Schlumberger

PUTC

Kubernetes: From Beginner to Expert*

01. Introduction and Motivation

Ragavan Natarajan
rnatarajan7@slb.com

16 May, 2018

1 Introduction

Kubernetes was developed at Google incorporating some of the lessons they learned from building and running two internal systems, namely Borg[1] and Omega[2], responsible for managing applications within their infrastructure over the last decade. Both these systems had some drawbacks[3], and drawing from these experiences, Google designed Kubernetes to be a system that would be simple to use by humans and machines alike, while having a stronger developer-centric[1] focus. Kubernetes was made open source[4] mid 2014, donated to CNCF[5] soon after, and has been widely adopted in the industry ever since.

In an attempt to live up to the promise of this course, of making Kubernetes understood by entry level software engineers and non software engineers alike, this introduction breaks the norm of attempting to define Kubernetes upfront in an abstract fashion. A quick search in the Internet would reveal its definition, which is already a gobbledygook to most people. In fact, in this course its definition is deferred until the third lecture, by which time there is enough foundation laid for a wider audience to comprehend it.

Later parts of this series would visit various concepts of Kubernetes in detail, but for now, the following sections would focus on discussing various practical problems a software engineer would face in developing software applications on the cloud. The following section introduces two broad categories of applications, and visits the problems in detail in each of these categories.

[1] In the later part of this course we will see what developer-centric focus means.

2 Demands of Cloud Applications

In this section we will study the anatomy of user-facing, and back-end applications running on the cloud, and explore the challenging demands these applications place on the infrastructure, fuelled by the underlying business' needs. We will also discuss a variety of problems that software engineers[2] have to find a solution to, in order to overcome these challenges.

Section 2.1 discusses the problems faced in developing, deploying, and maintaining a web application and its necessary infrastructure in the context of an organization with a rapidly growing Internet-facing business. The subsequent section visits a class of problems that emerge often as consequence of the organization's growth, which places increasing demands on the capabilities of its applications, their underlying infrastructure, and engineering resources.

2.1 User-facing web applications

2.1.1 A naïve web deployment

Figure 1 shows a naïve deployment of a web application in a company. There is a single machine running a web server *myserver.com* serving a web application *myapp*. This machine also performs the additional effort of hosting the database, which the web application depends on.

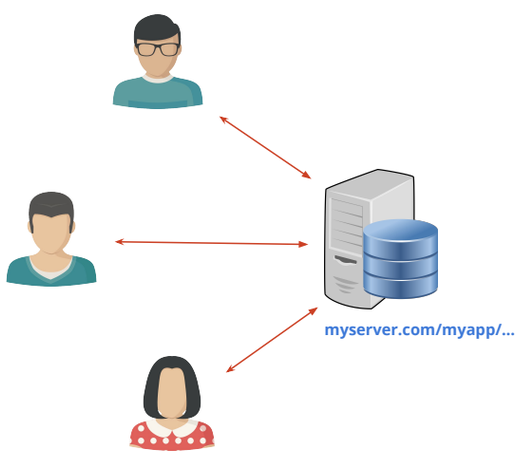

Figure 1: Anatomy of a naïve web deployment

2.1.1.1 Problems with the architecture

This architecture is bad for a lot of reasons, some of which should be obvious to anyone with moderate software development experience. Still, a overwhelming number of websites are still deployed in this fashion, or only in a slightly better manner, and their administrators constantly try to battle some (or all) of the problems listed below.

[2]The term *software engineer* is broadly used in this course to refer to anyone who develops, deploys, and maintains software along with its necessary software infrastructure.

Single point of failure

The machine that runs the server is a single point of failure, and failures could happen due to a variety of reasons. Following are some of them:

- **Spike in traffic:** The endorsement of the website by a celebrity could drive excessive amout of traffic to the web server beyond what it could handle. This could cause the web server to crash, and worse - the webmaster may not be aware that it did, causing all the potential sales to be lost until the web server is brought back up.
- **Hardware failures:** Hardware failures could occur in the machine due to electrical spike, for example. This is potentially much more expensive to repair because this server also runs the database. If the hard disk crashed, all the data is potentially lost. There is no data redundancy or failover in this architecture.
- **OS failure:** The operating system on this server could have a critical bug in its latest update which causes it to reboot for mysterious reasons. The web server goes down during that time. Similarly, planned upgrade to the operating system could cause unprecedented downtimes.
- **Failed upgrades:** A planned upgrade to the web application could fail because a critical bug was not caught in the regression testing cycle. Or, even worse, a new feature introduced in the application could cause memory leaks causing the server to crash once in a while. Rolling back to previous working version of the application could prove to be tedious too, as there is no clear rollback mechanism.

