Achieving Continuous Delivery of Immutable Containerized Microservices with Mesos/Marathon

Shashi Ravula

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

San Francisco 19.5.2017

Thesis supervisor:

Assoc. Prof. Keijo Heljanko

Thesis advisor:

M.Sc. Tapio Tolvanen

Aalto University
School of Science

AALTO UNIVERSITY
SCHOOL OF SCIENCE

ABSTRACT OF THE
MASTER'S THESIS

Author: Shashi Ravula

Title: Achieving Continuous Delivery of Immutable Containerized Microservices with Mesos/Marathon

Date: 19.5.2017 Language: English Number of pages: 7+75

Department of Computer Science

Professorship: Computer Science

Supervisor: Assoc. Prof. Keijo Heljanko

Advisor: M.Sc. Tapio Tolvanen

In the recent years, DevOps methodologies have been introduced to extend the traditional agile principles which have brought up on us a paradigm shift in migrating applications towards a cloud-native architecture. Today, microservices, containers, and Continuous Integration/Continuous Delivery have become critical to any organization's transformation journey towards developing lean artifacts and dealing with the growing demand of pushing new features, iterating rapidly to keep the customers happy. Traditionally, applications have been packaged and delivered in virtual machines. But, with the adoption of microservices architectures, containerized applications are becoming the standard way to deploy services to production. Thanks to container orchestration tools like Marathon, containers can now be deployed and monitored at scale with ease. Microservices and Containers along with Container Orchestration tools disrupt and redefine DevOps, especially the delivery pipeline.

This Master's thesis project focuses on deploying highly scalable microservices packed as immutable containers onto a Mesos cluster using a container orchestrating framework called Marathon. This is achieved by implementing a CI/CD pipeline and bringing in to play some of the greatest and latest practices and tools like Docker, Terraform, Jenkins, Consul, Vault, Prometheus, etc. The thesis is aimed to showcase why we need to design systems around microservices architecture, packaging cloud-native applications into containers, service discovery and many other latest trends within the DevOps realm that contribute to the continuous delivery pipeline. At BetterDoctor Inc., it is observed that this project improved the avg. release cycle, increased team members' productivity and collaboration, reduced infrastructure costs and deployment failure rates. With the CD pipeline in place along with container orchestration tools it has been observed that the organisation could achieve Hyperscale computing as and when business demands.

Keywords: DevOps, Continuous Delivery, Docker, Microservices, Containers, Consul, Jenkins, Marathon, Mesos, Vault

Acknowledgements

This thesis would not have been possible without help, the support, and the encouragement of family, friends, and mentors.

My sincere gratitude goes to my supervisor Assoc. Prof. Keijo Heljanko, for being so patient with me and taking time to guide me remotely via video sessions week after week. I thank him for all the discussions, ideas and encouragement.

I want to thank all my colleagues at BetterDoctor. Especially, I want to express my gratitude to M.Sc. Tapio Tolvanen, for being my instructor during this period and for his good guidance. A big thanks goes to Dylan Clendenin, for his daily help, support, patience during the implementation process of this project.

As my Master's degree allowed me to travel in many countries, my mentors and friends are spread across Europe and USA. I would also like to extend my gratitude to Prof. Guillaume Pierre from my first year of masters at University of Rennes 1, for his great mentorship.

Thanks to all my friends, for the priceless time we spent together in Rennes, Espoo and San Francisco. Last, but not the least my deepest gratitude goes to my family, a constant source of energy and love.

San Francisco, 19.5.2017

Shashi Ravula

Contents

Abstract ii

Acknowledgements iii

Contents iv

Abbreviations vii

1 Introduction 1
- 1.1 Cloud Computing . . . 2
- 1.2 Rise of Cloud-Native Applications . . . 4
- 1.3 Key Terminologies . . . 5
  - 1.3.1 Continuous Integration . . . 5
  - 1.3.2 Continuous Delivery . . . 6
  - 1.3.3 Continuous Deployment . . . 6
  - 1.3.4 DevOps . . . 7
- 1.4 Motivation . . . 8
- 1.5 Research Questions . . . 8
- 1.6 Thesis Structure . . . 9

2 Background 10
- 2.1 System Architecture . . . 10
  - 2.1.1 Monolithic Applications . . . 11
  - 2.1.2 Service-Oriented Architecture . . . 12
  - 2.1.3 Microservices . . . 13
  - 2.1.4 Twelve-factor app . . . 15
  - 2.1.5 Deployment models . . . 16
  - 2.1.6 Immutable Microservices . . . 17
- 2.2 Containers . . . 18
  - 2.2.1 Namespaces . . . 20
  - 2.2.2 Cgroups . . . 21
  - 2.2.3 Docker . . . 21
- 2.3 Container Orchestration . . . 23
  - 2.3.1 Scheduler . . . 23
  - 2.3.2 Monolithic Schedulers . . . 24
  - 2.3.3 Two Level Schedulers . . . 25
  - 2.3.4 Shared State Schedulers . . . 25
  - 2.3.5 Docker Swarm . . . 27
  - 2.3.6 Kubernetes . . . 28
  - 2.3.7 Apache Mesos and Marathon . . . 30
  - 2.3.8 Marathon LB . . . 33
  - 2.3.9 Marathon Pods . . . 34

3 Service Discovery & Configuration Management 36
3.1 Infrastructure as code . . . 36
3.1.1 Configuration Management in an Immutable World . . . 36
3.2 Terraform . . . 38
3.3 Why do we need Service Discovery? . . . 39
3.3.1 Consul . . . 40
3.3.2 Architecture . . . 40
3.3.3 Secret Management . . . 41
3.3.4 Vault . . . 42

4 Requirements 43
4.1 Underlying Requirements . . . 43
4.2 Developer Requirements . . . 44
4.3 Optional Requirements . . . 45
4.4 Classification of Requirements . . . 45

5 Design and Implementation 46
5.1 Analyzing Existing Setup . . . 46
5.1.1 Source Code Management & Git Branching Model . . . 46
5.1.2 Automated Testing and Building . . . 46
5.1.3 Deployment . . . 47
5.2 Design choices . . . 48
5.2.1 Jenkins . . . 49
5.2.2 Containerization . . . 49
5.3 Building a CI/CD Pipeline with Docker . . . 51
5.4 Drafting the Continuous Delivery Pipeline . . . 52
5.4.1 Deploying the Containers . . . 54
5.4.2 Auto Scaling . . . 54
5.4.3 Monitoring and Centralized Logging . . . 54
5.4.4 ELK (Elasticsearch,Logstash,Kibana) Stack . . . 56
5.4.5 Continuous Delivery Workflow . . . 56
5.4.6 Blue Green Deployments . . . 57

6 Learnings and Evaluation 59
6.1 Improved Cycle time . . . 59
6.2 Increased Productivity . . . 59
6.3 Breaking Down the Silos . . . 60
6.4 Impact of Docker on CD . . . 60
6.5 Moving to Immutability Infrastructure . . . 60
6.6 Achieving Dynamic Configuration Management . . . 60
6.7 Reduced risk of deployment failure . . . 61
6.8 Reducing Infrastructure Costs . . . 61
6.9 Achieving Hyper-scale . . . 61
6.10 Testing and Fine tuning Docker Environment . . . 61
6.11 Developer Concerns . . . 62

6.11.1 Allow to Debug inside the containers 62
6.11.2 Learning curve for Docker and Mesos 62

7 Discussion **64**
7.1 Summary 64
7.2 Conclusion 65

References **66**

References **68**

A Configuration Files **69**
A.0.1 Continuous integration 69
A.0.2 Marathon Configuration file 70
A.0.3 Terraform IAC template 72

B Miscellaneous **75**

Abbreviations

CI	Continuous Integration
CD	Continuous Delivery
CMs	Configuration Managers
SDLC	Software Development Life Cycle
ALM	Application Life Cycle Management
VMs	Virtual Machines
LB	Load Balancer
IaaS	Infrastructure as service
PaaS	Platform as service
SaaS	Software a a service
IAC	Infrastructure as code
SOA	Service Oriented Architecture
ESB	Enterprise Service Bus
DNS	Domain Name System
API	Application Program Interface
CLI	Command Line Interface

1 Introduction

In the last couple of decades we have seen a lot of changes in the way organizations have been delivering software, especially the speed and quality of delivering the software has always been going up. Traditionally, Software Development Life Cycle (SDLC) was composed of many stages and for decades this was managed through the Waterfall Model [20]. Waterfall Model was the only well known and widely accepted method back in the day, it had many distinct phases like planning was the first in line, then comes identifying requirements, followed by design, then development of the whole product, user acceptance testing, launch into production, and finally the corresponding support and maintenance activities. In the mid 1990's it was observed that success rate of software development was woeful. For example, A 1999 review of failure rates in a sample of earlier Department of Defense projects drew grave conclusions: Of a total \$37 billion for the sample set, 75% of the projects failed or were never used, and only 2% were used without extensive modification [12]. Similar experiences were common for many organizations undertaking the development of large software projects. There were many drawbacks that came to light with the Waterfall Model like for example most projects rarely follow the sequential flow of process it defines and as Humphrey's requirements uncertainty principle states — requirements will not be completely known until after the users have used it [11]. It became evident that a change in software development methodology was needed to help mitigate these problems. In 1996, Barry Boehm published a well-known paper summarizing failures of the waterfall model [4], he also introduced a risk-reducing, iterative and incremental approach called the spiral model [3].

With the time it has been seen that organizations moved from Waterfall Model to Agile methodologies to improve the overall Software Delivery Life Cycle. This is because the way organizations deliver software is going through a wave of change as there is more pressure to adapt to market needs and deliver rapidly [26]. Gone are the days when an organization kept the customer waiting for a release every 6 months, if you want to be on the top you have to make rapid iterations and able to respond to continuous feedback. In order cope up with the impatient market a Manifesto for Agile Software Development was published in 2001[1] which is embraced by many IT organizations in today's world. Over the years organizations have adopted many agile methods which refine and optimize their software development practices but this focus has been mainly in software development and operations side of the software delivery was lagging behind in the race. Eventually, operations team could not keep up with the rate at at which artifacts and builds are being delivered [26]. This affected the entire delivery process since.

DevOps [9] is set of practices that is trying to bridge developer-operations gap at the core of things[2] but at the same time is not limited to this development and operations hand-off instead covers all the aspects which help in speedy, optimized and high quality software delivery. The two key components of a DevOps methodology

[1] https://www.agilealliance.org/agile101/the-agile-manifesto

[2] "Understanding DevOps-Infrastructure as code - Sanjeev Sharma" https://sdarchitect.wordpress.com/2012/12/13/infrastructureas-code

are Continuous Integration (CI) and Continuous Delivery (CD). CI is a development practice that requires developers to integrate code into a shared repository several times a day [10]. This helps in detecting errors quickly and test build every time someone pushes code to the repository , there by increasing communication among developers when integrating their code. CD ensures that software always remains fully deployable without needing to release changes before they are ready, tested, and approved. CI and CD help us achieve true agility in SDLC. We will discuss more in detail about CI/CD later in this chapter.

With the advent of Cloud Computing, applications are now decoupled from the physical infrastructure[3] and can be deployed on to public or private clouds. Trthe application can utilize the elastic nature of the cloud and scale up or down on demand and these applications are termed as cloud-native. In addition Docker containers and microservices architecture quickly are becoming standard on cloud-native applications, this deadly combination simplifies deployment in computing environments of all types. This transformation of moving from hundreds of VMs to thousands or tens of thousands of containers and deploying them at hyperscale requires advanced technologies for container management. In this thesis, we showcase a state of the art continuous delivery pipeline which deploys containers on to a mesos cluster using a container orchestrating framework called marathon. Within the DevOps context containers and microservices disrupt the way any enterprise or startup builds and delivers applications.

1.1 Cloud Computing

Broadly speaking, cloud computing refers to a distributed system of resources and services that communicate across boundaries of geographical and ethernet space. But that said, the cloud is not just a collection of resources because it possesses the capability to manage them as well. A cloud computing platform is responsible for provisioning, de-provisioning and monitoring servers and balancing their workload. Servers in the cloud could be both physical and virtual machines. Other than those, cloud platforms also provide a slew of applications, storage services, network filters etc. These offerings are billed on a consumption basis. Everything is offered as a utility, thus encompassing the traditional pay-as-you-go subscription model for services. A major characteristic of the cloud is that its services are transparent, meaning, the user need not be concerned about how the service is made available to him. All he needs is to know how to use it. And this is usually simplified by interactive User Interfaces. Most services on the cloud require little or no administrative knowledge on the part of the customer. An access is simply enabled on the request of a customer. The latter has a view of only the abstraction of the service he needs- the details of implementation and accessibility are hidden from him. The cloud is thus a stack of different services which can be classified as Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS) [17]. Figure 1 shows us the difference between each of these services and Traditional Enterprise IT.

[3]https://goo.gl/Z5gNJq

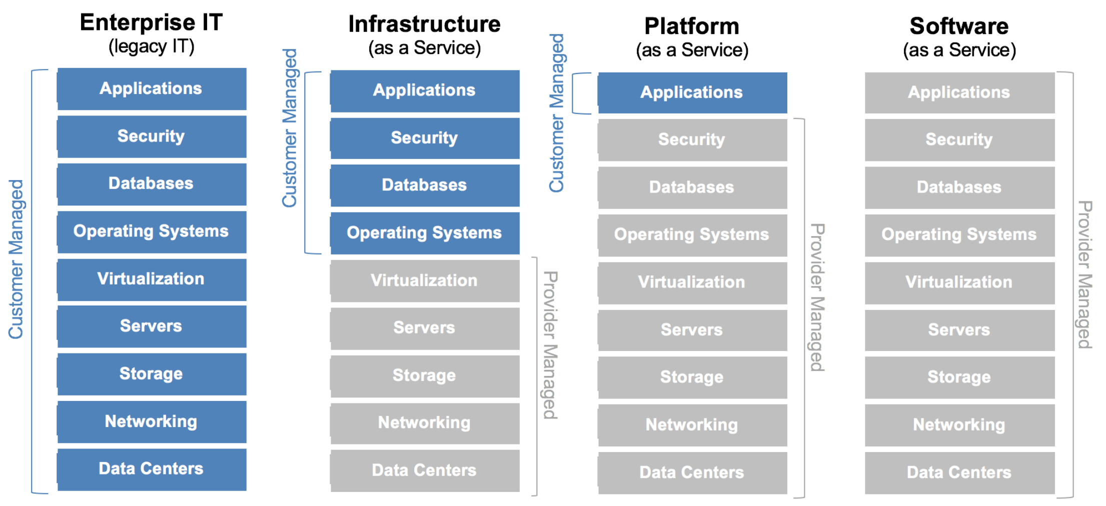


Figure 1: Cloud Computing Service Models[4]

IaaS - Infrastructure as a Service

Cloud providers of IaaS offer infrastructure starting from the simplest and most basic of resources like physical and virtual machines to complicated storage, network and operating systems as utilities. A hypervisor node is responsible for the running of these virtual machines. Clusters of such hypervisors administer the functioning of hundreds of virtual machines, thus serving the purpose of scaling in cloud services. Users need to buy access rights to infrastructure components they need, without having to buy the actual components. Not only does this cut costs, it also rids them of the pains of infrastructure management. Such a cloud service is ideal for organizations that have low capital or require temporary infrastructure. IaaS can further be divided into private and public clouds–the former refers to resources on a private network while the latter refers to resources on a public network available to external users. Amazon Web Services is the most renowned instance of IaaS.

PaaS - Platform as a Service

This offers an integrated environment that would allow application developers or the like to create, deploy and test applications without having to worry about the cost and complexity of underlying hardware and software infrastructure. A computing platform would typically include atleast an operating system, programming language execution environment, database, and web server, mostly with integrated test and deployment environments. Force.com from Saleforce.com is a renowned PaaS system that empowers developers to develop multi-tenant applications hosted on Salesforce.com servers.

[4]https://goo.gl/UW4GHR

SaaS - Software as a Service

This brand of cloud computing offers a single application built on a multi-tenant architecture, over the internet to thousands of customers. The software is provided on-demand, either free of cost or on a pay-per-use basis. All updates and security patches are taken care of by the provider. For the customer, this implies no efforts to support or maintain software, thus reducing significant costs, while for the provider, this means reduced management effort as it is only one app at a central location, needing administration. Google Apps is one of the best known SaaS offerings.

1.2 Rise of Cloud-Native Applications

Migrating to the cloud is a natural evolution of the the IT service delivery. Cloud computing refers to generally an environment in which computing, networking, and storage resources can be provisioned and released elastically in an on-demand, self-service manner. A cloud-native application should have an isolated state, is distributed in its nature, is elastic in a horizontal scaling way, operated via an automated management system and its components should be loosely coupled [7]. The capacity to operate cloud-native applications can generate enormous business growth and value to an organization [16]. The most common motivations to move towards a cloud-native application architecture are

Speed

Speed wins in the marketplace. The sooner a new feature of a software product hits the market the more competitive advantage the organization gains. High-performing organizations like Netflix, Amazon, Airbnb, etc are renowned for their practice of deploying hundreds of times per day. This practice of frequent deployment cycles helps these organizations to recover form their mistakes instantly which means they learn and innovate faster from the feedback loop. According to 2016 State of DevOps Report[5] High-performing organizations deploy 200 times more frequently, with 2,555 times faster lead times[6], recover 24 times faster, and have three times lower change failure rates. The elastic and self-service nature of cloud perfectly align with the frequent deployment model as we can setup the infrastructure and application environment via API calls to the cloud service.

Fault-Isolation

Cloud-Native architectures often employ microservices which helps to isolate the failure to just one component(service) in comparison to monolithic architectures where the entire system is brought down if there is one component misbehaving. In order to limit the risk associated with failure, we need to limit the scope of components or features that could be affected by a failure [1]. For example, a memory

[5] `https://puppet.com/resources/whitepaper/2016-state-of-devops-report`

[6] time elapsed between the identification of a requirement and its fulfillment

leak in one service only affects that service. Other services will continue to handle requests normally. More about microservice architecture is discussed in Section 2.1.3.

Scaling

With high demand for your service on the market it is very likely that you need to scale your service to meet the increasing demand and before migrating to the cloud, companies often bought hardware (physical servers) by scaling vertically. Although there was capacity planning and forecasting of the peak usage of the service, there were times when the estimations were wrong. Also, installation and maintenance of these servers on premise was challenging and most of the startup capital was used just to maintain the servers. With more public cloud infrastructures like Amazon Web Services became available, virtualization costs and maintenance costs were delegated to the cloud provider. Recently, when containers became the unit of application deployment instead of VMs, it not only improved the speed of deploying an applications but also reduced the time to respond to changes in demand. Additionally, cloud-native application architectures keep the application instance stateless and store the state in persistent external object stores or in-memory cache stores. Stateless applications can be quickly created and destroyed, as well as attached to and detached from external data stores. Cloud providers also enable us to scale up or down these external data stores on-demand.

According to Stine [24] cloud-native application architectures like to deliver software-based solution quickly, in a more fault isolating way and enable us to scale the applications horizontally rather than vertically. The twelve-factor app developed by engineers at Heroku is a collection of patterns for cloud-native application architectures. For many people, cloud native and 12 factor are synonymous. Section 2.1.4 describes the twelve patterns.

1.3 Key Terminologies

1.3.1 Continuous Integration

Continuous Integration is a 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[7]. It basically refers to integrate early and not to keep the changes localized to your workspace for long, instead share your changes with team and validate how code behaves continuously. Not only share within component teams but integrate beyond component boundaries, at product integration level [26]. This can be further optimized by automating, as soon as a developer delivers a change the build system detects and triggers a build, carries out sanity and integration tests and notifies the result to the team. This has to be a repeatable continuous process all across the development cycle. The goal is to develop a robust build pipeline which helps us to identify failures as fast as possible (fail fast) so that no change which breaks the pipeline goes unnoticed beyond that faulty check in, and the build is

[7]https://www.martinfowler.com/articles/continuousIntegration.html

immediately flagged as failed and responsible parties are alerted. A broken build should always be fixed immediately. A good CI process will help the team to move fast, with cleaner code and with a high level of confidence in the work they are doing.

1.3.2 Continuous Delivery

Continuous Delivery is a software development discipline where you build a deployment pipeline in such a way that the software can be released to production at any time[8]. CI augments Continuous Delivery and it is just an extension of the CI pipeline. CI assumes that there are other manual steps to be performed once the pipeline is executed whereas a successful execution of CD pipeline results in artifacts which can be deployed to production. The artifact could be a Docker image or a Debian package, etc.

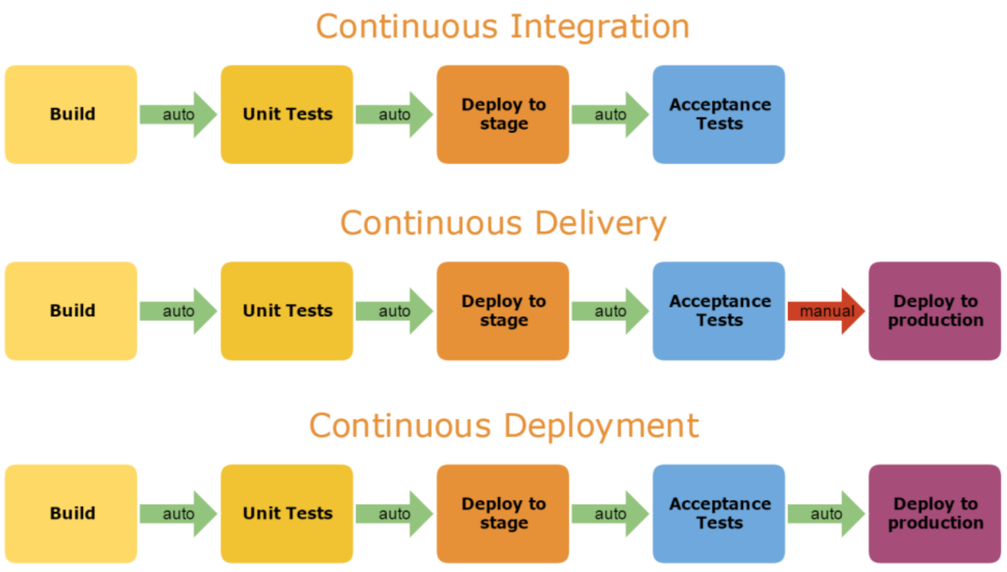


Figure 2: Continuous Integration vs Continuous Delivery vs Continuous Deployment[10]

1.3.3 Continuous Deployment

Continuous Deployment pipeline just goes a step forward and automatically deploys to production every build that passes the testing. Continuous Deployment needs more confidence when compared to Continuous delivery and Continuous Integration since there is no human intervention. The pipeline starts when a developer pushes commit to the code repository and ends with the application being deployed in production environment. The continuous deployment pipeline can include multiple stages where one of it could be deploying the application in stage environment and

[8]https://martinfowler.com/bliki/ContinuousDelivery.html
[10]https://goo.gl/HYFacQ

running end-to-end testing before deploying in production. Figure 2 shows the basic differences between each of the pipelines.

While continuous deployment may not be right for every company, continuous delivery is an absolute requirement of DevOps practices [11]. Only when you continuously deliver your code can you have true confidence that your changes will be serving value to your customers within minutes of pushing the *go* button.

1.3.4 DevOps

DevOps is a set of principles for streamlining and integrating the software development process. It is a management philosophy which was originally trying to solve the communication-gap between the development and operation teams which used to work in complete isolation. But later on the idea that came out was revised to improve collaboration between all the departments and this led to the evolution of few basic principles called the *Three Ways* [14].

Systems Thinking

The First Way emphasizes the performance of the entire system, as opposed to the performance of a specific silo of work or department.

Amplify Feedback Loops

The Second Way is to create feedback loops to make necessary corrections continuously by responding to both external and internal customers. A defect is not a defect unless it hits the customer, so creating these short and amplified feedback loops helps the organization to fix bugs iteratively.

Culture of Continual Experimentation and Learning

The Third Way is to create an environment that fosters experimentation and taking calculated risks and learn from these failures continuously.

The core values(CAMS)[12] of the DevOps Movement define DevOps as means to adopt a Culture of blame-free communication and collaboration, to break down barriers between teams, to embrace Automation to allow people to focus on important tasks, to introduce continuous Measurements to get feedback on the quality and to encourage Sharing of these measurements. This underpins the fact that DevOps is not about standards or tools, it is about enabling communication and collaboration between departments in an organization. Today DevOps principles have evolved completely depending on the company's interests and priorities. Most of the fundamental principles in agile methodologies, and lean management are now addressed by DevOps. These principles can broadly be listed as follows :

[11] https://puppet.com/blog/continuous-delivery-vs-continuous-deployment-what-s-diff
[12] http://devopsdictionary.com/wiki/CAMS

- Deliver more value to your customers - fast.
- Find better and faster solutions continuously.
- Manage risk and uncertainty better.
- Reduce waste continuously.

