Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Evaluating dynamic load balancing of ECM workload pattern employed in cloud environments managed by a Kubernetes/Docker eco-system

Pascal Hagemann

Course of Study:	Informatik
Examiner:	Prof. Dr.-Ing. Bernhard Mitschang
Supervisor:	Dipl.-Phys. Cataldo Mega
Commenced:	February 1, 2021
Completed:	September 1, 2021

Abstract

<Short summary of the thesis>

Contents

List of Figures

Acronyms

AI Artifical Intelligence. 17

CNCF Cloud Native Computing Foundation. 23

CNI Container Network Interface. 24

DC/OS Distributed Cloud Operating System. 24

DDoS Distributed Denial of Service. 44

DLB Dynamic Load Balancing. 13

DNS Domain Name System. 43

ECM Enterprise Content Management. 7

HPA Horizontal Pod Autoscaler. 26

HPC High Performance Computing. 13

HTTP Hypertext Transfer Protocol. 44

IaaS Infrastructure as a Service. 14

IaC Infrastructure as Code. 25

ICN IBM Content Navigator. 31

IIM Intelligent Information Management. 20

J2EE Java™ 2 Platform Enterprise Edition. 31

K8s Kubernetes. 7

KPI Key Performance Indicator. 20

LSDB Library Server DB. 31

MAPE Monitor-Analyze-Plan-Execute. 13

NAT Network Address Translation. 41

NFS Network File System. 28

NIST National Institute od Standards and Technology. 13

OCI Open Container Initiative. 22

OS Operating System. 11

PaaS Platform as a Service. 14

QoS Quality of Service. 11

REST Representational State Transfer. 27

RMApp Ressource Manager Application. 31

RMDB Ressource manager DB. 31

SaaS Software as a Service. 11

SLA Serice Level Agreement. 11

SLB Static Load Balancing. 15

VM Virtual Machine. 7

WAS WebSphere Application Server. 31

YAML YAML Ain't Markup Language. 25

1 Introduction

As more organizations transition to advanced solutions for their ECM systems they need solutions which can adapt to a dynamic environment [DH13]. This includes long term changes like a future growth or shrink in users as well as daily swings in usage driven by the day and night cycle. The adaption is relevant for cloud users as well as providers who need to fullfil their contracts while being profitable [Gun20]. To achieve this, modern ECM systems use a non-monolithic structure and are deployed into cloud environments. This approach follows the increasing abstraction from the underlying hardware where traditionally the deployment was issued directly on the server. Than in a first step the Operating System (OS) was abstracted by moving the deployment into virtual machines increasing security and control and remove dependency conflicts. To eliminate the additional overhead of the virtual machines in a next step, containerization is used to deploy applications [ZX20]. This allows for fine-grained control over different parts of the system especially by vertical scaling via the containerization but also horizontal scaling via constraint sets. The transition was driven by a increase in performance of clouds through efficient distributed computing which leans on efficient networking. A report by sysdig [sys21] suggests a growing and diverse market for containerized applications. It also mentions security aspects of container systems, which is a key factor for ECM applications. The usage of additional software to scan and analyze containers increased rapidly and while there are still insecurities they can mostly be identified and fixed. This analysis also includes monitoring which gets even more important the bigger and complex these systems get.

By exploiting different new aspects of these systems, organizations can provide their services in an efficient, fast and flexible manner. Efficiency is especially relevant because it can leverage the full potential of pay-as-you-go models used by cloud providers. An efficient use of the resources used and paid for is a big economic insentive. Even for self-hosted clouds the reduced energy usage generates an economic benefit. The software is preferably provided as a service and thus follows Software as a Service (SaaS) principles. The SaaS model opens new approaches to fulfil Serice Level Agreements (SLAs) for the system and achieve a high Quality of Service (QoS) in different multi-tennant scenarios. [MWL+14] lists some examplary Service level agreements which are usually time or space contraints on certain operations. As the software is handled by an automatized system it reduces manual intervention and thus management cost. A structured approach to expansion is also preferred as it follows a clear path and does not lead to unstructured and unmonitored systems with low utilization in certain areas.

For this purpose we want to research the deployment of an ECM system in a cloud environment together with a dynamic load balancing system. Our approach tries to leverage the microservice architecture where the application is split up into small parts which are loosly coupled [Red]. This enables each part of the application to be maintained separatly while only keeping the communication interface to the other components synchronized. The different parts of the system therefore need to be separated into loose-coupled components which are then containerized. The starting point is an ECM system deployed into Docker by Shao [Sha20] who took an existing ECM system

and containerized the components. These are now modified and translated to be controlled by a container orchestrator. To test the dynamic nature of the system different predefined workloads are applied with different intensities to represent mulit-tennancy. The architecture consists of a set of virtual machines on no specific architecture and running a container orchestrator. This is similar to a cloud environment where only the outlines for performance and processor architecture are known. The transformation of this application and of the application landscape in general is at least in part driven by the open-source software community [sys21]. Many big companies have released their software as open-source to speed up and outsource development while also making it more popular. The approach presented here leverages the open-source software to make it accessible to a broad range of users. Therefore all used software is, unless otherwise noted, open-source and free to use.

Structure

The following Chapter 2 describes the techniques and concepts used in this work. This includes the context in which the software is deployed as well as different approaches to tackle the issue. Chapter 3 describes our prototype implementation and the individual challenges of each stage. Also discussed are the test workloads and metrics which are used to perform the hardware evaluation to gather replication heuristics. Section 3.3 shows the results of the analysis for the defined metrics under the given workload and scenarios. Finally in Chapter 4 we draw conclusions from the previous results and mention further goals and improvements which where out of scope for this work.

2 Background

This chapter states the varous techniques, systems and structures used. There are variouse approaches to Dynamic Load Balancing (DLB) which need different amounts of input data and thus lead to different results. Mega et al. [MWL+14] list three categories of methods. First there are utility-based optimization techniques which try to maximize the utilty of each component by minimizing unused resources. A performance model for the specified system needs to be build and updated if the system changes. The second method group are based upon machine-learning. These try to build a model for resource consumption which is then applied to determine workload even before they arise. A similar approach for High Performance Computing (HPC) is presented in [TAZ+17] but the technique used can also used for other applications. They use time series data and a cloud environment which is similar to the environment used herein. With ever more data collected and used for the model the system gets better over time if the learning algorithm works correctly. The third approach, which is also used in [MWL+14], is based on a Monitor-Analyze-Plan-Execute (MAPE) feedback loop which is fed heuristics for system performance. The MAPE loop widely used to forecast and predict resource usage in the cloud [Koe14; WBW14]. The loop has four stages:

1. The **Monitor** stage collect metrics from all components.
2. Then in the **Analyze** stage the information is used to gather the relevant information for the
3. **Plan** stage where heuristics are applied on the metrics-information to create an execution plan.
4. The plan from the third stage is then **Executed** to the system after which a new cycle begins.

In certain intervals the system is also load tested to gather information on relations between the metrics and SLA restrictions. The data from the load tests is then further used to update the heuristics to match the current conditions of the system. This is needed to ensure that Service level agreements are not breached due to changes induced by usage of the system. The approach proposed in Chapter 3 is based on this approach.

2.1 Cloud Computing Structure

Cloud computing describes a architecture which can provide ressources to different tasks on a on-demand basis. This allowes the underlying hardware to be used in an versatile and efficient manner. The National Institute od Standards and Technology (NIST) provides a recommendation for definitions on cloud computing [MG11]. Therein NIST defines the following characteristics of cloud computing platforms:

1. *On-demand self-service*. The ability to automatically provision capabillities to a consumer.

2. *Broad network access*. That those capabillities are available over standard network clients.
3. *Resource pooling*. Multiple customers are asigned resources dynamically on demand.
4. *Rapid elasticity*. The capabilities can be scaled with demand at any time.
5. *Measured service*. Controlling the resource usage by metering capabilities and provide transparency over usage to the consumer and provider.

On that infrastructure the computing can be disclosed in different service models. The lowest level is the Infrastructure as a Service (IaaS) model in which the hoster provides the raw infrastructure. These are typically VMs running a predefined OS and are connected by a high bandwith connection to each other and the internet. On the second level there are Platform as a Service (PaaS) systems which provide access to a given platform running on VMs. One example of this is a cloud hosted K8s cluster on which an containerized application can be deployed by the developer. The advantage of this model is the reduced effort in managing the K8s cluster. All management tasks are performed by the provider automatically for example when a node needs to be added or removed from the cluster. The obvious drawback is the lack of freedom in certain configurations and choices in terms of software as only a small subset is supported by each provider. The last and most abstract model is the SaaS model. Here a specific software stack is automatically deployed with a developer defined config and scaling dependent on the availability and performance constraints. The applications used often would require high maintenance cost which can be outsourced specialized providers and thus lowering the cost. As this service is not offered for every application developers are constrained to a small set of applications for each field of application to choose from.

Additionally the cloud environments can be separated by the location of the hardware. Clouds hosted by a provider are called "public"clouds because the infrastructure can be rented by everyone. Therefore an application might run on a VM which shares CPU and RAM with one or more other VMs. This leads to problems in high security relevant applications because of exploits which can leak data on the same hardware like the relativly new discovered Spectre[] and Meltdown[]. It is possible to detect these with performance counters but this method is not reliable in all cases [MSHN17; Sze15]. On the other hand most applications and the companies using them profit from the shared hardware by reducing the expenses for electricity and maintenance. Especially for smaller firms without a huge IT department this can open them up to new possibilities which where not economic before. For the cloud hosting providers there is also a economic benefit if they can hold all their hardware at a high saturation at all times. This can only be achieved if users from different parts of the world participate in the service so daytime differences compensate each other. For companies which need to comply to higher security standards or want to be in control of the underlying hardware "private"clouds can be an alternative. The clouds are typically used only by one company or a limited set of customers and thus are not usable by the public. Therefore the saturation is not as good as with multiple tenants and needs to be layed out to fullfill peak loads properly. The owner is also solely responsible for maintenance and failure of its system.
To mitigate the negative properties of private clouds while still keeping some of the advantages a hybrid cloud approach can be used. This allows for some applications to be hosted in a private cloud while keeping others in the public cloud. The processes can then communicate over the internet. The hybrid cloud approach can therefore enables cloud computing for applications which need to run and communicate with local machinery despite for example a internet connectivity outage. As a drawback for the hybrid approach one could name higher security considerations by a

potentially higher amount of open communication channels between the public and private cloud. Also the effort for setting up the system on two separated clouds can be higher depending on the abstraction level.

2.2 Load Balancing

Load balancing describes the task of distributing tasks to a set of resources respectively nodes in a cloud environment. This is accomplished by load distribution function which assigns a task to a resource. The challenge is to design the distribution function in such a way that it optimizes specific system characteristics or metric. In general this can be the system load but also any other metric of the system. This leads to a diverse set of algorithms which vary greatly in terms of complexity and field of application. Zhiyong et al. [ZX20] describe and compare several task scheduling algorithms. These are divided into independent of workload and workload dependent. For multi-tennant applications it is important to distinguish different workloads as they can differ in complexity.

Load balancing mainly divides into two categories [NMNA12]: Static Load Balancing (SLB) and DLB. SLB bases its descisions for load distribution on a predefined system model. The system characteristics are defined by measurement or evaluation of the used hardware and software. From this model a central load balancer tries to minimize the distribution function for each resource it assigns tasks to. Because the system is assumed as static the distribution function has only the already distributed tasks and task to distribute as inputs. These functions in SLB are therefore less complex than in DLB. DLB as opposed to SLB does not use a static system model and instead relies on the current system state. Thus the distribution function tends to be more complex because it adds the system state as input. In regard to cloud systems with a dynamic structure of components and connections the complexity further increases. The load balancers need to also react to these changes to optimize the system.

The advantage of SLB is the simplicity of the approach which leads to a higher performance of static load balancers. This can be used efficiently if the nature of the tasks is well known and thus task execution can be accuratly predicted. A static load balancer can therefore be more cost effective and easier to integrate within existing systems. The disadvantage of SLB is the unawareness of the balancer in terms of system state. Because SLB only assumes the system state there is no guarantee that the real state does not differ significantly. This is especially true for highly dynamic systems and cloud archtiectures where there are numerous different tasks on a shared architecture. DLB on the other hand needs to be integrated more tightly with the system to react to the system state. This leads to higher efforts in setup and initialization for systems using DLB. Dynamic balancers are also more expensive in terms of computation and storage for metrics and system state. One advantage comes from the wide applicability of this approach. Because the distribution of tasks is tied to the system state a task does not need to be as accuratly characterized as for the static approach. The impact of every task on the system is measured and fed back into the next scheduling descision. If for example a task with unusual demands is scheduled an imbalance between resource utilization is created. A static load balancer does not recognize this and continues scheduling to all resources whereas a dynamic balancer notices the imbalance. He can then schedule further tasks to different resources to even out utilization.

In this constellation the load balancer marks a single point of failure. All requests pass through the balancer therefore, if the balancer fails no requests can be fulfilled. So the state of the balancer needs to be monitored as well as the state of the system to determine potential problems. To mitigate this problem, the load balancer can be distributed. This increases reliability of the service as well as performance of the load balancing system. One load balancer can only handle a specific number of requests per second. If more requests than this arrive they either get lost or are queued up what leads to higher response times. To tackle this issue load balancers need to be scalable to split the load. This however requires load balancing for each balancer.

Typical load balancing algorithms include Round-Robin, Throtteled, MapReduce, Least Connection / Response Time / Bandwidth and Hashing. Nuaimi et al. [NMNA12] compare seven different static and dynamic algorithms on replication, speed and other properties. Load balancing is a broadly researched topic since it is used before cloud computing existed. There are several papers comparing different algorithms [AS15; CS06; SNA14; SS14] and variations to existing algorithms improving certain aspects [AAA19; AAME19; PB19; SJ20; TZZ+11]. They all provide a lead in one or more categories with drawbacks in others. As a cloud provider or user it is important to identify the own bottlenecks and choose an appropriate approach.

2.3 Replication

To improve performance of a service more instances of this service can be created. Each of these replicas offers the same service and can be used interchangeable. Replication needs to be performed either if the current resources are overwhelmed by the current workload or if there is not enough workload for all instances to work efficiently. In the cloud computing environment resources are typically computing and storage nodes who are located in server complexes. Cloud description languages are used to describe the resources and the topology. There are a bunch of different languages and therefore no real standard for these descriptions. Yongsiriwit et al. [YSG16] use three different languages to create a framework that allows for interoperability between them. If resources under- or overwhelmed the amount of instances needs to be reevaluated and further increased or decreased depending on the current state. In case of a decrease excess instances need to be scheduled to stop and subsequently be destroyed. Before a instance is stopped teardown actions and active requests in the instance need to be finished. For a increase of instances new instances must be created and started.