Cloud compute instances are no exception to any of this. All of this could also happen to a VM on the cloud any time.

Inflexible compute resources

Initially the hardware for the server could have been chosen based on guesstimates on the application's compute requirements and traffic forecast. That would soon prove wrong when the marketing initiatives start to drive a lot of traffic to the website causing very high memory and CPU utilization beyond what the machine could handle.

Any attempt to address this by purchasing a more powerful machine[3] to run the application would only temporarily addresses the problem while opening the door to several new problems.

- **Limit to scaling:** The ability of a single server to handle 10k concurrent requests and beyond, commonly known as the **c10k**[6] problem, becomes an issue as the application grows in size. Use of *vertical scaling* to try and address this issue wouldn't help because a 2× scaling of compute resources does not correspond to a 2× increase in the number of concurrent connections a server could handle. For a thorough treatment of this subject, please refer to [7].
- **New hardware release:** When there is availability of new hardware promising good return on investment, upgrading the machine to that new hardware becomes more painful than it ought to be. The server needs to be prepared, the application deployed, and various testing performed on it before the new machine could start servicing clients.

  Again, one might be tempted to think that they would be able to get around this problem in a cloud computing environment by taking a snapshot of the VM, and by creating another VM with the snapshot with the desired hardware configuration. However, it is important to

[3] This method of purchasing a more powerful machine to meet the demands of an application is called *Vertical Scaling*.

note that new hardware might require completely different set of device drivers and libraries to be installed from what was there in the snapshot.

Seasonality

- **Overutilization:** During certain times of the year such as during the months when the customer purchase behaviour is very high, the new powerful machine too runs out of resources (CPU, memory, and network bandwidth).
- **Underutilization:** On the other hand, for most part of the year when the customer purchase behaviour is generally low, the resources of this powerful machine could be very much underutilized. In business terms, this machine does not fetch good return on investment for most part of the year.

Contention for resources

In this architecture, the machine hosting the web server is also loaded with the additional responsibility of hosting the database, and potentially other web applications too. It causes the compute resources on the machine to be shared with the resource hungry DBMS. This leads to starvation of resources for all the applications including the DBMS during peak traffic loads.

Security

Any vulnerability in your application means that the attacker potentially has access to all the other applications running on the machine, and could cause irreparable damage to the system or to the reputation of the organization.

Incompatible software needs

It is not uncommon for two software installed on a machine to have conflicting library requirements. In the architecture presented here, if the DBMS were to expect specific versions of system libraries incompatible with what the web server's needs are, then we have a problem.

Dedicated manpower

The highly failure prone nature of this architecture means that there is need for dedicated manpower to keep constant watch on the system, and to take corrective actions when it fails. This could become very frustrating for the personnel involved, as there is no incentive for them in terms of skills development to keep doing the same thing over and over again.

Environment mismatches

The application that worked well in the developer's environment could fail in the production environment for a variety of reasons.

- **Library mismatch** The developer's workstation might run an operating system whose system library versions are different from the one run on the production server. This could cause strange issues impossible to detect during the development phase.
- **Environment mismatch** The environment variables in the developer's workstation which the application depends on may have different values on the production server. The developer may have no control over the production server's environment variables lest it impacts other applications running on the server.
- **OS mismatch** The machine that runs the production server may run an entirely different operating system where everything from library path to software versions are different from what the developer has in their workstation.

Down time between upgrades

Any hardware or operating system upgrades on the machine running the server, or upgrades to the web server itself, causes down time.

Lack of fine-grained telemetry

Monitoring the resource consumption of the web application is non-trivial in this architecture, as the information from the monitoring tools needs to be demultiplexed in order to get the resource utilization at an application level. This involves a lot of specifics and is not always easy to do.

2.1.1.2 Summary

This section presented a naïve architecture for web deployment, and discussed in detail its various problem areas. However, this is not an exhaustive list, and there are potentially more problems with this architecture. The following section presents an apparent improvement to it, and evaluates it on the same grounds.

2.1.2 A thoughtless improvement

Figure 2 shows an architecture that aims to address some of these problems. There are two key improvements in this architecture.

