Tampereen yliopisto

Eero Häihälä

OPERATING STATEFUL APPLICATIONS IN CONTAINER SPACE

BSc thesis
Faculty of Information Technology and Communication Sciences
Kari Systä
February 2021

ABSTRACT

Eero Häihälä: Operating stateful applications in container space
BSc Thesis
Tampere University
Degree Education in Computing and Electrical Engineering (TISKO)
February 2021

Containers are becoming more and more common in hosting scalable cloud applications due to their inherent efficiency benefits. However, this efficiency comes with caveats, especially when it comes to maintaining state.

This thesis will examine ways to work around these caveats and how containers are managed in the field. In addition to pulling from research papers, I will bring my experience from working as a Junior System's Specialist. Combining the research and the experience into guidelines on managing stateful applications when working with containers.

Keywords: Containerization, Kubernetes, Docker, Statefulness, Cloud, Orchestration

TABLE OF CONTENTS

TABLE OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

ACID	Atomicity, Consistency, Isolation and Durability, the properties of an ideal database
AWS	Amazon Web Services, a public cloud provider
Container	An isolated minimalistic collection of requirements for software
Docker	A read only container made with the docker pipeline
GCP	Google Cloud Platform, a public cloud provider
Guest OS	An operating system that resides within a virtual machine
Host OS	An operating system that hosts one or more virtual machines
Kubernetes or k8s	An orchestrator that manages fleets of containers
LXC	Linux Containers, a type of container
VM	Virtual Machine, a virtual environment that simulates a physical computer. Usually contains a guest OS.

1. INTRODUCTION

As more and more services are moved to the internet, the web hosting industry has become very important. The global web hosting services market size was valued at USD 56.7 billion in 2019, which to give perspective is more than the film industry, which was worth 42 billion USD in 2019 [1] [2]. The growth shows no signs of stopping and a forecast shows that in 2027 the industry will be worth USD 171.4 Billion [1].

Currently the leading technologies used to host these websites can be divided into virtual machines (VMs) and container technologies like Docker and LXC. Multiple containers together are called a fleet. Recently containers have overtaken the previously dominant VMs[3]. Containers are easier to manage in large scale operations, faster to start up and more resource efficient, but achieve these benefits through optimizations with caveats and those caveats need to be addressed.

One of the more significant caveats is that usually containers themselves don't maintain state. This becomes a problem with something called stateful applications that seize to function when they cannot maintain state. Maintaining state is most problematic when converting existing older software to a container or when the case is something inherently stateful such as a database. Which rely on files on a disk or memory to maintain data, neither of which is permanent in the container space by default.

Systems called orchestrators like Kubernetes have been built to manage fleets [4]. The goal of this BSc thesis is to shed light on different ways of working with containerized applications. This is done by reiterating research on the subject supported by an anecdotal information. As for the anecdotal side, I worked with virtual machines and containers as a Junior System's Specialist for nineteen months, during which I completed the Google Cloud Professional Cloud Architect -certification. I will apply some of what I learned in that program to supplement the thesis.

This BSc thesis will first examine what others have said on the subject and compile terminology used in the field. Essential concepts will be explained and examined through the lens of existing scientific publications. After that various common situations and pitfalls with stateless and stateful applications will be examined. Followed by a short description of the journey to certification and finally everything will be summed up. This BSc thesis will not provide clear and concrete instructions on how specific systems should be implemented for multiple reasons. The industry is constantly in flux and any practical advice will likely be outdated in a few years or sooner. Selecting a type of implementation is also extremely situational which complicates the process.

2. TERMINOLOGY AND PRACTICAL APPLICATIONS

It is important to discuss how the technology got to where it currently is. This chapter will examine various currently used technologies. As a brief synopsis on the industry, initially software was run on the bare metal on top of an operating system. However, due to complications like several systems running on the same machine requiring different versions of the same dependencies and interfering with each other along with an increase in the availability of cheap computing power, the method of choice changed. Virtual machines (VMs) running on a hypervisor became the new paradigm of choice. This insulated the environments from each other at the cost of performance. The newest solution is to use various kinds of containers, which aim to have high performance and the insulative properties of VMs.

