created with a policy that allows everybody to read files stored in the bucket. The bucket name is referenced by the parameter BucketName, which defaults to "example-bucket-name". When invoking the template, all parameters in Parameters map can be changed, making above template to be re-useable.

4.2 Server Configuration Tool

A server configuration tool can also be called configuration management tool. A configuration management tool helps to deal with details of servers like managing processes, configuration of services running on the servers. There are two models in configuration management tool: push and pull.

In the push model, we only need network connections to all the nodes/servers that we want to manage. When executing the changes, the tool will translate the definitions into machine executable code (for example shellcode), send/push the code over the network to the nodes, run the code and return responses. Ansible is an example of push model.

In the pull model, the managed nodes have an agent running in the background, listening to commands from master nodes. When executing the changes, the definitions stored in master nodes are changed, then agents of slave nodes will periodically fetch definition and execute the changes. Puppet and Chef are examples following pull model.

Immutable server

Traditionally, a configuration management tool such as Ansible, Puppet, Chef are used to deploy new code changes to the servers in place. For the system using pull model tools like Puppet or Chef, the background agent might have an impact on the services inside the server. For example, when the agent executing the changes, computing resources are taken from the services. In the worst case, the agent can crash the whole server, in turn, shut down the server, leading to downtime. Another problem is configuration drift. Configuration drift is a phenomenon where servers in an infrastructure become more and more different from one another as time goes on, due to manual ad-hoc changes and updates, and general entropy [15].

Immutable server is another approach to deploy changes without having the above problems. The main benefit of this approach is that it allows us to be certain what the state of a server is once it has been provisioned [16]. Instead of introducing changes in place to the servers, we build new image (amazon machine images - AMI) containing configured configuration, then boot up new servers and destroy the old servers. If newly deployed servers have problems, we can simply boot up new servers using the previous known-good-version AMI to rollback. The process of building and immutable server AMI can be described in 8 steps:

- Boot a temporary server

- Install and update latest packages and security patches

- Install the configuration management tool like Ansible or Puppet
- Download the configuration definitions from source code repository
- Run Ansible or Puppet with downloaded configuration definitions to configure itself (masterless mode)
- Uninstall Ansible or Puppet
- Create server image AMI by dumping the storage blocks of the server (AWS provided application programming interface to do this)
- Shutdown the temporary server

Of course, immutable server has cons. For example, because of dumping the whole storage blocks, it increases the cost of AWS charge for data storage usages. Addition steps needed to tag and manage the versions of AMI. However, experience shows that advantages outweigh the disadvantages.

5. Building the Infrastructure

The author’s main responsibilities are to automate the releasing process of all microservices and to ensure high availability for all infrastructures. The tasks include:

- Research available CI, CD tools.
- Evaluate and select suitable CI, CD tools by creating a deployment pipeline for a fake microservice.
- Automate creating the deployment pipeline by applying IaC practice. Put resources and configurations to code that can be reused.
- Provision a development infrastructure for real backend microservices by adapting and modifying the deployment pipeline of the fake microservice.
- After having the development infrastructure, start looking for solutions to make the infrastructure more resilient and fault tolerant.
- Continue by looking for solutions to auto scale the infrastructure.
- Propose solutions and discuss with colleagues.
- When the company gets new customer, replicate the infrastructure to a new one.
- Repeat all above steps and continuously improve both the development infrastructure of the company and customers’ infrastructures.

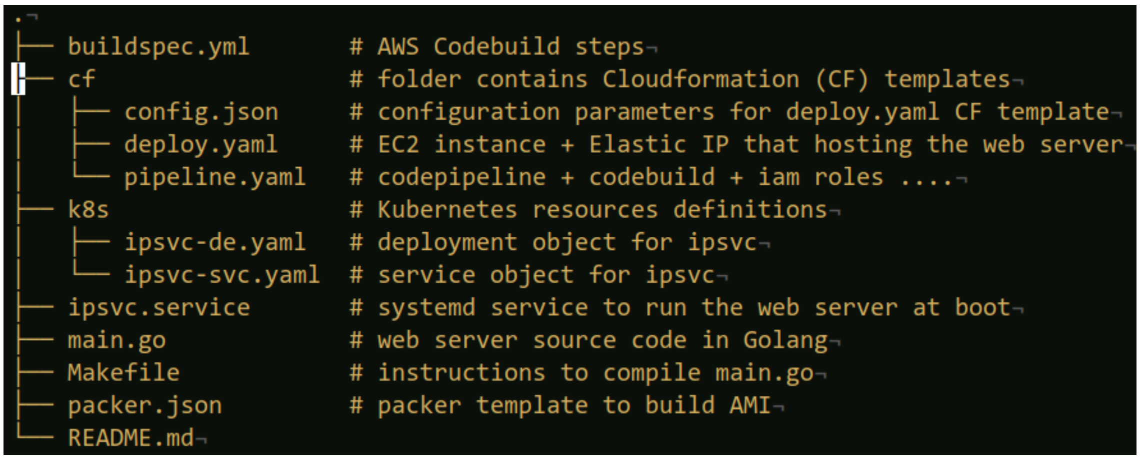
There are many free and self-hosted tools for CI, CD, IaC like Gitlab CI, Jenkins, GoCD, Terraform. However, self-hosting those tools requires the author maintaining another set of servers. With the strict deadline for the minimum viable product, the author decided to reuse available AWS services as much as possible. The decision was made that Cloudformation for templating infrastructure as code, Codebuild for CI builder and Codepipeline for deployment pipeline.

There are two methods to run a microservice in AWS cloud. The first and simpler way is to run the microservice in an EC2 instance (virtual machines). The second one runs the microservice in a container. The deployment of both EC2 instances and containers is demonstrated in the following chapter. While showing both ways of deployment, observations and discussions when and how to select a proper deployment method in different situations are given.

Since the source code of the company's microservices is confidential, a simple microservice for demonstration purpose is made. There is no need for a complex microservice because the goal is to show how to apply IaC and CD practices to do EC2 instances and containers deployment.

5.1 The Example Microservice

The example service for this section will be a web server that renders a simple HTML page showing the local IPv4 address of the server and the version number of the service. Showing the local IPv4 address enables checking the autoscaling when multiple instances or containers of the same microservice are load balanced. Figure 8 shows the structure of the source code repository.