1. Instead of the web application running on only one machine, now identical copies of the same web application run on multiple machines[4]
2. The database is no longer a part of any of the machines that run the web server, but is instead moved to a separate layer, and is also potentially deployed in a highly-available fashion.

The logical boundary shown by the green hexagon in figure 2 contains the machines and software components that together make the application *myapp*. In this architecture nginx[8][5] (pronounced engine-X) is configured as a reverse-proxy and load-balancer that lies outside of the logical boundary of the application.

[4]The use of multiple machines in some distributed fashion to scale an application is referred to as *Horizontal Scaling*. It does not stipulate however that the machines run identical copies of an application.

[5]Nginx is shown in this architecture purely for the purposes of simplicity of demonstration. There are better alternatives to Nginx as of this writing, capable of doing much more than reverse-proxying and load-balancing. The later parts of this course would visit some of those.

Unlike before where the client requests directly hit the server, now the client requests reach the reverse-proxy. Listing 1 shows a sample configuration of this reverse-proxy.

The nginx server listens on port 80 for any requests to myapp, and routes it to one of the pool of machines described in the `upstream` configuration `myappservers`. The routing strategy could be *round-robin*, for example. This configuration also allows mixing heterogeneous compute instances, by specifying in the `weight` parameter that a particular instance could take more workload. In this example, the server at IP `10.1.2.74` is configured to take 8× more workload. Similarly, when a server goes down for maintenance, for example, it is possible to mark its state in the configuration as `down`, which prevents nginx from routing requests to that server. Optionally, it is also possible to specify that a server should be automatically marked as *failure* upon n unsuccessful routing attempts, and that routing attempt to that server should resume after t seconds, as this example shows.

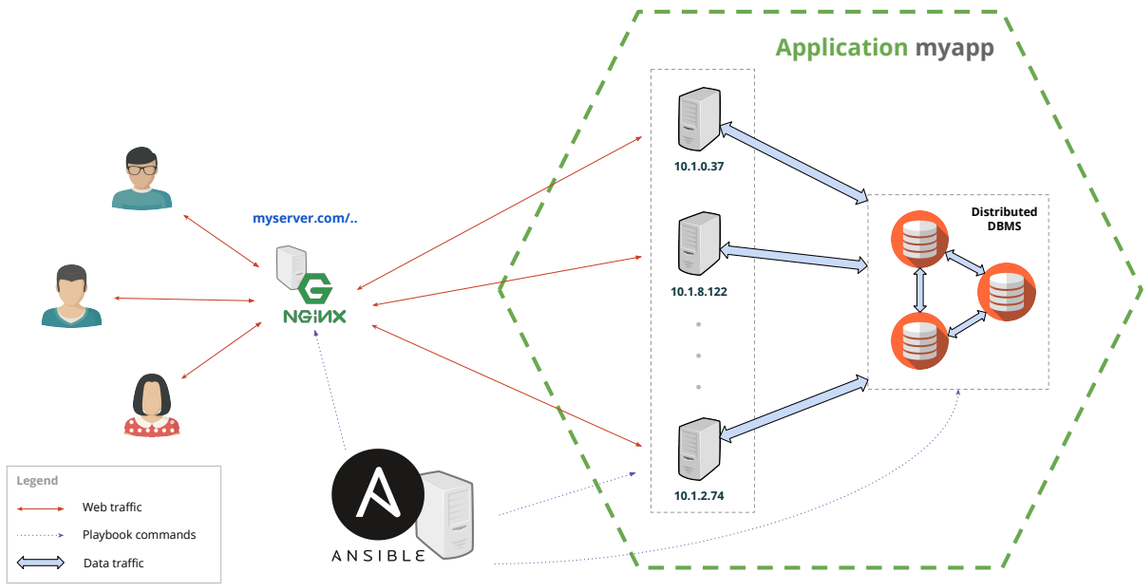

Figure 2: A slightly better architecture

```
http {
    upstream myappservers {
        server 10.1.0.37:80   max_fails=5  fail_timeout=30s;
        server 10.1.8.122:80  max_fails=5  fail_timeout=30s weight=2;
        server 10.1.2.74:80   max_fails=5  fail_timeout=30s weight=8 down;
        ...
    }

    server {
        server_name www.myserver.com;
        listen 80;

        location /myapp {
            proxy_pass http://myappservers;
        }
    }
}
```

Listing 1: nginx.conf

Deployment and Configuration management

The use of multiple machines to have identical applications running on them poses new challenges. When a machine goes down, due to hardware failure, for example, another machine that is brought as a replacement should have the set-up done in identical fashion as required by the application. Until a few years ago this would be achieved by means of shell scripts that are often executed remotely to set the target machine up.