While it is not commonplace yet, containers have slowly been invading the consumer space with projects like Flatpak, a way to distribute software as containers instead of binaries with dependencies [5]. And Pop!_OS, a Linux distribution that prioritizes the use of Flatpaks which are supplemented by ubuntu apt repositories [6]. These are important efforts in their own fields, but this BSc thesis will focus on hosted software like websites and various servers.

2.1 Containers and images

Docker is an open-source containerization engine, which automates the packaging, shipping, and deployment of any software applications that are presented as lightweight, portable, and self-sufficient containers, that will run virtually anywhere. [3] Instead of executing a regular binary in an isolated space, a container is an ephemeral running instance of a read only image in a monitored environment. This means that whenever the container is stopped, it loses any state information it had. There are ways around this but all of them have potential problems. The image the container is running is read-only. This means that the images can be stored on disk and do not take nearly as much space when several containers are running the same image since they do not mutate themselves. It also means that storing data within the container is not reliable since it could get terminated any minute wiping everything in the process. Docker containers operate through a monitored environment called the docker engine which is used to start,

stop, restart and interact in any other way with the containers. The engine runs as a daemon on the host machine and any interaction with the containers goes through the engine. [3] Containers share a kernel with the host system so for example a Linux image can only run on Linux. This drastically reduces the size of containers and speeds them up since each container does not have to run a virtual kernel on top of the host kernel. Docker is based on Linux containers (LXC), but they are not interchangeable [7], [8]. A docker container is running a simple application and does not support things like services or daemons whereas an LXC container can be thought of as almost a virtual machine. For future references to containers, assume docker containers or similar. Ultimately, the difference between a docker configuration and a typical VM setup with a hypervisor is best explained in figure 1.