In order to achieve the above, a company must easily be able to deploy and scale cloud infrastructure, improve the speed of software delivery, continuously test in a production like environment, ability to react to change more quickly, etc. There are many tools, frameworks, design patterns that are used to achieve this DevOps Philosophy. Efficient DevOps combined with efficient use of resources leads to high IT performance, which is a must for today's highly competitive IaaS business. Being highly agile and lightweight, containers combined with the microservice architecture have shown great promises in achieving the desired level of efficiency [13].

1.4 Motivation

In early 2016, BetterDoctor[13] pivoted from an online doctor search platform to a data validation company. The online doctor search platform was monolith and was a big pain to deploy with frequent releases. The new Data Validation service is comprised of several applications talking to each other in the data-pipeline. The Data team was very insightful to design these applications around Microservices architecture. Getting the new product out and receiving immediate feedback on the service was quintessential and therefore, the need for CI/CD has crawled in. I took opportunity to build the CI server, we push code every day to our repositories and CI is playing its part in iterating rapidly. Later on we started dockerizing and launching these Microservices on to the Mesos cluster and the transition is currently in progress. It was a great motivation to look at the problem and work towards designing a continuous delivery pipeline in order to achieve true agility. The journey to achieve CI/CD following modern state of the art DevOps practices and tools is showcased in this thesis project.

1.5 Research Questions

Below is a list of questions that needs to be answered in order to be able to understand the containers impact on continuous delivery, and continuous delivery's overall impact on the project:

- What problems or blockers does BetterDoctor have today that prevents them from achieving continuous delivery?
- How can we solve these issues with the use of containers, Microservices and container orchestration frameworks?

[13]BetterDoctor solves America's provider directory problem https://betterdoctor.com/

- Will containers have the desired impact on the implementation process?
- How will continuous delivery impact the Organization?

The goal throughout this Master's thesis is to establish a discussion and conclusion based on the questions presented above. These questions will hopefully give the reader a better overview of this research, and acts as guidelines and goals during this study.

1.6 Thesis Structure

This thesis is organized into seven sections.

- **Section 1** introduces the project by describing motivation and research terminologies that are common in a continuous delivery pipeline.
- **Section 2** gives the context/background for the project by introducing Microservices Architecture, Containers and Container Orchestration Frameworks. We also compare and contrast monolithic architecture with microservices. We understand how microservices and docker fit in the CD pipeline. We also discuss different deployment models for these microservices.
- **Section 3** gives an overview of Configuration Management and Service Discovery. We also discuss Immutability and how does it impact Configuration Management Tools. We will go through the architecture overview of Consul and see how does it help microservices to discover each other. We briefly discus secret management using Vault.
- **Section 4** gathers all the requirements for the continuous delivery pipeline.
- **Section 5** discusses the design and implementation of the project and the design choices we made.
- **Section 6** evaluates the project based on some learnings.
- **Section 7** presents a conclusion along with summary.

2 Background

This section gives a clear background on the microservices architecture and compares it with its predecessors. More over we analyze the twelve-factor application patterns and discusses in detail about technologies like containers and container orchestration tools which streamline the deployment of these microservices. We also discuss about immutable microservices and how do they change the way applications are deployed.

2.1 System Architecture

In the last few years web applications architectures have been evolving with a decent pace. With the rise of Internet services we can notice can see a paradigm shift from tiered monolith architectures to Service Oriented Architecture, and in turn from SOA to microservices. Choosing a particular software architecture is totally dependent on the requirements like scalability, complexity, ease of deployment, etc. Scalability is one of the most important characteristic of a web application. As you can see in the Figure 3 the comparison between the three major architecture models considering scalability and decoupling as parameters. Before we discus about the three major architecture patterns followed in the software industry, let us understand different types of scaling. Martin and Michael in their book Art of Scalability [1] describe a three dimensional scalability model called the scale cube 4.

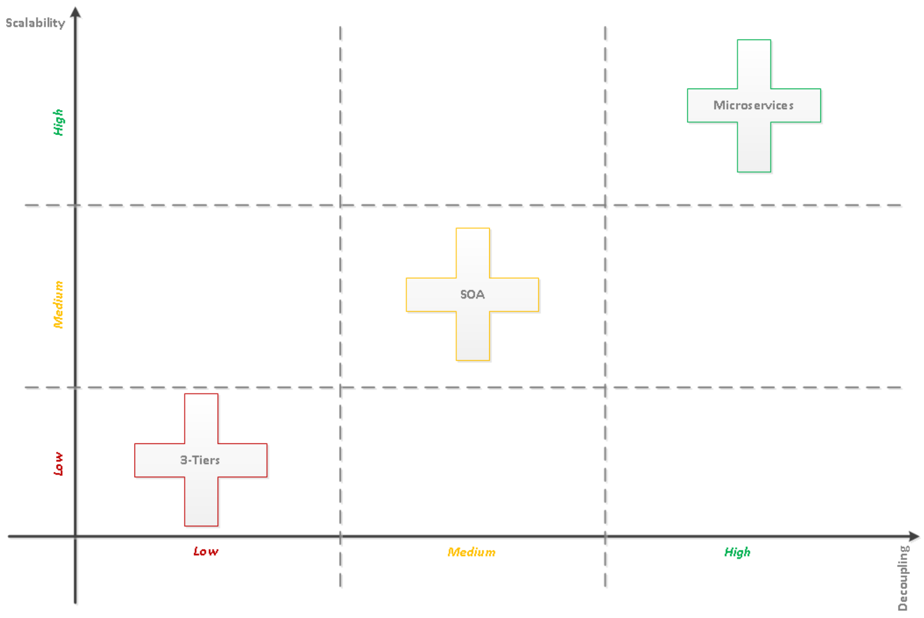


Figure 3: Monoliths vs SOA vs Microservices[14]

In this model, the commonly used approach of scaling an application by running multiple identical copies of the application behind a Load Balancer is represented as X-axis scaling. The Z-axis Scaling is similar to X-axis scaling but each server which runs

[14]https://goo.gl/HLsFud

the identical copy of the code is responsible for only a subset of data. Some component of the system is responsible for routing each request to the appropriate server. One commonly used routing criteria is an attribute of the request such as the primary key of the entity being accessed, customer type (high SLA for paying customers). Both X-axis and Z-axis scaling improve application's capacity and availability but increase application complexity and development costs. They represent Monolithic and SOA architectural scalability models. We can achieve the Y-axis scaling for applications which follow microservices architecture. Y-axis scaling is also known as functional decomposition. In contrast to Z-axis scaling which splits things that are similar, Y-axis scaling splits things that are different. At the application tier, Y-axis scaling splits a monolithic application into a set of services. Each service implements a set of related functionality such as order management, customer management etc.

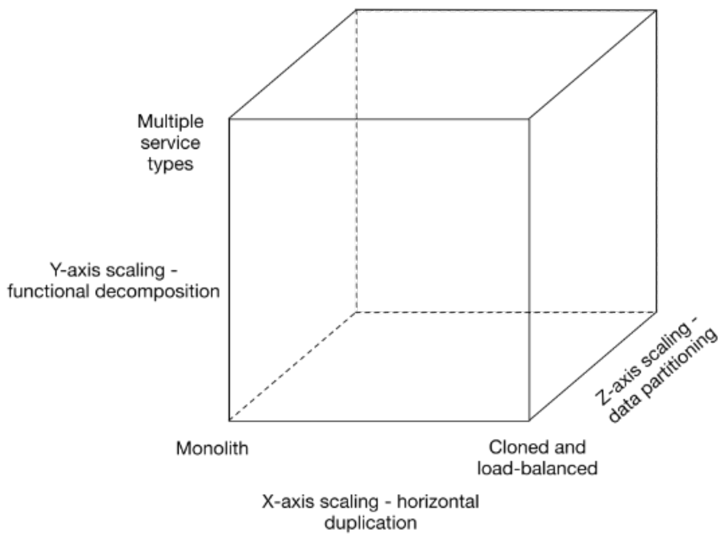

Multiple service types
Y-axis scaling - functional decomposition
Z-axis scaling - data partitioning
Monolith
Cloned and load-balanced
X-axis scaling - horizontal duplication

Figure 4: Scale Cube [1]

2.1.1 Monolithic Applications

Monolithic Applications tends to bundle together all the functionalities needed and are developed and deployed as a single unit. It is the simplest form of architecture and runs pretty well as long as we keep the complexity low. The problems tend to arise when the architecture needs to scale up feature-wise. As time passes, we continuously increase the size of the application and complexity which results in decrease in development, testing and deployment speed. Even if we just want to develop a simple feature which would only under different circumstances require a few lines of code but, due to the complex architecture we created , those few lines turnout to be thousands thus decreasing the development speed. Testing and deploying Monoliths with increased number of features requires hours to test, build and deploy.

Scaling a monolith often results in unbalanced resource utilization. We cannot scale individual portions of the application as all the application code runs in the same process on the server. For instance, let us look at the monolith in Figure 5, suppose we just want to scale up the order service, the other two services are also scaled along and duplicated on a new server. If one service is memory intensive and another CPU intensive, the server must be provisioned with enough memory and CPU to handle the baseline load for each service. This can get expensive if each server needs high amount of CPU and RAM, and is exacerbated if load balancing is used to scale the application horizontally[15]. Also, we have to redeploy the entire application every time its updated. There is always a single point of failure and fault isolation is not achieved.

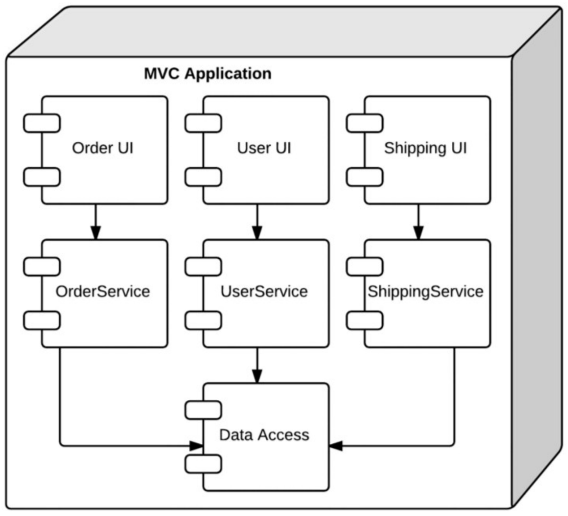


Figure 5: Monolithic Architecture [16]

To summarize Monoliths work well in the early stages of the project but as they grow they tend to become more complex and difficult to scale. Continuous delivery and Continuous deployment is difficult to achieve for these applications. They also have a barrier to adopt new technologies. Since changes in frameworks or languages will affect an entire application business layer and thus becomes expensive in both time and cost.

2.1.2 Service-Oriented Architecture

SOA emerged as a way to solve tightly coupled architecture created by the monolithic applications. SOA architecture can seen as a component based distributed architecture. Componentisation (into services), loose coupling, high performance/throughout, service lifecycle management (ie managing the services - taking central custody, promoting reuse, managing their lifecycle events through creation to eventual deprecation and retirement) were essential features of SOA [28].

SOA consists of two roles, a service provider and a service consumer. The Consumer Layer is the point where consumers (human users, other services or third

[15] https://goo.gl/H5tzTR
[16] https://goo.gl/85Sygt

parties) interact with the SOA and Provider Layer consists of all the services which are loosely coupled. The both layers communicate through ESB or the Enterprise Service Bus as shown in the Figure 6 which publish business capabilities as a service to the consumer and routes the messages to back-end with a capability of transforming, securing and handling delivery exceptions. These ESB's are themselves very complex and often contribute to create another big monolith application out ofthe SOA architecture. This led to the birth of microservices.

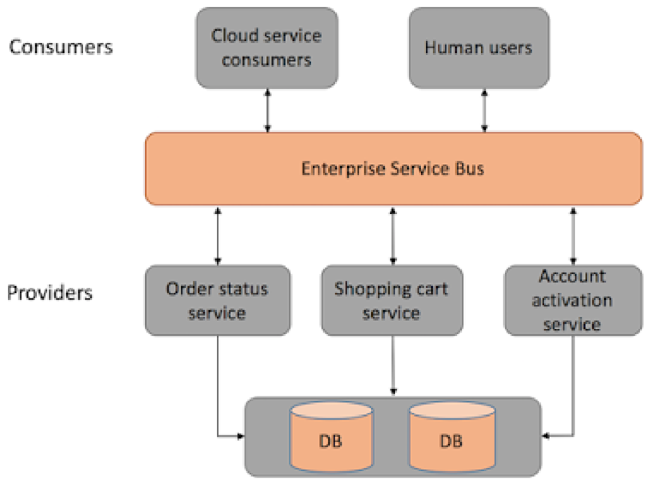


Figure 6: Service Oriented Architecture [17]

2.1.3 Microservices

The term "Microservice Architecture" has sprung up over the last few years to describe a particular way of designing software applications as suites of independently deployable services. While there is no precise definition of this architectural style, there are certain common characteristics around organization around business capability, automated deployment, intelligence in the endpoints, and decentralized control of languages and data. The idea is to split the application into a set of smaller, interconnected and independent services instead of building a single monolithic application. Each microservice is a small application that has its own hexagonal architecture consisting of business logic along with various adapters and is developed,tested and deployed separately from each other. These microservices would expose a REST or message-based API to communicate with different microservices.

Microservices Architecture is often confused with traditional SOA-type services. Although there's a great deal of overlap, there are ample differences between them. According to Martin Fowler microservices are one form of SOA, perhaps service orientation done right[19]. We can categorize Microservices as light weight version of SOA. SOA was more oriented towards IT side of organization, while microservice is

[17]https://dzone.com/articles/microservices-vs-soa-2
[19]https://martinfowler.com/articles/microservices.html

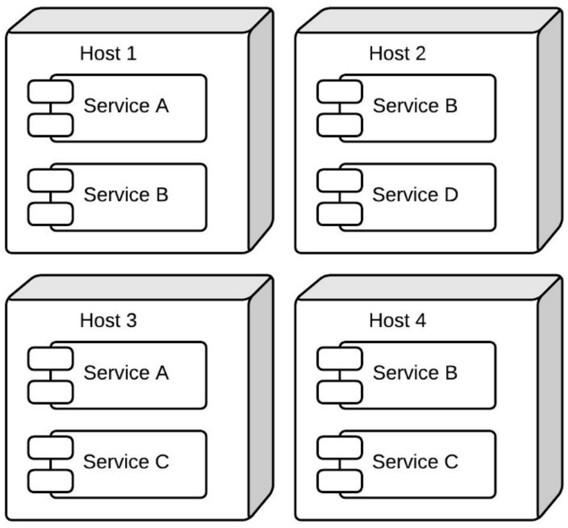


Figure 7: Microservices Architecture[18]

more inlcined towards SAAS based products. SOA mostly use XML/WSDL whereas Microservice architecture adopt REST API. The major difference between SOA and microservices is that the later is more self sufficient and deployed independently. There are many advantages to choose microservices over monolithic applications.

Scaling

Microservices scale much easier than the monolithic applications. We can scale the service which needs to be scaled unlike monoliths where we need to scale the whole application in to a new machine. Also, microservices enable us to group high resource demanding services to better hardware helping us to distribute services better in our infrastructure.

Agility and Innovation

Microservices are independent and autonomous which implies that the degree of freedom for developer to develop and maintain their own codebase and ability to quickly react to changing scenarios compared to monoliths which due to their nature are slow to changing things and experimenting is perilous and expensive.

Minimal and Lightweight

Due to their small size and normally a single service does not provide a considerable complex functionality they are easier to develop and even IDE work faster with because importing a small project with less dependencies and libraries means less load on IDE.

[19]https://goo.gl/85Sygt

Deployment, Rollback and Fault Isolation

We can achieve faster deployment with microservices because of the size. Its easier to roll back just that one service which is having issues in production to older version. Until we roll back, the fault is just isolated to that one service. Continuous delivery and deployment can be easily achieved with microservices.

To summarize, migrating monolithic architectures to microservices brings in many benefits. In particular, it provides adaptability to technological changes to avoid technology lock-in and, more important, reduced time-to-market and better development team structuring around services.

2.1.4 Twelve-factor app

A common methodology known as twelve-factor app[20] has emerged with the purpose of providing an outline for building well structured and scalable microservices. A twelve-factor app should follow the following principles.

- **Codebase -** Each deployable app is tracked as one codebase tracked in revision control. It may have many deployed instances across multiple environments.
- **Dependencies -** An app explicitly declares and isolates dependencies via appropriate tooling (e.g., Maven, Bundler, NPM) rather than depending on implicitly realized dependencies in its deployment environment.
- **Config -** Configuration, or anything that is likely to differ between deployment environments (e.g., development, staging, production) is injected via operating system-level environment variables.
- **Backing services -** Backing services, such as databases or message brokers, are treated as attached resources and consumed identically across all environments.
- **Build, release, run -** The stages of building a deployable app artifact, combining that artifact with configuration, and starting one or more processes from that artifact/configuration combination, are strictly separated.
- **Processes** The app executes as one or more stateless processes (e.g., master/-workers) that share nothing. Any necessary state is externalized to backing services (cache, object store, etc.). Port binding The app is self-contained and exports any/all services via port binding (including HTTP).
- **Concurrency -** Concurrency is usually accomplished by scaling out app processes horizontally (though processes may also multiplex work via internally managed threads if desired).

[20] https://12factor.net/

- **Disposability -** Robustness is maximized via processes that start up quickly and shut down gracefully. These aspects allow for rapid elastic scaling, deployment of changes, and recovery from crashes.
- **Dev/prod parity -** Continuous delivery and deployment are enabled by keeping development, staging, and production environments as similar as possible.
- **Logs -** Rather than managing logfiles, treat logs as event streams, allowing the execution environment to collect, aggregate, index, and analyze the events via centralized services.
- **Admin processes -** Administrative or managements tasks, such as database migrations, are executed as one-off processes in environments identical to the app's long-running processes.

2.1.5 Deployment models

There are two types of deployment strategies which are commonly used today mutable deployment and immutable deployment . The server which handles mutable deployments is called a Mutable or Snowflake server and the one which adopts immutable deployments is called a Immutable or Phoenix server.

Mutable-Snowflake Server

Generally Monolith applications are deployed on a massive mutable server or what are commonly called as a snowflake server. A snowflake server can be defined as a production server where application code is updated regularly along with configuration changes. These keep on changing every release and thus they are called mutable servers. The main disadvantage of mutable servers is that we cannot replicate or easily mirror the production environment for testing. When we get production faults, we can't investigate them by reproducing the error in a development environment. Since it is also a monolith monster server, it is not uncommon that after few release it grows in complexity and size. And then the only way to reproduce any production issues is to copy the VM it is running on and run it on another server. This snowflake mutable server needs a restart every time the configuration or the application artifacts (JARs, WARs, DEBs, etc) and the time this server can be down while restarting can be considerable and in today's world businesses need to run 24x7 with any downtime and in most of these deployment models a team is dedicated to work during nights and out office hours (weekends). This kind of deployment strategy fails to adopt continuous deliver and deployment. Apart form this fast roll back to previous version of code is almost impossible. Its mutable nature induces so much state in to the running system that its hard to redeploy the previous working version unless a snapshot of the whole VM was deployed to a new server which is cumbersome process.

Immutable-Pheonix Server

An Immutable Server is a server that once deployed, is never modified, merely replaced with a new updated instance[21]. With this approach we know that whenever we deploy a new image or container to the production server, we know for sure that it was the same one we built and tested in the continuous delivery pipeline. The main idea behind this model of deployment is that instead of updating the server with new application code, configuration, etc., we deploy another server with the immutable package (application code, configuration, etc.) which is packed as a container or image and can run in parallel with the old version of the application on a different node. Immutable deployments gain immediate benefits especially with zero downtime by using a reverse proxy that points to this self-sufficient immutable package or the application image (run as container or a VM).

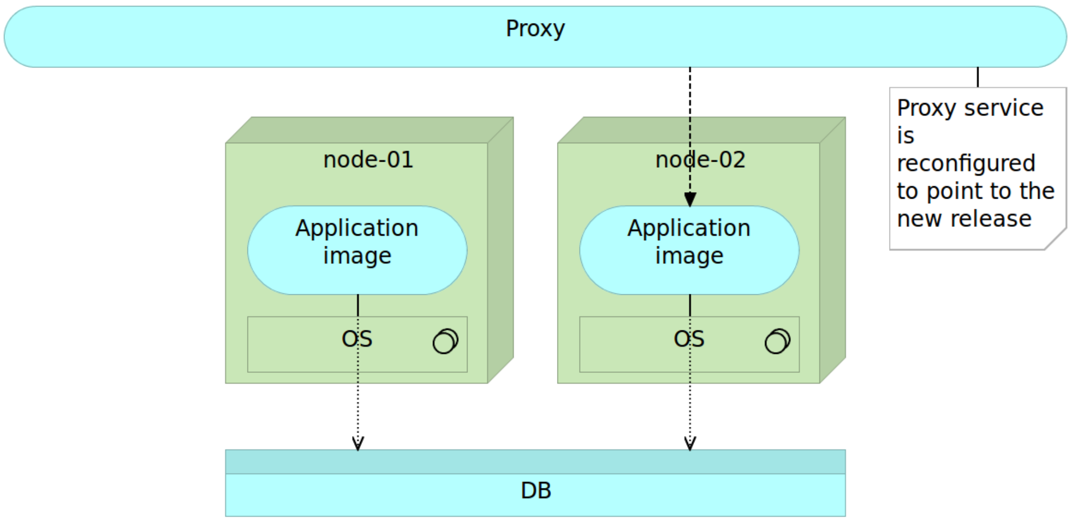


Figure 8: Proxy being routed to the new release of the application image[22]

All the traffic is passed through this reverse proxy, and whenever a new version of the application image is deployed on a separate server or the same server (if it s not using high resources), we then have two instances of the application running an older and a new version. We can hold of traffic to the new server until the final set of tests are passed(health checks and sanity checks) on the new instance and then let the proxy point to the latest application image as pointed out in Figure 8. Later we can remove the older version which does not serve the traffic anymore. This is called blue green deployment and can be achieved by a immutable deployment model.

2.1.6 Immutable Microservices

Immutable deployments can be even faster and easily manageable if the applications follow a microservices architecture since bigger applications take up lot of resources,

[21] https://martinfowler.com/bliki/ImmutableServer.html
[22] https://leanpub.com/the-devops-2-toolkit/read

testing time and deployment time. With microservices we have small independent services which allow us deploy, scale and test easily. They even can be deployed on the same machine. The same immutable deployment model works well for the microservices but instead of a serperate server we can deploy the latest release of the microservice in the same node. The mciroservice can be spread across the network on multiple hosts.

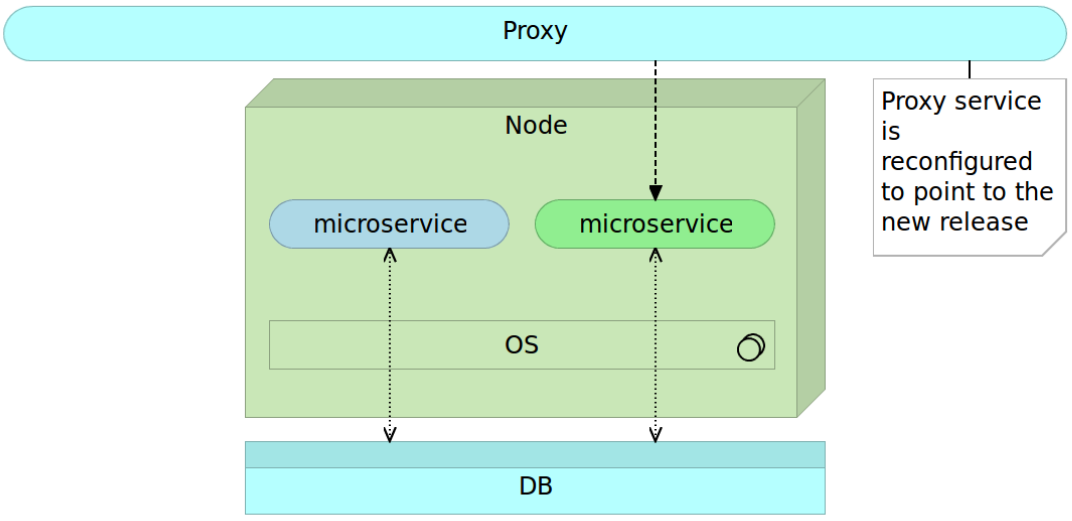


Figure 9: Proxy being routed to the new release of the microservice[23]

But soon as the architecture scales out, the number hosts grow faster than it will in a monolithic architecture and results in over provisioning and increased costs. If services are implemented in different programming languages, this means the deployment of each service will require a completely different set of libraries and frameworks, making deployment to a server complex. Linux containers[24] can help mitigate many of these challenges with the microservices architecture. Linux containers make use of kernel interfaces such as cnames and namespaces, which allow multiple containers to share the same kernel while running in complete isolation from one another[25]. Microservices are generally packaged as docker containers which are self sufficient and lightweight. We will learn about containers in the next section.