This state-machine like approach to solving this problem has numerous drawbacks, the biggest of which are lack of parallelism and idempotency. For example, if the shell-script had a command to add an environment variable to the file `/etc/profile`, and if it was run multiple times due to failures in the rest of the script, how many times the environment variable gets added to that file on the target machine? Idempotency concerns such questions. What if the set of target machines each run different flavour of Linux, each having its own package manager different from the other?

Deployment automation and configuration management tools such as Ansible[9], Chef[10], etc., emerged to solve this problem. These tools view deployment and configuration as a desired state of the target machine described by means of simple structured files such as YAML, or by means of a DSL (Domain-Specific Language). They would then ensure that the target machine's current state matches the described desired state and would run in a loop to ensure that it always stays that way. This approach naturally enables parallelism, idempotency, and so on.

In Ansible such configuration files are referred to as *Playbooks*. Engineers (titled DevOps, who are specifically hired to maintain a healthy state of the systems from development, to staging, to production) typically write and maintain these playbooks, which are also often source-controlled, having their own versioning and life cycle management.

In the following sections we will see the set of problems that are addressed by this improvement, and other set of problems that are left unaddressed, or even introduced by this architecture.

2.1.2.1 Problems addressed by this improvement

✓ Single point of failure

Although it is obvious from the figure that nginx could become a single point of failure, it is typically configured in an $N+1$ fashion offering failover, with optional network layer load-balancer on the front. Similarly Ansible is also configured in a highly-available fashion. These are not shown here for they would cause the reader's attention to digress from the core topic.

Assuming that nginx is configured with failover, this architecture addresses the issue of single point of failure. There are multiple machines each possibly in a different subnet, or even in different geographical location. Any failure in any of the machines could be addressed as described before, without it having a strong impact on the application as a whole.

✓ Down time between upgrades

The upgrades could now happen one machine at a time, or one set of machines at a time, while the other ones still continue to service the client requests. Previously this was not possible.

✓ **Inflexible compute resources**

If a new hardware was available, the operations engineer would be able to seamlessly integrate that into the system by means of automated configuration and deployment tools, without affecting the overall availability of the system.

2.1.2.2 Problems unaddressed by this improvement

✗ **Seasonality**

It might seem on the surface that resource over-utilization and under-utilization due to seasonal traffic is also addressed by this architecture, since it seems to be possible to bring more powerful machines and tear them down later on. However that is not really the case. Even during high traffic seasons, there could be powerful machines in this pool that are not fully utilized. Not only that, the low-power compute machines may be overloaded and become unable to service client requests in full throttle.

Also, the task of bringing up additional and more powerful machines, adding them to the pool for automated deployments, and later tearing them down, is non-trivial. As often the high- and low-traffic situation occurs, so often the pool needs to be expanded and shrunk, and with this architecture there is no easier way to achieve this. This is not ideal. Attempts to address this problem by means of deployment automation tools is an anti-pattern, and it should be strictly avoided.

✗ **Dedicated manpower**

In this architecture, the need for dedicated manpower to administer the infrastructure becomes even stronger. The newly employed personnel, typically titled *DevOps* must have a strong understanding of the operational semantics of various systems, including sound knowledge of configuration management tools, underlying network infrastructure, and so on.

✗ **Machines vs. Applications**

This architecture views the infrastructure as a set of machines, as opposed to a set of applications, meaning that any machine inside of that green hexagon in figure 2 is seen as belonging to the application myapp. The problem with this approach is that, even if any machine in that pool had enough resources to run other applications, the deployment strategy, lack of resource isolation, and scaling mechanisms of this architecture would prevent the possibility of achieving that.

✗ **The same old problems**

It should now be easier to reason why this apparent improvement to the architecture leaves many of the problems described earlier unaddressed.

- Resource isolation
- Security
- Incompatible software needs

- Environment mismatches
- Lack of fine-grained telemetry

2.1.2.3 Summary

So far we looked at a simple architecture and the issues associated with it in section 2.1.1 and a thoughtless improvement to it in section 2.1.2. We learned that not only did the improvement not solve all the problems, but also introduced some more problems.