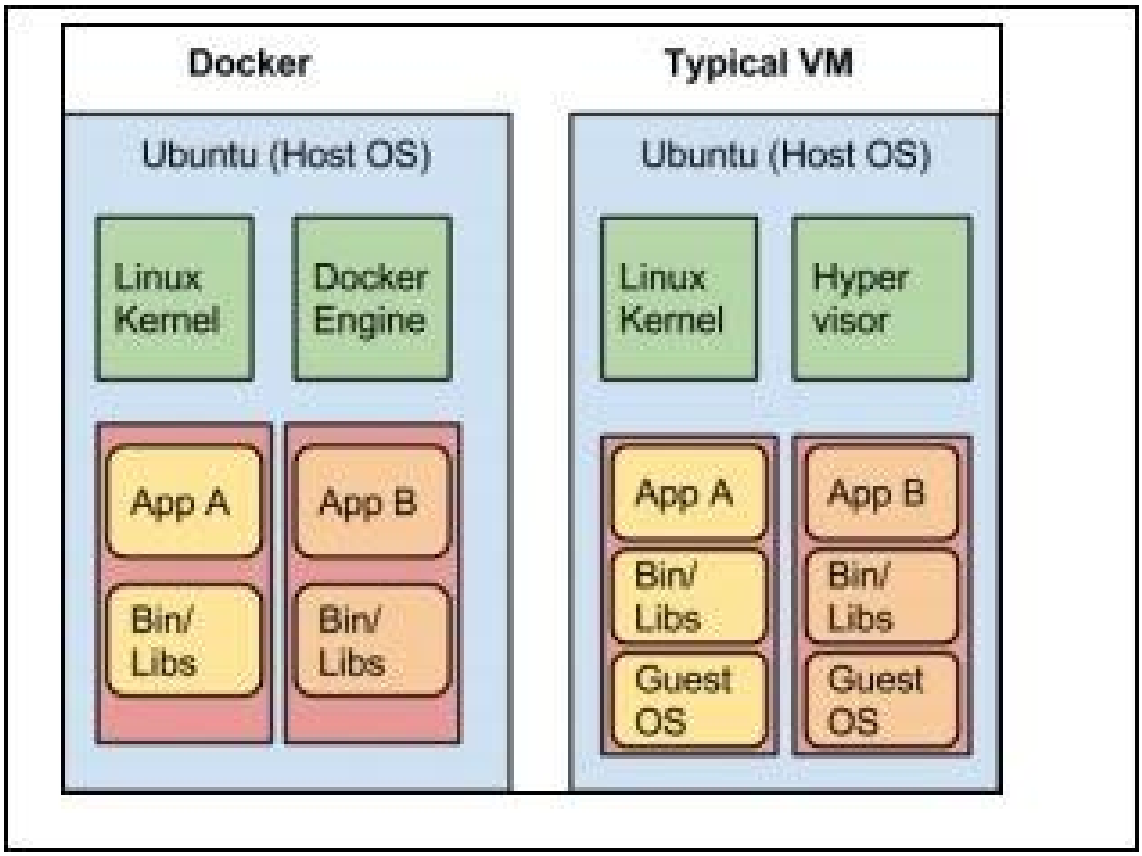

Figure 1. *The difference between a containerized and a typical VM based setup [9]*

As figure 1 shows, the difference between a hypervisor-based paradigm and a containerized one is that each app in the typical VM configuration requires a Guest OS within their VM, but as one can see in figure 1, each container doesn't require a Guest OS, which lessens the overhead, thus making the system easier to maintain and more efficient in terms of performance.

The images used by docker are immutable and built from layers with everything baked into the image. As Ankita Thakur put it, “a Docker container is a software bucket comprising everything necessary to run the software independently. There can be multiple Docker containers in a single machine and containers are completely isolated from one another as well as from the host machine. In other words, a Docker container includes a software component along with all of its dependencies (binaries, libraries, configuration files, scripts, jars, and so on). Therefore, the Docker containers could be fluently run on x64 Linux kernel supporting namespaces, control groups, and file systems, such as Another Union File System (AUFS). AUFS is a layered copy-on-write file system that shares common portions of the operating system between containers.” [9] The importance of copy-on-write should be emphasized and James Turnbull, a software developer and security specialist who is the vice president of services at Docker said the following in an interview with Charles Anderson for the software engineering radio podcast. “One of the other interesting technologies Docker relies on is a concept we call copy-on-write. Many file systems, such as Btrfs, Device Mapper, and AuFS, all support this copy-on-write model, which is what the kernel developers call a union file system. Essentially, what happens is that you build layers of file systems. So, every Docker container is built on what we call an image. The Docker image is like a prebaked file system that contains a very thin layer of libraries and binaries that are required to make your application work, and perhaps your application code and maybe some supporting packages.” [3] This greatly improves caching when building or running multiple images, since the host does not have to store duplicate operating system or other common components. A lesser benefit is that if an image is built from a well established and tested base image, the scope of debugging is limited to whatever is added on the well tested layers.

2.2 Orchestration and Kubernetes

To manage a fleet of containers, it is customary to employ an orchestrator. Kubernetes (k8s) is the most popular orchestrator and shares many characteristics with the other orchestrators so this BSc thesis will focus on it. Google Kubernetes Engine (GKE), Azure Kubernetes Service (AKS) and IBM Kubernetes Service (IKS) are all cases where a public cloud uses Kubernetes as the provided orchestrator [10]-[12].

When a single container in a cluster looks like it might have a problem, an orchestrator will simply restart it. Assuming the container will boot up correctly, this will solve the issue, but for the cases when it goes down again immediately after restarting an alert is

sent so an engineer can fix the situation. The end user may experience a brief service outage on their end, but eventually they will go through a network load balancer which will redirect them to a working container and the issue is resolved. In addition to managing multiple containers in a machine, an orchestrator typically manages multiple physical or virtual machines called hosts as well and deals with issues like hardware load balancing and resource management.