This typically involves the creation or assignment of storage to these new instances thus extending over a longer period of time to initialize the storage. After the creation the first startup also typically involves more setup steps and can take considerably longer. These startup and teardown times need to be considered in the determination of the reschedule intervals of the rules engine. If the interval is set too low instances are started so frequently that they are still in the startup phase but already count towards the instance count. This distorts the calculation of the instances and can lead to a escalation circle and overshooting in which new instances are added but do not contribute to the work until the next evaluation. This leads to new instances being added to meet the demand and so on until the first added instances start working regularly. The before mentioned circle of escalation can also similarly be created by instances which are in the teardown phase which leads to a underestimation of instances needed. A too high interval on the other hand can lead to low utilization by holding on to instances while the demand is already too low. New systems designed

for this kind of scheduling therefore try to keep their initialization, startup and teardown times as low as possible. With lower times in these steps demand curves can be followed even more accuratly for greater utilization. For systems with reactive scheduling startup times can be even more important to meet Service level agreements when demand increases.

When replication is applied the correct amount of instances is determined by a rules engine. The rules engine consists of a set of rules for each component that needs to be scaled. The rules for each component get evaluated in specifc intervals and combined together to create a new amount of required instances. If the determined number of instances changes the appropriate action needs to performed. If instances need to be removed a set of instances from the existing are selected. The selection can be random or if enough information is given an intelligent choice can be made to select the most irrelevant ones. If new instances are scheduled a set of nodes from the network is selected to host them. This process can also be informed to improve system performance and latency. Potential pitfalls and considerations for these processes are further described in chapter 2.3.1.

The rules engine can schedule changes in two ways: Reactive or proactive. If the scheduling is reactive the engine waits until a rule is violated and then enforces countermeasures to return to a valid state. With proactive scheduling the engine determines a point in time when the a rule is violated based on an estimate and then schedules a change so that the violation is mitigated. The obvious advantage of the second approach is that in an optimal case no rule violation occurs at all. But this is based on optimal prediction capabillities of the rules engine. If the prediction is wrong unnecessary resources may be used. This method can also cause a big overhead when more sophisticated prediction methods like machine-learning or Artifical Intelligence (AI) is used. Reactive scheduling does not perform actions until a violation takes place. This leads to a timespan where the system does not conform to Service level agreements. The administrator needs to decide if the system can react quick enough to transition into a valid state. A often used compromise is to proactively create instances but destroy them reactivly to be sure to stay in a valid state.

In a cloud environment with public clouds it is possible that the network of nodes spans over different providers to increase availability. In that case the system should be capable to automatically add or remove nodes from the network on each provider to increase efficency. For the IaaS model this means to attach more (virtual) machines to the network. This is specific to the provider and therefore needs to adapted for each one separately. For PaaS systems the nodes are not directly accessible for the administrator and thus can not be explicitly added. The nodes to host the platform on are chosen by the provider on basis of the requested capabillities of the platform. On a containerized platform like Docker or Kubernetes new containers are automatically placed on a node in the network. Combining networks on different providers is therefore not possible unless nodes can be manually added. SaaS systems further abstract from the underlying hardware and only make the service available. The administrator has no direct influence on the hardware or platform the service runs on. If a service needs more or less performance the administrator can request the corresponding capabillity changes.

2.3.1 Scheduling

When the rules engine decides to create new instances they need to be scheduled on a specific node. This is relevant especially when serving the application on a global scale where physical location matters. Therefore the instances should be spread evenly over the area of use. For the selection of

possible nodes the capabillities of the nodes in the network must also be considered. Some nodes can potentially contribute more to performance than others. This affinity must also be considered when removing instances from the system to choose the ones contributing the least. As factors for the affinity several factors like latency, performance, storage space, location or existing instances can be accounted for.

Even before the rise of VMs fast scheduling of them was important as seen in [SCP+03] where a VM state is transferred over a slow DSL link in minutes. From there scheduling of VMs and later container got through different phases. Khanna et al. [KBKK06] show an approach of server consolidation for VMs by monitoring key performance metrics and migrating the VMs to different servers. They also use heuristics to solve the optimization on minimizing cost while maximizing utilization and respecting SLAs. In [DMNC13] scheduling of VMs in a cloud environment is compared with Round-Robin and throtteled load balancing. For containerized architectures rescheduling is even more important because containers have an even shorter lifespan and thus need to be scheduled more dynamically [sys21]. New or improved algorithms for this kind of scheduling like [QYL+21] are developed after the demand rises steadily. Scheduling is also important for *Edge Computing* where instead of large machines computing is mostly done on small machines ”on the edge” of the system [CZS19; PI20]. These small *Edge Devices* can improve delay and energy consumption by stripping of the overhead from the main system.

A distinction for scheduling needs to be made between instances that are stateful and stateless instances. Stateless instances do only save data temporarily and thus do not need persistant storage. When they are created they connect to external storage solutions to manage their data. This makes it easy to create and destroy instances of this type. If they have associated data storage no measures must be taken to save the data.

Examples for stateless instances are load balancers. They can keep temporary state for their last scheduling descisions but this information is not essential. Stateful instances on the other hand create data that should not be discarded. The data is either shared between all instances of this service or unique to the instance. In the latter case appropriate actions must be performed to save the unique information. One example for this type are FTP servers. The files in the server must be preserved when the instance is destroyed. Before destruction the files must be assigned to another instance where they can be accessed from. This can be either explicit by assigning a subset of files to each instance or implicit by making each file accessible from each instance.

2.4 Monitoring

Monitoring is the key aspect for providing insight into all processes in the cloud. Aceto et al. [ABdP13] survey the different aspects of monitoring systems in the cloud. They also provide an exaustive list of monitoring systems that existed in 2013 with their capabilities. For our special case of container monitoring many of the named properties also apply. This work focuses mostly on the replication detection and metrics gathering part.

To predict the correct amount of instances dynamically from the rules the system state needs to be monitored. Monitored components can be of a wide range from hardware states to specific properties of applications. The task of monitoring many components designed by different people can be a difficult because no general standard for publishing metrics has been established yet.

Therefore special monitoring systems are needed to unify the data collection for further evaluation. These systems act as a centralized instance for all metrics gathered from the monitored system. The monitoring systems can be located either local on each node or more integrated into the system as cloud application. The integration into the cloud has the advantage of using the same mechanics as other applications. This includes for example security and infrastructure management which can be leveraged as opposed to maintaining a separate system.

To get metrics from the system they can be either collected (pulled) by or send (pushed) to the monitorung system. The first option has the advantage that the monitoring system can decide the interval in which the metrics are gathered. It can manage the collection for all gathered components and adjust the timings if needed. This allows the monitoring system to actively decide from which components it needs more or less information. This way a higher precision can be reached for important metrics.
With a push configuration on the other hand the application decides when to send metrics to the monitoring system. It can decide on its own when to update metrics more or less frequent to reach a specific precision. The application needs to know how to reach the monitoring system and may also possess credentials to access the system. Also the application needs to ensure that in case of an error the metrics are still updated or not send at all. This gives the monitoring system the opportunity to recognize when an error occurs.

For further analysis the metrics are typically stored in a database. Since all metrics are related to a timestamp a time-series database is the most appropriate. Most systems use a short term database which stores the data temporarily before they are transferred to a long term database. One metric measurement consists of a value, a timestamp and a name. This is the minimum amount of information needed to work with the metric. Collected metrics also can have tags attached to them to allow for categorization and filtering. An example for a tag is the source of the metric if there can be multiple sources for the same name. For cloud environments where applications may be deployed several times this is a key factor.

There are many challenges to overcome for monitoring cloud applications especially on a bigger scale. Since all data is combined into a single database this database marks a single point of failure. To mitigate this the database and the monitoring system can also be replicated and load balanced. This also increases the performance to allow for more metric measurments and therefore a higher precision. If a decentralized monitoring system is used this is automatically the case but to access all information the systems need to be connected or synchronized. This leads to a higher latency and network usage. Hauser et al. [HW18] identify challenges for monitoring and propose an alternative solution. They see the overhead of monitoring as a central problem of monitoring at scale. Lightweight tools have been proposed earlier to keep the additional overhead as small as possible [MSA12]. [HW18] propose a monitoring system based on the physical level only. They also do not use a centralised instance but dedicated metrics for each node. To minimize complexity when quering metrics they are compressed with statistical analysis to keep only relevant information.

Performance indicators

Key Performance Indicators (KPIs) are a subset of metrics which are general applicable to all systems. These indicators relate direclty to the performance of the underlying hardware and therefore give a direct representation of utilization. The most used indicators of this kind are

- the CPU load which can be measured per process on a basis of the available cores.
- the main memory usage of each process
- the disk I/O which is the amount of data read and written on disk over time
- the network usage as up- and download direction over time

Additionally other components can be included if present. For example GPU usage which is getting more important due to the recent shift to more AI centered applications. These can leverage the manycore architecture of GPUs to a great extend. With the rising demand GPUs are present in many cloud systems to accelerate tasks. If a deep insight into the performance bottlenecks is needed standard sources are to coarse. The monitoring of performance through Performance Counters can be used to gain more fine grained results [TM14].

To adapt to a specific system the metrics build upon these indicators can be abstracted and transformed to compact information. For example instead of measuring the disk I/O directly only the number of files created and deleted are measured. The network usage could also be reduced to the number of connections if the number of parallel tasks is most relevant.

2.5 ECM-Systems

ECM is described by the AIIM[1] as a combination of strategies and tools to collect and organize information for a designated audience. The tasks of the ECM system includes capturing, storage and management of information as well as delivery through the entire lifecycle.[2] They also mention a transition from ECM to Intelligent Information Management (IIM)[3] to also include data management. IBM also includes the full lifecycle from capture to providing insight into their definition with focus on taking full advantage of information of the content.[4] The five main components of an ECM system are:

- Capturing of information from various sources
- Management of this captured information by handling and transformation to make it usable
- Storage of information for direct use followed by
- preservation of relevant information for long-term use
- Delivery of information to the user

[1] AIIM: https://www.aiim.org
[2] AIIM ECM Definition: https://www.aiim.org/resources/glossary/enterprise-content-management
[3] IIM: https://cdn2.hubspot.net/hubfs/332414/AIIM_Blog/Intel-info-Next-Wave-2017-updated.pdf
[4] IBM ECM definition: https://www.ibm.com/cloud/automation-software/enterprise-content-management

To deliver the information a frontend application is used. The application can be a standalone software or a web application. New systems which rely on a broad customer base prefer web application since it gives a standardized interface to the user. With this approach the systems get rid of dependencies from operating systems or special libraries and also hides more internal processes that could be exploited. The storage is divided between file objects and organized data. While file objects are stored on the filesystem directly organized data is typically stored in one or more databases. Another approach is to use NoSQL databases to store unstructured data like files for better handling and faster access times [RE13]. This improves the performance and concurrent access capabillities for organized data. Examples for organized data are user information, file meta data or configuration options. To handle the file objects a object manager is used to handle access permissions and meta data for these objects as well as storage location information. For preservation data is transferred to a secure storage with lower performance but more efficient storage.

Typical components of cloud ECM systems are (1) a webserver as frontend interface, (2) a library server for storing information and (3) an object manager to store and manage objects like documents. For (2) and (3) additional ressources must be reserved for the system to store the information. The webserver (1) only displays and forwards information and therefore does not need permanent storage allocated. The requirements for storage and performance must be validated and revalidated over time to meet the current maximum demand of the system. Different technology requirements are needed for each system which results in various combinations of capabilities [DH13]. Moving existing ECM systems to the cloud is more difficult than introducing a new system as they are not designed with cloud principles in mind. In [KSST05] the IBM system which will be used for the prototype in Chapter 3 is adapted to place replicas on optimal nodes. Therefore they use an optimization algorithm together with heuristics to find a good solution with respect to memory and load restrictions. An optimal solution is not feasible for most optimization algorithms due to their NP-hardness. Mega et al. [MS20] surveyed the legacy ECM system structure and evaluated the key points to make a transition into the cloud. The main problems identified are firstly the intertwining of all components which prevents separating into loose-coupled or independent components. Second the built-in topology organization which is managed externally in cloud applications and third the reliance on hardware instead of virtualized components. This includes virtual network and storage in contrast to physical hardwired resources. Overcoming these challenges especially for networking and topology is the main part of this work.

2.6 Container orchestration

A container image is a repository of image layers each representing changes to the images filesystem. The layers build up a filesystem which is complete to run on an existing kernel. To run the image a container runtime starts the initial process of the container within a separate kernel namespace [VRS20]. The kernel namespace seperates each container by giving them a new context for all ressources like process IDs, users or files as well as network interfaces. To start a container a container runtime is provided with the mountpoint and metadata. This runtime then starts a new kernel namespace with the specific ressources allocated to the container process. For storage a separate storage driver is used to generate a distinguished storage area. The runtime also sets up access and usage limits to all kinds of ressources to provide security from potentially malicious containers. Starting a container image this way creates a running *container*.

The next step is the container engine which sits on top of the runtime to provide a high-level interface for users. The engine manages container images from various repositories and also prepares the necessary ressources for running the container. It also provides the necessary parameters to the runtime dependend on the user configuration. For automation it provides an API to handle operations. This API can be used for another level of abstraction in container orchestrators which will use the API to manage underlying containers. Orchestrators provide a standardized way to deploy and connect different applications. They manage the automatic scheduling and distribution of containers on different nodes within their system. Each node runs a container host which is used by the orchestrator to form a network of nodes. They can also create an overlay network which spans over all nodes in the system. This contributes to security since all communication is tunneled by the orchestrator between nodes. Furthermore it simplifies the communication since all inter-node communication is hidden. Orchestrators also manage failures of all components used by the system. This includes the applications hosted as well as the underlying components and ressources. If a failure is detected either countermeasures can be taken or the failure can be reported to an administrator for further actions.

Figure 2.1 depicts the hierarchy of a running container instance and lists examples for programs fulfilling each step. Starting with the engines and orchestrators which manage the user input and configuration of containers. These pass the image and configuration parameters to the runtime interfaces in a Container Runtime Interface (CRI) format. The interfaces create a runtime description in the Open Container Initiative (OCI) container image specification format[5]. The runtimes are the only component interacting with the OS and create the namespace and storage from the specification. De Velp et al. [VRS20] show in their reasearch the influence of different engines, runtimes and storage drivers for different tasks. Figure 2.2 shows that the time from creation to finish of execution can vary significantly for different runtimes. Especially the create and startup phase can differ to a factor of three which is crucial for short-lived container instances. A smaller difference was also measured for different engines.