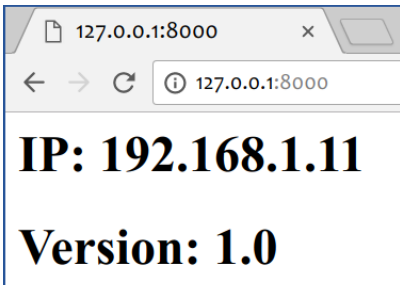


```
.¬
├── buildspec.yml       # AWS Codebuild steps¬
├── cf                  # folder contains Cloudformation (CF) templates¬
│   ├── config.json     # configuration parameters for deploy.yaml CF template¬
│   ├── deploy.yaml     # EC2 instance + Elastic IP that hosting the web server¬
│   └── pipeline.yaml   # codepipeline + codebuild + iam roles ....¬
├── k8s                 # Kubernetes resources definitions¬
│   ├── ipsvc-de.yaml   # deployment object for ipsvc¬
│   └── ipsvc-svc.yaml  # service object for ipsvc¬
├── ipsvc.service       # systemd service to run the web server at boot¬
├── main.go             # web server source code in Golang¬
├── Makefile            # instructions to compile main.go¬
├── packer.json         # packer template to build AMI¬
└── README.md¬
```

Figure 8. Structure of code repository. Github: https://github.com/mdk194/ipsvc

As shown in figure 8, the code repository contains source code and definitions of both infrastructure and the service configuration.

The example service will be called ipsvc. Figure 9 shows the example ipsvc running in localhost.



```
IP: 192.168.1.11

Version: 1.0
```

Figure 9. Example ipsvc service running on localhost.

As shown in figure 9, the example ipsvc service shows the local IP address and version number when accessed.

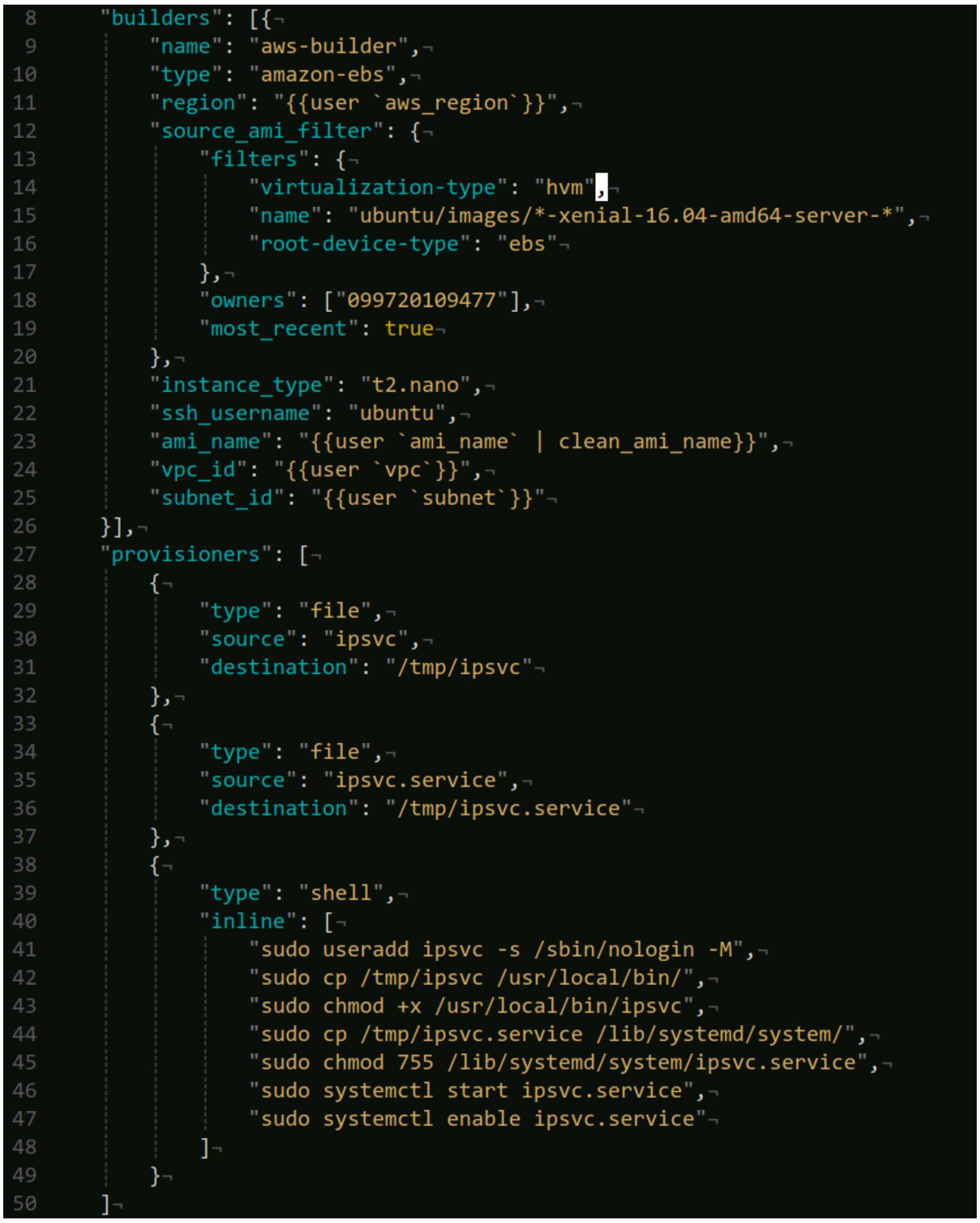
5.2 EC2 Instance Deployment

The main purpose of this section is to demonstrate immutable server, deployment pipeline, and Infrastructure as Code concepts. Codebuild is used as CI builder to compile server binary and build AMI. Cloudformation is used for defining AWS resources and for releasing process also. Codepipeline is used to make the deployment pipeline. We will use Codecommit as git version control.

5.2.1 Build Immutable Server AMI with HashiCorp Packer

Packer is an opensource tool to build virtual machine image from a source configuration [19]. We will use Packer to build custom AMI image base on Ubuntu image. The built AMI is configured so that when booted up, the init system will automatically run the web server.

Figure 10 shows the packer template to build the AMI for ipsvc.