In the following section we will step back to see the bigger picture, where an user-facing web application is merely another entity in the organization's larger infrastructure, which consists of other powerful entities such as distributed data processing and ML frameworks whose capabilities pose new challenges on the organization's infrastructure and other applications as a whole.

2.2 Explosion of back-end applications

In the context of an e-commerce website, for example, the user-facing web applications are capable of generating enormous volumes of data from user interactions such as mouse clicks and movements, duration of browsing, etc. Little over a decade ago most of this data would be discarded, as there was no cost-effective way of collecting, storing, and processing the data at scale in a timely manner, but that all changed with the emergence of open-source horizontally scalable Big Data processing frameworks such as Hadoop that run on commodity hardware.

Fast-forward, today we have an array of open-source data processing and machine learning frameworks, publish-subscribe and stream processing systems etc., all capable of scaling horizontally. This introduced the ability to solve problems that wouldn't have been cost-effective to solve previously. For example, today an e-commerce website's backbone would be able to analyze the user clicks and almost instantaneously provide relevant and appropriate recommendations back to the user, while running complex event processing and machine learning algorithms behind the scenes. Similarly there could be internal dashboards used by business analysts and other key decision makers, presenting data gathered and processed from various different endpoints in real time or near-real time.

This feedback loop like structure has strong impact on the infrastructure design philosophy of the organization as a whole. The following sections discuss in stages various challenges that are posed on the infrastructure as a result of this rapid evolution, and how some of these challenges used to be addressed.

2.2.1 The growth of back-end services

In this section we will discuss how the addition of few services on the back-end imposes more complexities on the infrastructure.

Figure 3 presents an architecture where the user-facing web application **myapp** now talks to multiple back-end services. Here, nothing changes about myapp – it continues to function in the same fashion of running from multiple machines with a reverse-proxy on the front, and with its own database layer in a highly-available fashion, the details of which are hidden inside the hexagon in this figure, in order to help us see the bigger picture.

Similarly each of the two services introduced here, namely *srvc1* and *srvc2* also have their own set of machines, owned by some engineering team developing it, and have their own database, the details of which are not shown in this this figure for the same reasons. As before, nginx load-balances the client requests to myapp, and deployment automation is achieved through Ansible configured in a highly-available fashion (not shown here for simplicity's sake).

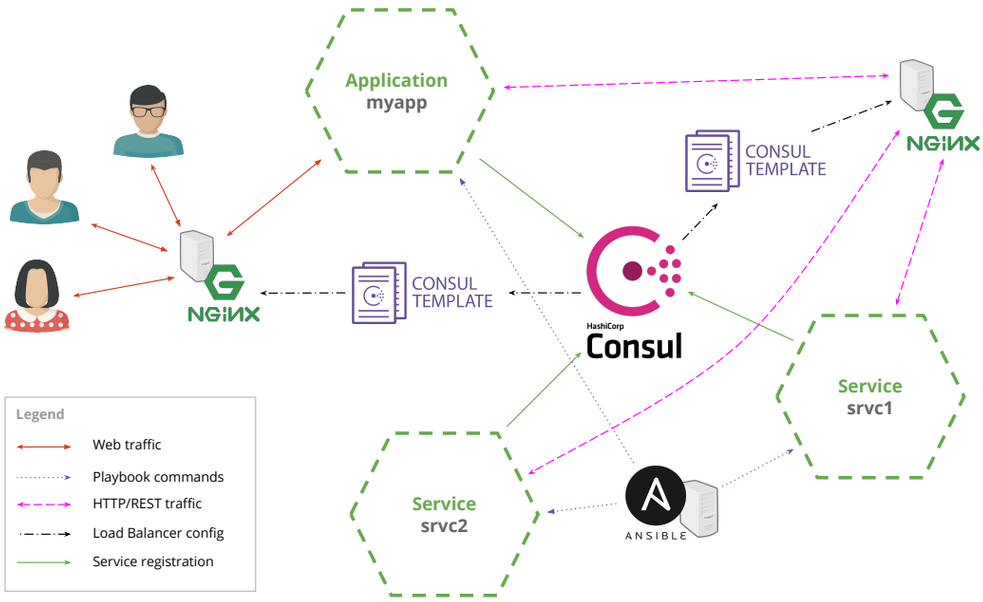

Figure 3: Complexity increases with more services

There are some differences, however. In this architecture, there is also another nginx deployment internally used by the services for load-balancing. Whereas the client-facing nginx server is exposed to the outside world, this internal nginx server is only used for load-balancing requests to internal services. There is the addition of two major components namely Consul, and Consul Template. The following section discusses discusses their purpose in the architecture in detail.