The container images are provided in a specific format to be recognized by the engine. Before united each container engine had its own image format and thus could only be used with this images. The main difference was in the structure of the images. Some use a layered format where changes in the structure between each step are saved while others use no layers or respectively a single layer containing all information. In 2015 the OCI was founded by Docker, CoreOS and other companies in the container industry to create standards for an open container environment. Docker donated their container image format and container runtime (runC[6]) at that time to OCI as initial cornerstone. Since then most container engines support and participate in the development of these standards. The container image specification[7] includes

- one or more filesystem layer describing the changes to the filesystem which when applied in order create the container filesystem,
- a manifest which generates an unique ID for the image as well as providing multi-architecture support in a single image by linking to manifests for each platform,

[5] OCI Image Format Specification: https://github.com/opencontainers/image-spec
[6] runC: https://github.com/opencontainers/runc
[7] OCI Image Format Specification: https://github.com/opencontainers/image-spec

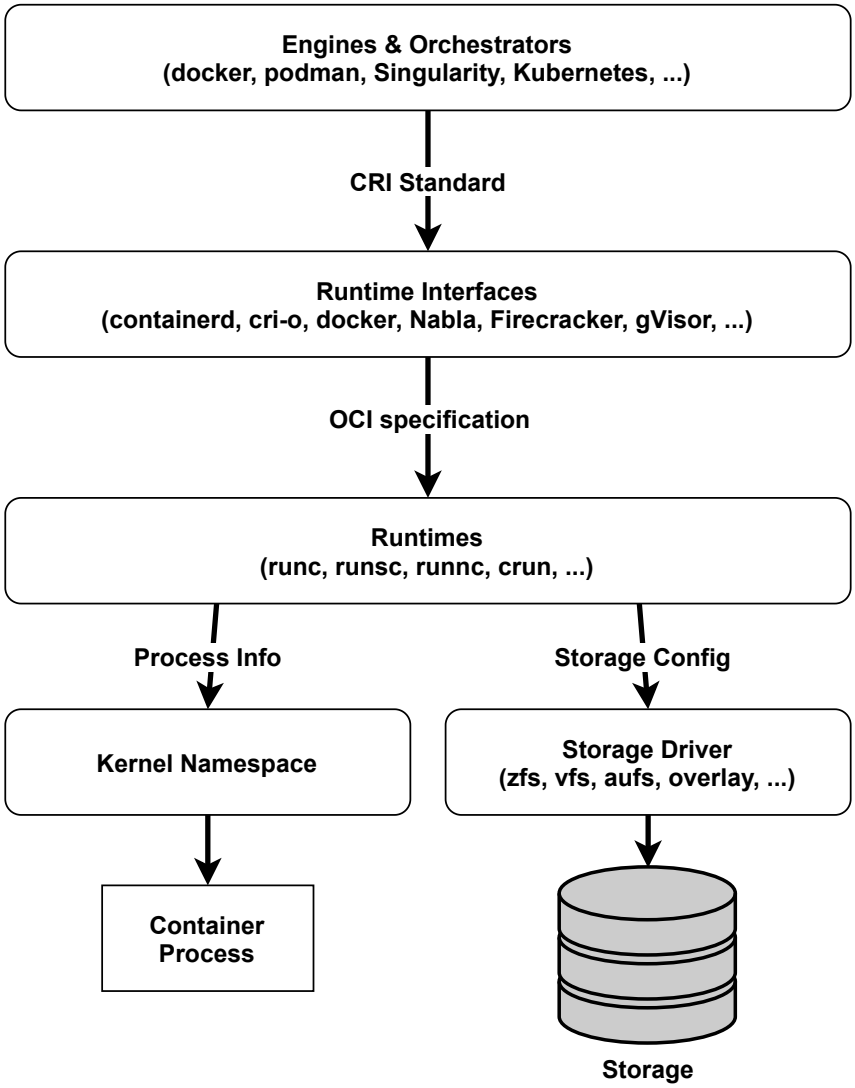

Figure 2.1: Process stack hierarchy of running a container from high to low level.

- and a configuration which is provided to the runtime to create and run the container from the image.

These are the main components for the image. Other components described in the standard are out of scope for this work.

The usage and adaption of an open standard for container images made it possible for a lot of different container systems to be able to use the same containers. This made it easy for users to adapt to the ecosystem since they did not need to choose a specific vendor. They can pick from a pool of vendors and an even bigger set of container images provided by different sources. This unification therefore played a big role in the surge of container applications since there where almost no barriers, neither financial nor ressource wise, to the container ecosystem. The OCI together with the Cloud Native Computing Foundation (CNCF) over time composed a set of companies and applications which build up and integrate into the container and cloud ecosystem. Most of

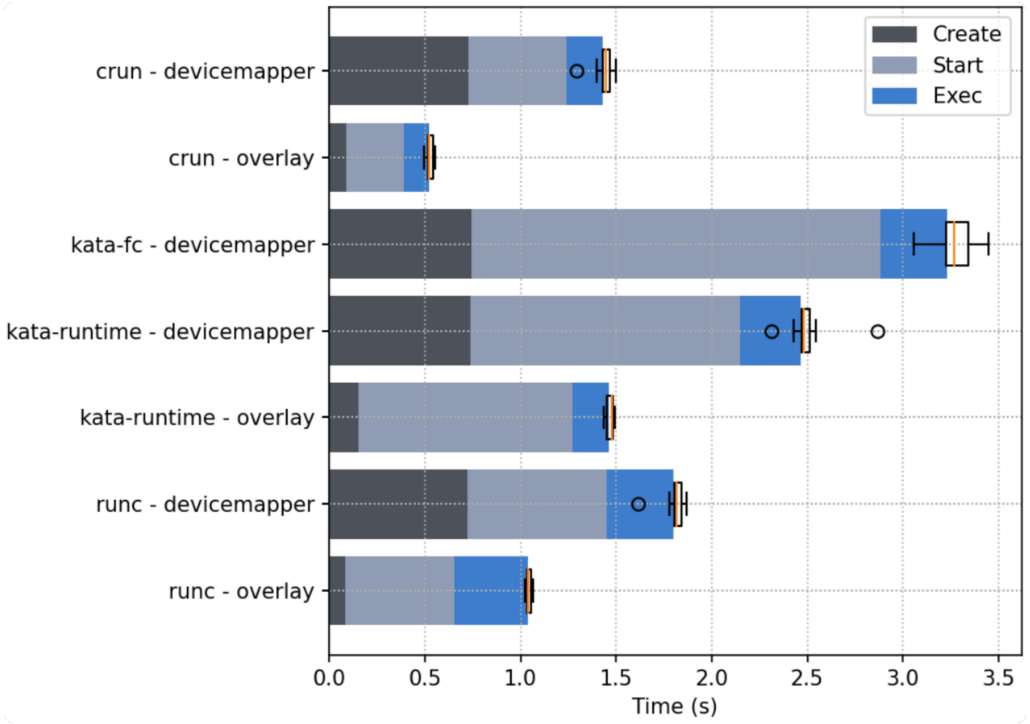

Figure 2.2: Creation, Startup, and Execution time for different container runtimes [VRS20].

them are open-source or free to use while some, especially applications for security or hosting, are closed-source or propietrary. Today the container infrastructure is widely adopted in most fields of web applications and services [Gar20; Red20]. In a professional environment the orchestration is currently gradually evolving to increase efficency and reduce management effort. With orchestration several tasks can be automated and therefore enable high performing wide scaled applications in places where these could not be implemented before. Either becuase of management capabillities or because of existing monolithic systems which can not be adapted.

2.6.1 Orchestrators

The following chapter presents a selection of container and cluster orchestrators. Unique features and ambitions of each orchestrator are shown and compared. The first orchestrator is Marathon[Mes18]. Marathon is a orchestration platform for Apache Mesos[8] and the Distributed Cloud Operating System (DC/OS). Apache Mesos is a distributed virtual kernel which isolates physical hardware and manages the distribution of the undelying hardware to the running applications. With Mesos the cluster setup is also cross-platform compatible and includes a cluster manager. Marathon can therefore not only orchestrate containers but also manage the cluster. Additionally to standard "Docker"(Container Network Interface (CNI) compliant) containers the system can also host co called *Mesos containers*. These can run applications from filesystem which can be archived and like containers configured with storage and environment variables as well as other properties. Another

[8] https://mesos.apache.org/

approach is given by the Nomad [Has21] workload orchestrator. The main task of nomad is to scale different types of workloads onto a cluster. As with Marathon, Nomad can scale different types of applications including containers, batch jobs and also VMs. To achieve this Nomad usses Infrastructure as Code (IaC) to define and setup clusters. The main objective of Nomad is to be as lightweight as possible. Therefore it is not a complete container orchestration suite but integrates with other tools to achieve complete functionality. Additionally Nomad follows a decentralized approach to achieve high scalability. They showed that networks with two million containers spread over various servers on the world can be set up within minutes[9].

One of the most recognizable orchestrators is Docker Swarm [Doc21]. As the name suggests it builds up on the Docker engine. A swarm is composed of a cluster of hosts running Docker engines which form a network. To configure the cluster Docker Swarm uses Docker compose definitions of applications. In contrast to a single Docker instance the application definition is applied to the whole cluster. Docker Compose is not aware of the cluster but only defines application services consisting of multiple containers and how they are linked together. Configurations are written as YAML Ain't Markup Language (YAML) files which contain container, network and volume definitions. If a application is deployed the defined ressources are either created if not yet existing or updated if the definition changed. With this approach a configuration update can be performed intelligently by only updating the parts of the services which need an update. A difference to the previous two orchestrators is the focus on containerized applications only. Thus only those applications can be scheduled instead of a general workload scheduler.

The most common orchestrator at the time of writing is K8s [The21]. It contains the most feature-rich implementation of a container orchestrator and is designed to be a complete orchestration environment. First designed by Google under the name Borg [VPK+15] and then transfered to the CNCF, which was also founded by Google, to speed up and outsource the development. Build upon the Docker runtime as backend service at first the system now is a independent orchestrator using the cri-o runtime[10] which is also managed by the CNCF. The combination of a lot of tools and services into one complex to facilitate all sorts of tasks automatically makes the system very powerful on one side but also highly increases complexity. These combined approach is against the programming principle of having a program for each task. Critizism for this and other missing features like missing version control and mutability [Dav20]. The configuration consists of centrally defined objects which are monitored to react to changes. Similar to Docker Swarm the configuration files are defined with the YAML syntax but with a more object oriented focus. The system includes among other things a DNS server, overlay network, task scheduling, ressource management, configuration management, state management (through key-value stores), failure management and scalability. But K8s does not manage the cluster on itself. The cluster consits of a minimum of one to many hosts which all run K8s. There must be at least one master node which all other nodes connect to. This way they form a network of fixed nodes for the system but nodes can be added at any time if they register themself at a master.

To further automate the cluster setup systems are build around K8s to manage these hosts. One prominent representative is RedHats OpenShift [Red21]. OpenShift integrates K8s into their ecosystem to extend it with additional capabillities. For this they use a modified K8s version which aims to be optimized for continous development and multi-tenancy by adding specific tools

[9] The Two Million Container Challenge: https://www.hashicorp.com/c2m
[10] cri-o runtime: https://cri-o.io

for developers and operators. This K8s instance runs in the background to support the container orchestration on the OpenShift cluster. Multicluster support is also added as well as security and data analysis. As OpenShift is targeted at businesses it is bundeled with additional support and is not open-source but a paid service. A open-source version of OpenShift K8s backend is available under the name OKD[11].

For the prototype detailed in chapter 3 we had the choice to pick from an available orchestration system. There were few requirements to this process for example to support Docker container and autoscaling which is supported by most orchestrators. In the end we decided to use K8s for the following reasons. Firstly it fullfills all the requirements we set to the system. K8s also is a complete solution which makes it complex but it can be set up more easy since all steps are combined. This also avoids the manual configuration of each component to work with each other properly. K8s has a large open community which results in a big pool of information regarding various scenarios. This makes an adoption easier from existing systems. Through the big community many applications are already available to use. This reduces the effort for porting existing applications to a containerized environment since only data and configuration must be ported. For development purposes the K8s cluster can be deployed in a virtual environment to avoid setting up a real cluster to speed up development. For this purpose several tools are being developed to autmatize this process. One tool is Minikube[12] which is directly published from K8s to set up a local one node cluster. A more elaborate approach is given by kind[13] which uses container to simulate an arbitrary amount of nodes. Each container represents a node and all container act as a multi-node cluster which can be set up automatically. The cluster definition is provided by a definition file following the IaC approach. For older applications that are not containerized yet but are instead bundeled into VMs there are virtualization layers like KubeVirt[14] which integrates these in the K8s enviornment. An alternative to K8s is Docker swarm as a competitor with comparable targets. But at the time of writing Docker swarm still lacks features which are available in K8s and is not as widely adopted in the industry. This prevents Docker swarm from being a viable alternative to K8s at the moment. K8s clusters can be rented from several big cloud computing providers and are thus widely available as PaaS or SaaS with a specific deployment.

2.6.2 Kubernetes

This section discusses a subset of concepts of K8s needed for the development of the prototype in chapter 3. Kubernetes offers implementations for many general concepts. One example is the Horizontal Pod Autoscaler (HPA) which is a rule engine for replicating pod instances from a given metric. The main components of K8s are depicted in Figure 2.3[15]. The system is split into a control plane and a number of nodes also called workers. The control plane components run on a worker or on a separate control plane node and can also be distributed among multiple nodes. It is preferred to run the control plane components separate from load to prevent any disturbances due to high load on the node. The control plane again consists of several components which coordinate the orchestration.