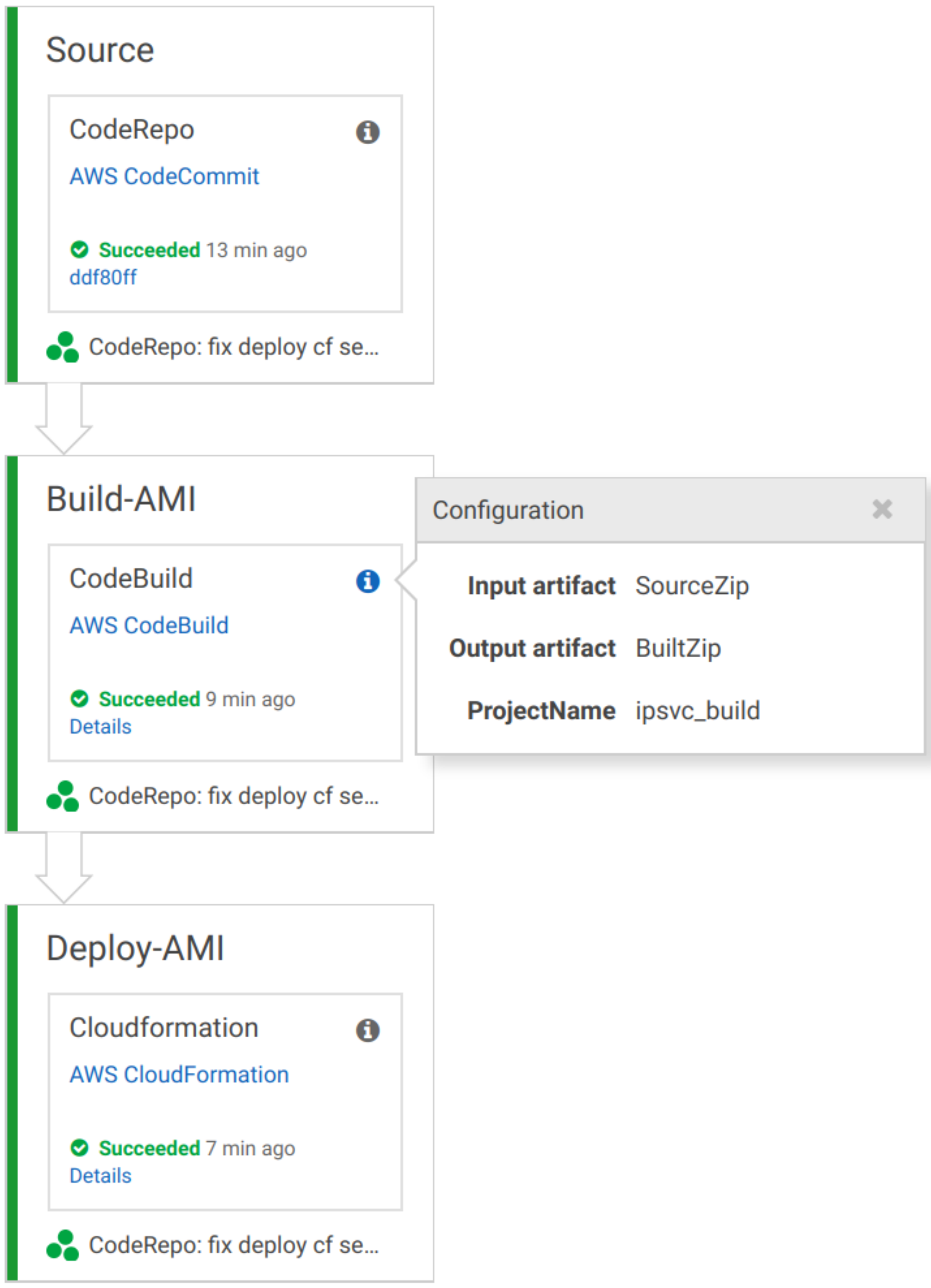
```
    "builders": [{
        "name": "aws-builder",
        "type": "amazon-ebs",
        "region": "{{user `aws_region`}}",
        "source_ami_filter": {
            "filters": {
                "virtualization-type": "hvm",
                "name": "ubuntu/images/*-xenial-16.04-amd64-server-*",
                "root-device-type": "ebs"
            },
            "owners": ["099720109477"],
            "most_recent": true
        },
        "instance_type": "t2.nano",
        "ssh_username": "ubuntu",
        "ami_name": "{{user `ami_name` | clean_ami_name}}",
        "vpc_id": "{{user `vpc`}}",
        "subnet_id": "{{user `subnet`}}"
    }],
    "provisioners": [
        {
            "type": "file",
            "source": "ipsvc",
            "destination": "/tmp/ipsvc"
        },
        {
            "type": "file",
            "source": "ipsvc.service",
            "destination": "/tmp/ipsvc.service"
        },
        {
            "type": "shell",
            "inline": [
                "sudo useradd ipsvc -s /sbin/nologin -M",
                "sudo cp /tmp/ipsvc /usr/local/bin/",
                "sudo chmod +x /usr/local/bin/ipsvc",
                "sudo cp /tmp/ipsvc.service /lib/systemd/system/",
                "sudo chmod 755 /lib/systemd/system/ipsvc.service",
                "sudo systemctl start ipsvc.service",
                "sudo systemctl enable ipsvc.service"
            ]
        }
    ]
```

Figure 10. Packer template to build and configure AMI.

As shown in figure 10, the packer builder type "aws-builder" is specified and the latest Ubuntu image is selected as the base image. The packer supports multiple provisioner types to configure the image, but in this example, file provisioner and remote shell provisioner is used. The file provisioner will copy statically compiled server binary and the systemd service to the AMI. The shell provisioner takes care of putting binary and systemd service into a correct location with correct permission then enable the service to run at boot.

5.2.2 Deployment Pipeline

Figure 11 shows the deployment pipeline created by the Cloudformation template.



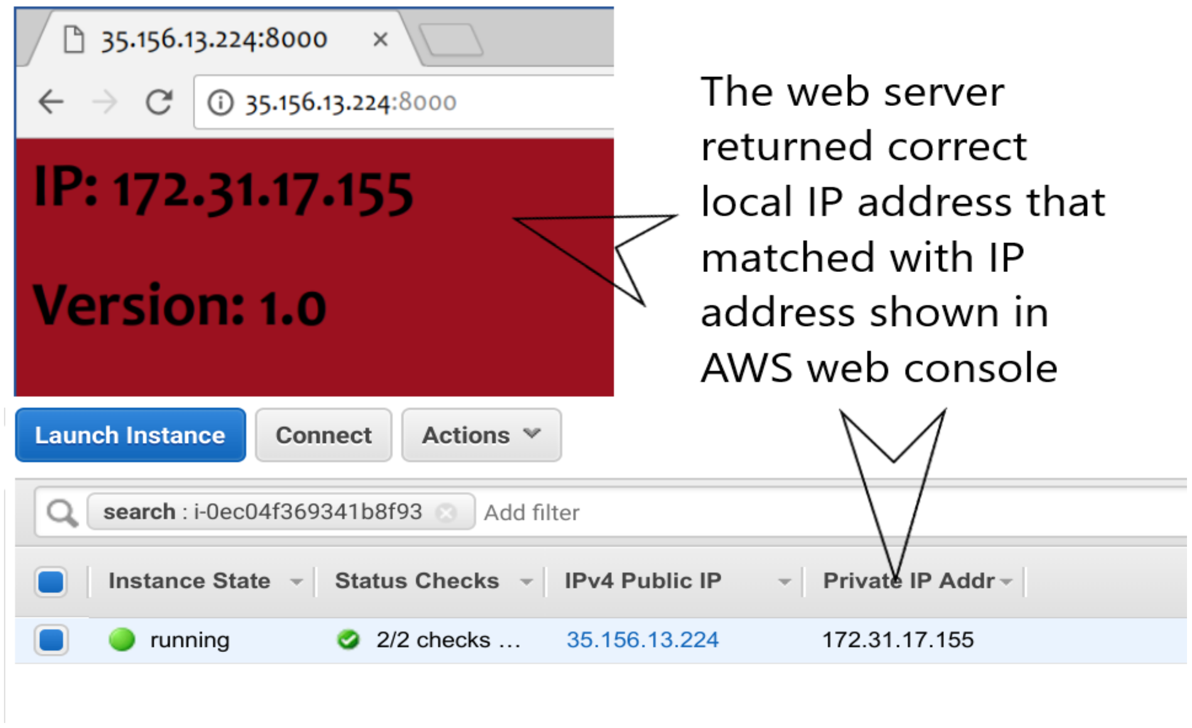
Input artifact	SourceZip
Output artifact	BuiltZip
ProjectName	ipsvc_build

Figure 11. Deployment pipeline for ipsvc service.