Kubernetes itself is an open-source orchestrator developed initially by Google and based on their previous system called Borg. Instead of singular containers, Kubernetes groups containers into pods[13]. A pod consists of multiple containers that are considered a unit. As David K. Rensin in the book 'Kubernetes' put it; "Kubernetes introduces some simplifications with pods vs. normal Docker. In the standard Docker configuration, each container gets its own IP address. Kubernetes simplifies this scheme by assigning a shared IP address to the pod. The containers in the pod all share the same address and communicate with one another via localhost. In this way, you can think of a pod a little like a VM because it basically emulates a logical host to the containers in it. This is a very important optimization. Kubernetes schedules and orchestrates things at the pod level, not the container level. That means if you have several containers running in the same pod they have to be managed together. This concept—known as shared fate—is a key underpinning of any clustering system." [13] Pods are distributed onto one or more nodes which is what hosts are called in Kubernetes terminology. When a pod is having issues, Kubernetes will attempt to eliminate nodes from the equation by restarting the pod on a different node if one is available. David K. Rensin wrote; "Another key concept in any clustering system—including Kubernetes—is *lack of durability*. Pods are **not** durable things, and you shouldn't count on them to be. From time to time (as the overall health of the cluster demands), the master scheduler may choose to *evict* a pod from its host. That's a polite way of saying that it will delete the pod and bring up a new copy on another node.[13]

Each Kubernetes setup has a master node that controls the worker or slave nodes. Interaction with the system goes through the master node API. Nearly all the components on the master and nodes accomplish their respective tasks by making API calls. These are handled by the *API Server* running on the master. [13] In addition to forwarding instructions to the worker nodes, the master node is running a configuration replication utility called Etcd, a scheduler and controller managers. Etcd is a service that keeps and replicates the current configuration and run state of the cluster. It is implemented as a

lightweight distributed key-value store and was developed inside the CoreOS project. [13] This ensures that the master and the worker nodes are in sync in terms of software versions and configuration. The scheduler decides which node is suitable for pods to run on, according to the resource capacity or the balance of the resource utilization on the node. It also considers spreading the pods in the same set to different nodes. [4] It is also the component that is responsible for handling health checks on the rest of the system. Faulty pods are moved to other nodes so that issues caused by a node can be ruled out of further investigations.

A single pod has limited processing power. Sometimes for demanding tasks like serving a busy website call for multiple pods performing the same thing. The process that manages these replicas is the *replication controller*. Specifically, the job of the replication controller is to make sure that the correct number of replicas are running at all times. That's its prime directive. Anything else it may do is to serve that end. [13] A controller manager is responsible for keeping other controllers up since they like containers and pods fail and must occasionally be restarted. It is usually a good idea to have two controllers that make sure the other one is operational so there is no possibility of a single point failure. If controllers are organized hierarchically, the master node hosts the highest-level controllers. Node setup is better explained in figure 2.

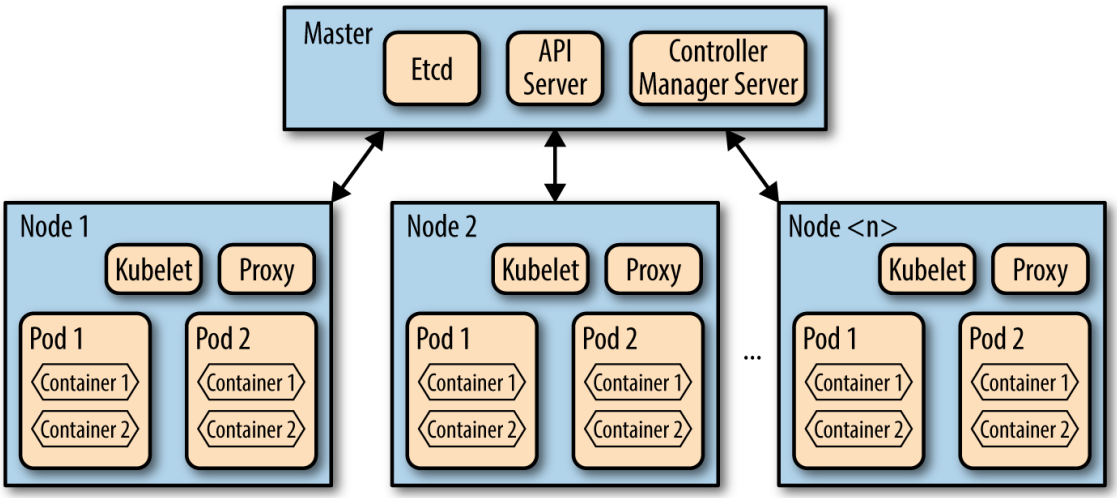

Figure 2. *Diagram explaining the configuration of Kubernetes nodes[13]*

Deployments are a way to control the cluster. According to the Kubernetes documentation, "A Deployment provides declarative updates for Pods and ReplicaSets. You describe a desired state in a Deployment, and the Deployment Controller changes the actual state to the desired state at a controlled rate. You can define Deployments to create new ReplicaSets, or to remove existing Deployments and adopt all their resources