[11] OKD: https://www.okd.io
[12] Minikube: https://minikube.sigs.k8s.io
[13] kind: https://kind.sigs.k8s.io
[14] KubeVirt: https://kubevirt.io/
[15] K8s components: https://kubernetes.io/docs/concepts/overview/components/

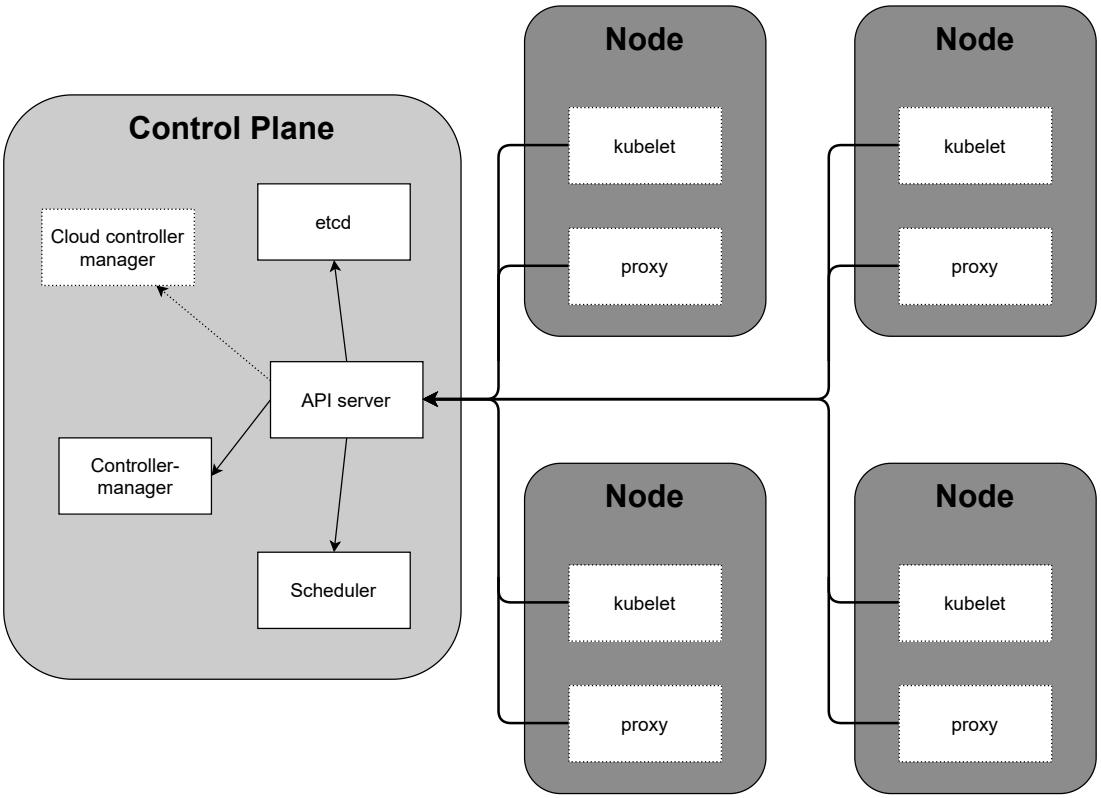

Figure 2.3: Components of the K8s control plane.

The Scheduler is a watcher which assigns containers to a node for execution while taking into account specific restrictions like resource requirements, policies and (anti-)affinity settings. etcd[16] is the key-value store containing all cluster specific data such as deployed objects and thus needs to be backed up regularly to rebuild the cluster in case of a failure. The controller manager consists of set of controllers which trigger actions when the state of a specific object diverges from the desired state to achieve the desired state. Examples of controllers include the node controller which monitors the status of all nodes in the system or the job controller which runs one-time tasks. The API server is the central instance for other nodes and administrator tools to communicate with K8s. It exposes the API to configure all objects in the system via Representational State Transfer (REST) operations. An optional component is the cloud controller manager which communicates with the specific cloud provider's interface. It manages the state and usage of nodes, routes and load balancers provided by the cloud.

The worker nodes consist each of a kubelet and a proxy module. The kubelet agent on each node ensures that the container scheduled on each node are running and healthy. It receives the specifications of container bundles from the control plane scheduler. The proxy module sets up and maintains the network rules for each node which spans up the overlay network. This allows intra-

[16] etcd: https://etcd.io

and internet communication for the container. If available the OS packet filtering and forward (e.g. iptable) is used. Additionally each node needs to run a container runtime on which the container are deployed. K8s currently supports three container runtimes: Docker, cri-o and containerd[17].

In addition to these standard components K8s provides several addons to add features to the cluster. Examples for common addons are a DNS server to reach all services by name, a UI Dashboard for monitoring, logging or generating metrics for cluster components. The following section describes a subset of core resource objects from K8s which are used for the prototype in chapter 3. All resources inside of K8s are defined as objects with specific names. These objects have well defined properties and can are often nested within each other. The resource object definitions and descriptions are defined in the API reference [18]. Figure 2.4 shows the objects and their logical dependencies. Arrows indicates the inclusion of the targeted object one or more times. The central object is the *pod* as the most basic unit with the finest granularity. A *pod* contains a set of containers which are logically coupled. For a container in a pod all other containers in that pod seem as they are running on the same machine. They can communicate via the loopback adapter. The *pod* is assigned an internal IP and therefore all containers in that pod share their network port space. Pods are a way to bundle tight coupled processes or applications into one unit. There are several workload sets which can be constructed from a combination of pods.

The *ReplicaSet* is a mechanism to ensure a specific amount of pods running. It contains a selector to identify the pods it acts upon and a template for new pods which it creates if the number of pods found with the selector is lower than the requested amount. This object is rarely used directly but instead is created automatically by a *Deployment*. Deployments provide additional operations and are typically used to deploy an application. The deployment automatically performs updates of the underlying pods in a rollout fashion by replacing one pod after another. This leads to an incremental update without downtime of the application since at every moment it is at least partially available. If a failure occurs in a new version it can also be "rolled back"to a previous version in the same fashion. Another property of deployments is that they can be scaled by an arbitrary factor to cope with an increasing load. A further type of sets is the *StatefulSet* which is the same as a deployment but for stateful pods which need for example persistent storage. The last set is the *DaemonSet* which ensures that an instance of the selected pods is deployed to each node by default. The set of nodes can be specifically reduced by setting constraints on ressources or other specific properties. The last workload set is the *Job* which runs a set of pods once until they complete. If they fail the job will try to rerun them a certain amount of times until they succeed. For periodic jobs the *CronJob* schedules pods using the same syntax as the cron scheduler[19]. This can be used for example to make backups at a time where the system is not likely to be stressed.

There are several types of storage that can be assigned to a pod each for a special use case. A wide variaty of sources for the storage is provided including local, Network File System (NFS) and various cloud providers. The first is the *Volume* which provides a volatile storage for the lifetime of a pod. If a pod or the node it is running on fails the state inside the volume is preserved until the pod is restarted. For persisten storage the *PersistentVolume* is provided which claims storage beyond the lifetime of the pod using it. Persistent storage is provided by the administrator for each storage source. A pod can choose the type of storage it prefers or let the system choose an appropriate

[17] containerd: https://containerd.io
[18] K8s API reference: https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.21/
[19] GNU mcron: https://www.gnu.org/software/mcron/

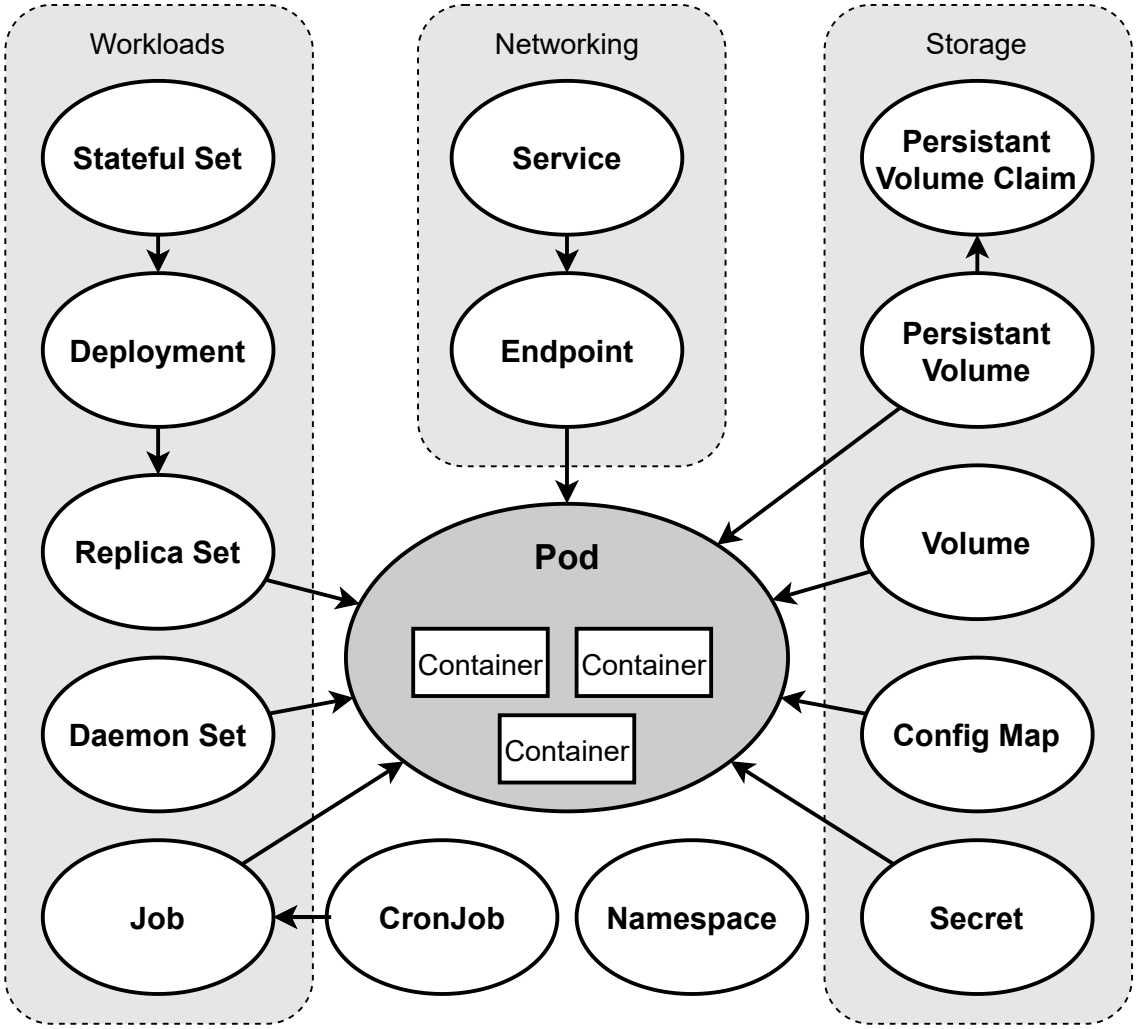

Figure 2.4: Overview of relevant K8s objects and their relations.

storage for examble restricted by the requested amount of storage. This type is most useful in combination with statful sets since a recreated pod can be reassigned to a storage by keeping the same id which is ensured by the stateful set. To provide configuration files or parameters to an application the *ConfigMap* can be used to define and inject configuration information into pods. A config map can be mounted as a volume which creates files for all data structures defined in the map. Single values can also be defined and added to the pod as environment variables. This seperates the configuration from the executables while also eliminates the need for a seperate persistent volume for the configuration. Config maps do not provide security for sensitive information such as passwords, keys and tokens. For these information a special type of config map the *Secret* is used. It stores sensitive information in encrypted form and is only decrypted when deployed inside a pod. Like config maps secrets can be mounted as files or environment variables.

With worload sets and volumes a pod can operate reliably on its own but it is missing connections to other pods or external services. This is accomplished by *Services* which defines a logical set of pods which is determined by a selector. Services expose a list of ports from each pod it selects

so that they can be accessed dynamically. The set of selected pods is updated automatically in a separate *Endpoint* object so that if a pod which matches the criteria is created or removed the set is updated accordingly. A unique IP address is assigned to each service as well as a domain name if a cluster DNS server is installed. K8s also injects the service information into each pod via environment variables. This enables all pods to communicate over these services which can be discovered by the specific name of the service either via hostname or environment variable. A request to the service is forwarded to one of the pods selected by the service. The algorithm for the forward selection is defined by the mode of the kube-proxy. By default the round-robin algorithm is used to distribute traffic but other methods including random, hash-based or least connections scheduling are also available. There are four types of services:

- ClusterIP: This gives the service an internal IP only. Thus it is accesible only from within the cluster.
- NodePort: Exposes the service on all nodes over a specific port to enable access to the service from outside the cluster.
- LoadBalancer: Uses a external load balancer to expose the service and forward load.
- ExternalName: Creates a service pointing to an arbitrary hostname instead of pods.

ClusterIP is the default type which is used for internal communication. *NodePort* is used in development to create access points to the cluster. In a production environment *LoadBalancer* is used to harness the efficiency of external load balancers provided in the cloud. *ExternalName* services can be used to add external ressources to the cluster in a unified fashion. When the external hostname changes only the service definition needs to be changed.

For automatized scaling of deployments or respectively replication in K8s the HPA is used. The HPA object defines a minimum and maximum number of replicas the metrics to scale on and the target pods to scale. The target refers to the object to which the scaling should be applied to. It consists of the type and the name of the object which uniquely identifies the object. One or more metrics can be used simultaneously to scale on each one of these metrics. The metrics refers to either a metrics bound to a pod or another K8s object or an external metric not referring to any internal object. A metric is defined by name and target. The target defines the desired value of the metric which is compared with the current value. The current value is calculated as average between all targets or percentage if limits for the metrics are defined.

Additionally K8s allows to add custom resource objects to extend the API. This is used by the newer concept of operator patterns to further automize the management of components.[20] Through the added operations it is possible to automate certain use cases and resolve failures automatically if they follow known patterns. The functionality can also include repeating tasks in the cloud like backups or updates checks on applications. This brings functionality which was previously provided by cronjobs or other task scheduling engines to the cloud and further reduces management overhead.

[20] K8s operator pattern: https://kubernetes.io/docs/concepts/extend-kubernetes/operator/

3 Prototype

3.1 Implementation

3.1.1 ECM-Components

The containerized reference implementation splits the ECM application into four components. First the central catalog or Library Server DB (LSDB) which contains all data and configuration of the services. This container hosts a DB2[1] instance with stored procedures containing the logic. Then the Ressource manager DB (RMDB) which contains an object catalog with physical file metadata corresponding to the file in physical storage. The third component is a Java™ 2 Platform Enterprise Edition (J2EE) Ressource Manager Application (RMApp) which manages the file ressource access by coordinating the retrival and storage of files from the storage with metadata from the RMDB. This RMDB metadata contains for example the path to file and the size as well as the last modified timestamp. The last and main component is the WebSphere Application Server (WAS) which hosts the ECM client web application. In this case an instance of the *IBM Content Navigator (ICN)* is used. This is the only component directly accessed by the user as a webservice. For administrative tasks the RMApp container also hosts a webinterface.

These four components are loosely coupled to each other. Connections and dependencies are further illustrated in As the main component the WAS depends on all other components to run. It fetches user- and configuration data from the LSDB container and ressources from the RMApp container. The RMApp container uses the RMDB container for requests on specific documents and files by using the metadata of the files for filter and search operations.

3.1.2 Translation to Kuberenetes

Since K8s builds up on the concept of containers for their system the ECM components can be translated to the Kuberenetes eco-system. This process is also part of another master thesis developed in parallel to this thesis.For each container a pod is designed with the same properties as the container. This pod is then embedded into a deployment which holds further properties from which the most relevant for us is the option to set a arbitrary number of replicas for the contained pod. This also gives us the opportunity to add one or more containers to a deployment to add more functionality or to further split up the application into more tight coupled containers.