As shown in figure 11, the deployment pipeline has three stages run in this order: source, build and deploy. The source state detects changes from code repository then trigger pipeline run. The build states will take the source code, compile binary and invoke packer to build AMI. Finally, the deploy state boot up new EC2 instance base on new built AMI, destroy the old instance, then swap the Elastic IP to serve the new version of the web server.

5.2.3 Working Web Server

When the deploy stage has finished successfully, testing the web server can be done by accessing to the instance public IP address. Figure 12 depicts a working web server.



IP: 172.31.17.155

Version: 1.0

The web server returned correct local IP address that matched with IP address shown in AWS web console

Instance State	Status Checks	IPv4 Public IP	Private IP Addr
running	2/2 checks ...	35.156.13.224	172.31.17.155

Figure 12. Working example ipsvc web server deployed in an EC2 instance.

As shown in figure 12, the deployed ipsvc web server returns local IP address of the server that matches with the IP address shown in AWS console dashboard.

After months doing the deployment with this method, the team noticed some issues. First, packer takes time to build the AMI. The minimum time to build an AMI is three minutes. For AMI with complex configuration, the time to build can take more than ten minutes. The deployment pipeline should run as fast as possible. A faster deployment pipeline allows developers to integrate and test changes more frequently. Second, it is a waste of CPU, RAM and blocks storage resources for many use cases. There are services that only need few megabytes of RAM and do not need storage at all. Since EBS volume is required for EC2 instance and the minimum size of EBS is four gigabytes, making the AMI size to be at least 4GB. Keeping multiple versions of AMI for rollback is needed, leading to the high cost of hosting the system. Third, the author needs to script and create quite lots of custom tools to maintain the system, even with lots of documentation and already available open-source tools. Those reasons lead to the exploration of Container and Kubernetes.

5.3 Container Deployment with Kubernetes

Container is new technology that is a good fit for microservices style system. AWS has built-in EC2 Container Service (ECS) to manage docker container. Another alternative for managing container is Kubernetes (k8s). After days evaluating and experimenting, it was decided to use k8s as container orchestration. With k8s, vendor lock-in is avoided and the infrastructure can be migrated to another cloud computing providers like Google Cloud Platform or Microsoft Azure. Section 5.3.1 introduces what is a container. Section 5.3.2 introduces k8s and some k8s terminologies. Section 5.3.3 demonstrates container deployment with k8s and show why most of the use cases, container deployment with k8s is a better solution than EC2 instance AMI deployment.

5.3.1 Container

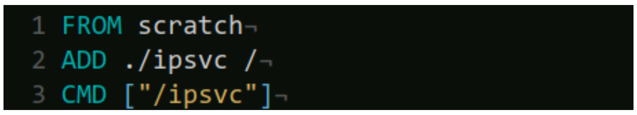
AWS EC2 or Google Compute Engine (GCE) use hardware virtualization (also known as virtual machines) to provide isolated computing environment. Hardware virtualization allows running multiple operating systems: Linux, Windows, BSD on same physical hardware: mainboard, CPU, RAM. When EC2 instance deployment is done like in section 5.1 above, a virtual machine is deployed. Container does not use hardware virtualization technique but relies on kernel (operating system) features to do isolation.

Historically, a UNIX-style operating system has used the term jail to provide resources protection for processes. Since 2005, after the release of Sun's Solaris 10 and Solaris Containers, container has become preferred term instead of jail [20, 4]. There are multiple implementations of container engine: LXD from Ubuntu, RKT from CoreOS, Docker from Docker with Docker is the most ubiquitous engine at the time of writing this document. Docker is implemented using Linux kernel features. The containers that Docker builds are isolated with respect to eight aspects [20, 6]:

- PID namespace: Process identifiers and capabilities
- UTS namespace: Host and domain name
- MNT namespace: File system access and structure
- IPC namespace: Process communication over shared memory
- NET namespace: Network access and structure
- USR namespace: Usernames and identifiers
- Chroot(): Controls the location of file system root
- Cgroups: Resource protection

To deploy a Docker container, a Docker image is built then shipped to a Docker registry. A Docker image is a bundled snapshot of all the files that should be available to a program running inside a container [20, 7]. Building Docker image in high-level viewpoint is same as building AMI. A base Docker image is selected, then install needed dependencies, add service binary and its configuration to the image. The result is a new docker image ready to be shipped to a Docker registry. The Docker registry is a stateless, highly scalable server-side application that stores and lets you distribute Docker images [21].

Figure 11 shows the content of the Dockerfile for the ipsvc service.



```
FROM scratch
ADD ./ipsvc /
CMD ["/ipsvc"]
```

Figure 11. Dockerfile for ipsvc service

As shown in figure 11, since the binary is statically linked, there is no need to install any libraries or dependencies, so the base image is the scratch image. A scratch base image means nothing or empty. After that, the compiled binary is copied to the Docker image and specify the starting command by executing the binary.

The Docker image of ipsvc service has the size of 5.53MB which is tiny compared to the AMI with the size of 4GB. Since the size of Docker image is small, the cost of storage on AWS is vastly reduced. Building the Docker image is faster than AMI, less than minutes, leading to quick feedback cycle for deployment pipeline.

5.3.2 Kubernetes

Kubernetes (k8s) is an open-source system for automating deployment, scaling, and management of containerized applications [22]. From the high-level point of view, Kubernetes can be described as a scheduler continuously looping through three steps:

- Keep track of state of current resources
- Look up the desired state
- Manage resources to meet the desired state

To get started with k8s, a running k8s cluster is needed. K8s cluster is constructed by a group of nodes. The nodes backing up a k8s cluster can be physical machines or virtual machines (EC2 instances) in a cloud platform. There are multiple ways to get a functional k8s cluster. For local testing and experimenting, minikube [23] can be used. When running in production, a managed k8s platform as a service such as Google Container Engine, Openshift can be used. AWS do not provide managed k8s cluster as service, but there are open-source tools to install k8s on top of AWS EC2: kops [24], kube-aws [25]. K8s abstracts the process of managing computing resources by introducing new concepts. The following explains a minimal set of concepts that need to be understand before being able to deploy the ipsvc example service.