with new Deployments.” [4] A deployment is a declarative document describing the state the cluster should aim towards and over time various controllers will enact actions to mold the system into whatever was requested. An important point that should be noticed is deployments are declarative. Instead of listing what the user wants the cluster to do, they tell the system what they want the end result to be. An essential tenet of the container paradigm is that the system is better off if individual pods are given as little consideration as possible and instead all effort is directed at the bigger picture. Pods and containers that for one reason or another get special treatment are called pets or darlings.

3. GCP EXPERIENCE

To support the more theoretical side of the thesis, I will share an anecdote of my experiences with the Google Cloud Professional Cloud Architect certification process. In order to get the certificate, I had to complete Google's interactive training in Qwiklabs [14]. Qwiklabs is a learning platform that offers courses of interactive lessons on a variety of topics. For the GCP certificate courses, Qwiklabs spins up a virtual environment for each lesson and the student is asked to complete some tasks in the environment. The testing environment works similarly to GCP, so if a student can perform a task in the virtual environment, they should be able to complete the same task in GCP.

I ended up doing a lot of courses on related subjects and reading through the documentation a lot. Turned out reading the documentation and especially checking for minor variations between similar tools. For example, Cloud SQL and Cloud spanner both offer database as a service (DBaaS) solutions. Cloud SQL is cheaper, but Cloud spanner automatically handles global accessibility, so if you do not need global access use Cloud SQL, if you do not use Cloud spanner. There are also important breakpoints, like how many connections can a single Cloud SQL instance handle in some time (250 for db-f1-micro VM, 1000 for a db-g1-small VM and 4000 for other machines[11]). The exam had multiple questions where the key to the solution was to remember the breakpoint and pick the correct option based on that.

Studying case studies provided by google is also a good way to understand how to apply the tools from GCP[15]. The case studies are a set of three different situations where a client would like to adapt a more cloud centric business model. One is about a company migrating from their own datacentre piece by piece to google cloud starting with their testing and development machines. Another is about a company providing large machines for mining and agriculture and a need to handle the data the machines send back. The last one is about a game company that had an unexpected hit and is having trouble handling their data. [15] The case studies are varied and do their job well. They also give very concrete solutions to problems which is something I personally value greatly.

4. STATEFUL APPLICATION, SPECIFICALLY DATABASE MANAGEMENT

4.1 Stateless and stateful applications

Nearly all applications maintain some information about their current state, but some of them can tolerate occasionally losing this information. These applications are called stateless applications and are generally easy to migrate to a cluster. A web server is a good example of a stateless application. If given a static site to serve, the server usually does not need to maintain any state and losing its state is not detrimental. The opposite of a stateless application is a stateful application, which will require persistent state to function. Databases are the most common type of stateful applications and thus naturally a focus of this BSc thesis. The purpose of a database is to maintain information given to it, so naturally in a container environment where state can be wiped this leads to issues.

Stateful applications requiring persistent state and containers not providing any by default will naturally mandate compromises with the ACID principles. Containerized stateful applications will require some apparatus to maintain their state due to not being able to do so by default by design. The design of this apparatus is not obvious and more importantly has no perfect solution that doesn't cause more albeit smaller problems. In this chapter I will examine the two main schools of thought about maintaining a database for a containerized system in a private cloud, within or outside the cluster. The benefits and downsides of both of these practices and how best avoid them. In short, within the cluster forces the ops engineer to work with the limitations of containers like their ethereal nature and outside the cluster the engineer needs to figure out how to connect the database into the cluster and how to scale with demand.