Another important difference to the Docker concept is the use of storage. In Docker a container is assigned a specific part of a storage system upon creation with a volume or file system mount. In K8s the pods are created and destroyed automatically on different nodes and can therefore not rely

[1] IBM DB2: https://www.ibm.com/analytics/db2

on a specific storage location. K8s therefore extends the concept of volumes to various mediums which are available on every node in the system instead of just local storage. For a list of all currently supported types refer to the K8s documentation[2]. Persistent volumes with specific amounts of storage are used if the storage needs to be preserved.

Therefore it is essential to identify stateless and stateful pods. Stateless pods are pods who do not depend on information which needs to be stored permanently. This would include files like those stored and managed by the RMApp. RMApp is therefore not a stateless but a stateful pod. Stateful pods need to share their data with their replicas to be independent. This can be done by sharing storage or via synchronisation. Thus stateless pods can be scaled easily by assigning enough temporary space to the pods at startup while stateful pods need more initialization and maintenance. Stateless pods are therefore preferred as they can created and destroyed easily and so also easily transferred to other nodes.

Another concept that needs to be accounted for is the access on the pods from outside of K8s by a client. With propably more than one pod instance running the inbound traffic needs to be load balanced to divide the load from a single instance. Therefore pods are combined into services which can are handeled automatically. The port mappings are defined similar to the port forwardings in Docker to make the access to application equivalent. All requests to the service will be load balanced between pods by the internal proxy. This serves the purpose of this prototype in terms of performance. In production deployments an external load balancer can be used to increase performance.

3.1.3 Monitoring

To load balance each component of the application they must be tied to corresponding metrics which subsequently must be gathered and organized. For this task we use the Prometheus[3] system. Prometheus is a monitoring and alerting system which gathers metrics from different sources in a standardized format. The metrics are provided by each source separatly with so called *exporters* which present the applications metrics via a HTTP rest interface. In Prometheus the process of gathering metrics is configured by so called *jobs* for each source which needs to be scanned for metrics to import. For each job transformations can be applied to for example rename or ignore some parts or attach metadata to metrics. The configuration also allows for modifying the scan interval to reduce overhead for slow changing metrics.

The metrics are then stored in a temporary time series database for a limited time after which thry are deleted from the temporary database. To store metrics permanently Prometheus can be coupled with an external time series database for long time storage of historic data. Evaluation of stored metrics uses Prometheus' own query language *PromQL* which is especially designed for time series data and metrics. For data analysis there is a adapter to Grafana[4] to visualize PromQL queries. The visualizations help in analysing and live monitoring a big amount of datasources at once on a single dashboard.

[2] https://kubernetes.io/docs/concepts/storage/volumes/
[3] Prometheus: https://prometheus.io/
[4] Grafana: https://grafana.com/

Prometheus also includes an alertmanager which handles sending alerts on specific conditions over several channels. The simplest being E-Mail but more sophisitcated solutions like Slack and a general HTTP POST request are also configurable. This informs system operators of irregularities or general events that are important like for example a nearly filled storage or a potential attack. The handling of alerts is out of scope for this thesis but is highly recommended for production application as it further automatizes the monitoring.

3.1.4 Metrics export

To use dynamic load balancing the state of the system must be monitored as detailed as possible. The limits are set by either the applications or hardware which may not make all metrics accessible or through performance losses from the interference. Since K8s is designed to already work with metrics the base components all already present their metrics. These can be fetched over the K8s Metrics API[5] which presents information over the control plane, the running pods and more. For information about the nodes in the network a *Node Exporter*[6] pod is added to every node with a daemon set. These pods generate hardware metrics for each node including CPU, memory, file I/O and network I/O which can be used to load balance between all nodes in the cluster. Custom pods must provide their own specific metrics on a per pod basis. For this purpose an existing webserver in a container can be used to publish the metrics. If none is available or can not be used, either a small webserver can be included in the container or a sidecar container with the metrics webserver can be added to the pod. For applications which are designed for deployment via an automated system it is more likely that metrics are already included and are or can be activated. If not, a custom script or program must be added or developed if none exists to fetch, preprocess and format the metrics appropriately.

In our case there is already an exporter for DB2 databases[7] which can fetch and publish results of SQL queries defined in the configuration. Since performance metrics of the DB can be queried with SQL there is no need for a special program connecting to the API of DB2. For Java applications like the WAS and RMApp there is a JMX exporter[8]. Unfortunatly this exporter does not expose the information neeeded to determine the ECM system state. Therefore we use a custom exporter based on generic logfile parsing. Promtail[9] is part of the log organisation system *Grafana Loki*. It can parse logfiles from various sources and prepare them for a centralized log management. For our use case we disable this functionality and only work with the option to generate Prometheus compatible metrics from the parsed logs. With this technique it is possible to add a complex logic to filter out relevant information without the need of designing a new program for each application.

[5] K8s Metrics API Definition: https://github.com/kubernetes/metrics/blob/master/pkg/apis/metrics/v1beta1/types.go
[6] Node Exporter: https://github.com/prometheus/node_exporter
[7] IBM DB2 exporter: https://github.com/glinuz/db2_exporter
[8] JMX exporter: https://github.com/prometheus/jmx_exporter
[9] Promtail: https://grafana.com/docs/loki/latest/clients/promtail/

3.1.5 Scaling

With the HPA K8s includes its own implementation of a rules engine for scaling components in their eco-system. It scales *deployments* on the basis of metrics provided by or to K8s. The algorithm to determine the number of required instances is shown in Equation (3.1). [10] The formula shows that the HPA uses a simple linear adaptation based on the metric values target and the current value as well as the current instances. For many applications this is sufficient but more sophisticated approaches can further improve the utilization and responsiveness [NMNA12]. Additionally there are a number of parameters to tweak the performance of the HPA scheduling. For example the evaluation interval, the maximum and minimum number of instances or the timeout after creating or deleting new instances. With these parameters the HPA can be fitted to more applications.

$$targetInstances = \lceil currentInstances \cdot \frac{currentMetricValue}{targetMetricValue} \rceil \tag{3.1}$$

K8s has three types of metrics it can use to scale pods on. *Ressource metrics* which are automatically provided by K8s. Currently supporting CPU and memory utilization. *Custom metrics* are metrics for properties of custom components like specific applications. These first two metrics are linked to objects in the K8s ecosystem. The third type are *external metrics* which are not linked with K8s objects and thus relate to external components. When using *External metrics* security needs to be considered because these metrics are fetched from outside and are thus not protected by the K8s security layer.

One drawback of the HPA is that it can only scale reactive and not proactive. As discussed in 2.3 this creates a period of time in which the requirements are not fulfilled. Proactive scaling can be simulated by adding prediction logic to the metrics before they are passed to the HPA. This logic can predict future load and adjust the output so that scaling is applied prematurely. Such a logic can for example take into account the acceleration of changes in a specific metric. This can also be used in monitoring to optimize the algoritm discrepancy.

Another approach is to abandon the HPA and use a separate tool to access the replication APIs directly. This involves more effort and potential less integration with the K8s ecosystem. The advantage of the direct access is the freedom of choice for the algorithm to determine and set the number of instances. For example AI can be used to train a model with the metrics an apply predictive scaling. Through the open interface definition and modular structure of K8s different services can be used or bypassed and replaced with own solutions.

3.1.6 Scheduling

The HPA decides when new pods are scheduled dependent on the metrics provided. When deploying new pods the scheduler must decide on which physical node it schould be created. For this purpose K8s can attach metadata to nodes and criteria to pods which can be used by the internal scheduler.

[10] HPA algorithm: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/#algorithm-details

When a pod needs to be scheduled the available nodes are scanned for nodes which meet the criteria. Then a node is chosen based on the utilization and already scheduled instances on these nodes. This allows for fine-grained control over scheduling location and distribution. The behaviour of the HPA can therefore be configured extensively.

3.2 System Design and Setup

To simulate a realistic scenario without the need for the actual infrastructure to be in place a simulated environment is used. Figure 3.1 shows the system with the basic component blocks. To make testing as simple as possible the system runs on a single host which is itself a VM. A VM can be created, deleted and copied easily by taking a snapshot of the current system. This allows for rapid testing of different configurations by simply copying the existing one without the need to set up the machine from the ground. On this host runs a Docker engine with several containers simulating real nodes and thus creating a multi-node system of virtual nodes (vNodes). A K8s cluster containing all vNodes as nodes is deployed. The cluster hides the underlying container structure from applications running inside with the internal overlay network. The cluster contains three main components:

1. The control plane managing all K8s components.
2. The ECM application consisting of several components
3. and the monitoring components.

Each component is independent of the others in terms of running the components. If there is no monitoring system the ECM application will run without load balancing and the monitoring system will just idle until the ECM application is there. This removes the constraints of relying on a fixed sequence of creation and destruction of components. The remainder of this chapter will further explain these components and how they interact.

3.2.1 Virtual Nodes

To simulate the multi-node cluster with vNode we use kind[11]. kind uses Docker container to setup a cluster of K8s nodes. A configuration file is used to specify the amount of vNodes and parameters on each node as well as parameters for K8s. This allows for example to give each container and therefore each vNode a specifc set of resources to simulate smaller and bigger nodes. The downside is that that the overall performance and ressources are limited to those of the host system with this simple configuration. By adding own certificates used for communication between K8s nodes it is possible to add other nodes to the cluster. Although the cluster is deployed this way it is still a standard K8s cluster and thus works the same as one created manually. The cluster is therefore equivalent with a real multi-node system.

For our testing we use a system with five vNodes as depicted in figure 3.2. Four worker nodes where the monitoring and ECM applications are running and a control-plane node for the control-plane only. This separates the performance impact of K8s components from the rest of the system allowing

[11] kind: https://kind.sigs.k8s.io/

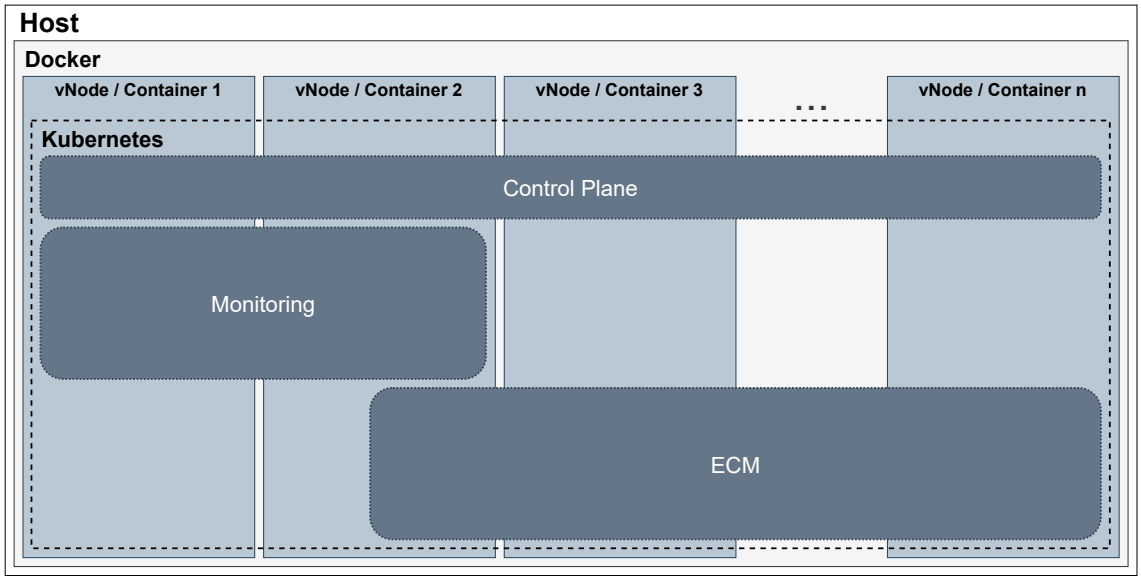

Figure 3.1: General system overview showing the K8s component blocks and how they are allocated.

for a more precise measurement. Listing 3.1 shows the configuration used. It creates n worker nodes without any ressource restrictions and a control plane node. The control plane node has additional port mappings to enable access to the cluster from the host node. In a production environment these access mappings are included in an external load balancer to serve applications on their standard ports (e.g. 443 for HTTPS).

3.2.2 ECM Deployment

The ECM application consists of several components which are separated into four pods. Figure 3.3 shows the structure of those pods. Dark grey boxes represent pods while light gray boxes represent containers and white boxes actual processes running inside the container. Each pod is deployed in a deployment which enables automated replication. The RMDB and LSDB pods have the same structure. Both contain an instance of a DB2 database accompanied by a exporter process which publishes the metrics of the database. RMApp and ICN are also similar. They each contain two container, one containing the WebSphere application and another sidecar container. The sidecar container contains a NGINX webserver and a promtail instance. Every access to the WebSphere application is routed through NGINX as a reverse proxy. NGINX records all aspectes of a request in a logfile which is further read by promtail. This data is then accumulated, processed and further published in the Prometheus metrics format by promtail.

The storage needed for the LSDB and RMDB pods is provided by two persistent volumes since the database needs to be preserved. Replicating of databases is out of scope for the tests performed here because it would require a level of effort which does not fit in the timeframe of this work. There are dedicated systems to deploy specific databases into the cloud while allowing for horizontal scaling

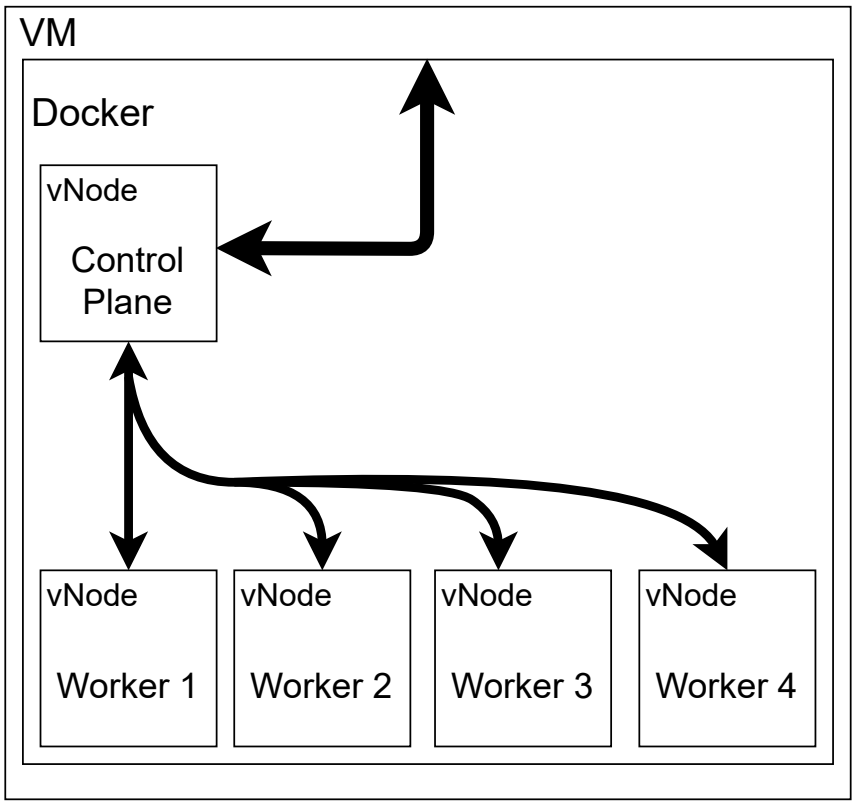

Figure 3.2: kind cluster on vNodes used for testing. Arrows symbolize data exchange.

like Vitess[12] for MySQL. But the deployment of databses into the cloud is still researched heavily and not generally advisable. To keep the data local and the setup simple a local NFS server is used as the storage ressource. The locality also helps in eliminating bottlenecks through communication over the internet. This also resembles a real deployment on a cloud cluster where the storage is often also provided by the cloud computing provider. Therefore the storage is optimized for usage with the computing cluster and propably local to the cluster.