Pod

A pod is a group of one or more containers with shared network and storage. K8s pod is the smallest object/resource that can be scheduled by the k8s scheduler. To run a service, the service is wrapped in a pod. Each pod will have its own IP address. K8s treats pod as an ephemeral entity and can re-schedule it at any time. Therefore, the service can be terminated at any time. Conversely, when the service crashed, k8s will auto re-schedule the service.

Labels and Selectors

Labels are key, value pairs attach to a k8s object. Selectors select and group k8s objects base on labels attached to the objects. Selectors support equality-based requirement (environment = production) or set-based requirement (environment in production and development). Labels and Selectors together is a powerful tool to implement different deployment strategies like blue-green deployment, canary deployment, which we will dive deep into later.

Replication controller and replica set

Replication controller ensures a specific number of pod replicas are running at any one time. K8s replica set is the newer version of replication controller. The only difference is that the replica set supports the set-based requirement selector. The replication controller or replica set are the components in k8s that make the service self-healing. For example, you define to run an application with replica count equaling one. When the service crashes, the replica set will automatically create another instance of the application. The replication controller and replica set are barebone components supporting horizontal scaling for the service.

Deployment controller

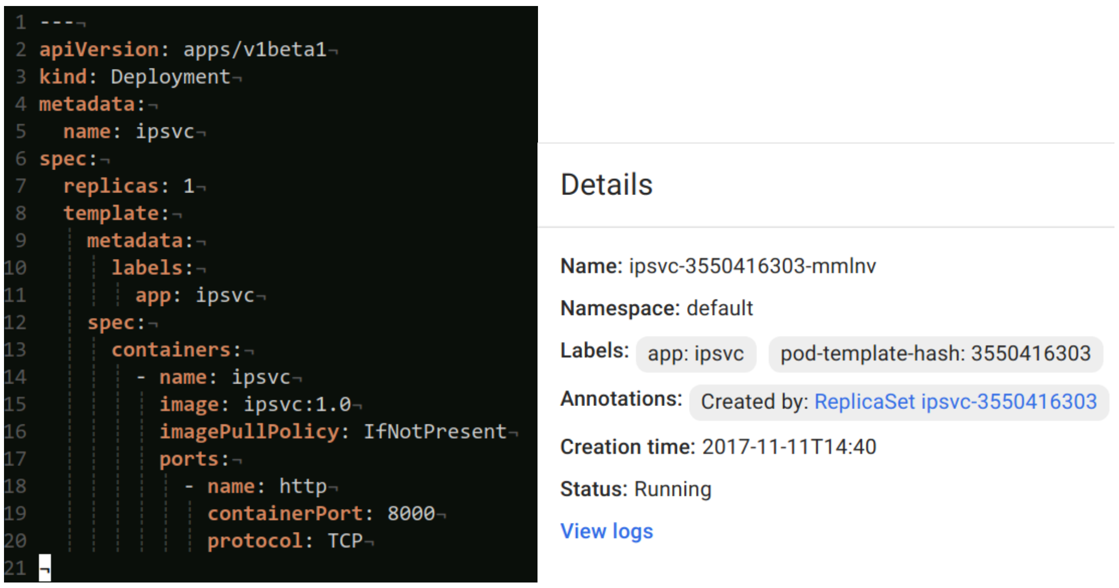
A deployment controller provides declarative updates for the pod and replica set. Typically, the pod and replica set for a service are defined in a deployment controller object. Therefore, definitions such as pod environment, storage and mount point, how many replicas of the pod, computing resources (CPU, RAM) requirement can be found in a deployment controller object.

Service

K8s service defines a logical set of pods and a policy to access them. Simply put, k8s service select the pods by label selector and then defines accessibility to the pods. A pod can be exposed for public access so that users or clients can connect to it, or be exposed internally so only other pods can connect to it. K8s service provides support for service discovery problem of microservices architecture. A service can be discovered by environment variables (IP address and port number) or by cluster DNS records.

5.3.3 Deployment with Kubernetes

Figure 12 show the definition of the k8s deployment object for ipsvc service.



```yaml
---
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: ipsvc
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: ipsvc
    spec:
      containers:
        - name: ipsvc
          image: ipsvc:1.0
          imagePullPolicy: IfNotPresent
          ports:
            - name: http
              containerPort: 8000
              protocol: TCP

```

Details

Name: ipsvc-3550416303-mmlnv

Namespace: default

Labels: app: ipsvc pod-template-hash: 3550416303

Annotations: Created by: ReplicaSet ipsvc-3550416303

Creation time: 2017-11-11T14:40

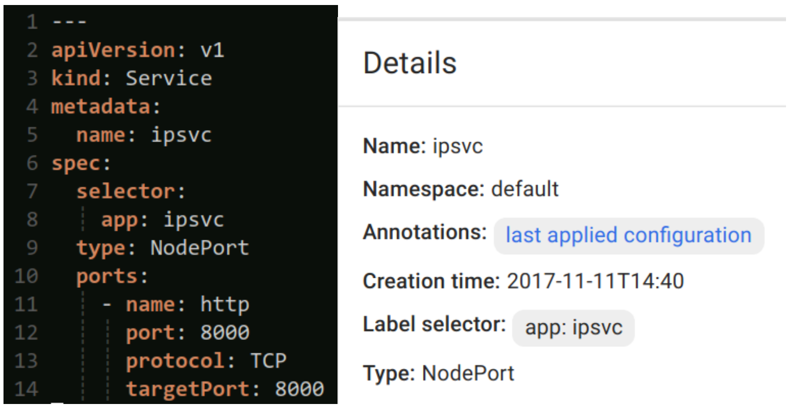
Status: Running

View logs

Figure 12. Ipsvc k8s deployment definition and details of pod created.

As illustrated in figure 12, a pod is created by k8s with labels defined by the template. The created pod has a random name managed by the k8s replica set.