4.2 Databases and ACID

As previously stated, databases are common and nontrivial to manage in a containerized environment. A database such as a Microsoft SQL, MySQL, Microsoft Access, PostgreSQL, or MongoDB instance usually exhibits a set of properties that form the acronym ACID [16]-[20]. These properties are what most users expect from a database, so it is important to understand what they stand for. The words these properties come from are Atomicity, Consistency, Isolation and Durability. Atomicity means that operations should either be done in their entirety or not at all. Consistency means that repeated read operations will return the same result provided the database was not

mutated between them. Isolation means that two operations will not interfere with one another. Two simultaneous write operations will be processed one at a time and will not interfere with one another. Durability means that the system will maintain state even if it loses power immediately after processing a transaction. [21]

Usually in a cloud environment, one or more of the ACID principles must get compromised for the sake of performance. If data needs to be available globally, there should be caches around the world to provide faster access. This will cause problems with consistency since updates made in a cache will not reach all the caches instantly and things like overlapping write operations are possible and then need to be resolved somehow. Cache containers can also go down with updates that have not yet been synchronized to the master database. These problems do exist, but most of them last for such a short time and rarely cause permanent problems that the noticeable increase in performance is worth the inconvenience, especially because when a system scales up beyond a point, it is going to require supplemental access points even if the application isn't supposed to work globally and these access points will cause all the same problems, since they effectively are the same thing just optimized for a different purpose. Accessibility caches are spread around to minimize the delay from users to the service whereas the described supplemental access points exist to alleviate the resource requirements, so they are likely to be packed in heavily populated areas presuming users connect to the nearest one.

By no means does the fact that ACID properties get compromised indicate that they are not important. It is generally encouraged to stick to the properties whenever reasonable. Users assume that the database at least roughly exhibits all the properties and in the long run, there is no solution that guarantees all the ACID properties and the speed the modern world requires.

Containers bring with them a set of additional problems unique to them. Like how an oversight during the configuration can lead to a rogue container that gets stuck on boot loop, possibly corrupting data because of it. Persistence of data is also a problem with containers, since they get wiped every now and then by design. Ultimately however there are two schools of thought when dealing with stateful applications like databases and containers; Put the database inside the cluster or outside the cluster. The following chapters will discuss the benefits and drawbacks of both approaches.

4.3 Managing a database outside the cluster in a virtual machine

Instead of integrating a database to the cluster, it is possible to move the persistent application out of the cluster to its own virtual machine. In this section I will examine running a database in a virtual machine instead of in a container cluster. This by itself is not a bad idea and does have upsides especially regarding ease of use but has problems scaling. A virtual machine running a database service could easily take minutes to get up and running. This leads to an issue when the load on the system is not constant, like for example with a website. If the system tries to dynamically adjust by spawning new slave database VMs to help with the load it must either predict a few minutes or so into the future, keep backup machines always running or simply adjust in real time and let the user wait whenever the system needs to scale up. None of the options given for scaling the amount of running units are perfect and bring with them either increased costs or slowness of use which negatively impacts the user experience (UX). The first option is not impossible, especially when data is gathered over a longer period, but when a user could spend only a short time, even under a minute using a service, the number of users will fluctuate wildly, which means that the optimal amount of running services will fluctuate wildly. Attempting to extrapolate from the current situation is very difficult because of this. The second option would guarantee the service stays operational and the user experience is smooth, but constantly running more machines than an ordinary situation calls for will cause higher operation costs. The third opinion would be optimal for costs, but the users will experience issues with the service when the system must scale up.

To help with the management of the database, a logging environment like Graylog, Splunk, or an ELK stack could be deployed and then have triggers when the logs stop coming or have or lack some specified content [22]-[24]. A monitoring service like Stackdriver could also be used to monitor the system with a daemon and alert an engineer when something goes wrong. While both are good ideas and should be implemented, neither of them automatically solves any issues. Because of this, managing VMs will become cumbersome with a large fleet and will require puppet, ansible, terraform or some other external infrastructure automation or configuration management tool. And even with one of those, the debugging process is very manual. All this being said, it is often easier to manually debug a VM than it is to debug a container if one proves troublesome due to the better visibility one gets regarding VMs.