The data flow is depicted in Figure 3.4. Tenants enter the system through the ICN web application where they perform operations. If an object is requested the metadata are fetched from the LSDB together with the ID of the file in the RMDB. The ICN can then request the object from a RMApp via the ID. The RMApp uses the ID to gather the location and other related information like size or last edited date from the RMDB. With the location it can then fetch and return the object from the object storage.

3.2.3 Monitoring Deployment

Figure 3.5 shows the relevant parts of the monitoring system which also consists of four components. First Prometheus who manages the collection, storage and access of all metrics. Prometheus depending on the load prometheus replicates itself to keep up with demand. But due to the small

[12]Vitess: https://vitess.io/

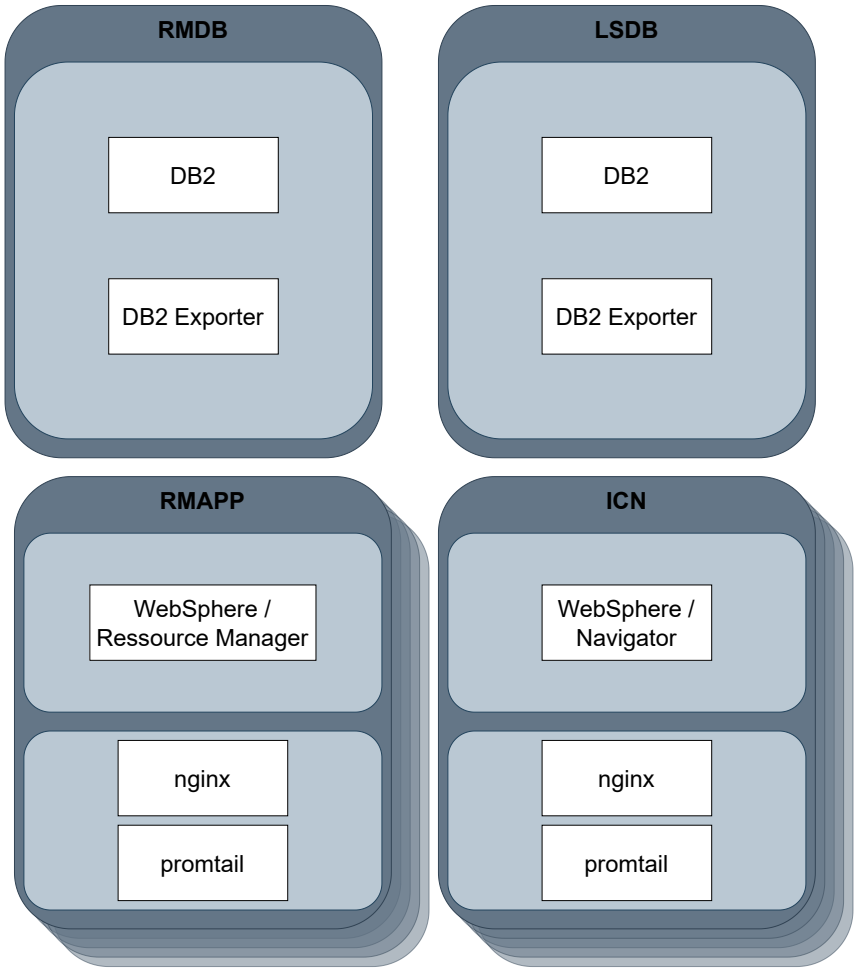

Figure 3.3: ECM component structure in K8s. Dark grey boxes represent pods, light gray boxes represent containers, white boxes represent processes.

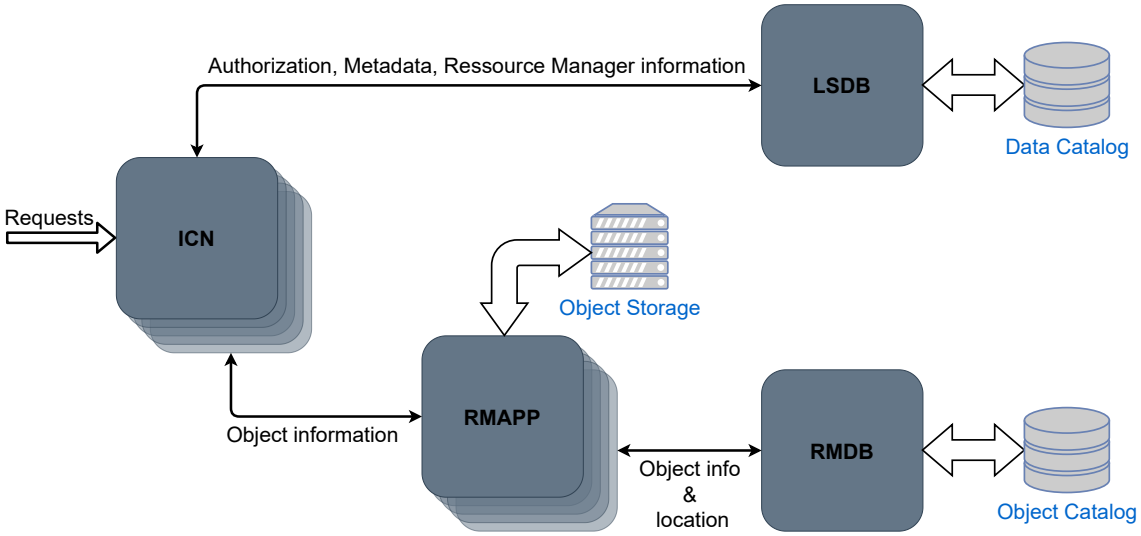

Figure 3.4: Data flow between ECM components.

Listing 3.1 kind configuration file in YAML format. Creates 5 nodes and several port mappings.