2.2 Containers

Cloud computing has offered a large amount of computational resources on an unprecedented scale and to deal with this abundance of customers from many fields with different resource needs, Cloud providers have used resource-sharing by leveraging virtualization to consolidate multiple customers onto the same hardware [27]. Until recently, only hypervisor-based systems (VMs) were popular with IaaS model but

[23]https://leanpub.com/the-devops-2-toolkit/read
[24]https://linuxcontainers.org/
[25]https://goo.gl/3XEDWd

today due to their light impact on performance, container-based systems, such as LXC[26], have gained space and becoming very popular under PaaS/IaaS Clouds with the emergence of Docker [25].

There are two main categories of virtualization, namely hypervisor-based virtualization and container-based virtualization. While containers provide OS-level virtualization, hypervisor-based virtualization is more at the hardware level. In VM based virtualization the host machine's resources are controlled by the hypervisor (Virtualbox, VMware Workstation) at Hardware-level In virtual-machine-based virtualization (Figure 10). In a container-based virtualization, also known as OS-Level virtualization the physical machine resources are partitioned, creating multiple isolated user space instances on the same OS. Despite this, users in these instances have the illusion they are working on their own independent subsystem of network, memory, and file system. For this reason, container-based systems are supposed to have a weaker isolation compared to hypervisor-based systems. However, from the point of view of the users, each container looks and executes exactly like a stand-alone OS [27]. Container based systems use LXC (Linux Containers)[27] which is an operating-system-level virtualization method for running multiple isolated Linux systems (containers) on a host using a single Linux kernel.

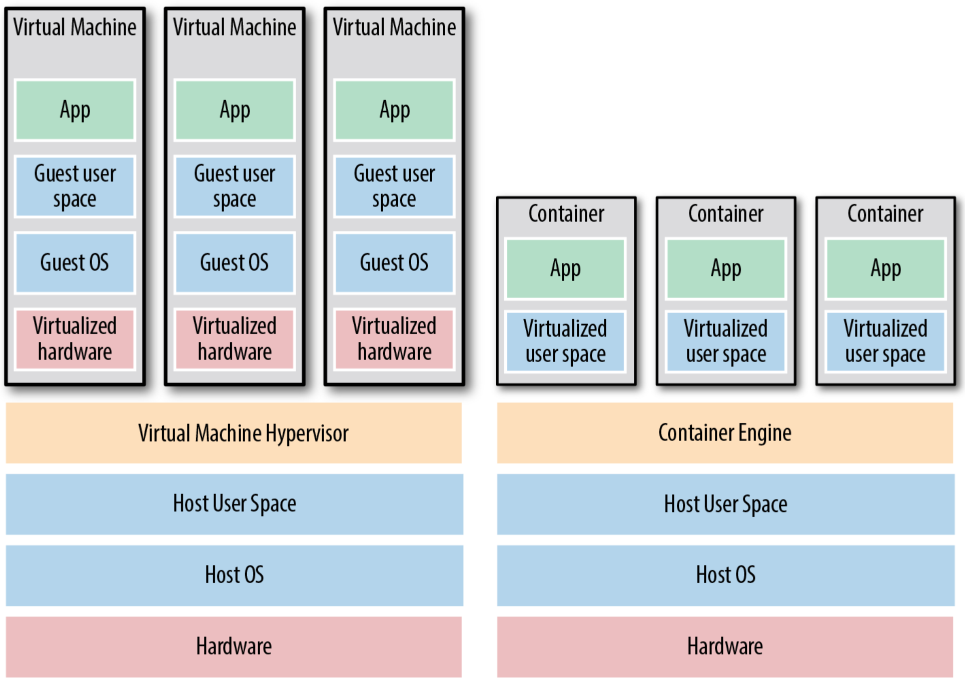


Figure 10: VMs vs Containers

Compared with hypervisor-based virtualization, container-based virtualization offers a completely different approach to virtualization. Instead of virtualizing with a system in which there is a complete operating system installation, container-based

[26]https://linuxcontainers.org/
[27]https://linuxcontainers.org/

virtualization isolates containers to work from within a single OS [28]. For example, a physical server running five virtual machines would have five operating systems in addition to a hypervisor that is more resource demanding than lxc. Five containers, on the other hand, share the operating system of the physical server and, where appropriate, binaries and libraries. As a result, containers are much more lightweight than VMs. With monolithic applications this is not so big of a difference, especially in cases when a single one would occupy the whole server. With microservices however, this gain in resource utilization is critical considering that we might have tens or hundreds of them on a single physical server. Put in other words, a single physical server can host more containers than virtual machines[29].

In order to achieve the isolation between the operating system and the top layer, container based virtualization uses Linux kernel functionalities namely namespaces and cgroups. These functionalities are usually wrapped into a more user friendly framework like Docker which is discussed in Section 2.2.3.

2.2.1 Namespaces

Since containers should not be able to interact with things outside and each container should be independent, the global host resources are wrapped in a layer of namespace that provides the illusion that the container is its own system. There are six different types of namespaces[30].

- **Mount namespace:** Isolates filesystem mount points seen by a container, in such way that processes in different containers might have different views of the file system hierarchy.
- **UTS namespace:** Allows each container to have its own hostname and NIS domain name.
- **IPC namespace:** Isolates the inter-process communication, meaning that processes containing in a containers have its own message queues, and they are completely independent from the others.
- **PID namespace:** Isolates the global PID space per containers, in such a way that might have processes with the same PID number running onto different containers. It allows containers to be migrated between hosts while keeping the same applications' PID number.
- **Network namespace:** Isolates the network subsystem, such as firewall tables, devices, IP address and IP route tables. Each container maintains its own networking configuration and the applications running on that can bind to the per-namespace port number space. This allows multiple web servers, for instance, to be hosted onto different containers with each server intensive to port 80 in its (per-container) network namespace.

[28] https://goo.gl/mFs4Ch
[29] https://leanpub.com/the-devops-2-toolkit/read
[30] https://lwn.net/Articles/531114/

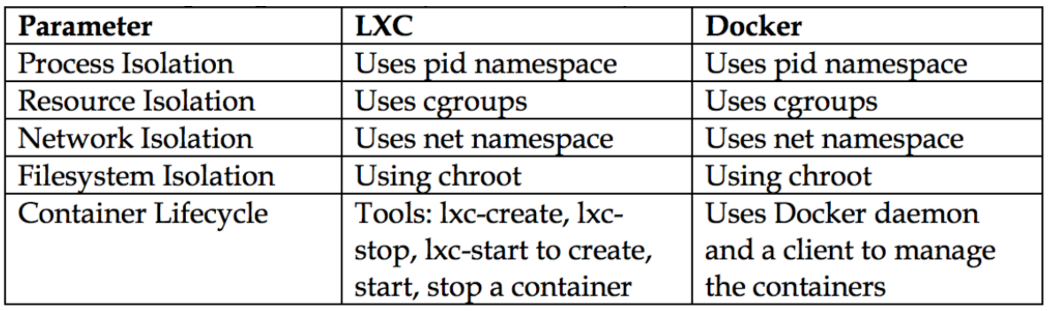
- **User namespace:** Isolates groups and users IDs from the host and other containers running on. It means that the user root (ID 0) has full privileges within a container, but without any privileges outside, ensuring safety and reliability.

2.2.2 Cgroups

Control groups, or cgroups for short, are a way to isolate shared resources. These resources include block IO, memory, CPU, and so on. Cgroups provide a mechanism for aggregating/partitioning sets of tasks into hierarchical groups with specialized behaviour[31]. A hierarchy is a set of Cgroups arranged in a tree, such that every task in the system is in exactly one of the Cgroups in the hierarchy. For example we can limit the resource usage for an application and all its child processes by adding it to one hierarchy. Cgroups make the containers running on the host machine use a fair share of CPU relative to the other containers. This prevents one or more containers use up all the resources and leave no computing resources to the others.

2.2.3 Docker

Docker rekindled the container technology which has been already known for many years but rarely used because of the complexity of building a container that is stable. Docker acts as an additional layer which abstracts away the complexity for the container execution environment for the users by utilizing a server-client architecture. Since version 0.9, Docker includes the libcontainer[32] library as its own way to directly use virtualization facilities provided by the Linux kernel, in addition to using abstracted virtualization interfaces via libvirt, LXC and systemd-nspawn[33]. Libcontainer is considered to be a tighter integration with the Docker framework where the execution environment is developed by the community in GO. The comparison between LXC and Docker are shown in Figure 11



Parameter	LXC	Docker
Process Isolation	Uses pid namespace	Uses pid namespace
Resource Isolation	Uses cgroups	Uses cgroups
Network Isolation	Uses net namespace	Uses net namespace
Filesystem Isolation	Using chroot	Using chroot
Container Lifecycle	Tools: lxc-create, lxc-stop, lxc-start to create, start, stop a container	Uses Docker daemon and a client to manage the containers

Figure 11: Comparing LXC and Docker [6]

The core component of the Docker framework is the Docker Engine which is sometimes also refered as docker daemon synonymously. The engine is basically a

[31] https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt
[32] https://goo.gl/obXl0t
[33] https://github.com/docker/libcontainer

daemon which is a long running program on the operating system that manages containers with the use of an execution environment. External Programs can talk to the daemon by using the REST API and instruct it what to do . These instruction are mostly given through the Docker CLI (Command Line Interface). he daemon creates and manages Docker objects, such as images, containers, networks, and volumes. Docker Images which are the building blocks of containers (also known as read-only containers) have intermediate layers that increase reusability, decrease disk usage, and speed up the image building process. The underlying technology behind these images is called UnionFS. Docker Engine can use multiple UnionFS variants, including AUFS (advanced multi layered unification filesystem), btrfs, vfs, and DeviceMapper. For instance if you have two images based on the same Linux distribution but two different programming language say Ruby and GO. These two images will point to the same base layer (the linux base image), and then put another layer on top of that (which is the programming language).

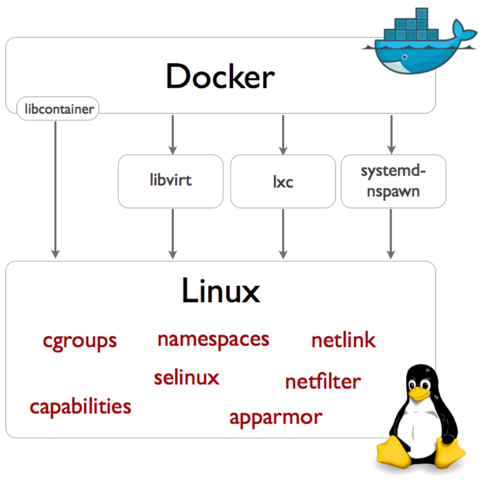


Figure 12: Docker and its Interfaces[34]

Docker Architecture

Docker uses a client-server architecture. The Docker container needs a host(a bare metal physical machine or a VM). The Docker client talks to the Docker daemon, which does the heavy lifting of building/pushing images to the registry, running, starting/stopping and inspecting containers, etc. The client and the server (docker daemon) communicate via API over UNIX sockets or a network interface. Docker Registry are used to store images built from the dockerfiles. There are many know registries available online where we can push our images. At BetterDoctor we use quay.io but we are in transition to moving towards an internally hosted private registry.

[34]https://goo.gl/NxOD8n

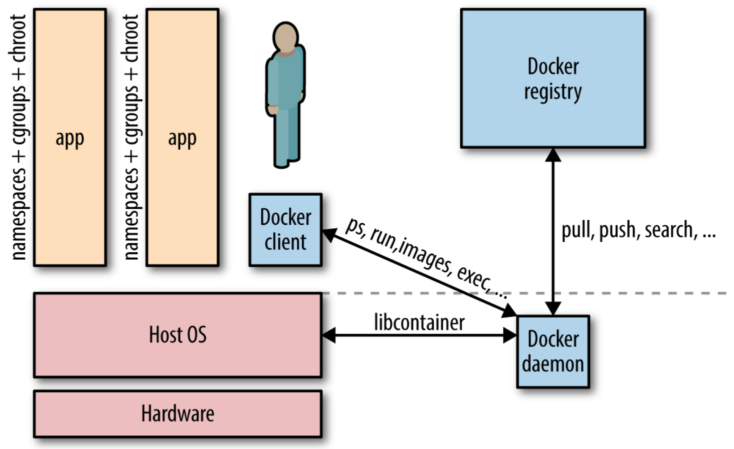


Figure 13: Docker Architecture Diagram[35]

2.3 Container Orchestration

The container ecosystem has become a very crowded space. Increasing rates of container adoption and usage in production have introduced new tools in the container management and orchestration space. While the container runtime APIs meet the needs of managing one container on one host, they are not suited to manage multiple containers deployed on multiple hosts. For example the Docker Command Line Interface (CLI) supports many container activities like pulling the image from the registry, building a new docker image from the docker file, uploading the image to the registry, terminating a running container, etc. But when it comes to managing multiple containers deployed on multiple hosts, docker CLI falls short and is not suitable for handling multiple containers. Therefore, the need for container orchestration tools crawled in.

Before we get in to details further about container orchestration tools, let us find out what a distributed cluster scheduler does which is the core component of a container orchestrating tool.

2.3.1 Scheduler

A distributed systems scheduler or a cluster scheduler takes an application by request of a user and places it on one or more of the available machines. In a docker world this means that the respective application image should be available on the host and also the local docker daemon must be present on the host to launch the application container. For example, a user might request the scheduler to deploy three instances (containers) of an application container. The scheduler then decides on which host the application should be deployed on the cluster based on its knowledge of the state of the cluster and resource utilization. It takes into account the necessary resources and constraints (launching an app on public facing host for example) required to launch the application on the host.

[35] https://www.oreilly.com/learning/docker-networking-service-discovery

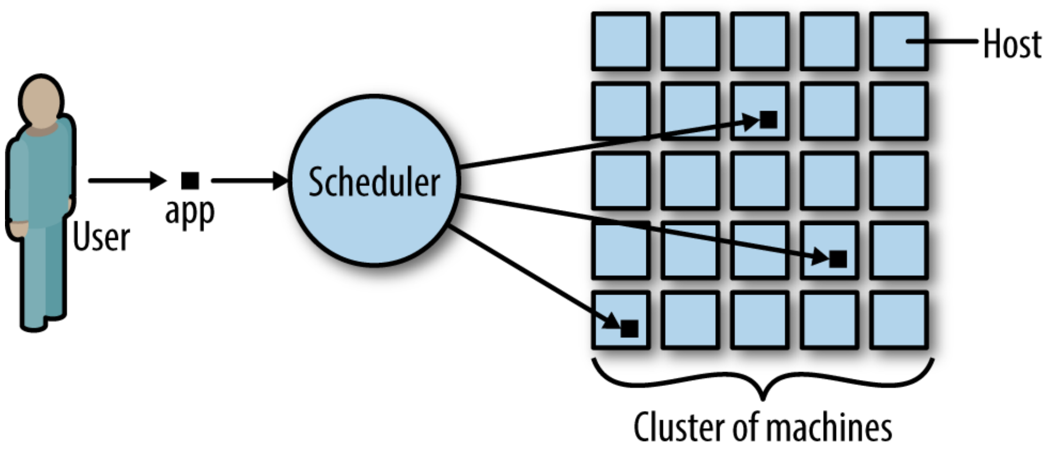


Figure 14: A distributed system scheduler[36]

A cluster scheduler has myriad of objectives which include efficient usage of cluster resources, working with user-supplied placement constraints, scheduling applications rapidly to not let them in a pending state, having a degree of "fairness", being robust to errors and always available [37]. According to the white-paper concerning Omega [21], a scalable scheduler for large compute clusters developed by Google there are three main types of schedulers.

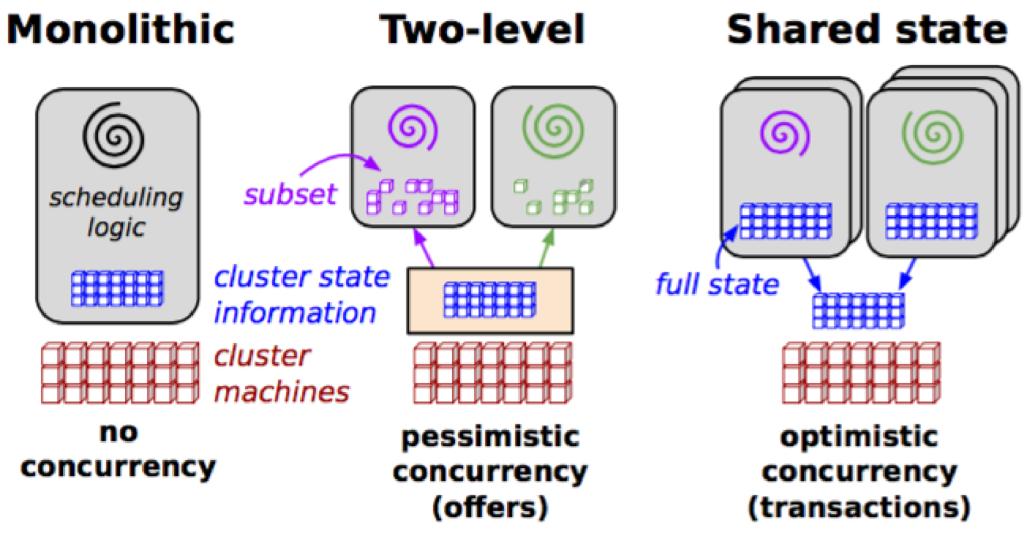

Monolithic
Two-level
Shared state
scheduling logic
subset
cluster state information
cluster machines
full state
no concurrency
pessimistic concurrency (offers)
optimistic concurrency (transactions)

Figure 15: Overview of the Scheduling Architectures [21]

2.3.2 Monolithic Schedulers

Monolithic schedulers use a single, centralized scheduling algorithm for all jobs. All workload is run through the same scheduler and same scheduling logic. Swarm,

[37]http://armand.gr/static/files/htise.pdf-ComparisonofContainerSchedulers
[37]https://www.oreilly.com/learning/docker-networking-service-discovery

Fleet, Borg and Kubernetes adopt monolithic schedulers. Kubernetes improvised on basic monolithic version of Borg and Swarm schedulers. This type of schedulers are not suitable for running heterogeneous modern workloads which include Spark jobs, containers, and other long running jobs, etc.

2.3.3 Two Level Schedulers

Two-level schedulers address the drawbacks of a monolithic scheduler by separating concerns of resource allocation and task placement. An active resource manager offers compute resources to multiple parallel, independent “scheduler frameworks”. The Mesos cluster manager pioneered this approach, and YARN supports a limited version of it. In Mesos, resources are offered to application-level schedulers. This allows for custom, workload-specific scheduling policies. The drawback with this type of scheduling architecture is that the application level frameworks cannot see all the possible placement options anymore[38]. Instead, they only see those options that correspond to resources offered (Mesos) or allocated (YARN) by the resource manager component. This makes priority preemption (higher priority tasks kick out lower priority ones) difficult[39].

2.3.4 Shared State Schedulers

Shared State Schedulers grant full access to the entire cluster resources by removing the central resource allocator. Each application level scheduler will have access to the entire cluster. The state of the cluster is shared between all the schedulers. An example for this type of scheduler is Omega and Nomad. By supporting independent scheduler implementations and exposing the entire allocation state of the schedulers, the architecture can scale to many schedulers and works with different workloads with their own scheduling policies. The major drawback with this architecture is that the schedulers must work with stale information (unlike a centralized scheduler), and may experience degraded scheduler performance under high contention.

Having briefly discussed about schedulers we can now move on to know more about the various container management tools. Container Orchestration tools extend lifecycle management capabilities to complex, multi-container applications deployed on a cluster of machines. These tools can treat an entire cluster as a single entity for deployment and management[40]. Container orchestration tools can automate all aspects from initial placement, scheduling and deployment to updates and health monitoring functions that support scaling and failover of the containers. The most common features of a container orchestrating framework are:

[38]It is referred to as Information Hiding in Omega paper [21]

[39]In an offer-based model, the resources occupied by running tasks aren’t visible to the upper-level schedulers

[40]https://thenewstack.io/containers-container-orchestration - Janakiram MSV

Declarative Configuration

The container orchestration tools allow the DevOps teams to declare a blue print for an application either in a JSON or YAML file before deploying it on to the cluster. The blue print (JSON file) includes information such as the application configuration, docker image repositories, networking and port configuration on the host, storage information (mounting volumes), etc. This also allows them to provide different configuration for application running in different environments (staging, development, production).

Rules and Constraints

Workloads often have special policies or requirements for host placement, performance and high availability. It is not rational to scale an application in a way that all the containers of that application are on the same host. Similarly, it may be a good idea to place in-memory cache on the same host as the web server. Orchestration tools support mechanisms for defining the affinity and constraints of container placement.

Provisioning

Container orchestration tools provide API calls to provision or schedule containers within the cluster and launch them. The orchestration framework will determine the right placement for the containers by selecting an appropriate host based on the specified constraints such as resource requirements, location affinity etc. The underlying goal is to increase utilization of the available resources. Most tools will be agnostic to the underlying infrastructure provider and, in theory, should be able to move containers across environments and clouds.

Discovery

In a distributed deployment consisting of containers running on multiple hosts as microservices, it is often very difficult to discover other microservices running in the cluster. Therefore, service discovery becomes a critical function. In most of the orchestration frameworks it is provided by a light weight DNS or proxy-based, etc.

Health Monitoring

Container orchestration tools will be responsible for tracking anf monitoring the health of the containers (mostly by polling the health-check endpoint) and hosts in the cluster. In case if a container crashes, a new one can be spun up quickly. If one of the slaves or hosts fail then the orchestration framework relocates the container to another host. It will also run specified health checks at the appropriate frequency and update the list of available nodes based on the results. Orchestration tools ensure that the deployment always matches the desired state declared by deployer in the blueprint (JSON).

There are many container orchestrating tools out in the market right now and its always a tricky situation to choose the right container orchestration tool to deploy and manage cloud-native applications. At BetterDoctor, we adopted Mesos/Marathon as our container orchestration framework. One of the main reasons was that Mesos was already production grade cluster management platform. Also, being a data company we could also benefit from running multiple frameworks on the cluster like Spark and other big data frameworks which could share the same cluster resources. The three widely used container orchestration platforms are Docker Swarm, Kubernetes and Mesos. We will go through Mesos more in depth while briefly covering the other two players.

2.3.5 Docker Swarm

Docker's native clustering tool Swarm is tightly integrated with the Docker API, making it well-suited for use with Docker. Swarm architecture is very simple and straightforward. Each host runs a Swarm agent and one host runs a Swarm manager. The sawrm manager is responsible for the orchestration and scheduling of containers on the hosts[41]. Swarm agents are just nodes having docker engine installed on each one of them. The Docker remote API available on these hosts is available for Swarm manager when starting the Docker daemon.

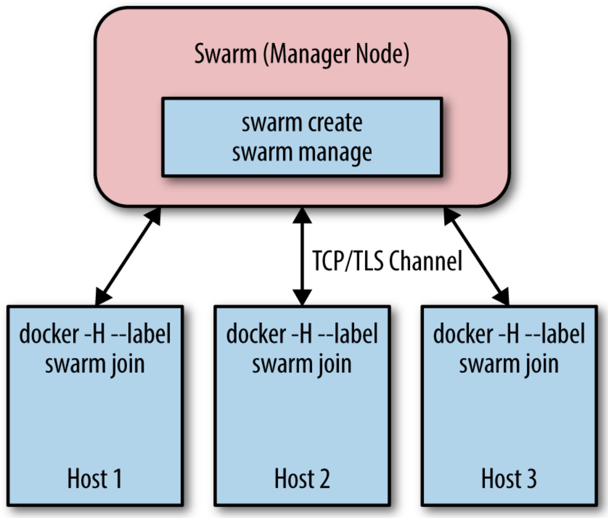


Figure 16: Docker Swarm Architecture[42]

Swarm mode[43] uses single-node docker concepts and extends them to Swarm. For example, to run a Docker cluster, we use the command run **docker swarm init** to switch to swarm mode. For adding more nodes under the manager, run **docker swarm join**[44]. Swarm has a YAML based deployment model inheriting Docker

[41]https://www.oreilly.com/ideas/swarm-v-fleet-v-kubernetes-v-mesos
[42]https://www.oreilly.com/learning/docker-networking-service-discovery
[43]https://docs.docker.com/engine/swarm
[44]https://goo.gl/lNLJzd

Compose. Auto-healing of clusters, overlay networks with DNS, adding constraints (Filters)[45], strategies (random, bin packing, spread) to distribute containers on the cluster, High availability through multiple masters are some of the core features of swarm. Swarm still does not yet support native auto-scaling or external load balancing. Scaling must be done manually or through third-party solutions. One thing to note about Swarm is that the scheduler is replaceable. We can, for example use Apache mesos scheduler instead of the default included scheduler. For discovering containers on each host, Swarm uses a pluggable backend architecture just like its scheduler. It supports different backends: etcd, Consul, and Zookeeper.

2.3.6 Kubernetes

Kubernetes is an opinionated open source container orchestration tool built by Google which claims that it generates, manages, and tears down 2 billion containers a week. It was first released in June of 2014, and is written in Go. Let us go through the Kubernetes architecture briefly. The architecture is made up of several components which are listed below

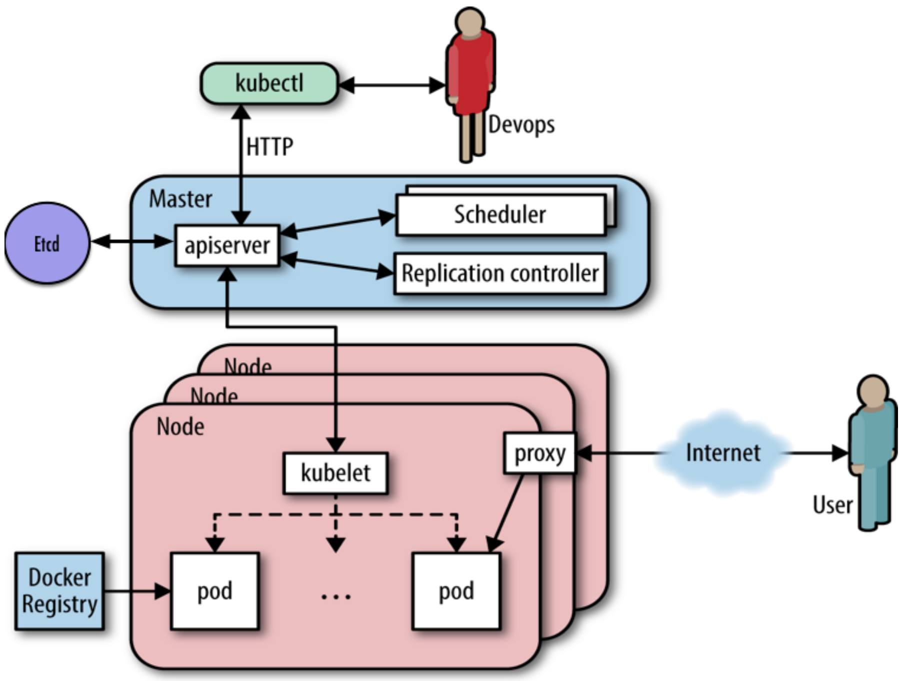


Figure 17: Kubernetes Architecture[46]

Master Components

The server that runs the Kubernetes management processes, including the API service, replication controller and scheduler. The API server is the only Kubernetes

[45] https://www.slideshare.net/rajdeep/docker-swarm-introduction
[46] https://www.oreilly.com/learning/docker-networking-service-discovery

component that connects to etcd; It validates and configures the data for pods, services, and replication controllers. It also assigns pods to nodes and synchronizes pod information with service configuration. The controller manager server watches etcd for changes to replication controller objects and then uses the API to enforce the desired state. Some examples of controllers are replication controller, namespace controller, endpoints controller, etc. The scheduler assigns workload to specific worker nodes (minions) in the cluster. Etcd is a distributed, reliable, and consistent key-value data-store for shared configuration and service discovery. Kubernetes stores all its cluster state in etcd. This state includes what nodes exist in the cluster, what pods should be running, which nodes they are running on, and a whole lot more. Etcd follows the principle of raft[47][19] consensus algorithm to operate as a data-store.

Node (Minion) Components

Each Kubernetes node has an agent called Kubelet runs, which is responsible for controlling the Docker daemon, informing the Master about the node status and setting up node resources. The Master exposes an API, collects and stores the current state of the cluster in etcd, and schedules pods onto nodes. The unit of scheduling in Kubernetes is a pod. Essentially, this is a tightly coupled set of containers that is always collocated. The number of running instances of a pod (called replicas) can be declaratively stated and enforced through Replication Controllers. Proxy service, available on each minion which is used to make applications available to the external world. This service forwards requests to the correct container by providing primitive load balancing.

Kubectl

kubectl is a command line interface for users to run commands against Kubernetes clusters[48]. Kubectl talks to the API server just like the docker CLI and is used to scale, delete list modify pods, do rolling updates, etc.

Service

A service is a grouping of pods that are running on the cluster. Services are *cheap* and you can have many services within the cluster. Kubernetes services can efficiently power a microservice architecture. The service definition, along with the rules and constraints, is described in a JSON file. For service discovery, Kubernetes provides a stable IP address and DNS name that corresponds to a dynamic set of pods. When a container running in a Kubernetes pod connects to this address, the connection is forwarded by a local agent (called the kube-proxy) running on the source machine to one of the corresponding backend containers.

Although kubernetes has a high learning curve, it is still one of the widely adopted container orchestration tools today because of its great community and supported

[47]`https://raft.github.io/`

[48]`https://kubernetes.io/docs/user-guide/kubectl-overview`

features like self-healing, secret and configuration management, scaling, rollbacks, service discovery, etc.

2.3.7 Apache Mesos and Marathon

Apache Mesos[49] is a general-purpose cluster resource manager that abstracts the resources of a cluster (CPU, RAM, etc.) in a way that the cluster appears like one giant computer to you, as a developer. We can say that Mesos acts like the kernel of a distributed operating system. It is hence never used standalone, but always together with so called frameworks, such as Marathon (for long-running stuff like a web server), Chronos (for batch jobs) or Big Data frameworks like Apache Spark or Apache Cassandra. Mesos began as a research project in the UC Berkeley RAD Lab by then PhD students Benjamin Hindman, Andy Konwinski, and Matei Zaharia, as well as professor Ion Stoica[50].

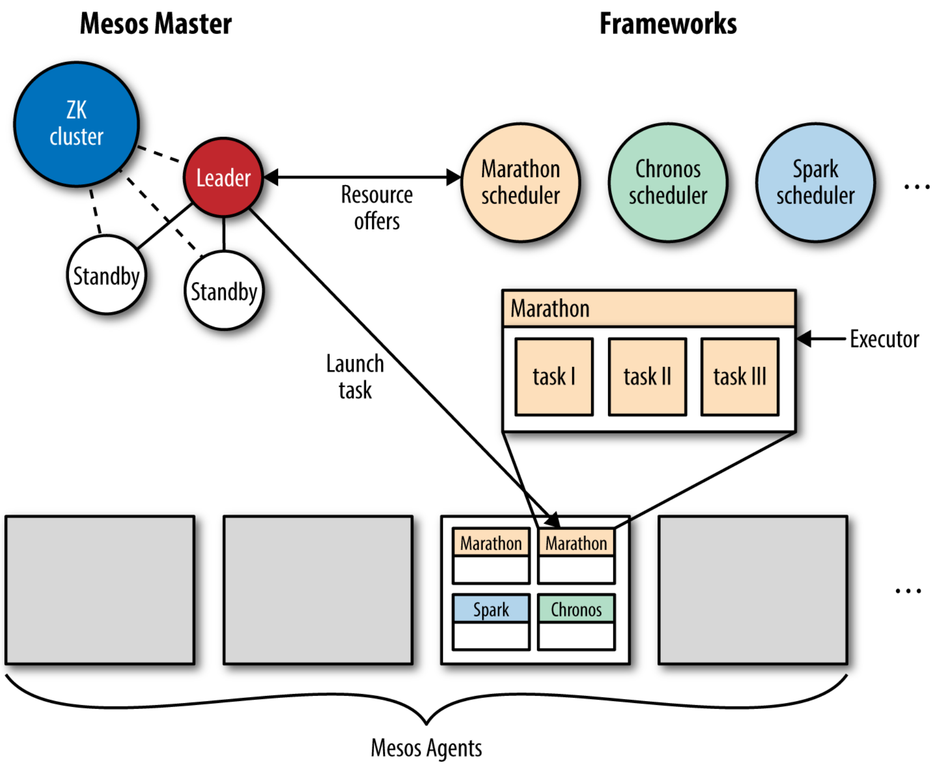


Figure 18: Apache Mesos Architecture[51]

As you can see from the architecture diagram 18, A Mesos cluster consists of one or more mesos-masters that manages slave daemons running on each cluster node, and frameworks that run tasks on these slaves. Each agent or slave is registered with the master to offer resources [8]. The master communicates with deployed frameworks to forward tasks to slaves. The master implements fine-grained sharing

[49] `http://mesos.apache.org/`
[50] `https://en.wikipedia.org/wiki/Apache_Mesos`
[51] `https://www.oreilly.com/learning/docker-networking-service-discovery`

across frameworks using resource offers. Each resource offer is a list of free resources on multiple slaves. If the Mesos master is unavailable, existing tasks can continue to execute, but new resources cannot be allocated and new tasks cannot be launched. Therefore, its very critical to make the master fault-tolerant. To achieve this, the cluster is run in high availability mode that uses multiple Mesos masters. There is only one active master running at a time which is elected as the leader with help of apache Zookeper, while the other masters will become followers. All the nodes in the system, including masters and slaves, communicate with ZooKeeper to determine which master is the current leading master.

As discussed earlier, about different types of scheduling, mesos comes under the two-level scheduler. In the first level, the master determines the free resources available on each node, groups them, and offers them to different frameworks based on organizational policies, such as priority or fair sharing. Organizations have the ability to define their own sharing policies via a custom allocation module as well. In the second level, each framework's scheduler component that is registered as a client with the master accepts or rejects the resource offer made depending on the framework's requirements. If the offer is accepted, the framework's scheduler sends information regarding the tasks that need to be executed and the number of resources that each task requires to the Mesos master. The master transfers the tasks to the corresponding slaves, which assign the necessary resources to the framework's executor component, which manages the execution of all the required tasks in containers. When the tasks are completed, the containers are dismantled, and the resources are freed up for use by other tasks.

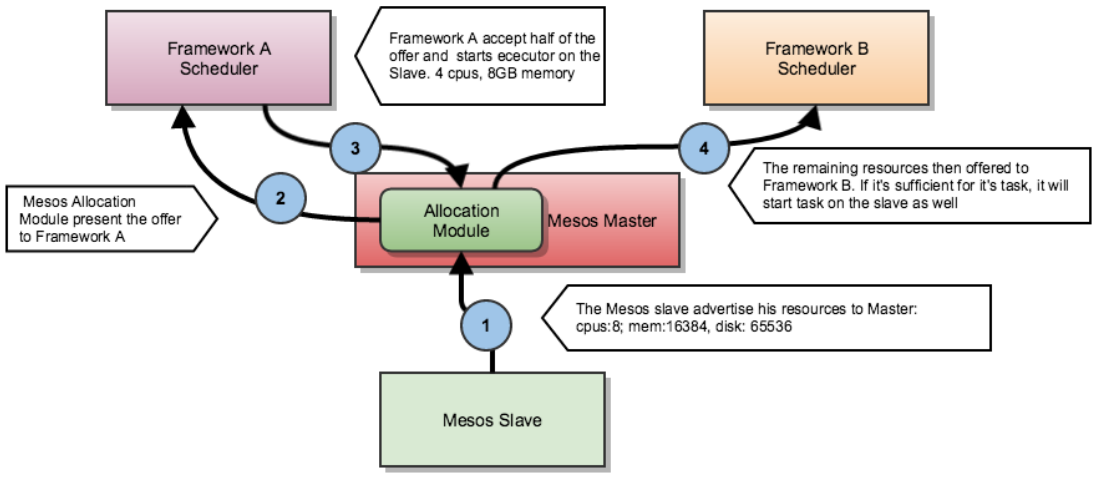

Framework A Scheduler
Framework A accept half of the offer and starts ececutor on the Slave. 4 cpus, 8GB memory
Framework B Scheduler
3
4
The remaining resources then offered to Framework B. If it's sufficient for it's task, it will start task on the slave as well
Mesos Allocation Module present the offer to Framework A
2
Allocation Module
Mesos Master
1
The Mesos slave advertise his resources to Master: cpus:8; mem:16384, disk: 65536
Mesos Slave

Figure 19: An example of resource offering

Figure 19 shows an example of how the resources are offered to the frameworks. In step (1) the mesos slave advertises its free resources to the master that it has 8 CPUs and 16 Gb of memory free. The master then invokes the allocation module which according to its fair-share or priority policy offers the resources to framework A (step 2). In step (3) the framework A's scheduler replies to the master with information about a task to run on the slave, using half of the advertised resources. The master then allocates the appropriate resources to the frameworks executor on the slave which launches the task. In step(4) the master advertises the remaining resources to the Framework B. Mesos also provides frameworks with the ability to reject resource offers. A framework can reject the offers that do not meet its requirements. This allows frameworks to support a wide variety of complex resource constraints while keeping Mesos simple at the same time. A policy called delay scheduling [29], in which frameworks wait for a finite time to get access to the nodes storing their input data, gives a fair level of data locality albeit with a slight latency trade-off.

Marathon

Marathon is a production-grade container orchestration platform for Mesosphere's Datacenter Operating System (DC/OS) and Apache Mesos.

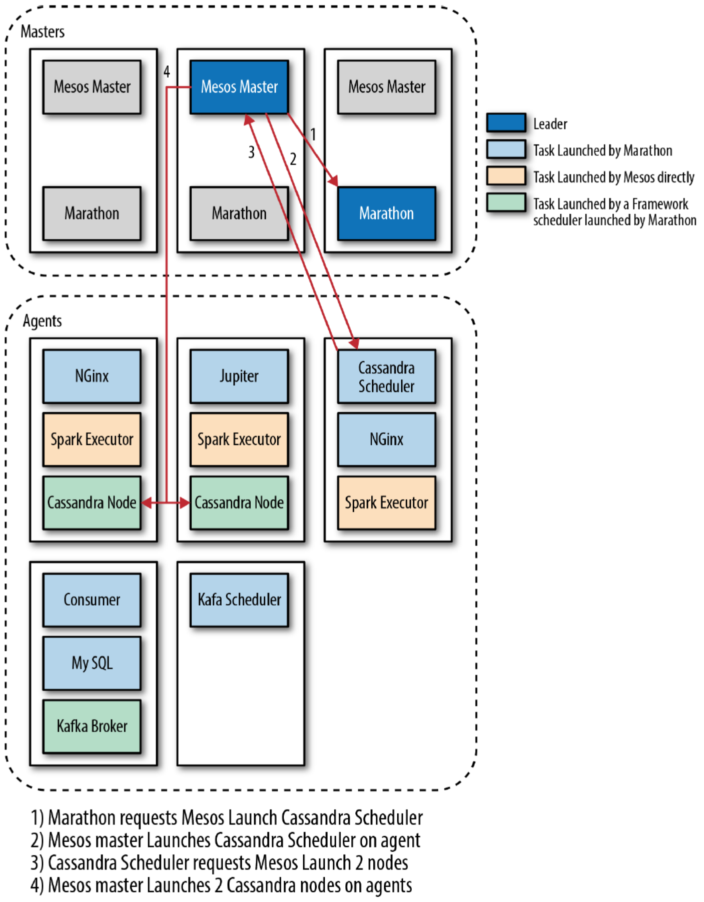

Masters
Mesos Master
Mesos Master
Mesos Master
Marathon
Marathon
Marathon
4
1
2
3
Leader
Task Launched by Marathon
Task Launched by Mesos directly
Task Launched by a Framework scheduler launched by Marathon
Agents
NGinx
Spark Executor
Cassandra Node
Jupiter
Spark Executor
Cassandra Node
Cassandra Scheduler
NGinx
Spark Executor
Consumer
My SQL
Kafka Broker
Kafa Scheduler
1) Marathon requests Mesos Launch Cassandra Scheduler
2) Mesos master Launches Cassandra Scheduler on agent
3) Cassandra Scheduler requests Mesos Launch 2 nodes
4) Mesos master Launches 2 Cassandra nodes on agents

Figure 20: Some interactions between Marathon, apps and Mesos[52]

It is a framework that is designed to launch long-running applications which serves as a replacement for a traditional init system. It has many features that

[52]https://mesosphere.com/wp-content/uploads/2017/03/Application_Delivery_with_Mesosphere_DCOS.pdf

simplify running applications in a clustered environment, such as high-availability, node constraints, application health checks, an API for scriptability and service discovery, and an easy to use web user interface. It is well known for its scaling and self healing capabilities. For example, if a Marathon app fails or the node that it is running on is lost, Marathon will automatically deploy a replacement task to Mesos and thus helps in maximizing up-time and reduce service interruption of the service.

Marathon can also be used to run other Mesos framework Scheduler as a Marathon app. Since it is designed for long-running applications, it will ensure that framework (running as a Marathon application) it has launched will continue running, even if the slave node(s) they are running on fails. Figure 20 illustrates the interaction between Marathon, its applications and Mesos. The framework scheduler in this case Cassandra is launched and monitored by Marathon. Now its up to Cassandra scheduler to request Mesos Master to launch its tasks.

2.3.8 Marathon LB

Marathon LB is one of the core component of DC/OS. Its main functionality is to manage HAProxy. It does that by connecting to the marathon API to retrieve all running apps and then generates HAProxy config and reloads HAProxy. By default, marathon-lb binds to the service port of every application and sends incoming requests to the application instances.

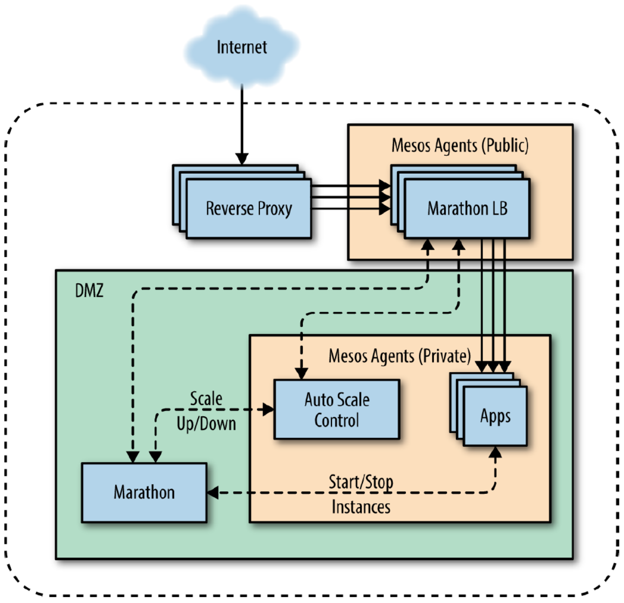


Figure 21: Marathon Load Balancer[53]

Services are exposed on their service port as defined in their Marathon definition JSON. It is also used to create virtual hosts for applications by exposing ports 80 and 443, in addition to their service port. As you can see from the Figure 21 it

[53]https://mesosphere.com/wp-content/uploads/2017/03/Application_Delivery_with_Mesosphere_DCOS.pdf

also is used to perform auto-scaling based on the amount of traffic hitting it from Internet. Marathon LB also plays an important role in performing the Blue Green Deployments.

2.3.9 Marathon Pods

A pod in Marathon links multiple apps together into a group of tasks that are executed together on a single agent. Pods allow interdependent tasks to be deployed together and to share certain resources. Tasks within a pod share a network interface. Pods allow quick, convenient coordination between applications that need to work together. Pods are particularly useful for transitioning legacy applications to a microservices-based architecture[54]. Listing 6 shows an example of a Pod with 2 containers.

DC/OS

DC/OS is a an opensource distributed operating system based on the Apache Mesos distributed systems kernel. It enables the management of multiple machines as if they were a single computer, pooling distributed workloads and simplifying rollout and operations. As a datacenter operating system, DC/OS is itself a distributed system, a cluster manager, a container platform, and an operating system[55]. As a distributed system, DC/OS includes a group of agent nodes that are coordinated by a group of master nodes. It uses Marathon framework for container orchestration and Mesos for cluster management.

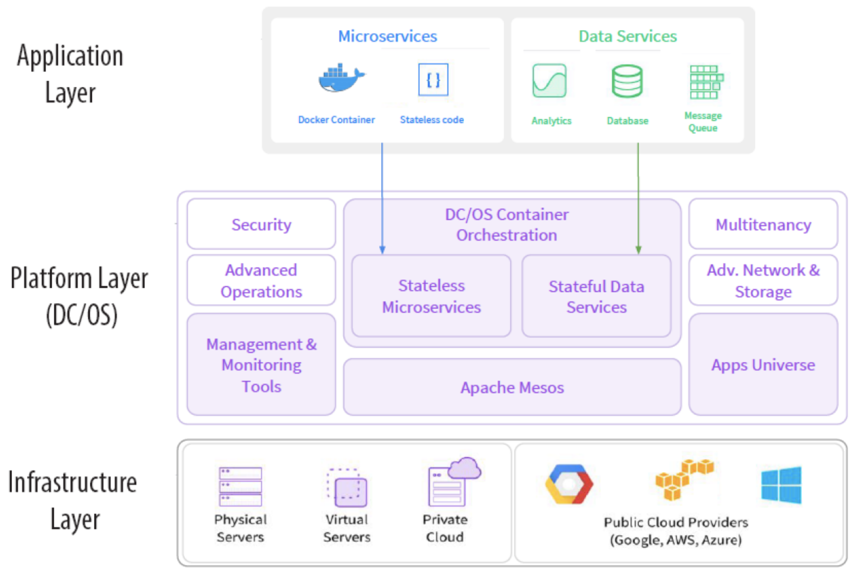


Figure 22: DCOS Architecture[56]

[54] https://dcos.io/docs/1.9/deploying-services/pods
[55] https://dcos.io/docs/1.9/overview/what-is-dcos/

To sum up DC/OS is a system made up of different software components, written in a range of programming languages, running on multiple Linux nodes in an appropriately configured TCP/IP network. There are many different DC/OS executables (components) running on each of the nodes along with their dependencies. Each of these DC/OS components provides some specific function or service (for example internal load balancing). DC/OS is the system that results from the combination of these individual services working together. The modern application, composed of microservices, containers, and stateful big data services, is key for startups/enterprises to capture new value chains in the fast paced market. The DC/OS model facilitates the adoption of microservices, big data and containers.

[56] https://mesosphere.com/blog/2016/09/16/dcos-1-8-networking-container-security

3 Service Discovery & Configuration Management

Configuration management and Service Discovery are two very well known problems which arise when we are dealing with modern application architectures which scale up and down according to our business needs. Often these microservices need to discover other microservices and data sources on which they are dependent on. When we need a urgent update to software on existing servers, logging into each one and making those updates is not the right solution as it is very manual and time consuming. Configuration management allows your servers to be configured in bulk, reducing labor, reducing operator error and improving overall management of your infrastructure. On the other hand, service discovery is important because in cloud and container based environments, static mapping of a service to IP address and port is no longer viable and as we scale up our business, we need better tools which can register and locate services which our microservices need to function.

3.1 Infrastructure as code

Infrastructure as code (IAC) is an approach to infrastructure automation based on practices from software development. It emphasizes consistent, repeatable routines for provisioning and changing systems and their configuration [18]. The motive behind IAC is that you write and execute code to define, deploy, and update your infrastructure. This represents an important shift in mindset where you treat all aspects of operations as software even those aspects that represent hardware [5] (e.g., setting up physical servers). In fact, a key insight of DevOps is that you can manage almost everything in code, including servers, databases, networks, log files, application configuration, documentation, automated tests, deployment processes, and so on.

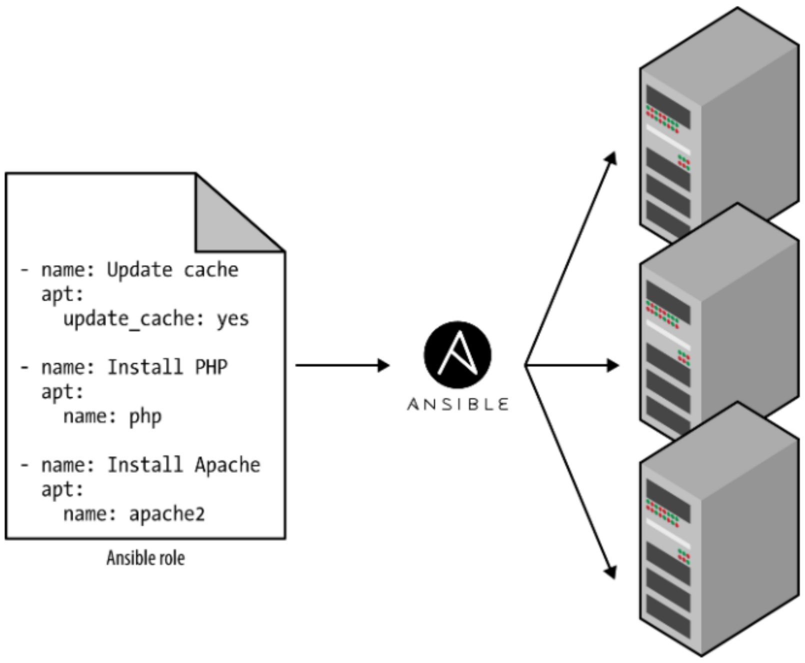
Adopting IAC means the infrastructure is in source files that anyone can read rather than a sysadmin's head and one can always roll back to previous version with the help of version control system. The infrastructure code can be reviewed and tested before the infrastructure change is applied. IAC is a very powerful methodology, by converting the manual practices to code one can improve the overall software delivery. According to the 2016 State of DevOps Report, organizations that use DevOps practices, such as IAC, deploy 200 times more frequently, recover from failures 24 times faster, and have lead times that are 2,555 times lower[57].

3.1.1 Configuration Management in an Immutable World

Configuration Management system is an IAC tool. It was considered as a revelation for the system administrators a decade ago. Since the IT systems never remain static and typically have countless dependencies that make up the entire system, even a small unnoticed configuration change can cause havoc on an entire infrastructure. To control the chaos of frequent releases, configuration managers like Chef, Ansible, Puppet, Salt, etc. provide the needed visibility and granular control of the performance,

[57]https://puppet.com/resources/whitepaper/2016-state-of-devops-report

functions, and physical aspects of IT infrastructure, thereby giving systems team the ability to standardize and sufficiently document the entire lifecycle of a given platform. This in turn facilitates effective change management, increases system predictability and scales the speed and agility of software deployments according to demand. As you can see in the Figure 23 configuration management tool like Ansible can execute the required code across a large number of servers.



```
- name: Update cache
  apt:
    update_cache: yes

- name: Install PHP
  apt:
    name: php

- name: Install Apache
  apt:
    name: apache2
```

Figure 23: Configuration Manager in action[58]

Today, it is interesting to see where CM's stand, with advent of Immutable infrastructure, provisioning a new server with required updates is not tedious task anymore. Configuration management tools such as Chef, Puppet, Ansible, and SaltStack typically default to a mutable infrastructure paradigm. For example, if we tell Chef to install a new version of Apache, it will run the software update on your existing servers and the changes will happen in-place. Over time, as we apply more and more updates, each server builds up a unique history of changes. This often leads to a phenomenon known as configuration drift [5], where each server becomes slightly different than all the others, leading to subtle configuration bugs that are difficult to diagnose and nearly impossible to reproduce. The alternative is immutable infrastructure. We can achieve this by either baking the AMI (using packer) into the VM or with Docker. In fact, Docker makes it even easier. Docker images are immutable and creating them is easy, fast and painless. Replacing the old containers with new ones is also easy to do with the help container orchestration tools. Moreover, In a container world, the ability of tools like Chef and Puppet to

[58]https://www.oreilly.com/learning/why-use-terraform

do convergence[59] is becoming less exclusive, when we don't really have to converge anymore, and one can essentially just settle for a simple script that bakes the Docker images right up for you, and can even roll back changes on demand. *Never upgrade a server again. Instead, create new servers, and throw away the old ones.*

CM's are also an anti pattern in a way. The major motive behind IAC is to improve DevOps methodology by promoting self-service infrastructure where a team of developers can deploy their code to the servers without the help from the Ops team. But the Ops team is responsible for all the changes in the production and only they have the commit access to the configuration management repository. This is again more like everything is in the sysadmin's head. This is the anti-devops way of doing things. We also do not expect developers to run configuration management on the clusters that they are responsible for because CM's have a high learning curve.

Configuration management had its great days and it really did change the way we managed our infrastructure and allowed us to scale our infrastructure in ways we never had before. The advantages it brought us in reliability and consistency should not be understated, but it is also a technology, that like the hand-crafted perl scripts that came before it, who's time has come. With the adoption of the cloud and application containers, we can now do better than configuration management, we can run with immutability. This will make infrastructure more consistent, more reliable, more secure and more scalable than ever before[60].

3.2 Terraform

While there are many CM tools like Chef, Puppet, Ansible which are designed to install and manage software on existing severs, IAC tools like CloudFormation and Terraform are designed to provision infrastructure (servers). They are often known as Orchestration or Provisioning tools which can be used along side other CM tools which can configure the provisioned servers. Today, for a enterprise or a startup adopting technologies like Docker and Packer, CM tools no longer play an important role in their technology stack. With Docker and Packer, you can create images (such as containers or virtual machine images) that have all the software your server needs already installed and configured. Once you have such an image, all you need is a server to run it which can be provisioned by an orchestration tool like Terraform. It can provision infrastructure across many different types of cloud providers, including AWS, Azure, Google Cloud, DigitalOcean, etc. At BetterDoctor, we provision our Mesos (DC/OS) cluster on AWS infrastructure using terraform and an example code for the same is shown in the Listing 8 and Listing 7.

Terraform stores state about your managed infrastructure and configuration. This helps terraform to keep track of your infrastructure and lets you update or destroy the infrastructure you manage. This state is stored by default in a local file named "terraform.tfstate", but it can also be stored remotely, which works better in a team environment. This helps in sharing this state across the team enabling the team

[59] CM's identify a fixed point that is, the final condition desired and then configuring the process by which to achieve that condition. this process is called convergence

[60] https://hackernoon.com/configuration-management-is-an-antipattern-e677e34be64c

members to modify the existing infrastructure or destroy them when they do not need the environment anymore. For example you can bring up a QA environment on-demand. At BetterDoctor, we bring up an on-demand Mesos Spark Cluster and tear it down when we no longer need it. By adopting Immutable infrastructure, redeploying all servers even for a minor change is not a pain anymore. As the modern DevOps principle says *If it hurts, do it more frequently*[61]. Which means frequent deployments will need significant automation to keep up with the number of deployments per day and setting up this automation for immutable infrastructure takes a fair amount of work, but it helps to maintain and understand our infrastructure better.

3.3 Why do we need Service Discovery?

Building cloud-native applications that live in ephemeral environments has lot of advantages but it has few new problems that need to be addressed and service discovery is at the top of this list. When deploying docker microservices on marathon each deployment will result in instance or container of a service being instantiated on different machines in the cluster. We no longer have static set of servers to which we deploy our applications too. Therefore we need to have service discovery mechanism so that applications can find each other at any given point. It is not uncommon to find people using Chef, Puppet, and other configuration management tools to build service discovery mechanisms. This is usually done by querying global state to construct configuration files on each node during a periodic convergence run. Unfortunately, this approach has a number of pitfalls. The configuration information is static and cannot update any more frequently than convergence runs. Generally this is on the interval of many minutes or hours. Additionally, there is no mechanism to incorporate the system state in the configuration: nodes which are unhealthy may receive traffic exacerbating issues further.

The most common service discovery mechanism is using DNS[62] (Domain Name System). But DNS is not suitable for discovery in applications where multiple hosts are behind a single record. It is more suitable where we access to individual hosts is required since DNS cannot be used for load balancing except for simple round-robin. Additionally, many applications have no way of reloading cached DNS entries to pick up changes. Then we have proxy-based solutions like HAProxy[63] which again introduce a single point of failure. Therefore a service discovery tool should be able to provide the following

- **Discovery** - Services need to discover each other to get IP address and port detail to communicate with other services in the cluster.

- **Health check** - Only healthy services should participate in handling traffic, unhealthy services need to be dynamically pruned out.

[61] http://enterprisedevops.org/article

[62] https://en.wikipedia.org/wiki/Domain_Name_System/

[63] http://www.haproxy.org/

- **Load balancing** - Traffic destined to a particular service should be dynamically load balanced to all instances providing the particular service.
- **KV Store** - Applications can make use of hierarchical key/value store for any number of purposes, including dynamic configuration, feature flagging, coordination , etc.

3.3.1 Consul

Consul is a highly opinionated[64] tool for service discovery and configuration. It is distributed, highly available, and extremely scalable. Consul is built to be datacenter aware, and can support any number of regions without complex configuration. It also has a key/value store in place to support dynamic application configuration. The key/value store can be nicely paired with envconsul[65] which is also a HAashicorp tool which allows applications to be configured with environment variables, without having knowledge about the existence of Consul. It lets you choose how the application can be restarted, what values from Consul are going to be injected into the application and what the behaviour should be if something fails. This makes it especially easy to configure applications throughout all the environments: development, testing, production, etc.

3.3.2 Architecture

Every host that provides services to Consul runs a Consul agent. The agent is responsible for checking the health of the services on the node as well as for the node itself. Any unhealthy host is deregistered from the Consul service. Agents communicate with other agents and servers via specific ports and use both TCP and UDP protocols. The Consul servers store and replicate the data. Consul is also highly fault tolerant. If the whole service cluster for Consul goes down then this doesn't stop discovery. It does this by using a Gossip[66] Protocol to manage membership and broadcast to the cluster. This makes Consul not only a typical client-to-server system but also a client-to-client system. In addition the server nodes use the Raft consensus algorithm to provide consistency for leader election. While Consul can function with one server, three to five servers are recommended to avoid failure scenarios leading to data loss. A diagram of the interaction between the Consul agents and servers is shown below in the Figure 24.

The following steps take place in the life-cycle of an application with a running consul agent on the host.

- At startup, the service or container registers itself as a member of a given service with the consul cluster.

[64]Opinionated software does things in the right way (software development best practices) and trying to do it differently will be difficult and frustrating.

[65]https://github.com/hashicorp/envconsul

[66]https://www.consul.io/docs/internals/gossip.html

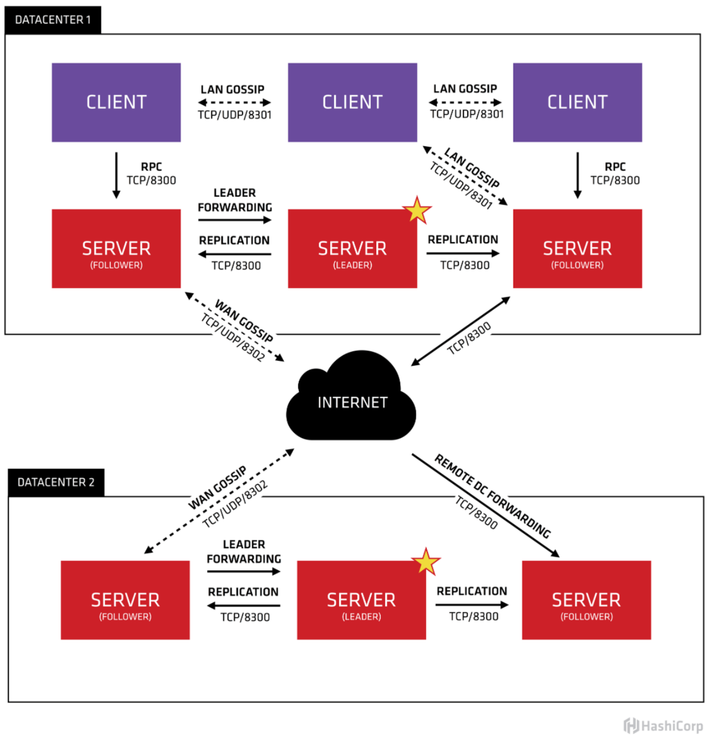


Figure 24: Consul Architecture[67]

- Introspects itself to determine whether it is healthy and send periodic heartbeat messages to consul to knows whether it is healthy.
- Check-in periodically with the consul agents which forward the request to the consul masters to see if changes have been made to upstream services.
- Any changes to the configuration cause the application to reload its configuration or otherwise respond properly to changes in the upstream services.

3.3.3 Secret Management

Secret Management is one the most neglected areas in an IT organization. Having secret keys spread around the organization's digital space is always dangerous. Some of the problems include

- Sensitive credentials and keys are stored (pushed) in certain code repositories (Github) .
- Sensitive credentials and keys are stored in plain text.
- Shared credentials and keys are used in numerous places.
- No scheduled key rotation.

[67] https://www.consul.io/docs/internals/architecture.html

BetterDoctor being a healthcare company had to take up this project for handling secrets and sensitive data properly to be compliant. Any good solution will have to follow security best practices, such as encrypting secrets while in transit; encrypting secrets at rest; preventing secrets from unintentionally leaking when consumed by the final application, etc. The best solution in the market right now is Vault - A secret manager developed by Hashicorp.

3.3.4 Vault

Vault[68] is the current gold standard in secret management and provisioning. The documentation is robust and have decent tutorials on how to set it up. At Better-Doctor, Consul is the storage backend for Vault which provides high availability. Vault tokens (e.g. *9cfced14-91ae-e3ad-5b9d-1cae6c82362d*) are the core method for authentication. Tokens bind to a list of Vault policies and are immutable. The secrets can be fetched from Vault API or via Consul if one has access to the application namespace. Every team member is assigned a vault token by which he can have access to allowed secrets. Even applications are assigned their own tokens through which they can access their secret keys like db passwords, API tokens etc. As you can see from Figure 25 *myapp-name* and *another-app* are two different applications storing their secrets in their own namespaces.

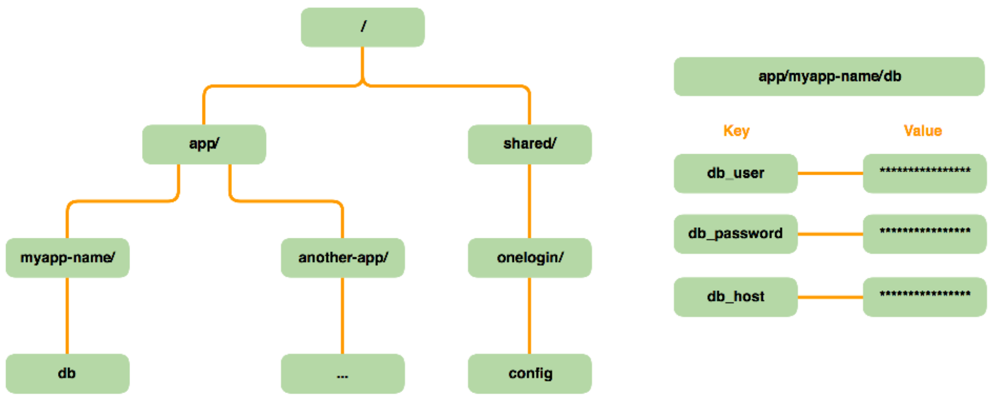


Figure 25: Vault Tree and Node[69]

[68] https://www.vaultproject.io/
[69] http://engineering.nike.com/cerberus/docs/architecture/vault

4 Requirements

Before designing the new Continuous Delivery pipeline the requirements for the same were gathered during a series of architectural meetings. In this section we go through the various requirements collected based on the suggested methodologies and architecture of the proposed CD pipeline. Apart form the underlying requirements which are implicit for building a CD pipeline, we also gathered specific set of requirements from developers and new requirements which arise due to the introduction latest technologies and frameworks. Eventually these requirements can be classified into functional or non-functional and also prioritize them as mandatory or optional requirements.

4.1 Underlying Requirements

Integrate with the existing technology stack

Analyzing existing technology stack and determine which components of the CD pipeline can be smoothly integrated with the existing stack and justify replacement for components which do not fit in the pipeline. The best case scenario for the CD pipeline is an add on to existing technology (Github, Jenkins, Slack, Chef, AWS).

Automated Testing on CI server

Analyze the need for automated testing of the feature branches and justify the need for CI server. Using a dedicated Jenkins Ci server which is close to production system is preferable. Not having the integration environment as production-like as possible inheres the risk that tests passing in the former could fail in the latter because of the differences.

Achieving Immutability

Why do we need immutability? Justify technology going to be used to achieve the Immutability. Although Docker seems to be the perfect candidate, can Immutability only be achieved via containers?

Deploy/Deliver the Same Way to Every Environment

It is essential to use same pipeline to deploy to every environment whether to staging or production. The difference between the environments have to be as small as possible also in the deployment and integration pipeline. This way, we test the deployment process many, many times before it gets to production, and again, we can eliminate it as the source of any problems.

Support Dynamic Runtime Configuration for Docker Containers

Analyze existing ways to configure a Docker container and come up with a centralized configuration management tool which can load environment variables in to application at run-time. Consul key/value store seems to be the top contender with envconsul driving the dynamic configuration changes.

Enable push button releases

This requirement comes from Continuous Delivery paradigm perform push-button deployments [10] of any version of the software to any environment on demand. This requirement makes it easy for developers to release the latest code into stage or production.

Enable Rollback

Being able to automatically revert to previous release easily. This helps in reducing the tension for deployment and encourages developers to deploy more frequently and thus pushing new features out to users quickly.

Mimic AWS auto-scaling for container environment

Since Marathon is capable of horizontal scaling - Ability to deploy multiple instances of the application and run a load balancer to route the traffic to the instances. It also needs to auto scale the application containers based on custom triggers. Since we no longer deploy applications to EC2 instances and Marathon does not provide features to auto-scale, a tool which mimics AWS auto-scaling is needed. For example, some of the applications in the data-pipeline use sidekiq queue length as a custom trigger to scale up the web and worker instances/containers.

Blue-Green Deployments

Blue-green Deployment[70] is a key feature of the continuous * . It enables the product team to launch releases without causing any outage to users. This release technique reduces downtime and risk by running two identical production environments called Blue and Green. At any time, only one of the environments is live, with the live environment serving all production traffic. This ensures zero-downtime strategy for deployments resulting in no interruption of the service even during deployment of a new release.

4.2 Developer Requirements

Empower teams with self-service application deployments

Enable developer/tester to deploy applications through self-service portal (Jenkins) with push-button provisioning. This enables team members to deploy to an environment for reproducing production bugs. It also Empowers the team to be able to deploy to staging and productions environments. Robust permission system should be setup to confine each team to deploy/configure its own application.

[70] https://martinfowler.com/bliki/BlueGreenDeployment.html

Notify on build success/failed

Notify team the status of the build/integration/deployment on the slack channel. keep the pipeline transparent to every team.

4.3 Optional Requirements

These are the requirements which fall out of the current project scope but are related very closely.

Automate the provision of the Cluster

Automate the creation of the Mesos Cluster which host the containers. This also helps to recreate the whole cluster in case of Disaster Recovery or Cluster upgrades.

Container Logging and monitoring

Analyze and Discover tools which can monitor containers just like EC2 machines. Ingest container application logs to existing ELK stack.

Notify on image pushed to docker registry

Notify the team on slack channel whenever a new image is pushed to the docker registry.

4.4 Classification of Requirements

Having gathered the various requirements let us classified them in terms being functional or non-functional and further prioritize them as mandatory and optional.

	Functional	Non-Functional
Mandatory	Integrate with the existing technology stack.	Automated Testing and CI server
	Deploy/Deliver the Same Way to Every Environment	Enable Rollback
	Achieving Immutability	Support Dynamic Runtime Configuration for Docker Containers
	Mimic AWS auto-scaling for container environment	
	Notify on build success/failed	
	Empower teams with self-service application deployments	
	Enable push button releases	
	Blue-Green Deployments	
Optional		Notify on image pushed to docker registry
		Automate the provision of the Cluster
		Container Logging and monitoring

Table 1: Overview and classification of requirements

5 Design and Implementation

In this section we first evaluate the status quo of the delivery and deployment process at BetterDoctor and understand the pain points in the process. Having discussed about various technologies and frameworks in the previous sections lets us see how do they fit in the proposed delivery and deployment model.

5.1 Analyzing Existing Setup

It probably is a good idea to consider that the first requirement is to integrate into current delivery process and therefore we need understand the status quo of the existing setup. This also gives us a clear picture of what other requirement are already fulfilled and the ones which are not by the status quo enabling us to build upon the existing setup.

5.1.1 Source Code Management & Git Branching Model

At BetterDoctor we use git as the source code management system and a standard branching model is followed[71]. It is important to understand the the current branching model because it defines the delivery and deployment pipelines. The pipeline begins when the local developer checks in the new code (feature) to the SCM. The current branching scheme dedicates a distinct feature branch for each new feature or a bug-fix while holding the current development mainline in a branch named “develop” and the version currently in production in a branch named “master”. The names of the development branches are by convention constructed from the ID of the corresponding bug report/feature request in JIRA[72] and a short, descriptive title, mostly also taken from JIRA.

Merging back of a branch to mainline, i.e. the “develop” branch, then requires the creation of a pull request and approval/code-review of another developer or the repository owner. A release tag might be created against a develop branch or a release branch which will be merged back to master. This git work-flow also suits pretty well for the proposed system.

5.1.2 Automated Testing and Building

The existing setup at BetterDoctor did not have any CI server to perform automated testing of the feature branches whenever a local developer checks in new code to SCM. The feature branch was tested locally and then if all the tests (unit and integration) pass, a pull request is created for the review/approval from peers or the repository owner. Although test-driven development was followed at BetterDoctor it was not leveraged to its full extent. Implementing a CI server will help in many ways. Some are listed below .

[71] http://nvie.com/posts/a-successful-git-branching-model/

[72] bug tracking and feature tracking software https://www.atlassian.com/software/jira

- Running tests immediately for every change helps us tho know if something is broken right away. If you are going to fail, then fail early, fail fast, fail often [10]. It is not so much the successful builds that are important, but the unsuccessful ones.

- Continuous Integration paves way towards Continuous Delivery/ Continuous Deployment by frequently building the software.

- Acceptance/Integration/End to End/Really long running tests may be run on the CI server that would not be run on a developer box usually.

- A developer may make a tiny change before pushing/committing to SCM and not run tests thinking its a safe change.

- Everyone can see whats happening - CI server displays lights that glow green when the build works, or red if it fails. This also helps in faster code review.

- The CI server is closer to the production system which can eliminate 'But it works on my machine' [22] syndrome.

There are several reasons for implementing CI, but the main point of CI is to get an idea what the state of the code is over time. The main benefit (out of several) this provides, is that we can find out when the build breaks, figure out what broke it, and then fix it immediately thus improving the software health and quality. Thus, building a new CI server was a major requirement for a continuous delivery pipeline.

5.1.3 Deployment

The systems team at BetterDoctor uses Jenkins to build, package and deploy latest releases to stage and production environments. A Jenkins job is manually triggered to pull the latest code for the release tag from the respective git repository. The deployer can select from a list (git tags) of releasable code and submit the job for the deployment to happen. Behind the scenes the job packages the code in to a debian binary (.deb) and then uploaded to AWS S3 storage [73]. Then the Chef Client is run on the respective application boxes (VMs) which then pulls the debian package and sets up the environment for the application to run. The current deployment model is illustrated in the Figure 26.

A dedicated chef role [74] for both stage and production environment is assigned for each application which performs all of the steps that are required to bring the node into the expected state when the chef client is run. For our proposed deployment model we replace chef with immutable containers and consul key value stores.

[73]Highly-scalable object storage https://aws.amazon.com/s3/

[74]A role in Chef is a categorization that describes what a specific machine is supposed to do. What responsibilities does it have and what software and settings should be given to it. https://docs.chef.io/roles.html

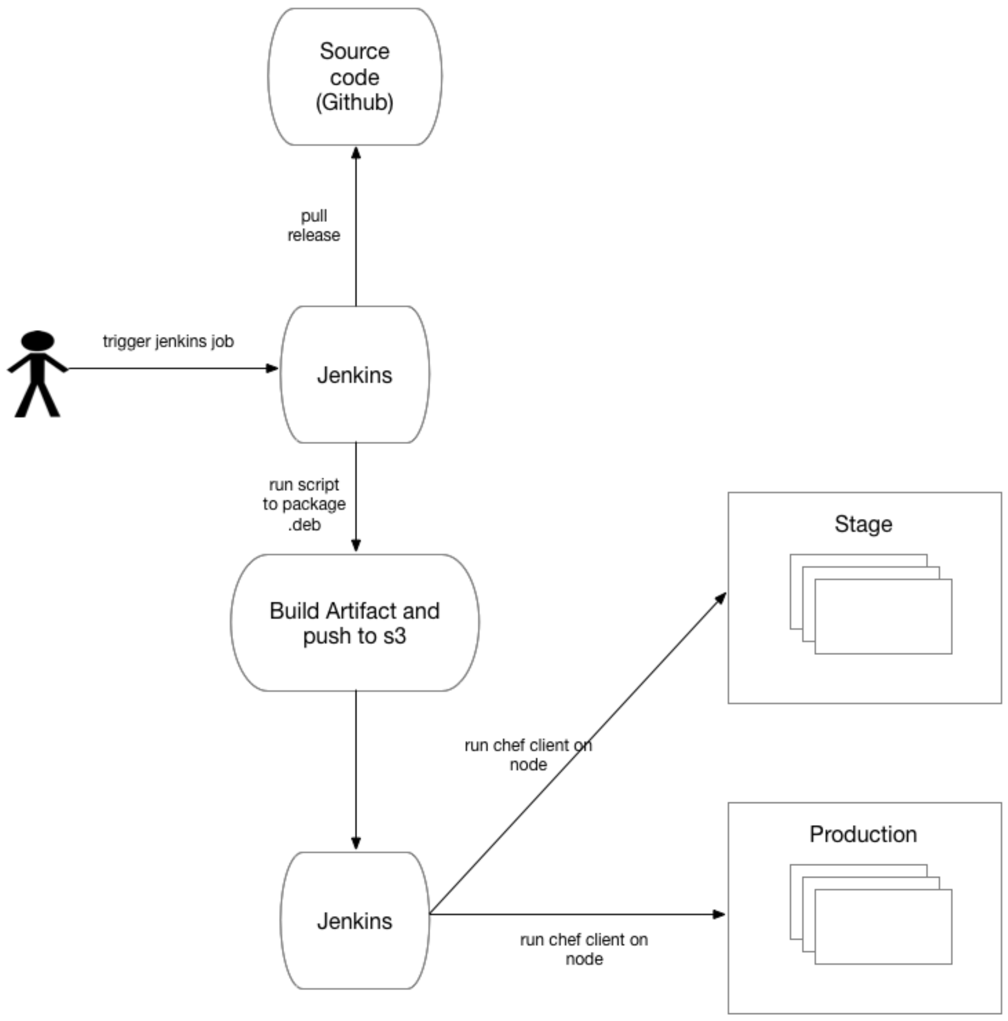


Figure 26: current deployment model.

5.2 Design choices

Continuous delivery could have also been accomplished with the existing setup but immutability cannot be achieved and Configuration management tools such as Chef typically default to a mutable infrastructure paradigm. This often leads to a phenomenon known as configuration drift. Moreover, with time Chef scripts and configurations turn into a enormous pile of cruft[75] and become a nightmare to maintain them.

Immutability can be achieved if the application processes are stateless. Their state is stored in a service outside of the "immutable infrastructure". It would be completely possible to deploy new VMs for every new version of your application (which can be automated) by baking your VM images for every release using tools

[75]unnecessarily complicated, or unwanted code or software

like Packer[76]. The objective is to be able to deploy new application nodes from scratch without upgrading your application in place which may induce some state over time causing the application to break or behave in an unexpected way. Attaining Immutability through VMs is a tedious task which is not ideal due to the fact that the artifact is very large (snapshotting a VM) and makes the process cumbersome. It is also important note that slow deployment process of the AMI (Amazon Machine Images) and vendor locking are the major frustration points. The AMI can be only used with Amazon Cloud vendor making it less portable and rigid.

Containers on the other hand were born to be immutable. They are much faster to build, test and deploy than VMs or running scripts to configure servers. "Immutable" containers model by rebuilding Docker images containing the application code and spinning up new containers with every application update. As soon as your application image has been built, tested and tagged, deploying it is a very efficient process. The two major advantages of using immutable containers are

Portability

Since the container will be self-sufficient, we need not worry about the underlying configuration of the OS. We build once and run them anywhere as the docker image is packed with all the necessary environment for the application to run.

Predictability

With immutable images, you can be sure that a given tag of that given image will always have the same behaviour because the code is contained in the image. This will also aid in rollback.

5.2.1 Jenkins

At BetterDoctor we are already familiar with Jenkins for automating daily manual tasks and deployment purposes. It also fits as a suitable platform to host a continuous delivery pipeline. Jenkins has become the most-used orchestrator for the different phases of the product lifecycle [2]: from the checkout of the code and the unit-tests, to the static code-analysis, to the performance tests, to the release of the binaries until the deployment into test/staging/production. It quickly transitioned from being a Continuous Integration tool to a Continuous Delivery one: thanks to the development of new plugins which also embrace docker. With many features and huge community support it manages to be positioned as the hub of CD pipelines.

5.2.2 Containerization

Containerizing or Dockerizing an application is the process of encapsulating an application with its operating system environment (a full system image) using

[76]Packer is an open source tool for creating identical machine images for multiple platforms from a single source configuration https://www.packer.io/intro/index.html

container software like docker and similar technologies. This helps in making the application's architecture more "cloud-friendly".

Benefits of Containerization:

- Streamlines deployment process and makes it easy to automate deployments, even having them driven completely from a CI system.
- Easy rollbacks on Production environment by just switching to older image version.
- The same container image can be tested in a separate test environment, and then deployed to the production environment. You can be sure that what you tested is exactly the same as what is running in production.
- Makes application portable and achieve cloud interoperability.
- Developers can also run containers locally to test their work in progress in a realistic environment.
- Hardware can be used more efficiently, by running multiple containerized applications on a single host that ordinarily could not easily share a single system.
- Containerizing is a good first step toward supporting no-downtime upgrades, canary deployments, high availability, and horizontal scaling.

At BetterDoctor all of the applications are developed in ruby. So, we needed a base ruby image which can access the custom gems from github repositories. Listing 3 shows the base ruby image which also has an envconsul binary copied. Envconsul helps in dynamic configuration management of the container. To run tests on container we need the following files to be available in the code repository

Application Image Docker file

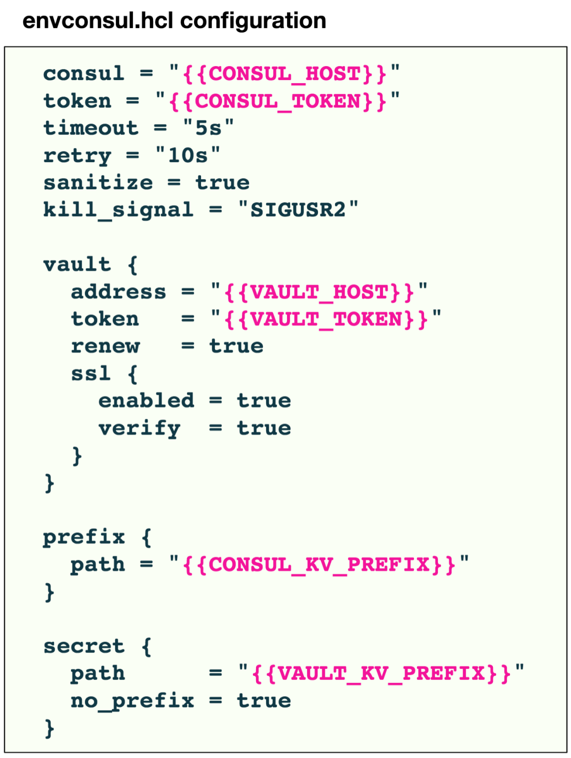
The Docker-File 5 encapsulates the whole code repository including the env-consul and docker-compose files.

Env-Consul Configuration file

This configuration file enables env-consul to talk to consul api to retrieve key value pairs (environment variables) for respective application . The consul token restricts the application to only read keys from its path. Similarly the vault token also allows the application only to read secrets on its path. Listing 4 shows the file for reference. A template env-consul file shown in the Figure 4

Docker-Compose file

The Docker-Compose file executes the test inside the container. Docker Compose makes it easier to configure and run applications made up of multiple containers. You can see an example docker-compose file below on Listing 2.


envconsul.hcl configuration

```
consul = "{{CONSUL_HOST}}"
token = "{{CONSUL_TOKEN}}"
timeout = "5s"
retry = "10s"
sanitize = true
kill_signal = "SIGUSR2"

vault {
  address = "{{VAULT_HOST}}"
  token   = "{{VAULT_TOKEN}}"
  renew   = true
  ssl {
    enabled = true
    verify  = true
  }
}

prefix {
  path = "{{CONSUL_KV_PREFIX}}"
}

secret {
  path      = "{{VAULT_KV_PREFIX}}"
  no_prefix = true
}
```

Figure 27: Env-Consul Configuration File

5.3 Building a CI/CD Pipeline with Docker

The workflow for the CI pipeline.

1. Developer makes a pull request on the feature branch on Github.
2. GitHub uses a webhook to notify Jenkins of the pull request.
3. Jenkins pulls the GitHub repository, including the Dockerfile describing the image, as well as the application and test code.
4. Jenkins builds a Docker image on the Jenkins slave node
5. Jenkins instantiates the Docker container on the slave node, and executes the appropriate tests
6. The status of tests are then communicated back to developer/team on the slack channel. The status is also updated on the Github pull-request page.

Listing 1 shows the shell script on Jenkins which is executed every time a pull request is made to the application repository on github. Jenkins automatically loads the the entire Pull Request in to Jenkins job workspace.

```
#bin/bash --login

set -e
```

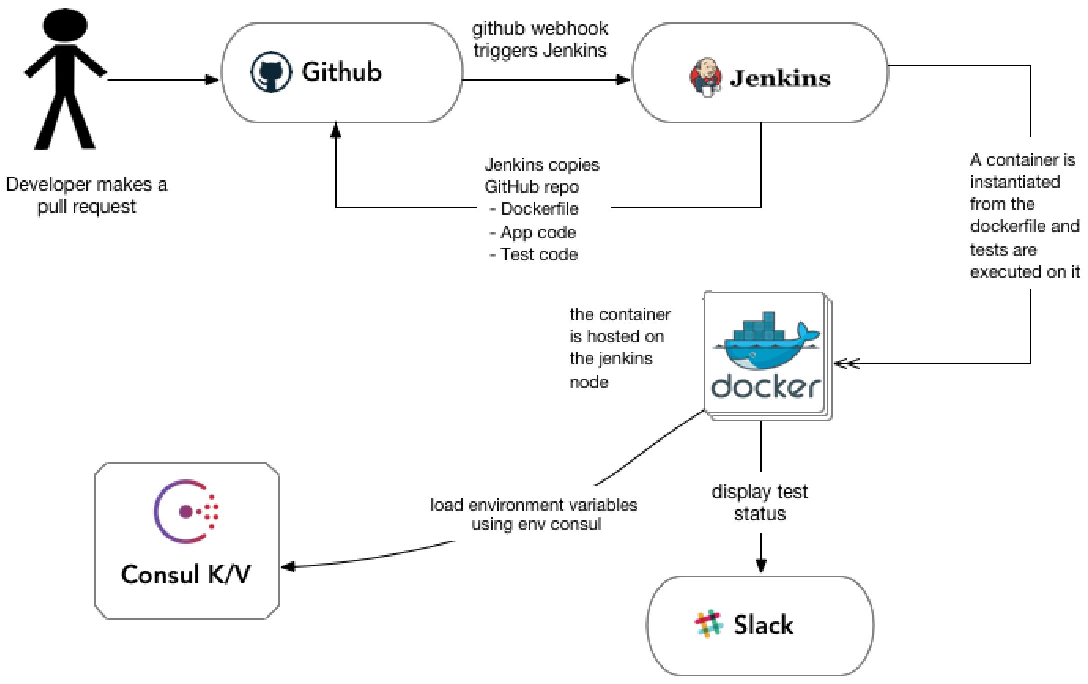

Developer makes a pull request
Github
github webhook triggers Jenkins
Jenkins
Jenkins copies GitHub repo
- Dockerfile
- App code
- Test code
A container is instantiated from the dockerfile and tests are executed on it
the container is hosted on the jenkins node
docker
load environment variables using env consul
Consul K/V
display test status
Slack

Figure 28: Continuous Integration with Docker

```

# Build Project
echo -e "\033[34mBuilding Project...\033[0m"
docker build .

# Run Tests
echo -e "\033[34mRunning Tests...\033[0m"

docker-compose -f docker-compose-tests.yml build
docker-compose -f docker-compose-tests.yml run -d mongo ;

echo \$?

docker-compose -f docker-compose-tests.yml run --rm etl; echo \$?
```

Listing 1: Jenkins CI Execute Shell

5.4 Drafting the Continuous Delivery Pipeline

The CD pipeline had to be just an extension of the CI pipeline. It is implemented with use of continuous integration tools, in our case it was Jenkins. The pipeline would generally consist of different types of jobs in Jenkins which are linked together or will be linked together as the process matures. By linking these jobs together we construct a chain of events and thus creating our customized continuous delivery and deployment pipelines. Although chaining jobs together involves high maintainability, thanks to the version control for Jenkins job configuration which helps us to rollback to previous configuration in the event of any issue.

A CD pipeline practicing DevOps methodology is not only responsible for releasing artifacts but also providing various tools for team members to help them deploy, monitor and search logs for their respective applications. The Jenkins deploy job deploys the respective container on to the Mesos Cluster. The Mesos Cluster is brought up by Terraform. Each Mesos slave is configured with user data. In our environment a typical Mesos Slave is made up of some default services which run on initial boot as a systemd[77] service. The Figure 29 shows the default services on a typical Mesos Slave. The services shown in the picture each have a specific use case.

1. **CAdvisor**[78] is a container running as daemon on the host (Mesos slave) that collects, aggregates, processes, and exports information about all the running containers on specific host. It gives a clear understanding of the resource usage and performance metrics of the running containers.

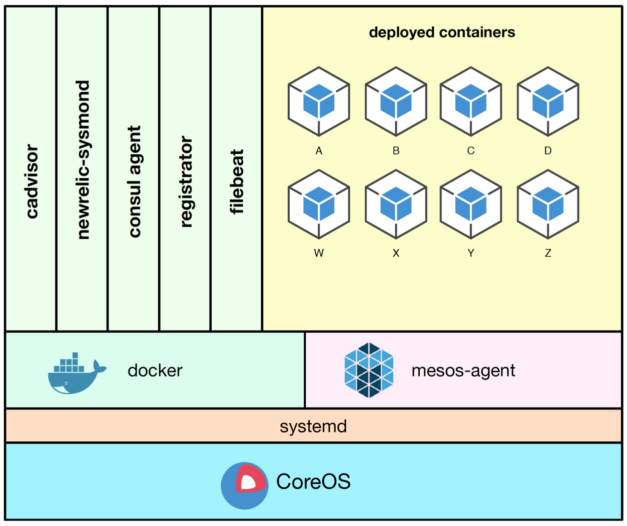


Figure 29: Mesos Slave running default services

2. **Newrelic Agent**[79] also collects server metrics and performance data to New Relic, whcih then is populated on to New Relic dashboards.

3. **Consul Agent**[80] is the core process of Consul. The agent maintains membership information, registers services (mostly docker containers), runs health-checks, responds to queries, etc.

[77] https://coreos.com/os/docs/latest/getting-started-with-systemd.html
[78] https://github.com/google/cadvisor
[79] https://docs.newrelic.com/docs/agents/manage-apm-agents/agent-data/agent-attributes
[80] https://www.consul.io/docs/agent/basics.html

4. **Registrator**[81] helps the runnig containers to register themselves. It is a docker container whose responsibility is to make sure that new containers are registered and deregistered automatically from our service discovery tool (consul). It teams up very well with consul agent.

5. **Filebeat**[82] helps to forward logs of the containers running on the host to our ELK (Elastic Search, Logstash and Kibana)[83] stack. This helps to send our logs to a centralized repository.

   By using these services each of the container deployed on to the Mesos cluster is monitored, registered and logged. All of the metrics can be visualized via the respective dashboards

5.4.1 Deploying the Containers

The containers can be deployed via a POST API call to Marathon API. To make this task easier for any member to configure, run, investigate containers which are going to deployed or running, we developed an internal Docker deployment tool called *Duncan*. Duncan talks to Marathon, Consul and Vault API under the hood. Using Duncan one can set secrets and environment variables for a running application dynamically which then triggers the env-consul process. They can also spin up a local docker container to debug an issues that crawled in. Some of the use cases of this tool are show in the Figure B1 and Figure B2.

5.4.2 Auto Scaling

Although Marathon offers auto-scaling based on resource utilization like memory, CPU [84] or also based on current RPS (requests per second)[85] for your apps based on HA-Proxy stats, we needed auto-scaling based on queue depth or size of the queue for our Sidekiq[86] workers. Therefore, the need for *Slythe* cropped up which monitors the Sidekiq queues of our applications and scales them up or down as needed. This tool also talks to Marthon API under the hood. More about this tool is shown in the Figure B3.

5.4.3 Monitoring and Centralized Logging

This was considered to be an optional requirement but it turned out to be a very important requirement for developers to troubleshoot and debug either in the pipeline or in any environment (production or staging).

[81] https://github.com/gliderlabs/registrator
[82] https://www.elastic.co/products/beats/filebeat
[83] https://www.elastic.co/products
[84] https://github.com/mesosphere/marathon-autoscale
[85] https://github.com/mesosphere/marathon-lb-autoscale
[86] https://github.com/mperham/sidekiq

Prometheus

Prometheus is a tool, initially built by soundcloud[87] to monitor their servers, it is now open-source and completely community driven. It works by scraping "targets" which are endpoints that post key-attribute machine parseable data. Prometheus then stores each scrape as a frame in a time series database allowing you to query the database to execute graphs and other functions like alerts. Prometheus servers scrape metrics from instrumented jobs (exporters), either directly or via an intermediary push gateway for short-lived jobs. The exporters[88] are long running jobs which collect metrics for a particular service and report it to a URL reachable by the Prometheus server. Once the exporter is running it'll host the parseable data on port 9100 and *http://<your-device-ip>:9100/metrics* end point gives you all the metrics captured. Prometheus can also use consul for service discovery and alert if any of the service registered becomes unhealthy. The metrics stored in the Prometheus server are then queried from Web UI like Grafana[89]. The Alert manager is responsible for notifying alerts to Slack or Pagerduty[90]. Figure 30 shows you the architecture of Prometheus monitoring system. Also, Figure B5 shows an example dashboard for Grafana.

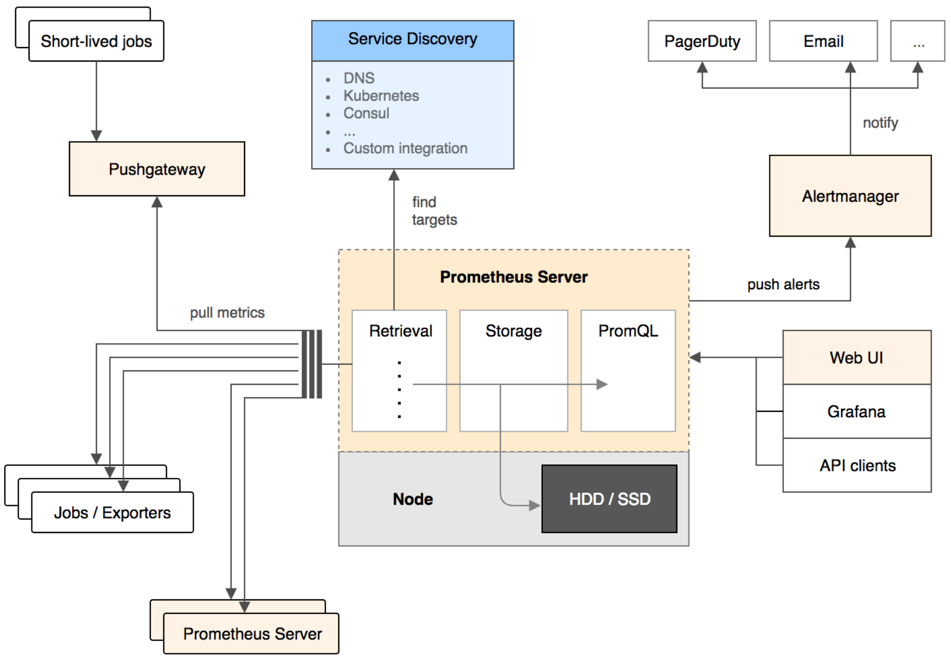


Figure 30: Internal Monitoring - Prometheus Architecture

[87]https://soundcloud.com/
[88]https://prometheus.io/docs/instrumenting/exporters/
[89]https://grafana.com/
[90]https://www.pagerduty.com/

5.4.4 ELK (Elasticsearch,Logstash,Kibana) Stack

The ELK Stack is a collection of three open-source products — Elasticsearch, Logstash, and Kibana — from Elastic[91]. Elasticsearch is a NoSQL database that is based on the Lucene[92] search engine. Logstash is a log pipeline tool that accepts inputs from various sources, executes different transformations, and exports the data to various targets. Kibana is a visualization layer that works on top of Elasticsearch. These three different open source products combined together are most commonly used in log analysis in IT environments and also help in achieving centralized logging. Filebeat on the other hand is a light weight shipper which forwards the logs to the logstash service from various hosts. As shown in the Figure 31 Logstash then sends the data to elastic search via REST protocol. We can also index the data based on the labels to segregate logs form each container. The logs can later be accesed and queried via Lucene syntax from the web UI called Kibana. Figure **??** shows the dashboard for Kibana.

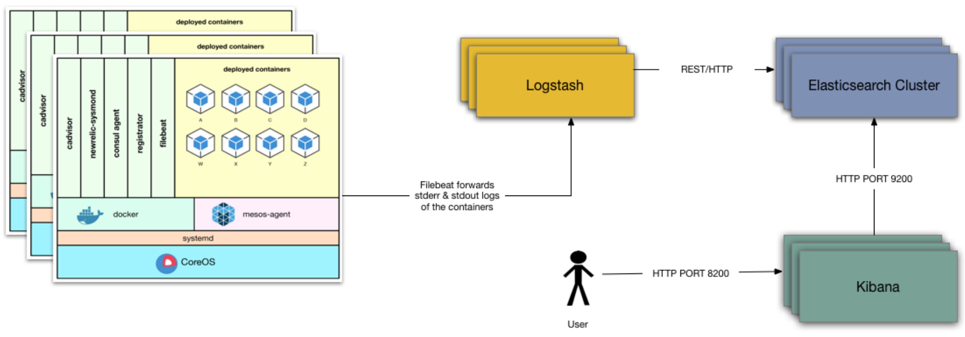


Figure 31: Containers forwarding the logs to the ELK stack

5.4.5 Continuous Delivery Workflow

Figure 32 shows the series of events involved in a deployment pipeline. Let us go through each one of them.

1. Developer makes a release tag on a branch after the feature passes CI.
2. GitHub uses a webhook to notify Jenkins of the new release tag.
3. Jenkins pulls the GitHub repository, including the Dockerfile describing the image, as well as the application and test code.
4. Jenkins builds a Docker image on the Jenkins slave node. Although testing the code again is not necessary, just to be on the safer side we do another round of testing on the new tag. Jenkins instantiates the Docker container on the slave node, and executes the appropriate tests.

[91]https://www.elastic.co/
[92]https://lucene.apache.org/

5. The status of tests are then communicated back to developer/team on the slack channel. If tests are successful, the image is then pushed to the quay.io private Docker repository.

6. Update slack with a message of the latest release tag which was pushed to quay along with the link which show the change or addition in code compared to previous release.

7. This step could be automated and just be part of a single Jenkins job but we are still not ready for continuous deployment and our requirement is to have a push button release. Using this deploy job we can deploy the application container on to stage or production environment on our Mesos Cluster.

8. The Jenkins deploy job posts the Json payload to Marathon API which in turn talks to the Mesos master regarding resources and constraints to launch the long running marathon task (container). An example of the json payload is shows in the Listing 6. An internal deployment tool called Duncan written in GoLang does the heavy lifting of posting this Json to the Marathon API. More about this tool discussed below.

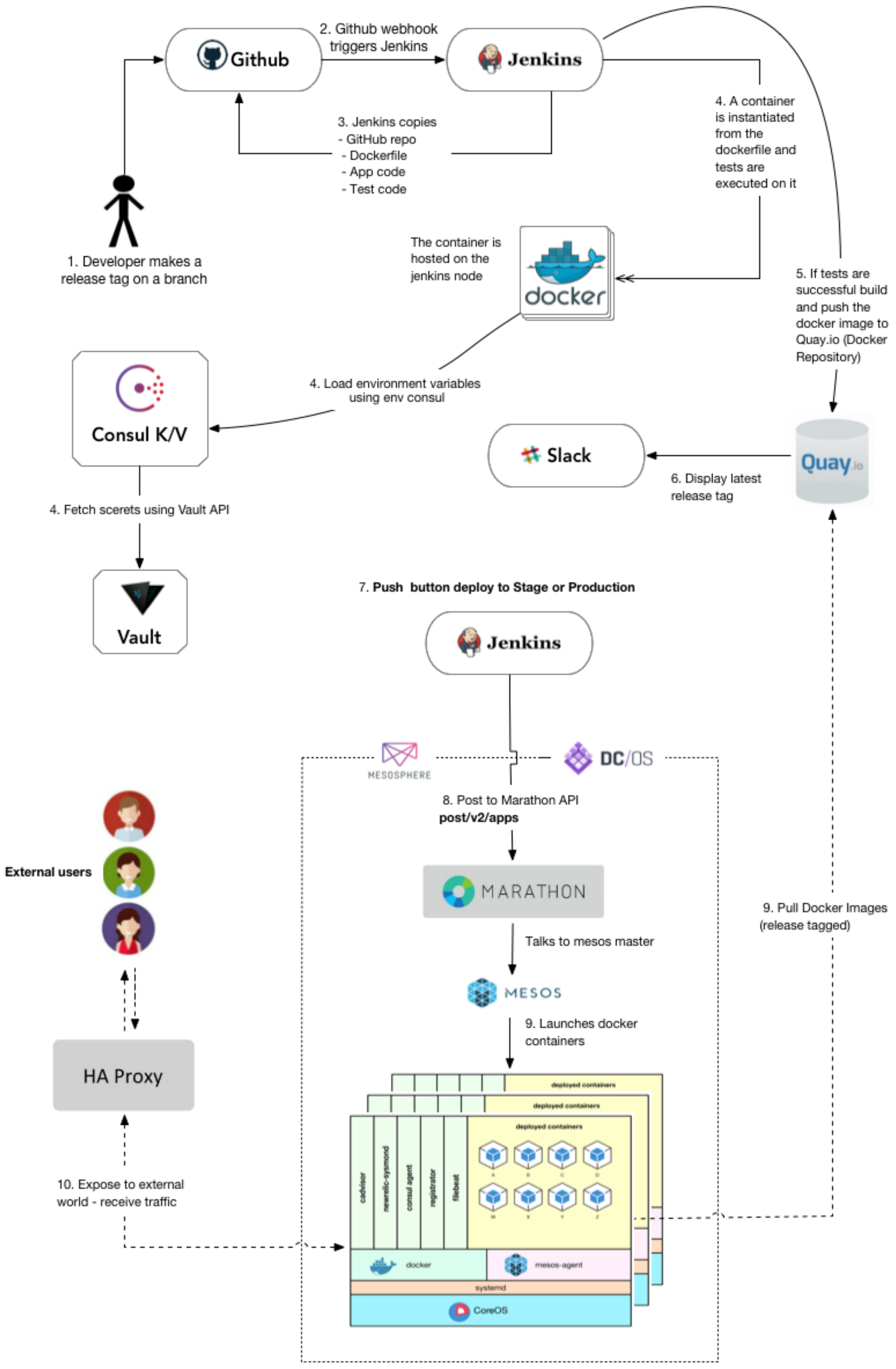
9. The Mesos Master then launches the task on the Mesos slave. The Mesos agent then runs the Docker Executor which then pulls the docker image from the quay.io repository. After successfully pulling the image the executor runs the container.

10. Public facing applications which are used by external users on the Internet are accessible via Marathon-lb[93] which is a HAProxy based load balancer. It provides proxying and load balancing for TCP and HTTP based applications.

5.4.6 Blue Green Deployments

Blue Green and Canary Deployments are supported by Marathon LB. This makes it very easy to implement in the CD pipeline. A newer version of application is only registered with the Marathon LB if it passes all the required health checks. If the health checks doesn't pass then Marathon will rollback the deployment. If everything goes fine then the older version of docker containers are replaced by the latest ones. This will enable users a smooth transition to the newer version of the application since there will zero downtime during deployments.

[93] https://mesosphere.com/blog/2015/12/04/dcos-marathon-lb/



Github
2. Github webhook triggers Jenkins
Jenkins
3. Jenkins copies - GitHub repo - Dockerfile - App code - Test code
4. A container is instantiated from the dockerfile and tests are executed on it
1. Developer makes a release tag on a branch
The container is hosted on the jenkins node
docker
5. If tests are successful build and push the docker image to Quay.io (Docker Repository)
Consul K/V
4. Load environment variables using env consul
Slack
Quay.io
6. Display latest release tag
4. Fetch sceret s using Vault API
Vault
7. Push button deploy to Stage or Production
Jenkins
MESOSPHERE
DC/OS
8. Post to Marathon API post/v2/apps
External users
MARATHON
9. Pull Docker Images (release tagged)
Talks to mesos master
MESOS
9. Launches docker containers
HA Proxy
10. Expose to external world - receive traffic
deployed containers
cadvisor
newrelic-sysmond
consul agent
registrator
filebeat
docker
mesos-agent
systemd
CoreOS

Figure 32: Series of events in the Continuous Delivery Pipeline

6 Learnings and Evaluation

This section tries to evaluate Continuous delivery pipeline based on the requirements defined in Section 4. We then list out all the learnings positive and negative during the implementation of this pipeline. It is worth mentioning that during the point in time in which this report was originally written, only half of our services where migrated to production Mesos cluster but all of the applications were running in a staged environment. So, these findings and learnings are just the first impression of the whole pipeline.

6.1 Improved Cycle time

Manual deployment which are considered to be an anti-pattern make it difficult to measure the cycle time accurately. The cycle time can be considered as the time from when work begins on a customer's request until a releasable artifact us produced through the continuous delivery pipeline. The major improvements in cycle time are seen for the bug-fixes. For example, a single line code-fix can be really benefit from the CD pipeline because there won't be any bottleneck in deployment and operations. As soon as the developer checks the latest code and makes a release tag, automated testing and artifact building is kicked off. Also, the respective team member no longer need to wait for a deployment expert to deploy the artifact on production since the developers can now deploy their application with their access-privileges. With continuous delivery in place it can be seen that it has vastly improved cycle time for smaller iterations and bug-fixes. Bigger code changes and features required some manual auditing and collaboration between team before deploying to production environments which made it difficult to analyze the cycle time.

6.2 Increased Productivity

Increased productivity is one of the main documented benefits of CD. CD helped in identifying issues earlier and enable fast feedback loops. By automating the building and testing of artifacts and providing easy tools to deploy the artifacts to production, developers could focus more on the actual development of the applications and less on menial, repetitive tasks related to operations and testing. However, it has to be noted that much effort has gone into establishing the new tools and practices. Developers agreed that some changes have required more time and effort than initially expected, mostly due to the fact that many of the tool (Docker, Mesos, etc) and techniques were unfamiliar to them when adoption started. But in the end it lead to a great development workflow which gives them enough confidence to make changes to their system at anytime. If a recent change did not break anything on production, the developer gets a moral boost for releasing to production often and early, which in turn improves speed of innovation and developer's productivity. Even when something breaks there are enough tools in the developers vicinity to act on the issue without seeking the help of any systems personnel which would cost time for both parties.

6.3 Breaking Down the Silos

Silo driven development is a barrier to the kind of large-scale cultural change that's required to build an effective DevOps process and fully realize the potential benefits of everyone's talents. The continuous delivery pipeline bridged the gap between development and operations, creating shared responsibility and ownership. Everyone now works together to get the code live. With improved end-to-end visibility (via slack) the team can now create a shared focus on the end result, and it encourages everyone to collaborate and work out how to tackle bottlenecks and mitigate production issues.

6.4 Impact of Docker on CD

As discussed in earlier sections the CD pipeline would also have been achieved if containers were replaced by VMs, but the big difference is the flexibility that comes with the light weight framework. Containers can start, stop and can be moved around different hosts, public clouds and private clouds within seconds which is not possible in case of VMs. This makes Docker perfect for quicker deployments and also highly scalable with high resource utilization. According to [6] and [15] container technology has better performance, scalability, and usability than hypervisor-based virtualization. In many cases, VM's performance tends to be inferior, whereas Docker containers perform equally to native performance of bare metal servers [23].

6.5 Moving to Immutability Infrastructure

One of the major advantages of moving to Immutable infrastructure is to get away from Chef (Configuration Manager). Why would we fix a faulty virtual server when we can create a new one in a matter of seconds. Our infrastructure provisioning tool terraform understands cloud better than any other CM or provisioning tool. It enabled us to write reusable scripts and it gave us better visibility on infrastructure updates. We now no need to worry about wasting our time fixing a VM when its faulty.

6.6 Achieving Dynamic Configuration Management

Envconsul has helped us to drive dynamic configuration changes from the Consul KV store and restart our application to take the values in a seamless way. Earlier we had to rebuild our Debian package by baking the configuration or via a Chef Role[94], even a single character mistake could have made us redeploy. It enabled us to achieve dynamic configuration in our services without taking them down to make the changes. We can now change configuration like the port number of any service, db password, access keys, etc. on run-time and don't have to worry about the service downtime.

[94] https://docs.chef.io/roles.html

6.7 Reduced risk of deployment failure

Manual steps in any part of the development process introduce risk of failure due to human error. The automatic testing of all changes has been proven to significantly reduce the risk of uncovering bugs in the applications after production deployments. Developers now have access to environments quite similar to those in production, which has maintains the dev-prod parity to reduce the risk of errors that can be discovered only once the software is running in the production environment. Not to forget the complexity and errors induced via Chef (configuration manger), which is great relief for sysops team. Apart from these, the deal breaker was the blue green deployment and self healing capability of the Marathon Framework.

6.8 Reducing Infrastructure Costs

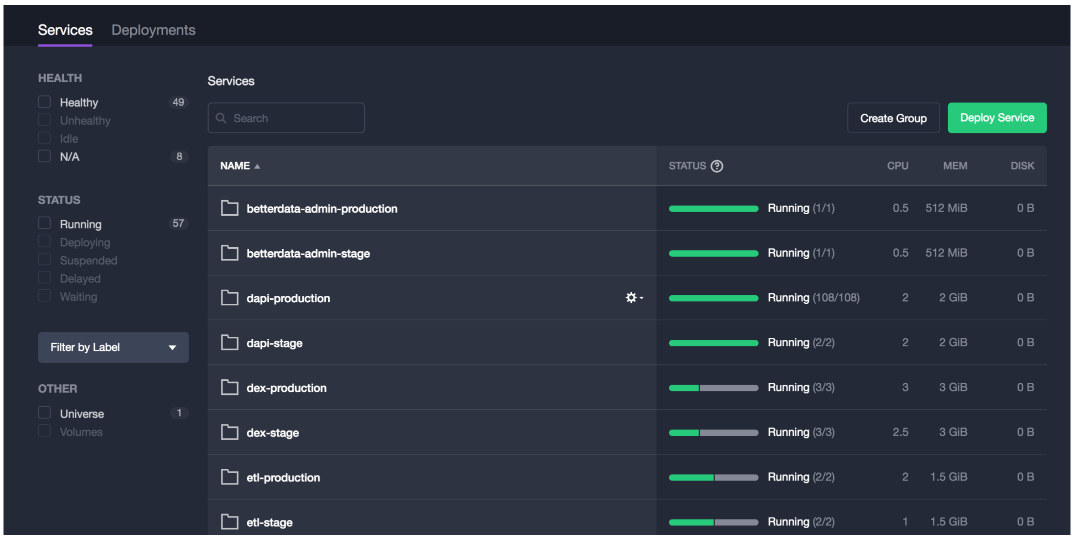
By moving just half of the applications from VMs to containers and running them on Mesos Cluster, an estimated 25-30 % savings on AWS infrastructure costs was observed. The overall resource utilization and optimization of the infrastructure has drastically improved. We have also noticed a significant decrease in VM sprawl or EC2 instance sprawling because we can now run staging and production environments in the same cluster and save resources and money. Although we had to be careful to set up resource limits so that our staging environment doesn't starve production for CPU, memory, or disk resources which was a bit time consuming due to testing.

6.9 Achieving Hyper-scale

Running containers at a large-scale is just a step away with help of Docker, Marathon and Mesos. Its now easy to run 100's and 1000's of Docker containers across 10, 50 Or even 10,000 hosts with help of the container orchestration and cluster management tools in place. Scaling our services to more than 15x or 20x is not at all tedious task anymore. We could see the true potential of the DC/OS platform when we needed to scale up our services to meet the client SLA. To achieve the same amount of scalability via VM's involves higher time and cost. This is really a game changer with regards to scaling your services instantaneously based on client demands.

6.10 Testing and Fine tuning Docker Environment

The most difficult phase in the project was to test and tune the Docker environment to match the VM production environment in terms of capacity, performance and acceptance. A large portion of developers time was required to team up with operations team in testing and fine tuning their respective applications in docker environment. The developers were already busy in their sprint tasks and couldn't dedicate enough time for testing their applications in docker stage environment. Later, a dedicated sprint task was allocated for each developer just to test and fine tune the applications in docker. Once all the acceptance criteria for each application was achieved, migration to production was kicked off.


Services Deployments

HEALTH
- Healthy 49
- Unhealthy
- Idle
- N/A 8

STATUS
- Running 57
- Deploying
- Suspended
- Delayed
- Waiting

Filter by Label

OTHER
- Universe 1
- Volumes

Services

Search

Create Group Deploy Service

NAME	STATUS	CPU	MEM	DISK
betterdata-admin-production	Running (1/1)	0.5	512 MiB	0 B
betterdata-admin-stage	Running (1/1)	0.5	512 MiB	0 B
dapi-production	Running (108/108)	2	2 GiB	0 B
dapi-stage	Running (2/2)	2	2 GiB	0 B
dex-production	Running (3/3)	3	3 GiB	0 B
dex-stage	Running (3/3)	2.5	3 GiB	0 B
etl-production	Running (2/2)	2	1.5 GiB	0 B
etl-stage	Running (2/2)	1	1.5 GiB	0 B

Figure 33: Scaling up to 100's (DAPI) of containers with ease

6.11 Developer Concerns

The developers were very accommodating in regards to the implementation of the CD pipeline. Although, there was a substantial learning curve involved to adjust to a new way of thinking and perhaps be trained in new technologies and practices. There were no major changes in regards to how they integrate code. They were already following a Test Driven Development model which made CD even easier.

6.11.1 Allow to Debug inside the containers

The developers were used to debug production and stage issues by logging on to the server using ssh. This practice is considered as an anti-pattern because the developers might induce state in to the server which may cause it to behave unexpectedly in the future. By providing enough tooling to bring up a local docker container which is running the same code as in production helped them to debug the issue locally. But occasional exec'ing into production containers is still performed. *Old habits die hard.*

6.11.2 Learning curve for Docker and Mesos

It's hard to deny that there was a significant learning curve for technologies like Docker and Mesos for everyone on the team. Docker was completely different to what it was in development environment compared to its production run. Scheduling a biweekly training session on Docker and Mesos along with architecture meetings to plan the release of the container migration helped the team members to get a grasp on the technologies and the delivery process. Building stateless dockerized server-based applications and launching them on Mesos (Marathon) was never so easy (at the cost of a steeper learning curve).

"I first had not enough confidence moving containers in to production and was not very comfortable with the technology stack. But, two weeks in to running the containers in production it made more sense to move everything to containers. It just made everything easy deploying, scaling and provisioning."

-developer XYZ

To conclude, there aren't really any disadvantages by moving towards a Continuous Delivery paradigm. We were manually building and deploying artifacts anyway. Instead, now Continuous Delivery as a process helps us streamline it. It definitely has far more benefits than disadvantages.

7 Discussion

7.1 Summary

Cloud Native Applications are the latest trend in IT that promises to develop and deploy applications in a rapid and cost-efficient way by leveraging cloud services. It drastically improves the pace of innovation among teams as they are able to focus more on functionality rather than worrying about underlying infrastructure. This helps the organisations to bring their innovations and capabilities to the market faster than ever before. Continuous delivery, DevOps and Microservices go hand in hand to fully implement a cloud-native strategy. All three aspects are required to achieve a successful low risk software-delivery process. CD helps in shipping software faster to reduce the time of your feedback loop. DevOps helps us to bring about the cultural and technical changes required to fully implement a cloud-native strategy. microservices is the software architecture pattern used most successfully to expand your development and delivery operations and avoid slow, risky, monolithic deployment strategies.

Containers revolutionize the way microservices are packaged. Their lightweight nature and tight resource management aligns well with the cloud native application approach, adding speed and resource efficiency. To use containers effectively they must be orchestrated. Container Orchestration tools like Marathon and Kubernetes can start, stop and distribute containers across a cluster of nodes. Some orchestrators like marathon for example are also responsible for providing functionalities like rollback and blue-green deployments which help reducing downtime for a cloud-native application.

Heavy-duty configuration management tools like Chef, Puppet, Ansible and Salt can decommissioned with the adoption of containers. Although you can still use these tools in conjunction with Docker to provision containers, the reality is you probably will not. The power, simplicity and speed of provisioning containers natively in Docker eliminates the need for configuration management tools. As discussed in earlier chapter why try to revive an faulty VM when you can re-deploy a new one. Provisioning tools like Terraform and CloudFormation help in achieving immutable infrastructure which reduces the potential for configuration drift and inducing state in the system. This increases predictability, since there is little variance between development, testing and production.

Service Discovery for microservices presents new challenges, which arise from the flexible and scalable nature of the infrastructure. Service Discovery tools like Consul help solve these challenges help the microservices to communicate to each other via API calls. Hashicorp tool Env-Consul which uses Consul KV store to load the environment variables in to the containers dynamically is a real game changer.

Monitoring and Centralized Logging are necessary for distributed services running on a cluster. By embracing microservices, containers and clusters, the number of deployed containers will rapidly increase at a great speed which makes it difficult look at logs for each service (by logging into a container or a node). By aggregating and visualizing these logs and monitoring metrics we can easily track down the production incidents and find patterns from the historical information.

7.2 Conclusion

Research in the field of continuous delivery is scarce and very recent. Thus, the validity of prior work has not been proven through years of scrutiny, and the opportunities for reflecting the results of this study against existing literature are limited. The organization has achieved and adopted CD pipeline which is very similar to that described by many authors in books and online material. The tools and practices adopted through out the project are current trending technologies which show a lot of promise in improving the maturity of the pipeline.

Introducing continuous delivery into an organization requires careful thought and planning. The transition is rarely something that can be done overnight, and usually involves a number of intermediate and incremental steps in order to shift away from the current development methodology. To succeed in building this CD pipeline the communication between business, development, QA and operations should take place on many different levels, which often leads to confusion and long wait times due to hand-offs between different people and groups. Continuous Delivery is not just about implementing a new workflow and pipeline for delivery. We cannot just automate the pipeline and expect to achieve a true agile development workflow. It has more to do with a getting everyone think in a DevOps manner to bring about the cultural change. The philosophy of *you build it, you run it* should be applied. Every team should be able to develop and deploy their code. The gap between development and operations can be bridged inducing a culture of collaboration. Share the tools and infrastructure code, accept and review pull requests, etc. The true potential of a continuous delivery pipeline is only realized when you have an operationally mature platform and when we move away from custom scripting, manual work and late-night outages. Developers and Operations personnel should transparently work together to plan, build, release and deploy.

References

[1] Martin L. Abbott and Michael T. Fisher. *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise*. Addison-Wesley Professional, 2015. URL https://books.google.com/books?id=yzUpD2YbhWwC.

[2] Valentina Armenise. *Continuous Delivery with Jenkins: Jenkins Solutions toImplement Continuous Delivery*. 2015. URL http://dx.doi.org/10.1109/RELENG.2015.19.

[3] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988. doi: 10.1109/2.59. URL http://dx.doi.org/10.1109/2.59.

[4] Barry W. Boehm. Anchoring the software process. *IEEE Software*, 13(4):73–82, 1996. doi: 10.1109/52.526834. URL http://dx.doi.org/10.1109/52.526834.

[5] Y. Brikman. *Terraform: Up and Running: Writing Infrastructure as Code*. O'Reilly Media, 2017. ISBN 9781491977125. URL https://books.google.com/books?id=ZH1VDgAAQBAJ.

[6] Rajdeep Dua, A. Reddy Raja, and Dharmesh Kakadia. Virtualization vs containerization to support paas. In *2014 IEEE International Conference on Cloud Engineering, Boston, MA, USA, March 11-14, 2014*, pages 610–614, 2014. doi: 10.1109/IC2E.2014.41. URL http://dx.doi.org/10.1109/IC2E.2014.41.

[7] Christoph Fehling, Frank Leymann, Ralph Retter, Walter Schupeck, and Peter Arbitter. *Cloud Computing Patterns - Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, 2014. ISBN 978-3-7091-1567-1. doi: 10.1007/978-3-7091-1568-8. URL http://dx.doi.org/10.1007/978-3-7091-1568-8.

[8] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*, 2011. URL https://www.usenix.org/conference/nsdi11/mesos-platform-fine-grained-resource-sharing-data-center.

[9] Shigeru Hosono. A devops framework to shorten delivery time for cloud applications. *IJCSE*, 7(4):329–344, 2012. doi: 10.1504/IJCSE.2012.049753. URL http://dx.doi.org/10.1504/IJCSE.2012.049753.

[10] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. A Martin Fowler Signature

Book. Addison-Wesley, 2010. ISBN 9780321601919. URL https://books.google.com/books?id=9CAxmQEACAAJ.

[11] Watts S. Humphrey. *A discipline for software engineering*. Series in software engineering. Addison-Wesley, 1995. ISBN 978-0-201-54610-1.

[12] S. Jarzombek. Proceedings of the joint aerospace weapons systems support, sensors and simulation symposium. *Government Printing Office Press, 1999*, pages 73–82, 1999.

[13] Hui Kang, Michael Le, and Shu Tao. Container and Microservice Driven Design for Cloud Infrastructure DevOps. In *2016 IEEE International Conference on Cloud Engineering, IC2E 2016, Berlin, Germany, April 4-8, 2016*, pages 202–211, 2016. doi: 10.1109/IC2E.2016.26. URL http://dx.doi.org/10.1109/IC2E.2016.26.

[14] G. Kim, K. Behr, and K. Spafford. *The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win*. IT Revolution Press, 2014. ISBN 9780988262584. URL https://books.google.com/books?id=qaRODgAAQBAJ. The 3 ways.

[15] Zhanibek Kozhirbayev and Richard O. Sinnott. A performance comparison of Container-based technologies for the Cloud. *Future Generation Comp. Syst.*, 68: 175–182, 2017. doi: 10.1016/j.future.2016.08.025. URL http://dx.doi.org/10.1016/j.future.2016.08.025.

[16] Nane Kratzke and René Peinl. ClouNS - a Cloud-Native Application Reference Model for Enterprise Architects. In *20th IEEE International Enterprise Distributed Object Computing Workshop, EDOC Workshops 2016, Vienna, Austria, September 5-9, 2016*, pages 1–10, 2016. doi: 10.1109/EDOCW.2016.7584353. URL http://dx.doi.org/10.1109/EDOCW.2016.7584353.

[17] Peter M. Mell and Timothy Grance. SP 800-145. The NIST Definition of Cloud Computing. Technical report, Gaithersburg, MD, United States, 2011.

[18] K. Morris. *Infrastructure as Code: Managing Servers in the Cloud*. O'Reilly Media, 2016. ISBN 9781491924389. URL https://books.google.com/books?id=4IdRDAAAQBAJ.

[19] Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.*, pages 305–319, 2014. URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

[20] W. W. Royce. Managing the development of large software systems: Concepts and techniques. In *Proceedings, 9th International Conference on Software Engineering, Monterey, California, USA, March 30 - April 2, 1987.*, pages 328–339, 1987. URL http://dl.acm.org/citation.cfm?id=41801.

[21] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, pages 351–364, 2013. doi: 10.1145/2465351.2465386. URL http://doi.acm.org/10.1145/2465351.2465386.

[22] J. Shore. *The Art of Agile Development: Pragmatic Guide to Agile Software Development*. Theory in Practice. O'Reilly Media, 2007. ISBN 9780596519353. URL https://books.google.de/books?id=g4BoxSensnoC.

[23] Aleksander Slominski, Vinod Muthusamy, and Rania Khalaf. Building a Multi-tenant Cloud Service from Legacy Code with Docker Containers. In *2015 IEEE International Conference on Cloud Engineering, IC2E 2015, Tempe, AZ, USA, March 9-13, 2015*, pages 394–396, 2015. doi: 10.1109/IC2E.2015.66. URL http://dx.doi.org/10.1109/IC2E.2015.66.

[24] Matt Stine. Migrating to cloud-native application architectures. 2015. URL https://download3.vmware.com/vmworld/2015/downloads/oreilly-cloud-native-archx.pdf.

[25] J. Turnbull. *The Docker Book: Containerization is the new Virtualization*. James Turnbull, 2014. ISBN 9780988820203. URL https://books.google.com/books?id=4xQKBAAAQBAJ.

[26] M. Virmani. Understanding DevOps bridging the gap from continuous integration to continuous delivery. In *Fifth International Conference on the Innovative Computing Technology (INTECH 2015)*, pages 78–82, May 2015. doi: 10.1109/INTECH.2015.7173368.

[27] Miguel G. Xavier, Israel C. De Oliveira, Fabio D. Rossi, Robson D. Dos Passos, Kassiano J. Matteussi, and César A. F. De Rose. A performance isolation analysis of disk-intensive workloads on container-based clouds. In *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2015, Turku, Finland, March 4-6, 2015*, pages 253–260, 2015. doi: 10.1109/PDP.2015.67. URL http://dx.doi.org/10.1109/PDP.2015.67.

[28] Zhongxiang Xiao, Inji Wijegunaratne, and Xinjian Qiang. Reflections on SOA and Microservices. In *4th International Conference on Enterprise Systems, ES 2016, Melbourne, Australia, November 2-3, 2016*, pages 60–67, 2016. doi: 10.1109/ES.2016.14. URL http://dx.doi.org/10.1109/ES.2016.14.

[29] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010*, pages 265–278, 2010. doi: 10.1145/1755913.1755940. URL http://doi.acm.org/10.1145/1755913.1755940.

A Configuration Files

A.0.1 Continuous integration

```
etl:
  build: .
  command: envconsul-config  env-consul-test.hcl nosetests-2.7 -v
                                    importer_test/
  links:
    - mongo

mongo:
  image: mongo:3.0.9
  ports:
    - "27017:27017"
  command: --smallfiles
```

Listing 2: Docker-Compose

```
FROM ruby:2.2.5

RUN apt-get update -y && apt-get upgrade -y
    && apt-get clean
    && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*

RUN mkdir -p /root/.

ADD id_docker /root/.ssh/id_rsa

ADD envconsul /usr/local/bin/envconsul

RUN chmod 600 /root/.ssh/id_rsa

RUN echo "Host github.com\n\tStrictHostKeyChecking no\n" >>
                                /root/.ssh/config
```

Listing 3: Base Ruby Docker Image

```
consul = "consul-api.betterdr.net:8500"

token = "634das184ssdf67f-fb213219a-d2cc-97ee-c6dasdc8bb49c9123123a8"

# max_stale = "10m"
timeout = "5s"
retry = "10s"
sanitize = true
splay = "5s"
kill_signal = "SIGUSR2"

vault {
  address = "https://vault.betterdr.net"
  token   = "cdfdsfds2gfefef-3a7d-ewrer7fa1-cf3462-43c6af56991a"
  renew   = true

  ssl {
    enabled = true
    verify  = true
  }
}

prefix {
  path = "env/etl/stage"
}

secret {
  path      = "secret/etl/stage"
  no_prefix = true
}
```

Listing 4: env-consul-test.hcl

```

FROM quay.io/betterdoctor/ruby:2.2.5

WORKDIR /tmp

COPY Gemfile* /tmp/

RUN bundle install

COPY . /app

WORKDIR /app
```

Listing 5: Application Dockerfile

A.0.2 Marathon Configuration file

```
{
  "id" : "etl-production",

  "apps":[
    {
      "id": "web",
      "instances": 1,
      "cpus": 1.0,
      "mem": 1024,
      "uris": [
        "https://s3.amazonaws.com/betterdoctor-operations-
    qhtumyvauxvxorwmeujn/Configs/docker.tar.gz"
      ],
      "container": {
        "type": "DOCKER",
        "docker": {
          "image": "quay.io/betterdoctor/etl:release-1.43.4",
          "network": "BRIDGE",
          "forcePullImage": true,
          "portMappings": [
            {
              "containerPort": 3000,
              "hostPort": 0
            }
          ]
        }
      },
      "env": {
        "environment": "production",
        "instance_type": "web",
        "RAILS_ENV": "production",
        "SERVICE_NAME": "etl-production",
        "SERVICE_TAGS": "etl,production,web"
      },
      "healthChecks": [{
        "path": "/",
        "portIndex": 0,
        "protocol": "HTTP",
        "gracePeriodSeconds": 300,
        "intervalSeconds": 30,
        "timeoutSeconds": 5,
        "maxConsecutiveFailures": 3,
        "ignoreHttp1xx": false
      }],
      "labels": {
        "HAPROXY_GROUP": "external",
        "HAPROXY_0_VHOST": "etl.betterdr.net",
        "HAPROXY_0_BACKEND_HTTP_OPTIONS": "  acl is_proxy_https hdr(X-Forwarded-Proto) https\n  redirect scheme https unless { ssl_fc } or is_proxy_https\n"
      }
    },

    {
      "id": "worker",
      "instances": 1,
      "cpus": 1.0,
      "mem": 512,
      "cmd": "supervisord -c /etc/supervisor/conf.d/supervisord-worker.conf",
      "uris": [
        "https://s3.amazonaws.com/betterdoctor-operations-qhtumyvauxvxorwmeujn/Configs/docker.tar.gz"
      ],
      "container": {
        "type": "DOCKER",
        "docker": {
          "image": "quay.io/betterdoctor/etl:release-1.43.4",
          "network": "BRIDGE",
          "forcePullImage": true
        }
      },
      "env": {
        "environment": "production",
          "instance_type": "worker",
          "RAILS_ENV": "production"
        }
      }
    ]
}
```

Listing 6: marathon JSON

A.0.3 Terraform IAC template

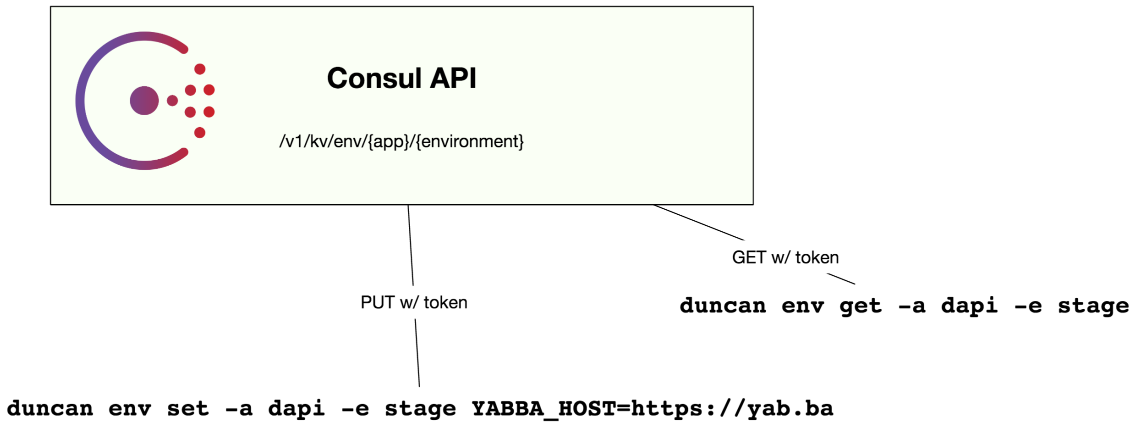
```

- "name": "filebeat.service"
    "command": "start"
    "enable": !!bool |-
      true
    "content": |
      [Unit]
      Description=ELK: Filebeat collects logs and sends them to
    Logstash
      Requires=filebeat-download.service
      After=filebeat-download.service
      After=docker.service
      [Service]
      StandardOutput=journal+console
      StandardError=journal+console
      ExecStart=/opt/filebeat/filebeat -e -c /etc/filebeat/filebeat.yml
    -d "publish"
      [Install]
      WantedBy=multi-user.target
  - "name": "cadvisor.service"
    "command": "start"
    "enable": !!bool |-
      true
    "content": |
      [Unit]
      Description=Google Container Advisor
      Requires=docker.service
      After=docker.service
      [Service]
      Restart=always
      RestartSec=5
      StandardOutput=journal+console
      StandardError=journal+console
      ExecStartPre=-/usr/bin/docker rm cadvisor
      ExecStart=/usr/bin/docker run --rm --volume=/:/rootfs:ro --volume
    =/var/run:/var/run:rw --volume=/sys:/sys:ro \
          --volume=/var/lib/docker/:/var/lib/docker:ro --publish=8888:808
    0 --name=cadvisor google/cadvisor:latest \
          --docker_env_metadata_whitelist=environment,instance_type
      ExecStop=/usr/bin/docker stop cadvisor
      [Install]
      WantedBy=multi-user.target
  - "name": "newrelic-sysmond.service"
    "command": "start"
    "enable": !!bool |-
      true
    "content": |
      [Unit]
      Description=Newrelic Server Monitoring
      [Service]
      EnvironmentFile=/etc/environment
      TimeoutStartSec=20m
      Restart=always
      StandardOutput=journal+console
      StandardError=journal+console
      ExecStartPre=-/usr/bin/docker rm newrelic-sysmond
      ExecStart=/usr/bin/docker run --rm --name newrelic-sysmond -e
    NEW_RELIC_LICENSE_KEY=${NEW_RELIC_LICENSE_KEY} \
        -e HOSTNAME=mesos-$${COREOS_PRIVATE_IPV4} -v /sys/fs/cgroup:/
    sys/fs/cgroup:ro -v /var/run/docker.sock:/var/run/docker.sock quay.
    io/betterdoctor/newrelic-sysmond:2.3.0.132
      ExecStop=/usr/bin/docker stop newrelic-sysmond
      [Install]
      WantedBy=multi-user.target

```

Listing 7: Terraform user data file

```
resource "aws_launch_configuration" "Mesos-Production-
    MasterLaunchConfig" {
  name_prefix                 = "Mesos-Production-MasterLaunchConfig"
  image_id                    = "${lookup(var.coreos_ami, var.region)}"
  instance_type               = "${var.master_instance_type}"
  key_name                    = "${var.key_name}"
  security_groups             = ["${aws_security_group.Mesos-Production
  -AdminSecurityGroup.id}", "${aws_security_group.Mesos-Production-
    MasterSecurityGroup.id}"]
  associate_public_ip_address = false
  user_data                   = "${template_file.user_data_master.
    rendered}"
  iam_instance_profile        = "${aws_iam_instance_profile.Mesos-
    Production-MasterInstanceProfile.id}"

  ebs_block_device {
    device_name           = "/dev/xvdb"
    volume_type           = "gp2"
    volume_size           = 100
    delete_on_termination = true
  }

  ephemeral_block_device {
    device_name  = "/dev/sdb"
    virtual_name = "ephemeral0"
  }

  lifecycle {
    create_before_destroy = true
  }

```

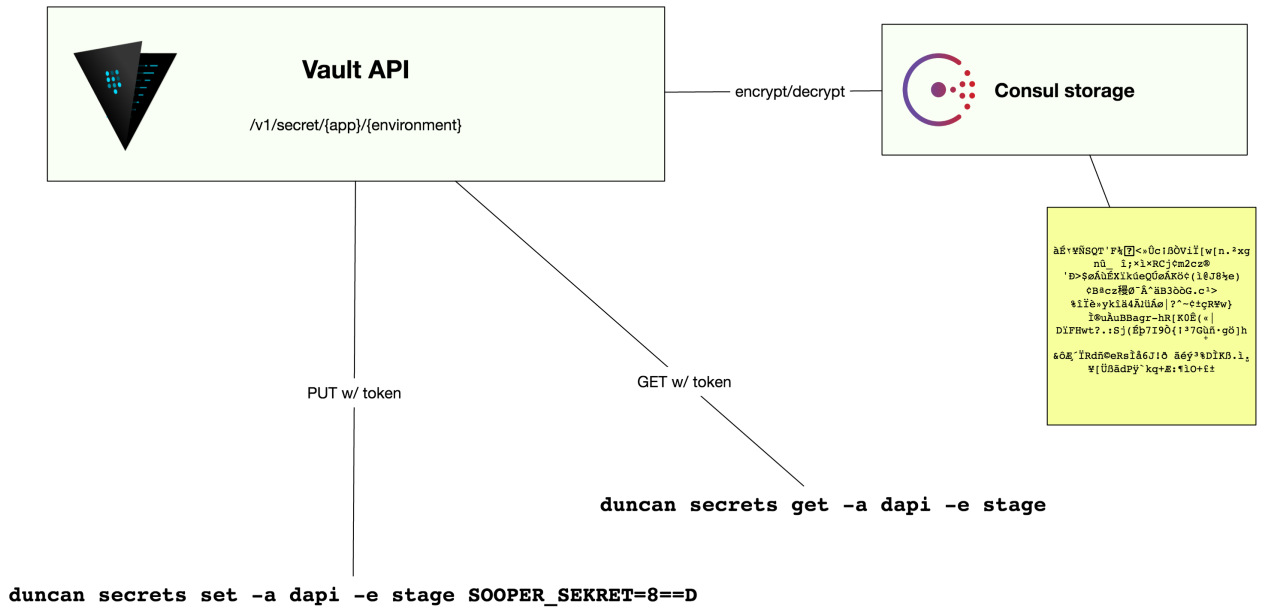
Listing 8: Terraform Amazon EC2 template file

B Miscellaneous



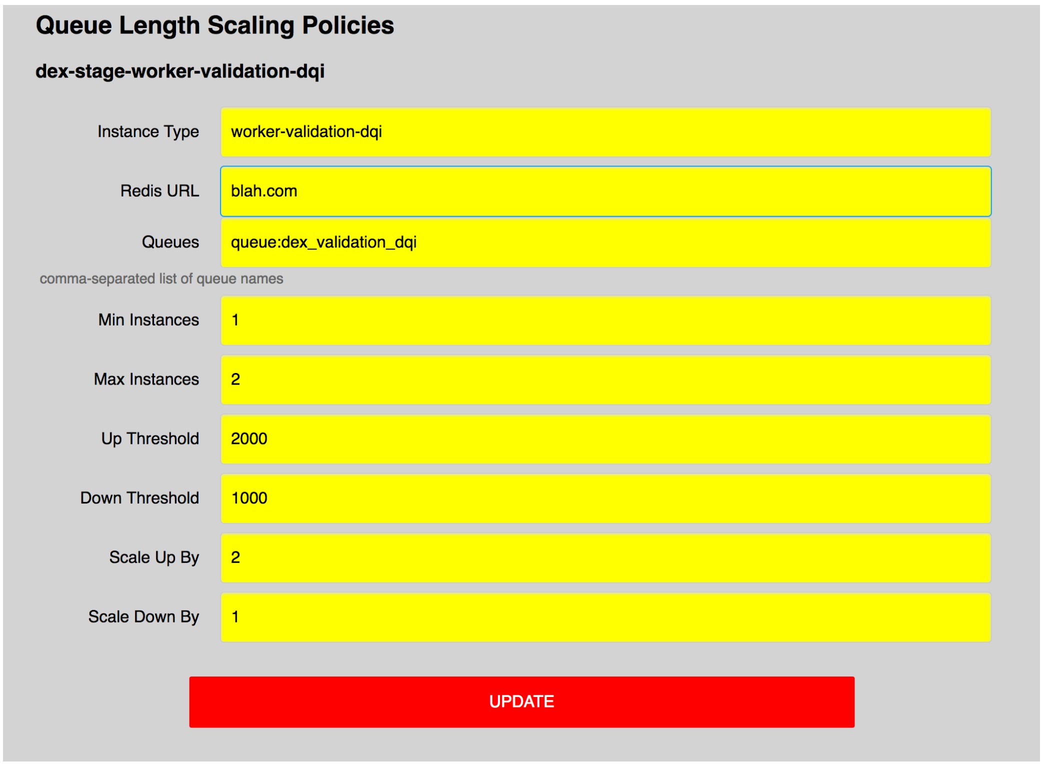
```
duncan env get -a dapi -e stage
duncan env set -a dapi -e stage YABBA_HOST=https://yab.ba
```

Figure B1: Duncan - get set environment variables from consul



```
duncan secrets get -a dapi -e stage
duncan secrets set -a dapi -e stage SOOPER_SEKRET=8==D
```

Figure B2: Duncan - get set secrets to vault



Queue Length Scaling Policies

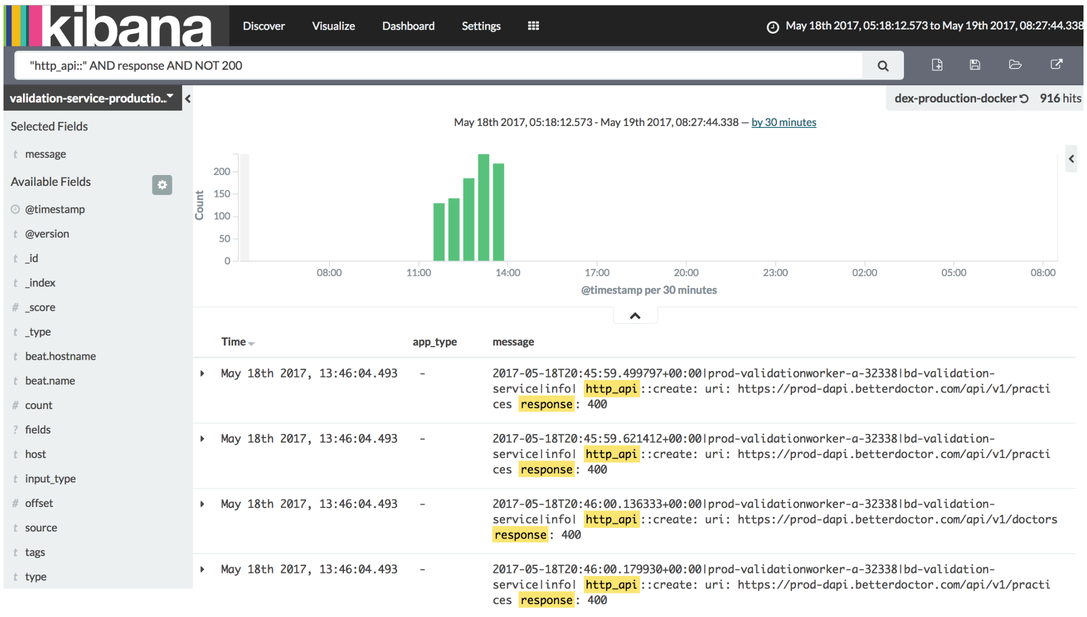
dex-stage-worker-validation-dqi

Instance Type	worker-validation-dqi
Redis URL	blah.com
Queues	queue:dex_validation_dqi

comma-separated list of queue names

Min Instances	1
Max Instances	2
Up Threshold	2000
Down Threshold	1000
Scale Up By	2
Scale Down By	1

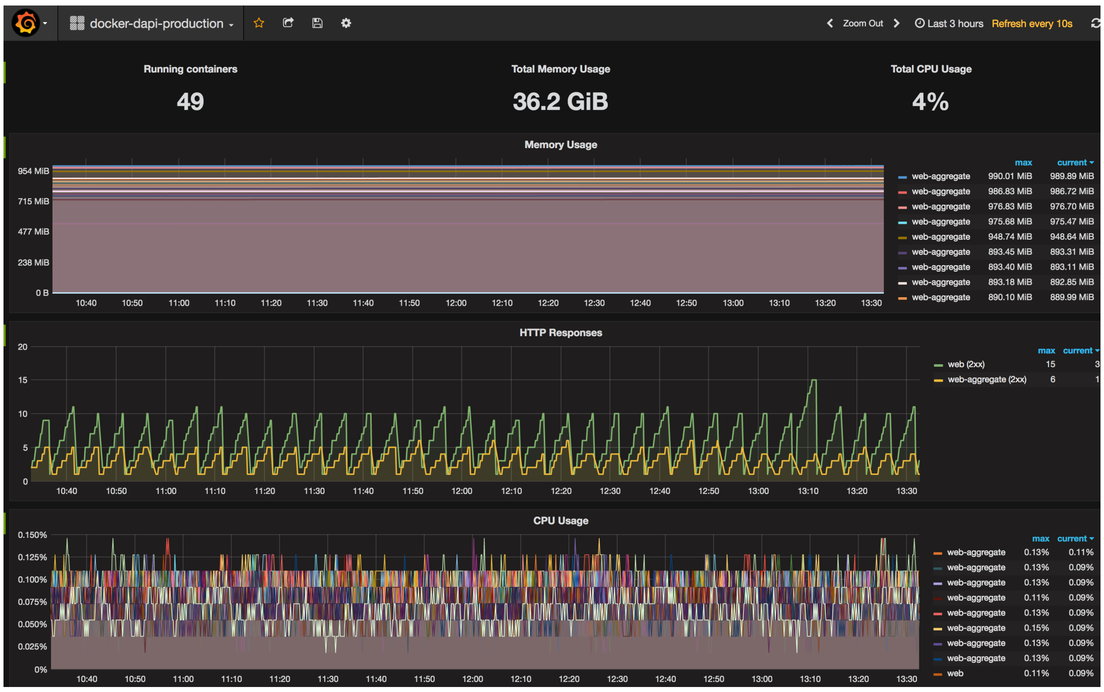
Figure B3: Slythe - Autoscaling based on queue length



"http_api::" AND response AND NOT 200

```
Time                          app_type   message
May 18th 2017, 13:46:04.493   -          2017-05-18T20:45:59.499797+00:00|prod-validationworker-a-32338|bd-validation-
                                         service|info| http_api::create: uri: https://prod-dapi.betterdoctor.com/api/v1/practi
                                         ces response: 400
May 18th 2017, 13:46:04.493   -          2017-05-18T20:45:59.621412+00:00|prod-validationworker-a-32338|bd-validation-
                                         service|info| http_api::create: uri: https://prod-dapi.betterdoctor.com/api/v1/practi
                                         ces response: 400
May 18th 2017, 13:46:04.493   -          2017-05-18T20:46:00.136333+00:00|prod-validationworker-a-32338|bd-validation-
                                         service|info| http_api::create: uri: https://prod-dapi.betterdoctor.com/api/v1/doctors
                                         response: 400
May 18th 2017, 13:46:04.493   -          2017-05-18T20:46:00.179930+00:00|prod-validationworker-a-32338|bd-validation-
                                         service|info| http_api::create: uri: https://prod-dapi.betterdoctor.com/api/v1/practi
                                         ces response: 400
```

Figure B4: Kibana Dashboard - Lucene Search Query



Running containers: 49

Total Memory Usage: 36.2 GiB

Total CPU Usage: 4%

Memory Usage

	max	current
web-aggregate	990.01 MiB	989.89 MiB
web-aggregate	986.83 MiB	986.72 MiB
web-aggregate	976.83 MiB	976.70 MiB
web-aggregate	975.68 MiB	975.47 MiB
web-aggregate	948.74 MiB	948.64 MiB
web-aggregate	893.45 MiB	893.31 MiB
web-aggregate	893.40 MiB	893.11 MiB
web-aggregate	893.18 MiB	892.85 MiB
web-aggregate	890.10 MiB	889.99 MiB

HTTP Responses

	max	current
web (2xx)	15	3
web-aggregate (2xx)	6	1

CPU Usage

	max	current
web-aggregate	0.13%	0.11%
web-aggregate	0.13%	0.09%
web-aggregate	0.13%	0.09%
web-aggregate	0.11%	0.09%
web-aggregate	0.13%	0.09%
web-aggregate	0.15%	0.09%
web-aggregate	0.13%	0.09%
web-aggregate	0.13%	0.09%
web	0.11%	0.09%

Figure B5: Grafana Dashboard Showing Container Metrics