Public clouds such as AWS, Azure and GCP offer one or more database as a service (DBaaS) and almost always when operating in a public cloud it is more efficient to use their built-in tools than it is to create any configuration with the available generic tools like VMs and containers. This is however only possible when operating in a public cloud. Google for example offers several options like Cloud SQL and Cloud Spanner for traditional relativistic database needs and Cloud Bigtable for as google puts it, “A fully managed, scalable NoSQL database service for large analytical and operational workloads.” [25]

So, to summarize, the benefits of keeping the database separated from the cluster is that it is easier to troubleshoot and to investigate should something go wrong, but the downside is that automatically scaling up or down based on traffic is difficult. One should set up a system to monitor the machines and if operating in a public cloud, use a tool like Cloud SQL instead of maintaining an SQL VM.

4.4 Managing a database inside the cluster with persistent storage

Maintaining a database within the cluster has some immediate obvious benefits. The networking can be simplified, since slave database containers can be placed into pods with the containers that require them to create systems that can survive independently of the system for a while. These slave database containers could then be connected to the pod that has a master database to reinforce ACID principles. This simplifies the networking, since now there only needs to be two connections, one between the users and the service and the slave and master databases, the later of which can be done internally within a VPN, so the only point of access from the internet is that which is required by end users. This configuration will also make the configuration more secure.

Another benefit of using a cluster is that if you operate in a public cloud like AWS, Azure or GCP, the operation costs can come down as a result of dynamic billing and scaling [26]. Sometimes however operating in a public cloud is not possible due to legal reasons and the situation calls for a private cloud solution.

There are downsides to containerizing the database infrastructure as well. First of all, as previously discussed containers are ephemeral and cannot maintain state beyond their uptime, violating the durability part of ACID. This is an issue if the goal of the container is to act as a database to specifically and exclusively maintain a state. Both Kubernetes and docker do provide systems called volumes. Kubernetes volumes are more useful for the purposes of maintaining a database within the cluster, so those will be the focus. Kubernetes volumes are located within pods and come in three forms, EmptyDir, NFS and PD. EmptyDir is an empty directory, the contents of which will be deleted when the pod is shut down. NFS or Network File System volume is a network disk that resides elsewhere and is mounted within the pod for access. Data will be retained after the pod goes down and multiple pods can mount the same file system albeit with possible side effects and harm to the isolation part of ACID. PD or Persistent Disk is an alternative to NFS available only in GCP. Whereas NFS requires configuration to upkeep, google maintains PDs on a more abstract level. The ops engineer will have access to a disk, on GCP but will not have to bother themselves with setting up RAID or other redundancy methods. Google guarantees that PDs are safe both in terms of information security and data corruption.

When compared to VMs, containers and pods can get start a lot faster, often it takes less than a second for a container to get up and running. The speed is a result of containers using the host kernel and thus they don't need to wait for the operating system kernel to start. Because of this, a container can simply be restarted when odd behaviour is detected, which often solves the problem. However, even if a pod can launch in a sub second that is sometimes somewhat misleading. The service can get up and running in less than a second, but for something like a database, internal caching is very important in terms of performance. Aggressively restarting pods will cause the caches to be frequently emptied and the overall system to slow down drastically.

To minimize the downtime caused by the occasional restart, a Content Delivery Network or CDN can be deployed as a cache between the service and its users. A CDN could also provide faster access to assets for users. [25] A CDN would be subject to the same problems itself, but the system as a whole only needs to have either the CDN or the database up, increasing the service's uptime when compared to the configuration without a CDN.

Lastly configuring the cluster to have a master database pod will create a single point where a failure would cause the entire system to fail. This is not necessarily as bad as it sounds, since if we include a slave database in each pod the system will appear completely normal to the end user and assuming controllers are doing what they are supposed to. After a while, the master pod should be brought back without issues which will let the slaves' updates through, and the service will carry on. If the master pod is out for a longer period, the end user could notice desync between users or issues with consistency or durability, since subsequent visits to the service may be redirected to a different access point which may use a different slave database. The scheduler should kill pods the instant they look at all suspicious to avoid a possibly misbehaving master pod causing serious damage. This can lead to tardiness in the system but since a majority of the processing is done by the slave nodes it should not be too much of an issue.

So, to summarize, the benefits of integrating the database into the cluster is the ability to package slave nodes into pods with the service providing containers for marginal security benefits. One can also implement a CDN that easily provides better performance for the end users. The drawbacks being the added complexity of a system with very little permanence and the change in mentality that this brings with it. As a compromise one could use slave databases in the pods that use a Cloud SQL instance or a database VM as a master node. This in theory could combine the benefits of both practices but requires upkeep of several differing systems which is needlessly complex and provides very little benefit.

5. SUMMARY

The modern consensus is to move to the cloud if you are planning on having thousands of daily users, especially if more. Maintaining a singular virtual or even physical machine is likely going to be cheaper in the long run with low user numbers, in which case software should be installed directly on the machine. Otherwise move to a cloud, preferably a public one if information you are storing allows it. In the public cloud setup, an orchestrated cluster with Kubernetes or equivalent and form your services into pods with the resources they require. Create a global master database pod to guarantee ACID principles and use it as a master database for the slaves in the pods. Have controllers and schedulers manage the system automatically and send automated alerts to your engineers with a monitoring solution with something like Google Cloud's operations suite (formerly Stackdriver).

REFERENCES

[1] Anonymous "Web hosting services market size, share & trends analysis report by type, by application, by deployment (public, private, hybrid), by end user (enterprises, individuals), and segment forecasts, 2020 - 2027," 2020.

[2] A. Watson, "Film industry - statistics & facts," Oct 26,. 2020.

[3] J. P. J. Mello, "Containers are quietly replacing VMs. Is your IT Ops team prepared?" 2017. Available: https://techbeacon.com/enterprise-it/containers-are-quietly-replacing-vms-your-it-ops-team-prepared.

[4] *Kubernetes Documentation*. Available: https://kubernetes.io/docs/.

[5] *Flatpak—the future of application distribution*. Available: https://flatpak.org/.

[6] *Pop!_OS by System76*. Available: https://pop.system76.com/.

[7] *Docker Documentation*. Available: https://docs.docker.com/.

[8] *Exploring simple Linux containers with lxc*. Available: https://www.redhat.com/sysadmin/exploring-containers-lxc.

[9] A. Thakur, *Docker : Become an Expert in the Innovative Containerization Tool to Unlock New Opportunities in the Way You use and Deploy Software : A Course in Five Modules.* 2016.

[10] *Azure Kubernetes Service (AKS) | Microsoft Azure*. Available: https://azure.microsoft.com/en-us/services/kubernetes-service/.

[11] (Last updated -10-22 UTC.). *Google cloud documentation*. Available: https://cloud.google.com.

[12] *IBM Cloud Kubernetes Service*. Available: https://www.ibm.com/cloud/kubernetes-service.

[13] D. K. Rensin, *Kubernetes.* (1st ed.) 2015.

[14] *Qwiklabs - Hands-On Cloud Training*. Available: https://www.qwiklabs.com/.

[15] *GCP Professional Cloud Architect Certification exam guide*. Available: https://cloud.google.com/certification/guides/professional-cloud-architect/.

[16] *PostgreSQL: The world's most advanced open source database*. Available: https://www.postgresql.org/.

[17] *Database Software and Applications | Microsoft Access*. Available: https://www.microsoft.com/en-us/microsoft-365/access.

[18] *SQL Server 2019 | Microsoft*. Available: https://www.microsoft.com/en-us/sql-server/sql-server-2019.

[19] *The most popular database for modern apps | MongoDB* . Available: https://www.mongodb.com/.

[20] *MySQL*. Available: https://www.mysql.com/.

[21] (June 20,). *What does ACID mean in Database Systems?*. Available: https://database.guide/what-is-acid-in-databases/.

[22] *Industry Leading Log Management | Graylog*. Available: https://www.graylog.org/.

[23] *The Data-to-Everything Platform Built for the Cloud | Splunk*. Available: https://www.splunk.com/.

[24] *ELK Stack: Elasticsearch, Logstash, Kibana | Elastic*. Available: https://www.elastic.co/what-is/elk-stack.

[25] *What does CDN stand or and how does it work?*. Available: https://www.akamai.com/uk/en/cdn/what-is-a-cdn.jsp.

[26] S. Singh, "Cost breakdown of Public Cloud Computing and Private Cloud Computing and Security Issues," *International Journal of Computer Science and Information Technology,* vol. 4, pp. 17-31, 2012.