Figure 13 shows the k8s service object for the ipsvc service.



```
---
apiVersion: v1
kind: Service
metadata:
  name: ipsvc
spec:
  selector:
    app: ipsvc
  type: NodePort
  ports:
    - name: http
      port: 8000
      protocol: TCP
      targetPort: 8000
```

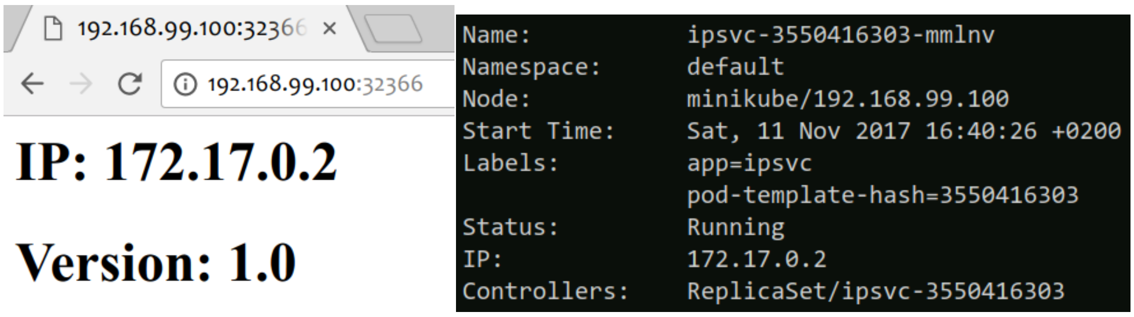
Details

Name: ipsvc
Namespace: default
Annotations: last applied configuration
Creation time: 2017-11-11T14:40
Label selector: app: ipsvc
Type: NodePort

Figure 13. Ipsvc k8s service definition and details of service created.

As illustrated in figure 13, the created k8s service selects the pod with label app: ipsvc. The k8s service exposes the pod and make it publicly accessible. Since the type of the service is NordPort, k8s will map a random node port in range 30000-32000 by default to the pod's port 8000. In this case, the ipsvc web server can be accessed at port 32366.

Figure 14 shows the working ipsvc service deployed to a k8s cluster.



IP: 172.17.0.2

Version: 1.0

```
Name:           ipsvc-3550416303-mmlnv
Namespace:      default
Node:           minikube/192.168.99.100
Start Time:     Sat, 11 Nov 2017 16:40:26 +0200
Labels:         app=ipsvc
                pod-template-hash=3550416303
Status:         Running
IP:             172.17.0.2
Controllers:    ReplicaSet/ipsvc-3550416303
```

Figure 14. The ipsvc container deployed to a k8s cluster.

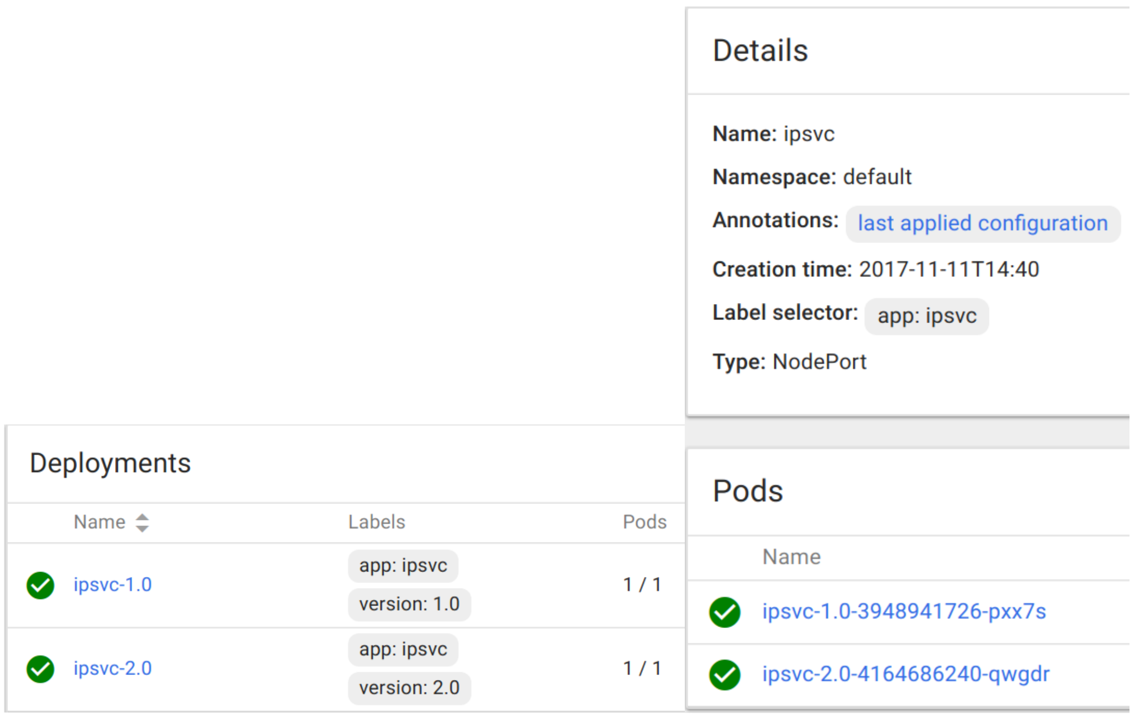
As shown in figure 14, the container running the ipsvc has local IP address matches with the returned IP of the ipsvc web server.

Canary deployment with kubernetes

Doing canary deployment in k8s is simple. Two conditions need to be ensured:

- Having two k8s deployment objects: one for old version, one for new version
- Making the selector of k8s service object select both those deployment objects

For example, the old deployment object, which running ipsvc pod version 1.0, has pods labeled: "app=ipsvc". The new deployment object has pods labeled: "app=ipsvc, version=2.0". The selector of service object selects the label: "app=ipsvc". Kubernetes service will then round-robin routed the connection to the old pods and new pods. If one pod runs version 1.0, another pod runs version 2.0, then half of traffic is routed to the new pod. Figure 15 illustrates canary deployment with k8s.



Details

Name: ipsvc
Namespace: default
Annotations: last applied configuration
Creation time: 2017-11-11T14:40
Label selector: app: ipsvc
Type: NodePort

Deployments

Name	Labels	Pods
✓ ipsvc-1.0	app: ipsvc version: 1.0	1 / 1
✓ ipsvc-2.0	app: ipsvc version: 2.0	1 / 1

Pods

Name
✓ ipsvc-1.0-3948941726-pxx7s
✓ ipsvc-2.0-4164686240-qwgdr

Figure 15. Canary deployment using k8s service and labels selector.