2.2.1.1 The problem of Service Discovery

There are two services in this architecture, each run from their set of own machines as described before, just like how myapp did in the architecture shown in figure 2.

Each machine that is a part of a network has a different IP address. When a machine goes down, naturally the service running on it goes down too. Unless someone told the load-balancer (either by marking the IP `down` in the nginx.conf file, or by removing the entry from it) that the machine went down, nginx has no way of knowing that, and might continue to route requests to that IP (although the routing itself would fail). When the machine gets replaced by another machine (with a possibly new IP address), and the service brought back up on it, an entry in nginx.conf has to be created with this new IP address for the service.

If there was no automated mechanism to add and remove mappings, the nginx.conf file would continue to have the dead machine's IP address mapped to this service. Worse things happen if some other machine started up and dynamically got the old machine's IP assigned to it. If this machine happens to run a totally different service on the same port, nginx would route the requests for the old service to this machine running a totally different service, due to the stale IP address mapping still present in its nginx.conf file.

Manually adding and removing entries in the conf file isn't practical, and is highly error prone even for small set of services. This problem is called **Service Discovery**.

Consul

Tools like Consul[11] help solve this problem[6]. There are multiple ways by which Consul gets to know about the running services, of which one is described below. It is out of the scope for this course to discuss all the various ways of implementing service discovery.

Every machine that runs a service also runs what is called a **Consul Agent**, which runs as a daemon (a background process), and accepts *service definitions* from one or more services running on that machine.

```
{
  "service": {
    "name": "srvc1",
    "tags": ["production"],
    "address": "",
    "meta": {
      "meta": "Production service"
    }
    "port": 7128,
    "checks": [
      {
        "id": "health_check",
        "name": "Service health check",
        "http": "http://localhost:7128/healthz",
        "interval": "10s",
        "timeout": "1s"
      }
    ]
  }
}
```

Listing 2: Service Definition

Listing 2 shows an example of a service definition, which says, there is a service by the name `srvc1` running in `production` environment on port `7128` on the same machine as the agent (i.e., `localhost`, hence the `address` field is left blank). It also instructs the Consul Agent to run health check by calling the specified URL every 10 seconds, and to declare the service as *failure* on a timeout of 1 second.

It is the responsibility of the Consul Agent to run the health check periodically, in addition to its primary responsibility of communicating the presence or absence of this service to a Consul Server, which is typically deployed in a distributed fashion spanning data centers. The internal details of how that happens is beyond the scope of this discussion.

Consul Template[12] is a standalone application that periodically queries the Consul Server and automatically updates any changes in the services to nginx.conf, and requests nginx to reload the configuration.

[6] Consul offers much more than service discovery, however. For more details please refer to its website

The drawbacks of this approach

On the surface it seems that the presence of a service discovery framework like Consul seems to solve most of the problems. Sure, it does – the effort of manually writing and maintaining nginx.conf is no longer required, thanks to its companion Consul Template. But it also introduces some problems.

1. **Agent reload** Each service is now required to provide a service definition along with it whenever it gets deployed. The service definition is required to be placed in a specific location on the machine, and the Consul Agent restarted to reload the service definition. If the agent crashed then there is no availability of a self-healing process to bring it back up, and the server would mark the node as critical, or remove its membership, although the service itself may still be running.
2. **Manual effort** It a service was found to malfunction, and requires debugging, this set up requires that the service definition be edited, to say, remove the tag *production* from it and the agent reloaded, so that it would not be listed under the available services under production. This is too much manual effort even for this small group of services, and is painful to scale. Engineers try to get around this by means of using deployment automation tools, but that is another moving part to deal with.
3. **Too many moving parts** Perhaps unwittingly too many moving parts were introduced to solve the problem of service discovery. There is Consul Server (deployed in a quorum of size > 3), Consul Agents (one per machine), service definition files (one per each service per machine), different versions of service definition files (for production, for staging, and so on), Consul Template, and even perhaps more deployment automation scripts (and their source control and versioning).
4. **Entropy** These moving parts also mean that any changes to the service definition takes time to propagate before a overall consistent state of the system is achieved, in an environment where each of these systems is possibly maintained perhaps by various teams. This attribute of a distributed system to increasingly get distorted over time is referred to as *entropy*. In a dynamically changing environment of distributed systems, this could result in unstable systems. For example, the time between a service going down and the overall system knowing about it, could cause failed routing requests, requests routed to wrong machines and wrong services, and so on.

This is not an exhaustive list of problems, nor did we see how other patterns of service registration work, and the trade-offs associated with those patterns. Nevertheless, it serves to illustrate some of the problems associated with service discovery, and helps us understand the need for a system that could achieve this in a simpler manner. The following section discusses an architecture where things could get even more complicated, not just with service discovery.

2.2.2 A Services Mesh (or mess?)

As an organization grows it typically attempts to tackle problems it (or, any other organization) never tackled before. As a result a diverse array of frameworks often gets created, or gets introduced into its infrastructure, often capable of solving a richer set of problems, while demanding radical approaches in how the infrastructure itself is designed around it.

Figure 4 shows an attempt to incorporate these diverse set of demanding applications into the same architecture discussed earlier. Now we have more hexagons representing a richer set of services and their own backbone. For example, there is a machine learning service that uses TensorFlow in some distributed fashion, there is Apache Spark which manages its own set of

machines for streaming and large scale data transformation needs, and there is Cassandra - a distributed row-store used here for storing time-series data, for example.

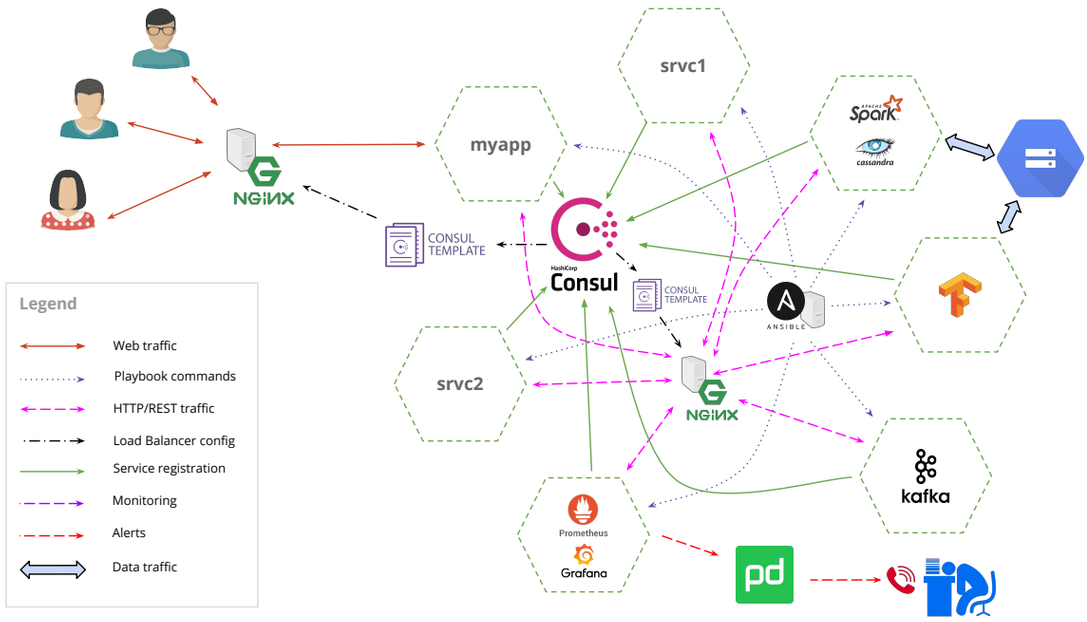

Figure 4: A services mesh (or mess?)

Unlike before, these new services cannot just be composed of a group of machines that all run identical copies of the same application. There is distributed computation, communication, and replication involved among the machines that run inside each of the hexagons running Apache Spark, Apache Cassandra, and TensorFlow on some kind of cluster manager.

Also, the failures on the machines inside these hexagons cannot be treated as how failures used to be treated in the former cases (like simply bringing another node up and deploying the service on it), as often a node that failed could be in the middle of a distributed computation that needs to be restarted upon a failure, and that requires the failed computations be reassigned to other available nodes and run from scratch. The services also involve complicated operational semantics that require people with specific skill sets to administer them.

For this architecture to even function properly, monitoring, alerting, and dedicated on-call staff become imperative. This architecture shows Prometheus and Grafana used together for monitoring, and PagerDuty for sending alerts to the on-call personnel. Organizations like Datadog and PagerDuty run successful businesses providing monitoring and alerting at scale.

Configuration management and deployment automation tools play a significant role in this architecture, and often the operations team expect more out of these tools – causing the very nature of these tools to evolve to become Turing-complete[7] in order to support the operational demands placed on these tools. And when something becomes Turing-complete it is not uncommon to see it abused like a full-blown programming language, but one without any unit-testing frameworks, community support, and so on – which naturally introduces more mess!

[7]Turing-completeness refers to the ability of a language to solve any computational problem – anything that a Turing machine can solve.

3 One infrastructure, various demands

Having discussed the various demands placed on the infrastructure and the engineering team in the context of an organization with rapidly growing business, this section takes a look at the desirable qualities in a system of infrastructure management.

3.1 What is desired?

Listed below is a set of desirable attributes in a system, some of which have been discussed in great depth in the previous sections.

- Resource isolation
- Ability to run low-latency jobs alongside batch jobs
- Ability to run parallel run-to-completion jobs
- Ability to support running fault-tolerant distributed processing frameworks
- Ability to support distributed persistence layers
- Auto-scaling
- Reliability, and Self-healing
- From machine-oriented to application-oriented
- Service Discovery
- Improved deployment reliability
- Quota management
- Composability
- Load balancing across application instances rather than machines
- Simple to learn (same API for people and machines)
- Ease of manual and automated system management
- Information by/for the applications
- Instance- and application level telemetry
- Security
- APIs around applications rather than machines
- Richer ecosystem and community support

This is not an exhaustive list, and we need a system that provides us most, if not, all of these qualities. After years of doing things the hard way, thanks to Google and the lessons they learned from running their infrastructure, and their willingness to share, we may have a system called *Kubernetes* capable of achieving this. Its wide acceptance, and the contributions it gets from the community has started to create a richer ecosystem around it, making it increasingly powerful. The remainder of this course would focus on how each of this is achieved in Kubernetes.

4 Summary

We started this chapter by *not* introducing Kubernetes in an attempt to make it well understood later, by first explaining the problems it was created to solve. We saw a naïve web application deployment and the various problems associated with it. We then discussed a thoughtless improvement to the architecture, and saw how it introduced more problems while leaving most of the problems that existed before, unaddressed.

We then moved on to discuss how the emergence of back-end services started to complicate the organization's infrastructure. We discussed a variety of tools in detail that needed to be introduced, in order to even run the infrastructure. Lastly we discussed how the emergence and incorporation of data processing and machine learning frameworks resulted in a mesh of services further complicating the infrastructure. We then concluded by taking a look at a set of desirable features in an infrastructure management system, and suggested that Kubernetes might well be one such system.

5 About the next chapter

The next chapter would take a brief tour of **containers** covering in-depth what is most important to know, in the context of their usage in Kubernetes. The topic of containers itself deserves a course on its own as containers have seen rapid evolution in the last 5 or 6 years, resulting in stronger demands for their standardization. Therefore, the next course would not just cover how to write Dockerfiles, but most importantly how *not to*. We will look at what problems containers solve, how they could be used, how they are abused, and how they would open the system to new security vulnerabilities when not used correctly, and how to run them securely, build them, package them, and so on. As before, the audience don't need to know anything about containers to attend the lecture.

References

[1] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, (Bordeaux, France), 2015.

[2] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *SIGOPS European Conference on Computer Systems (EuroSys)*, (Prague, Czech Republic), pp. 351–364, 2013.

[3] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *ACM Queue*, vol. 14, pp. 70–93, 2016.

[4] "Kuberentes on GitHub." https://github.com/kubernetes/kubernetes, accessed May 12, 2018.

[5] "Cloud Native Computing Foundation." www.cncf.io, accessed May 12, 2018.

[6] "The c10k problem." http://www.kegel.com/c10k.html, accessed May 12, 2018.

[7] R. D. Graham, "C10M – defending the Internet At Scale," in *ShmooCon IX*, 2013.

[8] "Nginx: High-performance Load Balancer, Web Server, and Reverse Proxy." https://www.nginx.com/, accessed May 12, 2018.

[9] "Ansible." www.ansible.com, accessed May 12, 2018.

[10] "Chef." www.chef.io, accessed May 12, 2018.

[11] "Service Discovery and Configuration with Consul." https://www.consul.io/, accessed May 12, 2018.

[12] "Consul Template." https://github.com/hashicorp/consul-template, accessed May 12, 2018.