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
name: ecm
nodes:
  - role: control-plane
    extraPortMappings:
      - containerPort: 30000
        hostPort: 50000
      - containerPort: 30001
        hostPort: 50001
      ...
      - containerPort: 30113
        hostPort: 9063
      - containerPort: 30180
        hostPort: 9064
    kubeadmConfigPatches:
    - |
      kind: KubeletConfiguration
      authentication:
        webhook:
          enabled: true
      authorization:
        mode: Webhook
    - |
      kind: ClusterConfiguration
      controllerManager:
        extraArgs:
          bind-address: 0.0.0.0
      scheduler:
        extraArgs:
          bind-address: 0.0.0.0
  - role: worker
  - role: worker
  - role: worker
  - role: worker
```

impact on performance from the ECM application tested there is no need to scale the monitoring system for testing. The second component is the Prometheus adapter who fetches the metrics from Prometheus and provides them to K8s as custom metrics. The node exporter is the third component. An instance of the node exporter is running on each node of the cluster to fetch machine metrics for this node. This uses the *DaemonSet* deployment form of K8s. The fourth and optional component is Grafana as a custom Dashboard provider for better overview and maintenance.

The dataflow for the monitoring components is depicted in Figure 3.6. Prometheus as central component coordinates all operations on metrics It actively fetches (pulls) metrics from the node exporters and the ECM components. Fetching metrics is semi automated with so called *Service Monitors*. These check the K8s API for Services with a specific set of labels. If they find a service who matches the specific labelset they add them to the pull configuration of Prometheus. This way

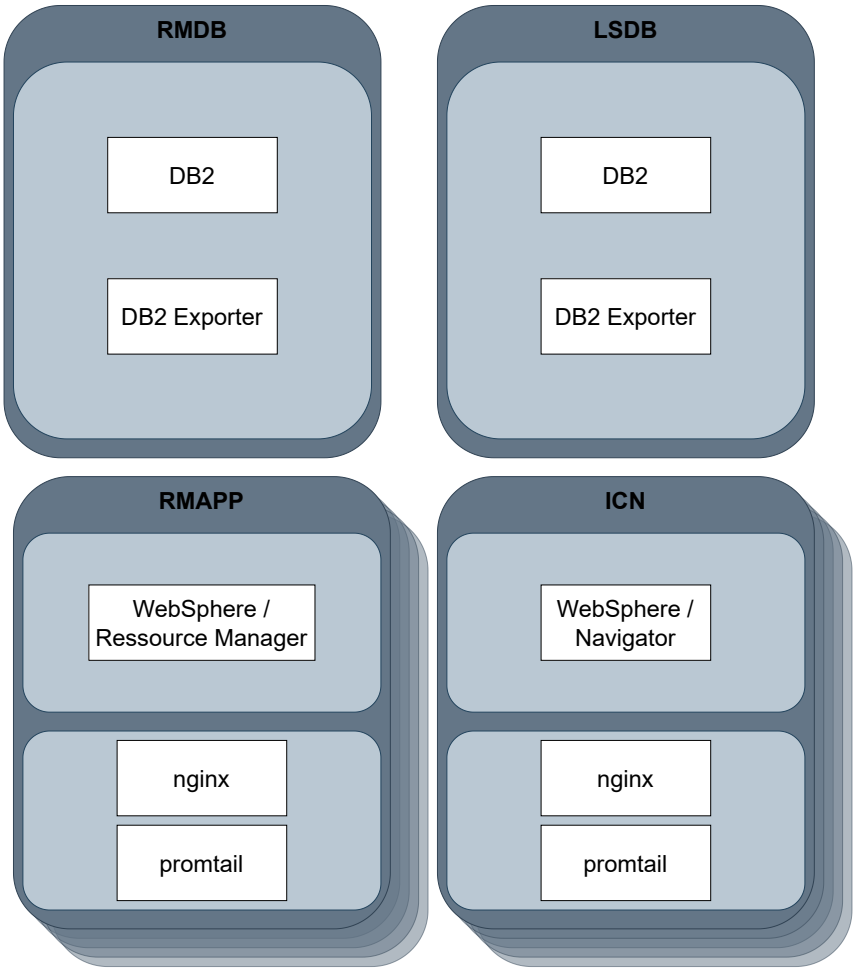

Figure 3.5: Monitoring component structure in K8s. Dark grey boxes represent pods, light gray boxes represent containers, white boxes represent processes.

it is not necessary to specify each Service to pull metrics from but instead define groups of Services. The *Service Monitor* includes a specification for the request to fetch the metrics. So as long as all Services publish their metrics the same way they can all be gathered automatically.

The Prometheus adapter can then be used to define custom metrics for K8s. A custom metric consists of a query on one or more metrics. By using the PromQL query language it is possible to combine metrics or apply other transformations on the metric. This includes for example a gradient analysis which can be used to detect a sudden increase or decrease of a value. Grafana is using the same query language to create graphs and charts. These can then be arranged and customized to create dashboards representing different aspects of the system. Because Grafana is not an active part of the monitoring system it is optional. The main purpose is for development and adminitration where it can help to find and isolate problems or detect long term changes.

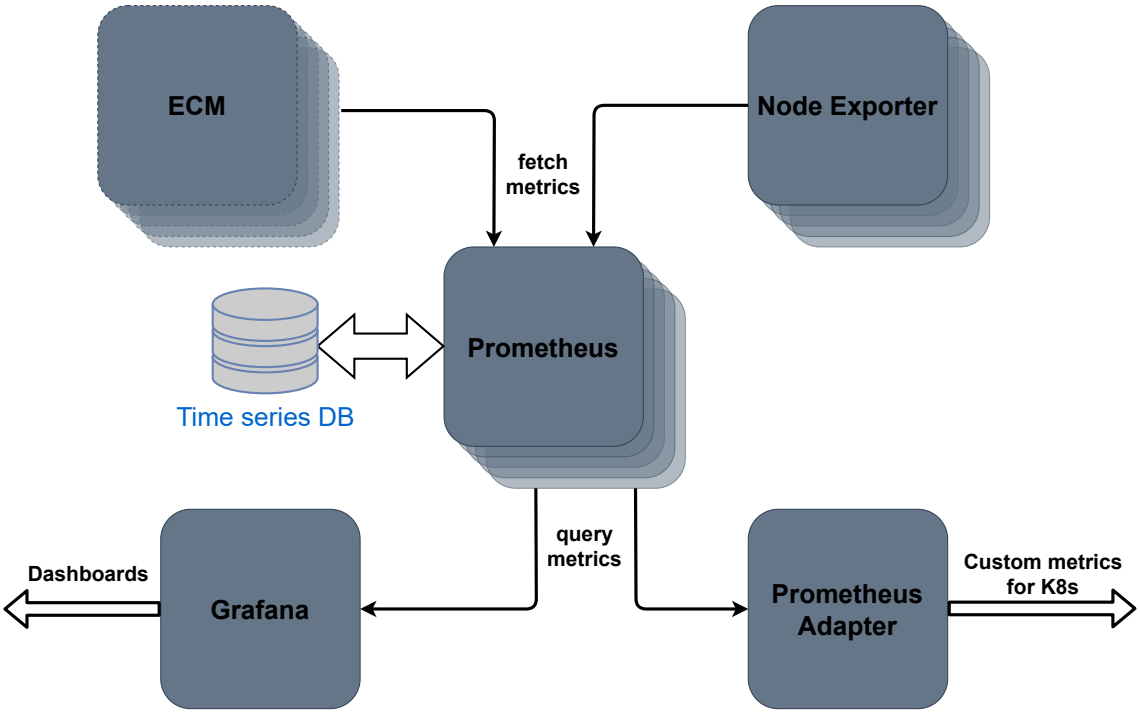

Figure 3.6: Data flow between monitoring components.

3.2.4 Networking / Port Mappings

The networking is done through different layers each having a different scope. An overview of the networking can be seen in Figure 3.7 There are four separated layers of networks which are separated through different mechanisms but mostly by packet filters and Network Address Translation (NAT). In the innermost layer are the pod networks where each pod uses its own portspace tied to the IP address. From here we use K8s' services to map the ports to ports on the control-plane where the IP address is also linked to a domain name. To access the application from outside of K8s NodePort services are used to publish the ports on dedicated ports on all vNodes. Since the vNodes are hosted as Docker containers we use the Docker port forwarding to make the ports available on the VM running the cluster. In the pod layer the ip addresses are bound to the pods and therefore port numbers can appear repeatedly but only once per pod. On the control-plane layer ports are bound to the IP address of the service and so each service can has its own port number range. In production the outer two layers are not used and instead a external load balancer is connected to the services on the control-plane. The external load balancer is bound to an IP address and further to a domain name under which it can be accessed from the internet.

The port mappings on each layer allow for a fine grained structurization and grouping of ports. For example specific ranges can be allocated to service, inner communication and publicly available ports. Specific ports can also be blocked on different layers which are protected by security mechanisms to control access separatly.

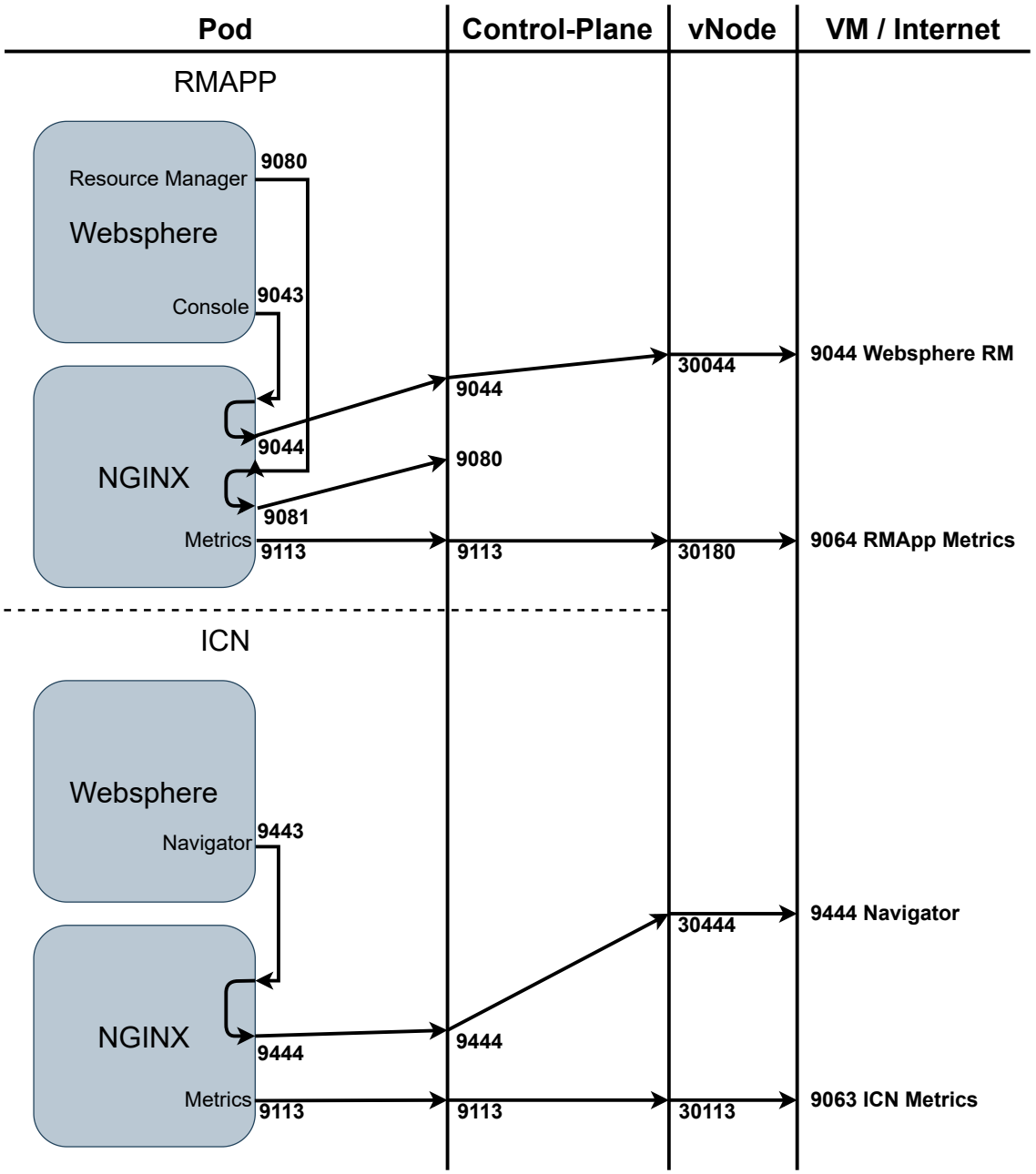

Figure 3.7: Network layers and port allocations from container level up to the VM running the cluster.

3.2.5 Metrics

The system state is provided by several custom metrics introduced by the ECM application. These come in addition to the standard metrics provided by K8s and general custom metrics by the node exporter. All metrics are centrally collected in Prometheus where they can be used by the Prometheus Adapter. The Adapter configuration is provided with queries to generate K8s metrics for the HPA which uses them to scale the deployments.

We divide the merics in three groups. General metrics (1) which are provided by K8s or the metrics pipeline. These metrics are mainly relevant for cluster monitoring and general administrative tasks. It is possible to monitor and even scale applications based on these metrics like for example CPU usage. But because they are superficial the status can only be imprecisely derived and does not give a deep insight.
As second group the Sidecar metrics (2) are metrics dedicated specifically to an application. They are gathered by observation of the application from the outside which is often done by adding a sidecar container to a pod. The application is considered a black box with well-known interfaces which can be targeted by explicit querying or interception. Due to knowledge of the application type these metrics are more precise than general metrics. They have potential access to every information published by the application. This also marks the limitation of these metrics. As they can only ”observe” the application it is not possible to gather internal information. This is relevant especially for older applications which are not designed with cloud and containerized deployment in mind. In this case relevant metrics may not be accessible from the outside.
Application metrics (3) as last group are the most sophisticated which are directly integrated into the application. As such they provide the most insight and fine-grained information. Most applications which are developed with cloud deployment in mind provide these type of metrics by default. Examples from the prototype are the components from the monitoring pipeline which provide metrics for themselves. The following paragraphs describe metrics available in the prototype from the specific groups.

General Metrics

Standard metrics are provided by K8s by default for CPU and memory usage for each pod. It also provides metrics for the internal Domain Name System (DNS) and API server. This helps in identifying bottlenecks on network resolution or the K8s API for big clusters. Also generally available are metrics of each kubelet which allows for monitoring each node on basic metrics. Examples for this are the number of pods and containers running on the kubelet and the operations occuring like creating or restarting a pod.

As an extension of these metrics the node exporter provides more precise information about the underlying hardware. The metrics include detailed information about the status of the network, storage, filesystem, memory and cpu. K8s can use these metrics to make better scheduling descisions when placing pods accross nodes. To use this feature specific pods need to be tagged with metadata labels. For big clusters these metrics give good insights for adminitrators on load distribution and bottlenecks.

Variable Name	Description	Example
$time_local	Timestamp of the request (ISO-8601 date & time)	2021-08-28T17:23:20+00:00
$request_method	HTTP request method	GET, POST, HEAD, PUT, ...
$request_uri	Path, Query and Fragment of the request URL	/index.html?go=1#footer
$status	HTTP status code number	200, 307, 404, ...
$body_bytes_sent	Length of the response in bytes	12453215
$request_length	Length of the request in bytes	567456153
$request_time	Request processing time in seconds	0.404

Table 3.1: Individual variables gathered for each request intercepted by NGINX.

Sidecar Metrics

As at least parts of the ECM application is proprietary software we intercept the datastream to gather metrics. This is achieved by adding a NGINX webserver as reverse proxy as shown in Figure 3.7. This way all requests can be analyzed on typical parameters included in the Hypertext Transfer Protocol (HTTP). NGINX exports statistics as logs in the JSON format. From there they are parsed by promtail and published as Prometheus metrics. Table 3.1 shows the individual variables fetched by NGINX. For a more detailed description of all available variables refer to the official documentation.[13] The *request_method* and *request_uri* are used as metadata to label the generated metrics with. This allows to identify which requests or group of requests contribute most to the overall number of requests. The *status* variable identifies successful and failed requests. Typically all status codes below 400 are considered successful while codes greater or equal to 400 are considered failed. This helps in identifying either a failure of the application or a potential Distributed Denial of Service (DDoS) attack. *body_bytes_sent* and *request_length* represent the traffic in each direction. These values exclude the header length but this is negligible for most requests which include sending or receiving data apart from HTML content. The *request_time* is one of the most important values as it indicates the response time of the application. This can be directly related to user experience and Service level agreements and therefore used to adjust the scaling.

Application Metrics

To get a deeper and more accurate insight into the processes inside the application we also want to generate application metrics. These are directly generated by the ECM system. As it was not possible to generate the metrics directly in the Prometheus format they are again exported via logs and further parsed and published by Promtail. For both databases in addition to the DB2 exporter which exports SQL queries the logs include procedure calls. This shows which actions are called more specifically than general metrics as there is no procedure call counter available. The approach is also less interfering as it does not use queries to fetch metrics and file accesses can be moved to virtual in-memory storage. The ICN publishes logs similar to NGINX but not listing every request but more precise for internal methods calls. This includes the execution time from which

[13] NGINX ngx_http_log_module: https://nginx.org/en/docs/http/ngx_http_log_module.html#log_format

the relative responsiveness can be derived. RMApp also releases logs for the methods called and how long it took to execute them. With these methods it is possible to identify the methods which are executed the most and how they relate to other metrics like CPU and memory usage.

3.3 Testing

- First determine the system state without scaling - Determine the tripping point for maximum efficency - Generate custom metrics resources to scale on - Add scaling - Test the scaling for the 3 different scenarios below - Describe workloads (low data exchange, high data exchange) - testing scenarios

3.3.1 Many users - low tasks

3.3.2 Few users - high tasks

3.3.3 Many users - high tasks

4 Conclusion

Cloud computing is being adopted at a high rate. Additionally the trend to microservices and containerization further increases integration into the cloud. Fueled at least in part by the open-source market Many applications and companies are still in the transitioning phase to adapt cloud technologies. This leads to systems which are distributed into classic deployments and cloud deployments. Connecting these systems is an additional challenge for administrators tackle. Industry reports suggest that the transition is speeding up fueled at least in part by the open-source movement. More professional applications are freely accessible which leads to a growing cloud ecosystem. As more standards are being adopted and container orchestration is getting more common these systems become more widely available. Administrators need to choose an appropriate system for their needs and a deployment method. Orchestration can be done either self managed in a private cloud or by a cloud computing provider. On the software site there are also proprietary solutions which offer addtional functionality and reduced managment effort. The decision is dependent on the capabilities of the company and the IT department. Similar considerations are needed for monitoring and other supporting tools like security analysis.
While research in classical implementations already lead to huge optimizations these insights can only be partially adapted. Major progress and research in the most important fields currently undertaken. This includes security of containerized applications by identifying potential risks and cluster management which includes monitoring and automatization. More automated systems are needed due to increasing dynamics and a diversified software market. While load balancing interfaces to the cloud can use classical methods the internal structure needs to be adapted. This adaption is part of current research and involves manual effort.

Specific requirements for ECM applications must be kept in mind when translation the application into the cloud. Accurate statistics must be turned into metrics which produce insight into the internal processes. K8s and supporting applications offer a wide range of functionalities which are sufficient for most applications without relying on proprietary software. The main effort lays in the extensive configurability of each component of the system as well as the transition of the ECM application. While the transition is made easier by having a big community to support all sorts of constellations with different software. Problems identified from our ECM application prototype for translation into the cloud are mainly down to two aspects. The separation of the application into loosly coupled or independent components. Identification of stateless and stateful parts and the following reduction and aggregation of states to keep the components stateless. For monitoring and further dynamic scaling for load balancing additional challenges arise. We choose an legacy applications to show these to an greater extend than a ”cloud-ready” application would. Firstly the applications need to be embedded into the networking such that a dynamic creation and deletion of containers is supported. Also components who are not supported inside the cloud need to be included into the cloud network system. Secondly metrics must be created and gathered for as many parts of the system as possible. As creation of metrics and exporting them to the cloud is not intended a translation system needs to be inserted. And lastly it is important for some applications