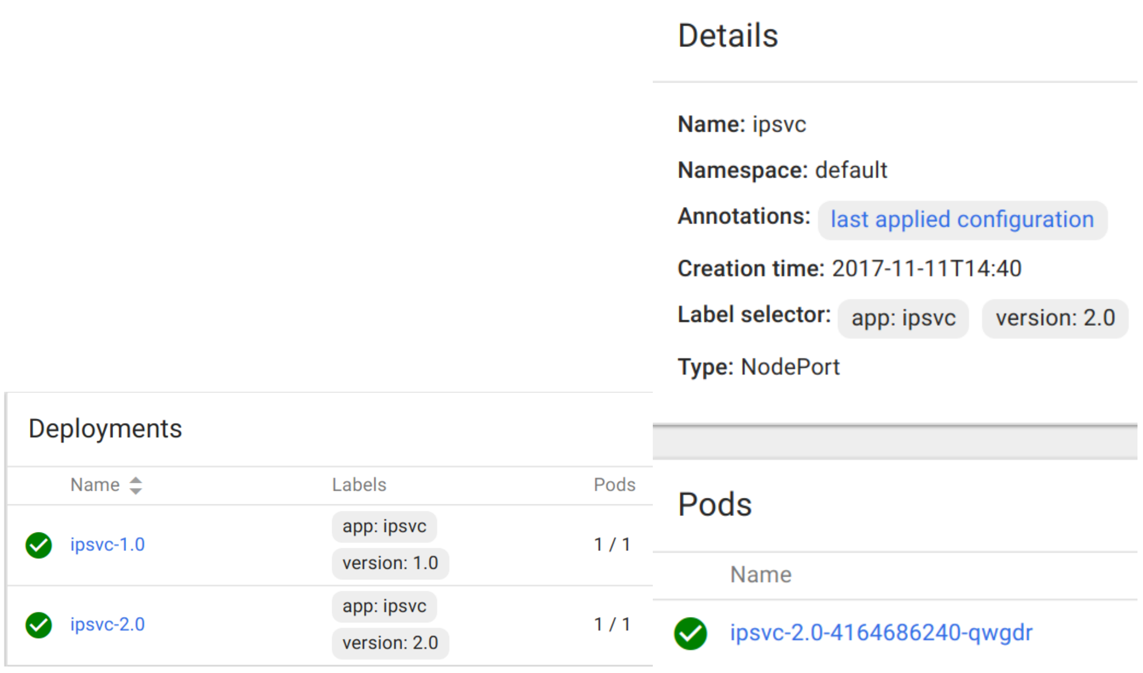
As illustrated in figure 15, the k8s service selects both old and new pods. Therefore, traffic is divided to both pods.

Blue-green deployment in kubernetes

For blue-green deployment with k8s, two conditions need to be ensured:

- Having two deployment objects: one for old version, one for new version
- Making the selector of k8s service object select only one version

For example, the ipsvc pod version 1.0 has labels: "app=ipsvc, version=1.0". The selector of k8s service selects version 1.0 only by selecting labels: "app=ipsvc, version=1.0". The ipsvc pod version 2.0 has labels: “app=ipsvc, version=2.0”. After the pod for version 2.0 created and initialized without error, the labels selector of k8s service is changed to: "app=ipsvc, version=2.0". Now traffic will only be routed to the pod running version 2.0 and the deployment for version 1.0 can be deleted.
Figure 16 illustrates blue-green deployment using k8s service and labels selector.


Details

Name: ipsvc

Namespace: default

Annotations: last applied configuration

Creation time: 2017-11-11T14:40

Label selector: app: ipsvc version: 2.0

Type: NodePort

Deployments

	Name	Labels	Pods
✓	ipsvc-1.0	app: ipsvc version: 1.0	1 / 1
✓	ipsvc-2.0	app: ipsvc version: 2.0	1 / 1

Pods

	Name
✓	ipsvc-2.0-4164686240-qwgdr

Figure 16. Blue-green deployment using k8s service and labels selector.

As illustrated in figure 16, after the service label selector is changed to “app: ipsvc, version: 2.0”, the k8s service routes traffic to the version 2.0 pod only even though there are pod running version 1.0 and pod running version 2.0.

5.4 Fault Tolerance and Autoscaling

This section discusses about fault tolerance and autoscaling for two types of deployment: EC2 instance and Containers.

5.4.1 Fault Tolerance

A fault tolerant microservice is the one able to withstand both internal and external failures. Internal failures are those coming from the microservice itself like code bugs, memory leak, deadlock, panic. External failures are the factors such as machine power outages, or network connectivity.

External failures

When hosting infrastructure in cloud computing platform, external failures cannot be predicted. Failures can happen from instance level to availability zone (AZ), to region or even all regions. One common strategy to mitigate Amazon EC2 outages is to spread the EC2 instances over multiple AZs or multiple regions. Implementing this strategy often require running redundant instances of one service for fast fail-over.

Implementing multi AZs failover in one region is simple. Instances are attached to an Elastic Load Balancer (ELB) and the health status of attached instances are continuously monitored. If an attached instance failed the health status ping, remove that server and boot up a new one then attaches it to the same ELB. This is straightforward for EC2 instance deployment style in AWS. K8s automatically re-schedule or spread the pods to multiple nodes. Therefore, implementing multi AZs failover in k8s is also simple, just ensure the nodes hosting the pods are spread evenly across AZs.

Multi regions failover can be implemented using DNS. Multiple entries can be added to one DNS record, with each entry is the DNS record of an ELB in a region. This strategy works for both EC2 instance or k8s deployment.

Spreading EC2 instances for fast failover is simple and fast to implement. The biggest concern with this strategy is cost-effectiveness. The smallest and cheapest instance type that AWS provides is t2.nano with 1 virtual CPU and 512MB of RAM. It is a waste of computing resources when running multiple EC2 t2.nano instances when the microservice take little CPU time and less than 512MB of RAM. For example, the ipsvc example service takes at peak only 5MB of RAM while running. Since k8s can allocate and manage computing resources in smaller trunks, it is the winner in this case.

Internal failures

Software development is hard. Even after lots of testing: unit tests, regression tests, integration tests, problems such as deadlock or memory leak can still happen and will eventually lead to downtime of the service. Those services are still functioning correctly. While running, they still provide value for the system. One way to mitigate this kind of problems is simply by restarting the whole service. This strategy works well with stateless service, but not for stateful service. For stateful service, often there is only one instance of the service running at any time. Restarting a stateful service will lead to downtime, depending on how fast the service will be re-spawned, recovered. Starting an EC2 instance takes more than one minute from experience. Starting a pod with k8s takes, most of the time, only a few seconds (when the container image is already present). Therefore, recovering a stateful service in k8s is often faster, leading to less downtime. Containers deployment with k8s is also the winner in this case.

5.4.2 Autoscaling

Nowadays, an application can suddenly become prominent because of social media. During that time, lots of traffic is generated. Depending on the kind of application, users might only use the service during a specified time of the day. On-demand scaling is a hard requirement for modern architecture. The process of scaling is the process of adding more power to the system.

There are two types of scaling: horizontal scaling (scale in/out) and vertical scaling (scale up/down). Vertical scaling means taking existing actors in a system and increasing their individual power [26]. Horizontal scaling means adding more actors to the system [26].

For vertical scaling of infrastructure, for EC2 instance deployment, the EC2 instance type can be changed to make the instance stronger or weaker. For containers deployment with k8s more computing resources are added to the pods. Since the maximum capacity for a pod in k8s is bound by the node hosting the pod. Therefore sometimes, vertical scale up of the pod requires scaling up the EC2 instance.

For horizontal scaling of infrastructure, for EC2 instance deployment, auto scaling group [27] feature can be used. The desired number of EC2 instances for idling system is set. That number often bigger or equal two for fault tolerance discussed above. The traffic coming to the ELB is continuously monitored, and the computing resources usage of all servers are aggregate. When the ELB has a certain amount of traffic or the aggregated computing resource usage meets a threshold (say 80% utilization), then the instances count can be increased to meet the demand. For k8s deployment, a feature called horizontal pod autoscale [28] can be used to automatically adding pod replicas. Since the capacity of a k8s cluster equals total capacity of all the nodes (EC2 instances). In conjunction with horizontal pod autoscaling, a strategy to scale the cluster is needed. The same autoscaling group feature can be used to scale k8s cluster with custom scripts, but k8s have a feature called cluster autoscaler [29] to manage the auto scaling group automatically.

Horizontal scaling is favored over vertical scaling. Since there is limit for the instance type that AWS provides, but no limit about the number of EC2 instances can be launched (to be precise, it is limited by physical capability, but the number is huge and can be considered as infinite). On-demand scaling works better with horizontal scaling. Changing EC2 instance type require restarting the instance leading to downtime. While adding/removing instances to an auto scaling group, attaching/detaching instances to an ELB can happen asynchronously, the power capacity immediately increased/decreased without any downtime.

In practice, horizontal and vertical scaling should be considered at the same time to find the balance. A robust monitoring system is a must to identify the bottlenecks and planning for the capacity to ensure a performant system.

5.5 Discussion

As discussed when experimenting the deployment of EC2 instances and containers with k8s, in most use cases, k8s tend to be the better solution. However, containers and k8s are quite young technologies. Container’s runtime engine is not as well studied as the operating system. Therefore, sometimes containers might not able to utilize the computing resources as effective as directly running on top of the operating system. Container’s runtime might also not be as stable as the operating system. For a critical system like a database for example, where reliability is the most important aspect, deploying the database system using EC2 instance style might be a better solution compared with container deployment. Analyzing and selecting the right solutions is significant to build a high availability infrastructure.

6. Conclusion

The main purpose of this study was to build a high availability infrastructure in cloud. The requirements were strict. The project team had only a few months of researching and implementing before releasing the first minimum viable product. Microservices architecture with Continuous Delivery practice helped the team meet the requirements.

There are many topics this thesis has not touched upon such as logging, monitoring, or security. They are important aspects to consider. For instance, logging and monitoring help to debug and to trace bugs, planning for scaling, optimizing performance and saving cost as well as alerting for a quick incident response. Security helps to protect the company brand reputation. Each one of these topics is worth its own study. Having solid infrastructure and the right engineering approach allow to continuously improve the quality of the product.

The project was a great learning process. Even though only one cloud service provider, AWS, was selected, core ideas of cloud computing could be directly mapped to other providers. Through this project, it was easy to see that automation is the key to success. To sum up, it enables both learning more and doing more.

References

1 Wittig, Andreas; Wittig, Michael. Amazon Web Services in Action. Manning Press; 2016.

2 Martin Fowler. Continuous Integration [online]. 01 May 2006. URL: https://martinfowler.com/articles/continuousIntegration.html Accessed 25 October 2017.

3 Jez Humble. Continuous delivery [online]. URL: https://continuousdelivery.com/ Accessed 25 October 2017.

4 Floyd Piedad and Michael W. Hawkins; High Availability: Design, Techniques, and Processes. Prentice Hall; 2000.

5 Sam Newman. Building Microservices. O'Reilly Media; 2015.

6 Eric Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley; 2003.

7 Melvin Conway. Conway's Laws [online]. URL: http://www.melconway.com/Home/Conways_Law.html Accessed 29 October 2017.

8 Susan J. Fowler. Production-ready microservices. O'Reilly Media; 2016.

9 James Lewis; Martin Fowler. Microservices [online]. 25 March 2014. URL: https://www.martinfowler.com/articles/microservices.html Accessed 29 October 2017.

10 Reed Hastings, CEO Netflix. Culture [online]. 1 August 2009. URL: https://www.slideshare.net/reed2001/culture-1798664 Accessed 29 October 2017.

11 Sam Newman. The principles of microservices. August 2015. URL: https://www.safaribooksonline.com/library/view/the-principles-of/9781491935811 Accessed 29 October 2017.

12 Charles P. Shelton. Human Interface/Human Error [online]. Spring 1999 URL: https://users.ece.cmu.edu/~koopman/des_s99/human/ Accessed 29 October 2017.

13 Jez Humble; David Farley. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley; 2010.

14 Kief Morris. Infrastructure as Code. O'Reilly Media; 2016.

15 Kief Morris. Configuration Drift [online]. December 2011. URL: http://kief.com/configuration-drift.html Accessed 04 November 2017.

16 Ben Butler-Cole. Rethinking building on the cloud: part 4: immutable servers [online]. June 2013. URL: https://www.thoughtworks.com/insights/blog/rethinking-building-cloud-part-4-immutable-servers Accessed 04 November 2017

17 Martin Fowler. Blue-green deployment [online]. URL: https://www.martinfowler.com/bliki/BlueGreenDeployment.html Accessed 04 November 2017.

18 Danilo Sato. Canary release [online]. URL: https://martinfowler.com/bliki/CanaryRelease.html Accessed 04 November 2017.

19 Hashicorp. Packer.io [online]. URL: https://www.packer.io/intro/index.html Accessed 05 November 2017.

20 Jeff Nikoloff. Docker in Action. Manning Publications Co; 2016.

21 Docker Registry [online]. URL: https://docs.docker.com/registry/ Accessed 09 November 2017.

22 Kubernetes [online]. URL: https://kubernetes.io Accessed 11 November 2017.

23 Minikube [online]. URL: https://github.com/kubernetes/minikube Accessed 11 November 2017.

24 Kubernetes Operations [online]. URL: https://github.com/kubernetes/kops Accessed 11 November 2017.

25 Kubernetes on AWS [online]. URL: https://github.com/kubernetes-incubator/kube-aws Accessed 11 November 2017.

26 Blake Smith. Understanding horizontal and vertical scaling [online]. URL: http://blakesmith.me/2012/12/08/understanding-horizontal-and-vertical-scaling.html Accessed 11 November 2017.

27 Amazon EC2 Auto scaling groups [online]. URL: http://docs.aws.amazon.com/autoscaling/latest/userguide/AutoScalingGroup.html Accessed 11 November 2017.

28 Kubernetes horizontal pod autoscale [online]. URL: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/ Accessed 11 November 2017.

29 Kubernetes cluster autoscaler [online]. URL: https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler Accessed 11 November 2017.