to be shut down gracefully before deletion to ensure data integrity. This is important regarding the responsiveness of the scaling and therefore efficiency which is determined by startup and shutdown time. The application needs to be optimized for this kind of usage internally to get the maximum efficiency by avoiding unnecessary resource allocation.

Outlook

As the cloud computing market sees rapid development it lacks in defining a unique direction. This results in different system which are incompatible and define own standards. As market leaders emerge, new de-facto standards get introduced like for container images. This process is normal for new technology adaption but leaves adopters in an open state where they may not want to rely on standards that may change rapidly. In the future more stable standards for configuaration, topology description and metrics should make it easier to choose and switch systems.
Another point of heavy research is the integration of stateful applications. These need to be taken with special care to ensure the integrity and organization of data. Many stateful applications like databases are not yet ready for adoption into a dynamic cloud environment. While deployment into cloud computing clusters is certainly possible the usage of dynamic load balancing is not achieved yet. For legacy applications users who want to bring their applications into the cloud there are two possibilities to consider. Either switching to applications which are developed with the cloud in mind or adapting their applications into the cloud environment. Further development may lead to easier adoption techniques which opens the market also for smaller businesses. For the prototype developed herein further steps involve the optimization of the components to reduce overhead in size and memory usage. This would also allow for faster startup and shutdown times. Due to time limitations it was not possible to analyse all metrics which is also relevant to improve accuracy on scaling. From these results a proper scaling behaviour based on a mix of metrics can be created. We also performed initial testing with a limited set of workloads. Additional testing with more realistic workloads should be performed to estimate the maximum supported users in realistic use cases.

Bibliography

[AAA19] H. Alazzam, E. Alhenawi, R. Al-Sayyed. "A hybrid job scheduling algorithm based on Tabu and Harmony search algorithms". In: *The Journal of Supercomputing* 76 (Dec. 2019), pp. 7994–8011. DOI: 10.1007/s11227-019-02936-0 (cit. on p. 16).

[AAME19] H. Alazzam, A. Alsmady, W. Mardini, A. Enizat. "Load Balancing in Cloud Computing Using Water Flow-like Algorithm". In: *Proceedings of the Second International Conference on Data Science, E-Learning and Information Systems*. DATA '19. Dubai, United Arab Emirates: Association for Computing Machinery, 2019. ISBN: 9781450372848. DOI: 10.1145/3368691.3368720. URL: https://doi.org/10.1145/3368691.3368720 (cit. on p. 16).

[ABdP13] G. Aceto, A. Botta, W. de Donato, A. Pescapè. "Cloud monitoring: A survey". In: *Computer Networks* 57.9 (2013), pp. 2093–2115. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2013.04.001. URL: https://www.sciencedirect.com/science/article/pii/S1389128613001084 (cit. on p. 18).

[AS15] S. Aslam, M. A. Shah. "Load balancing algorithms in cloud computing: A survey of modern techniques". In: *2015 National Software Engineering Conference (NSEC)*. 2015, pp. 30–35. DOI: 10.1109/NSEC.2015.7396341 (cit. on p. 16).

[CS06] A. Chhabra, G. Singh. "Qualitative Parametric Comparison of Load Balancing Algorithms in Distributed Computing Environment". In: *2006 International Conference on Advanced Computing and Communications*. 2006, pp. 58–61. DOI: 10.1109/ADCOM.2006.4289856 (cit. on p. 16).

[CZS19] F. Chen, X. Zhou, C. Shi. "The Container Scheduling Method Based on the Min-Min in Edge Computing". In: *Proceedings of the 2019 4th International Conference on Big Data and Computing*. ICBDC 2019. Guangzhou, China: Association for Computing Machinery, 2019, pp. 83–90. ISBN: 9781450362788. DOI: 10.1145/3335484.3335506. URL: https://doi.org/10.1145/3335484.3335506 (cit. on p. 18).

[Dav20] Dave Anderson. *A better Kubernetes, from the ground up*. 2020. URL: https://blog.dave.tf/post/new-kubernetes/ (visited on 08/24/2021) (cit. on p. 25).

[DH13] S. Dhouib, R. B. Halima. "Surveying Collaborative and Content Management Platforms for Enterprise". In: *2013 Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*. 2013, pp. 299–304. DOI: 10.1109/WETICE.2013.61 (cit. on pp. 11, 21).

[DMNC13] M. Dash, A. Mahapatra, Narayan, N. Chakraborty. "Cost Effective Selection of Data Center in Cloud Environment". In: *International Journal on Advanced Computer Theory and Engineering* 2 (Jan. 2013), pp. 2319–2526 (cit. on p. 18).

[Doc21] Docker Inc. *Docker Swarm mode*. 2021. URL: https://docs.docker.com/engine/swarm/ (visited on 08/24/2021) (cit. on p. 25).

[Gar20] Gartner, Inc. *Gartner Forecasts Strong Revenue Growth for Global Container Management Software and Services Through 2024*. 2020. URL: https://www.gartner.com/en/newsroom/press-releases/2020-06-25-gartner-forecasts-strong-revenue-growth-for-global-co (visited on 08/24/2021) (cit. on p. 24).

[Gun20] J. R. Gunasekaran. "Minimizing Cost and Maximizing Performance for Cloud Platforms". In: *Proceedings of the 21st International Middleware Conference Doctoral Symposium*. Middleware'20 Doctoral Symposium. Delft, Netherlands: Association for Computing Machinery, 2020, pp. 29–34. ISBN: 9781450382007. DOI: 10.1145/3429351.3431747. URL: https://doi.org/10.1145/3429351.3431747 (cit. on p. 11).

[Has21] HashiCorp. *Nomad - Workload Orchestration Made Easy*. 2021. URL: https://www.nomadproject.io/ (visited on 08/24/2021) (cit. on p. 25).

[HW18] C. B. Hauser, S. Wesner. "Reviewing Cloud Monitoring: Towards Cloud Resource Profiling". In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. 2018, pp. 678–685. DOI: 10.1109/CLOUD.2018.00093 (cit. on p. 19).

[KBKK06] G. Khanna, K. Beaty, G. Kar, A. Kochut. "Application Performance Management in Virtualized Server Environments". In: *2006 IEEE/IFIP Network Operations and Management Symposium NOMS 2006*. 2006, pp. 373–381. DOI: 10.1109/NOMS.2006.1687567 (cit. on p. 18).

[Koe14] M. Koehler. "An adaptive framework for utility-based optimization of scientific applications in the cloud". In: *Journal of Cloud Computing: Advances, Systems and Applications* 3 (Dec. 2014), p. 4. DOI: 10.1186/2192-113X-3-4 (cit. on p. 13).

[KSST05] T. Kimbrel, M. Steinder, M. Sviridenko, A. Tantawi. "Dynamic Application Placement Under Service and Memory Constraints". In: *Experimental and Efficient Algorithms*. Ed. by S. E. Nikoletseas. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 391–402. ISBN: 978-3-540-32078-4 (cit. on p. 21).

[Mes18] Mesosphere, Inc. *Marathon - A container orchestration platform for Mesos and DC/OS*. 2018. URL: https://mesosphere.github.io/marathon/ (visited on 08/24/2021) (cit. on p. 24).

[MG11] P. M. Mell, T. Grance. *SP 800-145. The NIST Definition of Cloud Computing*. Tech. rep. Gaithersburg, MD, USA, 2011. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf (cit. on p. 13).

[MS20] C. Mega, G. Shao. "How to evolve the architecture design of legacy enterprise content management systems for being able to exploit cloud technologies". 2020 (cit. on p. 21).

[MSA12] K. Ma, R. Sun, A. Abraham. "Toward a lightweight framework for monitoring public clouds". In: *2012 Fourth International Conference on Computational Aspects of Social Networks (CASoN)*. 2012, pp. 361–365. DOI: 10.1109/CASoN.2012.6412429 (cit. on p. 19).

[MSHN17] D. Molka, R. Schöne, D. Hackenberg, W. E. Nagel. "Detecting Memory-Boundedness with Hardware Performance Counters". In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ICPE '17. L'Aquila, Italy: Association for Computing Machinery, 2017, pp. 27–38. ISBN: 9781450344043. DOI: 10.1145/3030207.3030223. URL: https://doi.org/10.1145/3030207.3030223 (cit. on p. 14).

[MWL+14] C. Mega, T. Waizenegger, D. Lebutsch, S. Schleipen, J. M. Barney. "Dynamic Cloud Service Topology Adaption for Minimizing Resources While Meeting Performance Goals". In: *IBM J. Res. Dev.* 58.2–3 (Mar. 2014), p. 8. ISSN: 0018-8646. DOI: 10.1147/JRD.2014.2304771. URL: https://doi.org/10.1147/JRD.2014.2304771 (cit. on pp. 11, 13).

[NMNA12] K. A. Nuaimi, N. Mohamed, M. A. Nuaimi, J. Al-Jaroodi. "A Survey of Load Balancing in Cloud Computing: Challenges and Algorithms". In: *2012 Second Symposium on Network Cloud Computing and Applications*. 2012, pp. 137–142. DOI: 10.1109/NCCA.2012.29 (cit. on pp. 15, 16, 34).

[PB19] K. D. Patel, T. M. Bhalodia. "An Efficient Dynamic Load Balancing Algorithm for Virtual Machine in Cloud Computing". In: *2019 International Conference on Intelligent Computing and Control Systems (ICCS)*. 2019, pp. 145–150. DOI: 10.1109/ICCS45141.2019.9065292 (cit. on p. 16).

[PI20] H. Pydi, G. N. Iyer. "Analytical Review and Study on Load Balancing in Edge Computing Platform". In: *2020 Fourth International Conference on Computing Methodologies and Communication (ICCMC)*. 2020, pp. 180–187. DOI: 10.1109/ICCMC48092.2020.ICCMC-00036 (cit. on p. 18).

[QYL+21] B. Qiao, F. Yang, C. Luo, Y. Wang, J. Li, Q. Lin, H. Zhang, M. Datta, A. Zhou, T. Moscibroda, S. Rajmohan, D. Zhang. "Intelligent Container Reallocation at Microsoft 365". In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. Athens, Greece: Association for Computing Machinery, 2021, pp. 1438–1443. ISBN: 9781450385626. DOI: 10.1145/3468264.3473936. URL: https://doi.org/10.1145/3468264.3473936 (cit. on p. 18).

[RE13] J. Rats, G. Ernestsons. "Using of cloud computing, clustering and document-oriented database for enterprise content management". In: *2013 Second International Conference on Informatics Applications (ICIA)*. 2013, pp. 72–76. DOI: 10.1109/ICoIA.2013.6650232 (cit. on p. 21).

[Red] Red Hat, Inc. *What are microservices?* URL: https://www.redhat.com/en/topics/microservices/what-are-microservices (visited on 08/24/2021) (cit. on p. 11).

[Red20] Red Hat, Inc. *The future of container adoption*. 2020. URL: https://www.redhat.com/rhdc/managed-files/pa-hpe-containers-idg-analyst-material-f23712-202005-en.pdf (visited on 08/24/2021) (cit. on p. 24).

[Red21] Red Hat Inc. *Red Hat OpenShift*. 2021. URL: https://www.redhat.com/en/technologies/cloud-computing/openshift (visited on 08/24/2021) (cit. on p. 25).

[SCP+03] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, M. Rosenblum. "Optimizing the Migration of Virtual Computers". In: *SIGOPS Oper. Syst. Rev.* 36.SI (Dec. 2003), pp. 377–390. ISSN: 0163-5980. DOI: 10.1145/844128.844163. URL: https://doi.org/10.1145/844128.844163 (cit. on p. 18).

[Sha20] G. Shao. "About the Design Changes Required for Enabling ECM Systems to Exploit Cloud Technology". MA thesis. Germany: University of Stuttgart, 2020 (cit. on p. 11).

[SJ20] M. S. Sanaj, P. M. Joe Prathap. “An Enhanced Round Robin (ERR) algorithm for Effective and Efficient Task Scheduling in cloud environment”. In: *2020 Advanced Computing and Communication Technologies for High Performance Applications (ACCTHPA)*. 2020, pp. 107–110. DOI: 10.1109/ACCTHPA49271.2020.9213198 (cit. on p. 16).

[SNA14] H. Shoja, H. Nahid, R. Azizi. “A comparative survey on load balancing algorithms in cloud computing”. In: *Fifth International Conference on Computing, Communications and Networking Technologies (ICCCNT)*. 2014, pp. 1–5. DOI: 10.1109/ICCCNT.2014.6963138 (cit. on p. 16).

[SS14] S. B. Shaw, A. K. Singh. “A survey on scheduling and load balancing techniques in cloud computing environment”. In: *2014 International Conference on Computer and Communication Technology (ICCCT)*. 2014, pp. 87–95. DOI: 10.1109/ICCCT.2014.7001474 (cit. on p. 16).

[sys21] sysdig. *Sysdig 2021 – Container Security and Usage Report*. 2021. URL: https://dig.sysdig.com/c/pf-2021-container-security-and-usage-report?x=u_WFRi (visited on 08/24/2021) (cit. on pp. 11, 12, 18).

[Sze15] J. Szefer. “Leveraging Processor Performance Counters for Security and Performance”. In: *Proceedings of the 5th International Workshop on Trustworthy Embedded Devices*. TrustED ’15. Denver, Colorado, USA: Association for Computing Machinery, 2015, p. 41. ISBN: 9781450338288. DOI: 10.1145/2808414.2808421. URL: https://doi.org/10.1145/2808414.2808421 (cit. on p. 14).

[TAZ+17] O. Tuncer, E. Ates, Y. Zhang, A. Turk, J. Brandt, V. J. Leung, M. Egele, A. K. Coskun. “Diagnosing Performance Variations in HPC Applications Using Machine Learning”. In: *High Performance Computing*. Ed. by J. M. Kunkel, R. Yokota, P. Balaji, D. Keyes. Cham: Springer International Publishing, 2017, pp. 355–373. ISBN: 978-3-319-58667-0 (cit. on p. 13).

[The21] The Linux Foundation. *Kubernetes*. 2021. URL: https://kubernetes.io/ (visited on 08/24/2021) (cit. on p. 25).

[TM14] F. G. Tinetti, M. Méndez. “An Automated Approach to Hardware Performance Monitoring Counters”. In: *2014 International Conference on Computational Science and Computational Intelligence*. Vol. 1. 2014, pp. 71–76. DOI: 10.1109/CSCI.2014.19 (cit. on p. 20).

[TZZ+11] W. Tian, Y. Zhao, Y. Zhong, M. Xu, C. Jing. “A dynamic and integrated load-balancing scheduling algorithm for Cloud datacenters”. In: *2011 IEEE International Conference on Cloud Computing and Intelligence Systems*. 2011, pp. 311–315. DOI: 10.1109/CCIS.2011.6045081 (cit. on p. 16).

[VPK+15] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, J. Wilkes. “Large-Scale Cluster Management at Google with Borg”. In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys ’15. Bordeaux, France: Association for Computing Machinery, 2015. ISBN: 9781450332385. DOI: 10.1145/2741948.2741964. URL: https://doi.org/10.1145/2741948.2741964 (cit. on p. 25).

[VRS20] G. E. de Velp, E. Rivière, R. Sadre. “Understanding the Performance of Container Execution Environments”. In: *Proceedings of the 2020 6th International Workshop on Container Technologies and Container Clouds*. WOC’20. Delft, Netherlands: Association for Computing Machinery, 2020, pp. 37–42. ISBN: 9781450382090. DOI: 10.1145/3429885.3429967. URL: https://doi.org/10.1145/3429885.3429967 (cit. on pp. 21, 22, 24).

[WBW14] R. Weingärtner, G. Brascher, C. Westphall. “Cloud resource management: A survey on forecasting and profiling models”. In: *Journal of Network and Computer Applications* 47 (Oct. 2014). DOI: 10.1016/j.jnca.2014.09.018 (cit. on p. 13).

[YSG16] K. Yongsiriwit, M. Sellami, W. Gaaloul. “A Semantic Framework Supporting Cloud Resource Descriptions Interoperability”. In: *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. 2016, pp. 585–592. DOI: 10.1109/CLOUD.2016.0083 (cit. on p. 16).

[ZX20] C. Zhiyong, X. Xiaolan. “Overview of Container Cloud Task Scheduling”. In: *Proceedings of the 2020 Artificial Intelligence and Complex Systems Conference*. AICSconf ’20. Wuhan, China: Association for Computing Machinery, 2020, pp. 50–55. ISBN: 9781450377270. DOI: 10.1145/3407703.3407714. URL: https://doi.org/10.1145/3407703.3407714 (cit. on pp. 11, 15).

All links were last followed on 24.08.2021.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature