Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master Thesis

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:	2021-02-01
Completed:	2021-09-01

Abstract

The transition to cloud-based ECM solutions requires adaptation and enhancements for dynamic cloud environment. Cloud computing and containerization offer key concepts to create solutions for this task. They also open new opportunities to benefit from cloud usage by leveraging the pay-as-you-go model. Cloud users and providers both benefit from the increased efficiency of dynamic applications. But legacy applications are not yet able to leverage the benefits provided by cloud orchestration. This thesis, therefore researches the feasibility of a dynamic load balancing approach applied to an ECM application deployed into a cloud environment. To evaluate the approach, a prototype using open-source software is created on a Kubernetes orchestrated cluster. Previous work included the port of the containerized ECM application into the Kubernetes environment. The present prototype builds up on this approach by enhancing the ECM application components with metrics export capabilities. A monitoring system based on Prometheus is introduced to gather these metrics from the ECM application and other system components. Information provided by these metrics are used to add elasticity to application components. The prototype proves that dynamic load balancing of the ECM application in the cloud is feasible. Two major challenges for an efficient deployment of the application were identified, (1) the generation of useful metrics and (2) removing dependencies from individual components. Further research into optimizations of stateful service components is required. This further ensures an efficient usage in cloud based elastic topologies, especially considering stateful database applications.

Contents

List of Figures

Acronyms

AI	Artifical Intelligence. 17
CNCF	Cloud Native Computing Foundation. 24
CNI	Container Network Interface. 24
DC/OS	Distributed Cloud Operating System. 24
DDoS	Distributed Denial of Service. 48
DLB	Dynamic Load Balancing. 13
DNS	Domain Name System. 47
ECM	Enterprise Content Management. 7
HPA	Horizontal Pod Autoscaler. 26
HPC	High Performance Computing. 13
HTTP	Hypertext Transfer Protocol. 48
IaaS	Infrastructure as a Service. 14
IaC	Infrastructure as Code. 24
ICN	IBM Content Navigator. 31
IIM	Intelligent Information Management. 20
J2EE	Java™ 2 Platform Enterprise Edition. 31
K8s	Kubernetes. 7
KPI	Key Performance Indicator. 19
LSDB	Library Server Database. 31
MAPE	Monitor-Analyze-Plan-Execute. 13
NAT	Network Address Translation. 45
NFS	Network File System. 29
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. 26
RMApp	Ressource Manager Application. 31
RMDB	Ressource Manager Database. 31
RPS	Requests per Second. 51
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 require solutions which can adapt to a dynamic environment [MWL+14]. This includes long-term changes like a variation in the number of active users as well as workload changes driven by daily workload pattern. The adaption is relevant for cloud users as well as providers who need to fulfill their contracts while being profitable [Gun20]. To achieve this, modern ECM systems utilize 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. In a first step the Operating System (OS) was abstracted by using VMss, increasing security, control and removing dependency conflicts. To eliminate the additional overhead of VMs, containerization is used to deploy applications in a protected context [ZX20]. This allows for fine-grained control over different parts of the system by vertical scaling via containerization but also horizontal scaling via extended resource allocation. The transition was driven by an 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. The report 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. Many insecurities can be identified automatically and then fixed in a future release. The security analysis includes monitoring which gets more important for bigger and more complex systems.

By exploiting 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 paid resources is a great economic incentive. 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-tenant scenarios. [MWL+14] lists some exemplary SLAs which are usually time or space constraints on certain operations. As the software is handled by an automatized system, it reduces manual intervention and thus management cost. This leads to a structured approach of expansion which is preferred as it follows a clear path and does not lead to unstructured and unmonitored systems with low utilization.

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 loosely coupled [Red]. This enables each part of the application to be maintained separately 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 a containerized ECM system deployed using Docker by Shao [Sha20] who took an existing ECM system and containerized the components. These are now modified and ported 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 multi-tenancy. The architecture consists of a set of virtual nodes controlled by a container orchestrator. This is similar to a cloud environment where only the outlines for performance and architecture are known. The transformation of this application and of the application landscape in general, is driven by the open-source software community [sys21]. Many big companies donate part of their software as open-source to the public. This speeds up development while also making it accessible to a bigger userbase. The design presented here leverages only open-source software to evaluate the feasibility of our approach.

Structure

The following chapter 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 challenges emerging in the design and implementation. Chapter 3 discusses our prototype implementation and the individual challenges we identified during our investigations. It is divided into implementation considerations and the setup used for initial testing. Section 3.3 then describes the test workloads and metrics used to perform the evaluation of the prototype. The results of the analysis for the defined metrics under the tested workloads and scenarios. Chapter 4 summarizes the results and provides further goals and possible improvements.

2 Background

This chapter states the various techniques, systems and structures used. There are multiple approaches to Dynamic Load Balancing (DLB) which need different amounts of input data and thus lead to different results. Mega et al. [MWL+14] lists three categories of methods. First there are utilization-based optimization techniques which try to maximize the utilization 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 group of methods are based upon machine-learning. These try to build a model for resource consumption which is then applied to predict the workload pattern. A similar approach for High Performance Computing (HPC) is presented in [TAZ+17] but the same technique can be used for other applications as well. They use time series data and a cloud environment which is similar to the environment used herein. With more data collected to train the model and therefore the algorithm the system utilization can be improved over time. 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 to optimize 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 SLAs are met at any time when the system workload changes. The prototype proposed in Chapter 3 is based on this approach.

2.1 Cloud Computing Structure

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

- *On-demand self-service.* The ability to automatically provision capabilities to a consumer.
- *Broad network access.* That those capabilities are available over standard network clients.

- *Resource pooling.* Multiple customers are assigned resources dynamically on demand.
- *Rapid elasticity.* The capabilities can be scaled with demand at any time.
- *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 bandwidth 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 orchestrator cluster on which a containerized application can be deployed by the developer. The advantage of this model is the reduced effort in managing the 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. Public clouds are hosted by providers which allow everyone to rent a part of the cluster resources. Therefore, an application might run on a VM which shares CPU and RAM with one or more VMs. Due to shared hardware exploits can potentially leak data to other processes on the same hardware through vulnerabilities in CPUs or memory. [KHF+19; LSG+18]. This security implications need to be addressed in applications managing secure information. 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 profiting from the shared hardware by reducing operational costs and increasing utilization. Especially for smaller firms this opens opportunities which were not affordable before.
If companies need to comply to higher security standards or want to be in control of the underlying hardware they can operate private clouds. These clouds are typically used only by the operating company or a limited set of customers. Therefore, the utilization is dependent on a limited number of tenants. The responsibility for maintenance and failures must also be factored into the considerations of hosting a private cloud.
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 an internet connectivity outage. A drawback for the hybrid approach are the higher security considerations due to communication channels between the public and private parts of the cloud. This approach also includes the effort of setting up the system on two separated clouds.

2.2 Load Balancing

Load balancing describes the task of distributing workloads to a set of resources respectively nodes in a cloud environment. This is accomplished by load distribution functions which assigns a resource to a task. The challenge is to design the distribution function in such a way that it optimizes specific system characteristics or metrics. In general this can be the system load but also any other metric of the system. Potential characteristics are availability, response time or resource consumption. This leads to a diverse set of algorithms which vary greatly in terms of complexity and field of application. Zhiyong et al. [ZX20] describes and compares several scheduling algorithms. These are divided into workload dependent and independent algorithms. For multi-tenant applications it is important to distinguish different workloads on their operations, intensity and duration to estimate the complexity.

Load balancing divides into two categories [NMNA12]: Static Load Balancings (SLBs) and DLBs. SLB does load distribution on a predefined system model. The system characteristics are defined by the hardware and software metrics. From this model a central load balancer tries to minimize the workload distribution for each resource it assigns workloads to. Because the system is assumed as static the distribution function can only rely on the already distributed workloads and list of workloads to distribute. DLB as opposed to SLB does not use a static workload and instead relies on the current system state. Thus, the distribution function tends to be more complex because it adds the system state and workload information to the input. In regard to cloud systems with a dynamic topology the complexity further increases. The load balancers also need to react to these changes to optimize the system.

The advantage of SLB is the simplicity of the approach which leads to a higher performance. This can be used efficiently if the nature of the workloads is well known and thus workload execution can be accurately predicted. A static load balancer can therefore be more cost effective and easier to integrate within existing systems. The disadvantage of the 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 architectures where there are numerous different workloads 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 needs for metrics and system state. One advantage comes from the wide applicability of this approach. Because the distribution of workloads is tied to the system state, a workload does not need to be as accurately characterized as for the static approach. The impact of every workload on the system is measured and fed back into the next scheduling decision. If a workload with unusual demands is scheduled, an imbalance between resource utilizations 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 react by redirecting further workloads 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. The use of distributed load balancers increases reliability of the service in a failure condition.

Typical load balancing algorithms include round-robin, throttled, MapReduce, least-connection, -bandwidth and -response time scheduling and hashing algorithms. Nuaimi et al. [NMNA12] compares seven different static and dynamic algorithms on replication, speed and other properties. Load balancing is a broadly researched topic since it can be applied to many different problems in computer science. 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. It is therefore important to identify the specific 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 can be used interchangeably. Replication is needed if the current resources can not satisfy the current workload. If there is not enough workload for all instances to work efficiently unused replicas can be removed. In the cloud computing environment resources are typically computing and storage nodes who are located in server complexes. Resource description languages are used to describe the resources and the topology. There are a bunch of different standard languages for these descriptions. Yongsiriwit et al. [YSG16] uses three different languages to create a framework that allows for interoperability between them. If resources consumption increases or decreases the amount of replicas needs to be reevaluated and further increased or decreased depending on the current state. In case of a decrease excess replicas need to be scheduled to stop and subsequently be destroyed. Before an instance of a replica is stopped teardown actions and active requests in the instance need to be finished. For an increase of instances, new replicas 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. This startup and teardown times need to be considered to determine the reevaluation intervals of the system state. 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 replica instances and can lead to an escalation circle of overshooting in which new instances are added which do not contribute to the workload processing. An escalation can also similarly be created in the other direction by instances which are in the teardown phase which leads to a underestimation of replica 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. Workload patterns can be followed more accurately with this design approach which leads to a greater utilization. For systems with reactive scheduling startup times can be even more important to always meet Service level agreements if demand increases suddenly.

When the number of replicas is determined, the correct number of instances is determined by evaluating rules based on SLAs. The rules engine consists of a set of rules for each component that needs to be replicated. The rules for each component get evaluated in specific intervals and combined together to create a new desired number of instances of the replica. If the determined number of instances changes the appropriate action needs to performed. If replicas need to be removed a set of replicas from the existing ones is selected. The selection can be random or if enough information is given an algorithm can select the most irrelevant ones. If new replica instances are scheduled for creation, a set of nodes from the network is selected to host them. This process can also be improved with additional information. Potential pitfalls and considerations for these processes are evaluated in Section 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 rule is violated based on an estimate and then schedules replicas 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 capabilities of the algorithm. If the prediction overstates the real workload, unnecessary resources are 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. An often used compromise is to proactively create instances but destroy them reactively to be sure to stay in a valid state.

For PaaS systems nodes are not directly accessible for the administrator and thus can not be explicitly added. The nodes hosting the platform are chosen by the provider on the basis of the requested capabilities of the cluster. On a containerized platform like Docker or K8s 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 or automatically managed by orchestrators. SaaS systems further abstract from the underlying hardware and only make the service available. If a service needs more or less performance the administrator can request the corresponding capability changes.

2.3.1 Scheduling

When the rules engine decides to create new replicas of a resource, 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 from the cluster, the capabilities of these nodes 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. Several factors can contribute to the affinity like latency, performance, storage space, location or existing instances. Operators need to account for these when making decisions.

Before VMs were used widely fast scheduling was important already 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. In [DMNC13] scheduling of VMs in a cloud environment is compared with round-robin and throttled 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 clusters ”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.

Stateful and stateless instances need to be distinguished. Stateless instances save data temporarily and thus do not need persistant storage. When they are created they use 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 static load balancers. They can maintain a temporary state for their last scheduling decisions, but this information can be discarded. Stateful instances generate data that must be protected from deletion. 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 ECM systems. The files in the system 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 exhaustive 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 number of instances dynamically from the rules the system state needs to be monitored and saved. Monitored components can be of a broad 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 required 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 agents can be located either local on each node or more integrated into the system as cloud applications. The integration into the cloud has the advantage of using similar 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 are typically collected (pulled) by or send (pushed) to the monitoring system. The first option has the advantage that the monitoring system can decide the interval in which the published 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 certain metrics.
With a push configuration on the other hand, the application decides when to send its 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 applications access the monitoring system with the credentials provided. It 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. Systems can use a short-term database to store the data temporarily before they are transferred to a long-term storage 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 additionally contain tags 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 metric. 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 measurements 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] identifies challenges for monitoring and propose an alternative solution. They name 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 do not use a centralised instance but dedicated metrics for each node. To minimize complexity when querying metrics they are compressed with statistical analysis to maintain only relevant information.

Performance indicators

Key Performance Indicators (KPIs) are a subset of metrics which are applicable to all systems. These indicators relate directly to the performance of the underlying hardware and software 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
- and 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 too 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

To deliver the information, a frontend application is used. The application can be a standalone software or a web application. Modern systems, which rely on a broad customer base use web applications as a standardized interface which can be used by many users. 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 capabilities for organized data. Examples for organized data are user information, file meta data or configuration options. To handle the file objects, an object manager is used to handle access permissions and meta data for these objects as well as storage location information. For preservation the data is transferred to a secure storage with lower performance but more efficient storage.

Typical components of cloud ECM systems are a webserver (1) as frontend interface, a library server (2) for storing information and an object manager (3) to store and manage objects like documents. For (2) and (3) additional resources must be reserved for the system to store the information. The webserver (1) only displays and forwards information and therefore does not require 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

[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

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 initially the intertwining of all components which prevents separating into loose-coupled or independent components. The built-in topology organization which is managed externally in cloud applications and third the reliance on hardware instead of virtualized components is the second problem. This includes virtual networks 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 separates each container by establishing a new context for all resources like process IDs, users or files as well as network interfaces. To start a container, a container runtime is provided with the mount point and metadata. This runtime then starts a new kernel namespace with the specific resources 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 resources 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 resources for running the container. It also provides the necessary parameters to the runtime dependent 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 various 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 also create an overlay network which spans over all nodes in the system. This contributes to security since all inter-node communication is tunneled by the orchestrator. Furthermore, it simplifies networking since all inter-node communication is hidden from the applications. Orchestrators also manage failures of all components used by the system. This includes the applications hosted as well as the underlying components and resources. If a failure is detected either countermeasures are 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

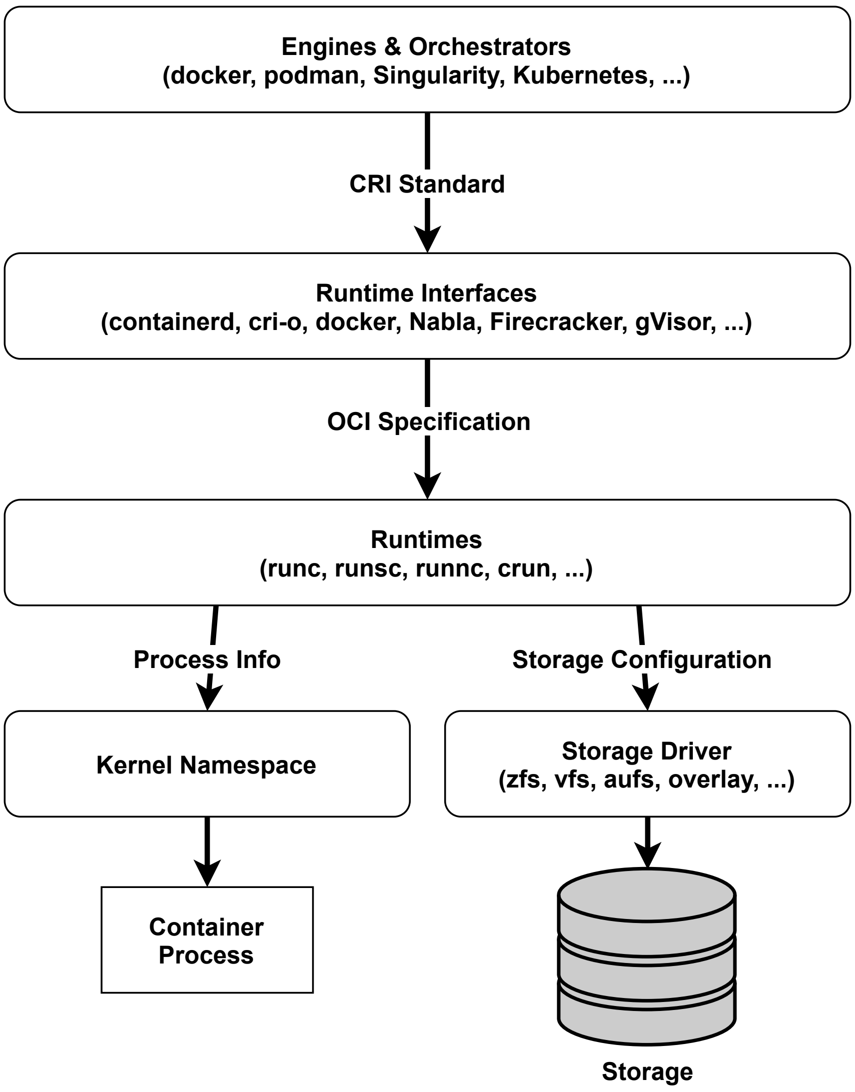

Engines & Orchestrators
(docker, podman, Singularity, Kubernetes, ...)
CRI Standard
Runtime Interfaces
(containerd, cri-o, docker, Nabla, Firecracker, gVisor, ...)
OCI Specification
Runtimes
(runc, runsc, runnc, crun, ...)
Process Info
Storage Configuration
Kernel Namespace
Storage Driver
(zfs, vfs, aufs, overlay, ...)
Container Process
Storage

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

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 research 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.

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

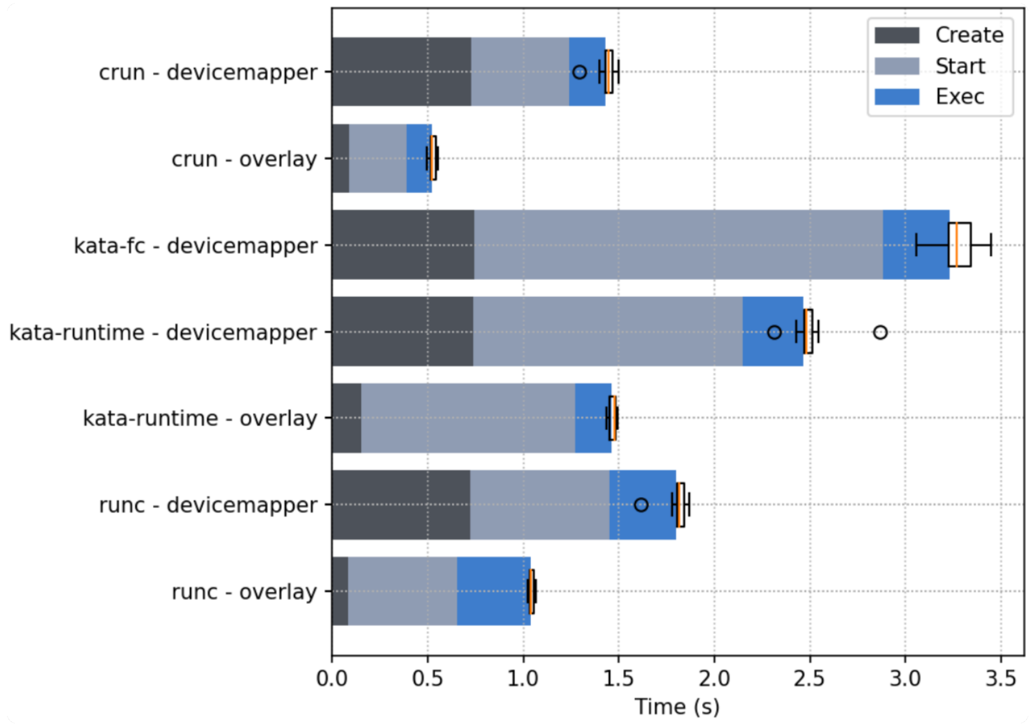


Figure 2.2: Creation, Startup, and Execution time for different container runtimes, by [VRS20].

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 these images. The main difference is in the structure of the image formats. 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 OCI image format specification[7] for containers include

- 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,
- 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.

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

The usage and adaption of an open standard for container images made it possible for many 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 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 them are open-source or free to use while some, especially applications for security or hosting, are closed-source or proprietary. 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 efficiency 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 because of management capabilities 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 underlying 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 so-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 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

[8]Apache Mesos: `https://mesos.apache.org/`

[9]The Two Million Container Challenge: `https://www.hashicorp.com/c2m`

YAML Ain't Markup Language (YAML) files which contain the container, network and volume definitions. If an application is deployed the defined resources 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 transferred to the CNCF, which was also founded by Google, to accelerate and outsource the development. Build upon the Docker runtime as a backend service at first the system presently is an 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 utilizing one program for each task. Criticism 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 others an internal DNS server, overlay networks, task scheduling, resource management, configuration management, state management (through key-value stores), failure management and scalability. But K8s does not manage the cluster on itself. The cluster consists 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 themselves 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 capabilities. For this they use a modified K8s version which aims to be optimized for continuous development and multi-tenancy by adding specific tools for developers and operators. This K8s instance runs in the background to support the container orchestration on the OpenShift cluster. Multicluster support is added as well as security and data analysis. As OpenShift is targeted at businesses, it is bundled with additional support and is not open-source but a paid service. An open-source version of the OpenShift K8s backend is available under the name OKD[11].

For our 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 fulfills 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 vast community many applications are already available to use. This reduces the effort for porting existing applications to a containerized

[10]cri-o runtime: `https://cri-o.io`
[11]OKD: `https://www.okd.io`

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 accelerate development. For this purpose several tools are being developed to automate this process. One tool is Minikube[12] which is used to set up a local one node cluster. A more elaborate approach is used by kind[13] which uses containers to simulate an arbitrary number 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 bundled into VMs there are virtualization layers like KubeVirt[14] which integrates these in the K8s environment. An alternative to K8s is Docker swarm as a competitor with comparable targets. But at the time of writing Docker swarm however 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 node or on a separate node dedicated solely to the control plane. They and can alternatively be distributed among multiple nodes for increased performance and availability. It is preferred to run the control plane components separate from the load to prevent any disturbances due to a excessive load on the node. The control plane consists of several components which coordinate the orchestration.

The Scheduler is a watchdog 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 a set of controllers which trigger actions when the state of a specific object diverges from the desired state. Controller examples 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. As optional component the cloud controller manager communicates with the cloud provider's interface to manage nodes, routes and external load balancers.

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

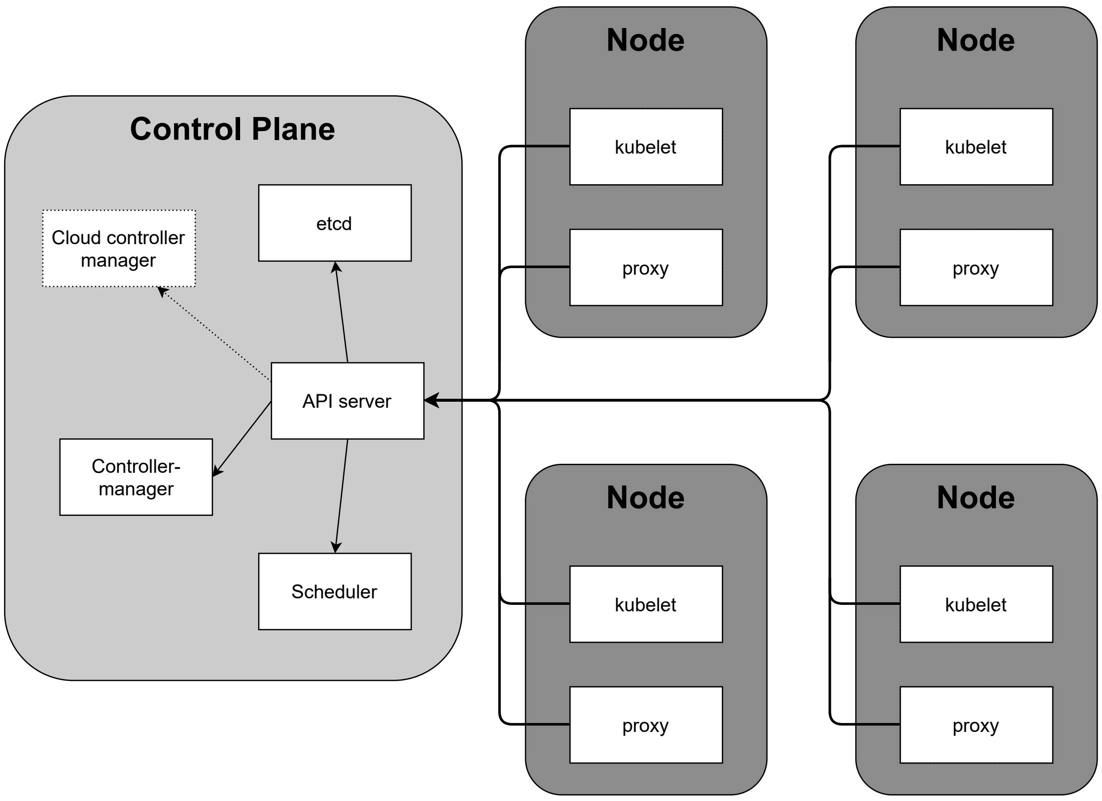


Figure 2.3: Components of the K8s control plane and cluster nodes.

The worker nodes consist each of a kubelet agent 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 enables intra- 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 contain 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 indicate 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 logically coupled into one virtual host. 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.

[17]containerd: https://containerd.io

[18]K8s API reference: https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.21/

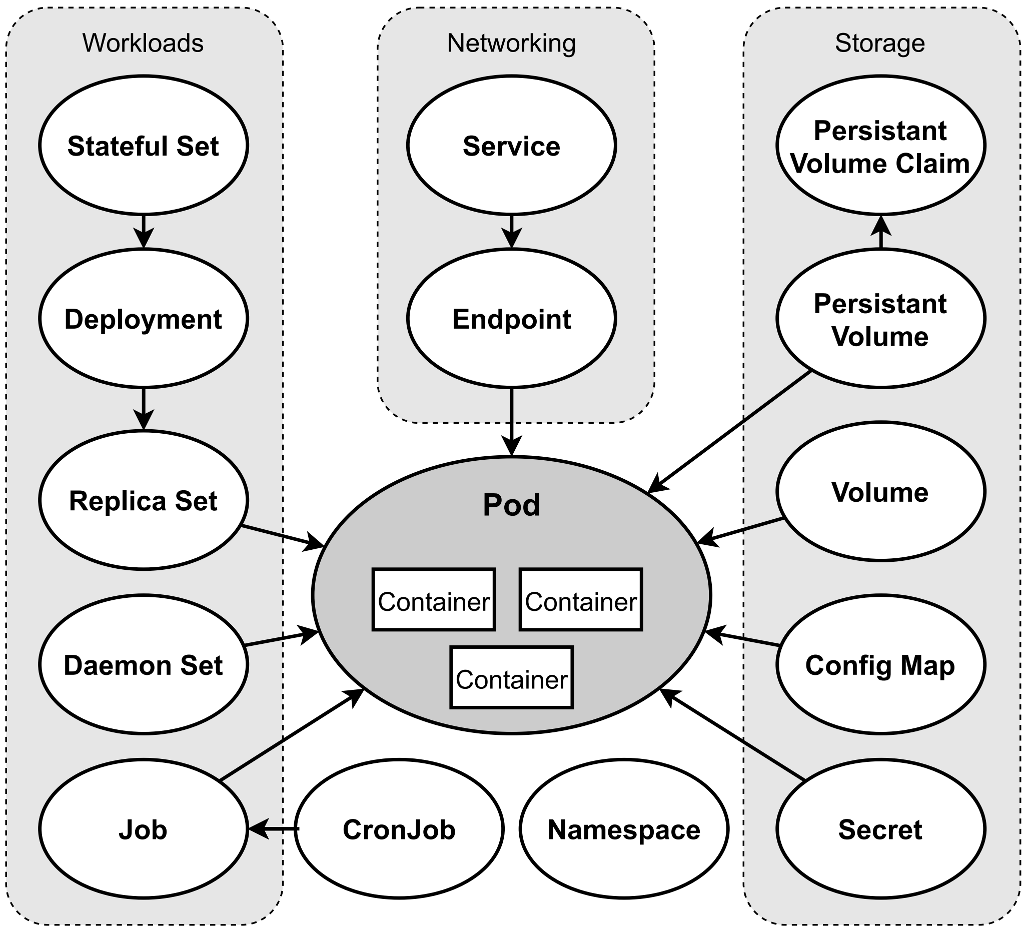


Figure 2.4: Selection of K8s objects and their relations.

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 includes and a template for new pods which it creates. Pods are created if the number of pods found with the selector is lower than the requested number of replicas. 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 the ability to be scaled by an arbitrary factor to cope with an increasing load. A further type of deployment is the *StatefulSet* which performs the same purpose as the deployment but for stateful pods which need persistent storage. The last deployment type is the *DaemonSet* which ensures that an instance of the selected pods is deployed and running on each node. The set of nodes can be specifically reduced by setting constraints on resources or other specific properties. The last workload set is the *Job* which runs a set of pods once until they complete. If the pods fail, the job will try rerunning them a certain number 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 unlikely to be stressed.

Several types of storage can be assigned to a pod each for a special use case. A wide variety 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 node fails the state inside the volume is preserved until the pod is restarted. For persistent 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 and not managed by K8s. A pod can choose the type of storage it prefers or let the system choose an appropriate storage restricted by the requested amount of storage. This type is most effective in combination with stateful 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 generates files for all data structures defined in the map. Singular values can also be defined and added to the pod as environment variables. This separates the configuration from the executables while also eliminates the need for a separate persistent volume for the configuration. Config maps do not provide security for sensitive information such as passwords, keys and tokens. For these informations 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 workload 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 define logical sets of pods determined by a selector. Services expose a list of ports from each pod they select 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 that fulfils 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 available. There are four types of services:

- ClusterIP: This gives the service an internal IP only. Thus, it is accessible 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 an external load balancer to expose the service and forward load.
- ExternalName: Creates a service pointing to an arbitrary hostname instead of pods.

[19]GNU mcron: https://www.gnu.org/software/mcron/

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 resources 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 values relate to either a metric 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 as 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 automate the management of components.[20] Through the added operations it is possible to resolve failures automatically if they follow known patterns. The functionality also adds a way to include repeating tasks like backups or updates checks on applications into the cloud. 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

The following sections discusses the components used to implement the prototype. A description of the previous work is included to give an overview of the base system. Then the additional monitoring application functionality is described covering different aspects separately. We also introduce the various tools used to achieve the desired functionality.

3.1.1 ECM-Components

The containerized reference implementation splits the ECM application into four components. First the central catalog or Library Server Database (LSDB) which contains all metadata and configuration of the services. This container hosts a DB2[1] instance with stored procedures containing the ECM application logic. Then the Ressource Manager Database (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 resource access by coordinating the retrieval 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 portal 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 web service. For administrative tasks the RMApp container also hosts a web interface.

These four components are loosely coupled to each other. ICN as the main component depends on the other components to function properly. It fetches user- and configuration data from the LSDB container and resources from the RMApp container. The RMApp container uses the RMDB container to serve requests on specific documents and files by using the metadata retrieved from the LSDB. The metadata are additionally used 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 K8s eco-system. This process is discussed in another master thesis developed in parallel to this thesis [Try21]. For each container a pod is configured with the same properties as the container but additional components specific to K8s. This pod is then declared in a deployment

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

definition file which holds further properties the most relevant being the option to set an arbitrary number of replicas for the contained pod. This 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 substantial 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 on a specific storage location. K8s therefore extends the concept of volumes to various available mediums on every node in the system instead of just local storage. Persistent volumes with specific amounts of storage are used if the storage needs to be preserved.

Therefore, it is essential to identify the need for stateless and stateful pods. Stateless pods are ephemeral pods who only use transient information. This includes log files and caches like those created by the ICN. RMApp on the other hand manages persistent files and is therefore not a stateless but uses a stateful pod. Stateful pods need to share data with their replicas. This can be achieved by shared storage or via data 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 be 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 probably more than one pod instance running the inbound traffic needs to be load balanced to divide the load between instances. Therefore, pods are combined into services which are handled automatically. The port mappings are defined similar to the port forwardings in Docker to make the access to the application equivalent in K8s. 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 bound to corresponding metrics which are subsequently gathered and analyzed. For this we use the Prometheus[2], a monitoring and alerting system which gathers metrics from multiple different sources in a standardized format. The metrics are provided by each source separately 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 the gathered metrics, for example to rename or ignore parts of the metric or attach metadata to metrics. The configuration additionally allows to modify the scan interval to reduce overhead for slow changing metrics.

In our prototype the metrics are then stored in a temporary time series database for a limited time after which they are deleted from the temporary database. To store metrics permanently Prometheus can be coupled with an external time series database like InfluxDB[3] for long time storage of

[2]Prometheus: https://prometheus.io/

[3]InfluxDB: https://www.influxdata.com/

historic data. Evaluation of stored metrics uses Prometheus' own query language *PromQL* which is explicitly designed for time series data and metrics. For data analysis there is an adapter to Grafana[4] to visualize PromQL queries. The visualization helps in analysing and live monitoring a big amount of data sources at once on a single dashboard.
Prometheus also includes an alert manager which handles sending alerts on specific conditions over several channels. The simplest being E-mail but more sophisticated solutions like Slack and HTTP POST request are also configurable. System operators can be informed of irregularities or events like for example a storage shortage or a potential attack. Handling alerts is out of scope for this thesis.

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 as not make all metrics might be accessible or through performance losses from the interference. Since K8s is designed to work with metrics the base components all publish their metrics in the Prometheus format. 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 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.

[4]Grafana: https://grafana.com/
[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 Dynamic Scaling

With the HPA acts as an implementation of a rules engine for scaling components in the K8s 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 current value for the metric 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 application scenarios.

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

K8s has three types of metrics it can use to scale pods on. *Resource metrics* which are automatically provided by K8s components. Currently supporting CPU and memory utilization. *Custom 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 accelerated changes in a specific metric. This is used in monitoring to optimize the algorithm discrepancy.

Another approach is to use separate tools 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 depending on the metrics provided. When deploying new pods the scheduler must decide on which physical node it should 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 satisfy the defined criteria. Subsequently a node is selected based on the utilization and already scheduled instances on these nodes. This allows for a fine-grained control over scheduling location and distribution.

3.2 System Design and Setup

To test a realistic scenario without the need for the actual cloud 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 efficient as possible the system runs on a single host which is itself a VM to allow for a fast setup. On this host a Docker engine runs several containers simulating real nodes and therefore creating a multi-node system of virtual nodes (vNodes). A K8s cluster containing all vNodes is deployed. The cluster hides the underlying container structure from applications running inside with the internal overlay network. Our K8s cluster contains three main components:

1. The control plane managing all K8s components,
2. the ECM application
3. and the monitoring system.

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. The monitoring system skips missing metrics until the ECM application is deployed. 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 vNodes we use kind[11]. kind is a toolkit that uses Docker containers to setup a cluster of K8s nodes. A configuration file specifies the amount of vNodes and parameters on each node as well as parameters for K8s. Each container and therefore each vNode can be limited to a specific set of resources to simulate smaller and bigger nodes. The downside is that the overall performance and resources are limited to those of the host system with this 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 a native K8s cluster. The cluster is therefore equivalent with a real multi-node system.

For our testing we use a system with three vNodes as depicted in Figure 3.2. Two worker nodes hosting the monitoring and ECM applications are created together with a control-plane node for the control-plane only. This separates the performance impact of K8s components from the rest of the system allowing for a more precise measurement. Listing 3.1 shows the cluster configuration used. It creates two worker nodes without any resource restrictions and a control plane node. The control

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

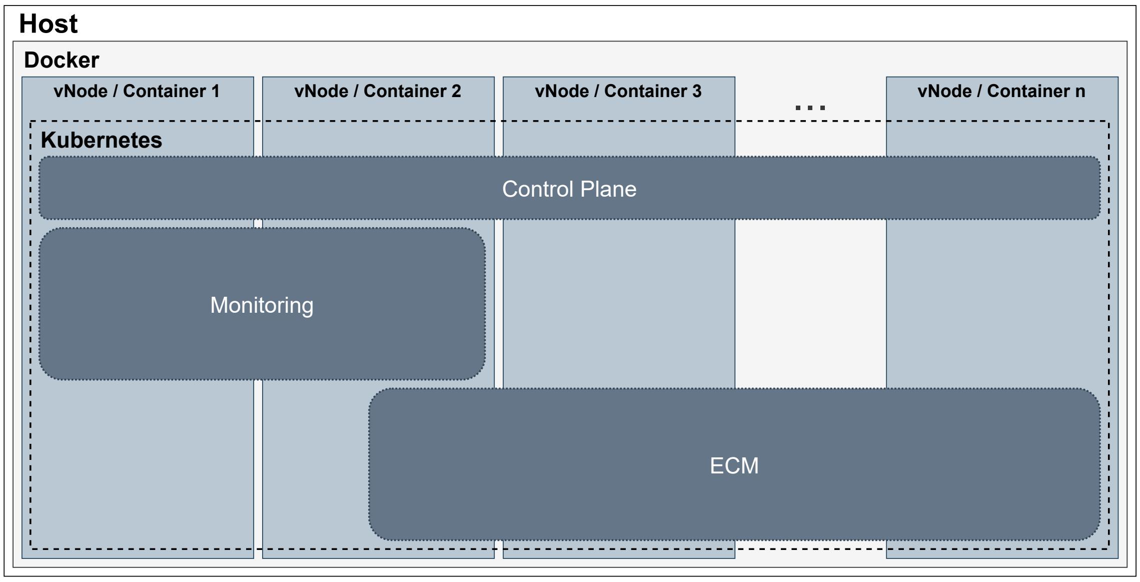


Figure 3.1: Abstract system overview showing the K8s component blocks and how they are allocated.

plane node has additional port mappings and configurations 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 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 containers, one containing the WebSphere application and another sidecar container. The sidecar container contains an NGINX webserver and a promtail instance. Every access to the WebSphere application is routed through NGINX as a reverse proxy. NGINX records all aspects 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 configuration for NGINX and Prometheus is declared in a ConfigMap object. The ConfigMap integrates and compacts multiple files into the K8s environment. This reduces the effort of handling each configuration file separately and mounting them into a container from external storage. Listing 3.2 shows parts of the ConfigMap used for the ICN NGINX. It defines four configuration files used by the webserver. At first in line seven the *nginx.conf* which defines the logging output format in the lines 16 and following. The second file *virtualhost.conf* defines the reverse proxy setup to forward traffic. The listening port as well as certificate file locations and the backend

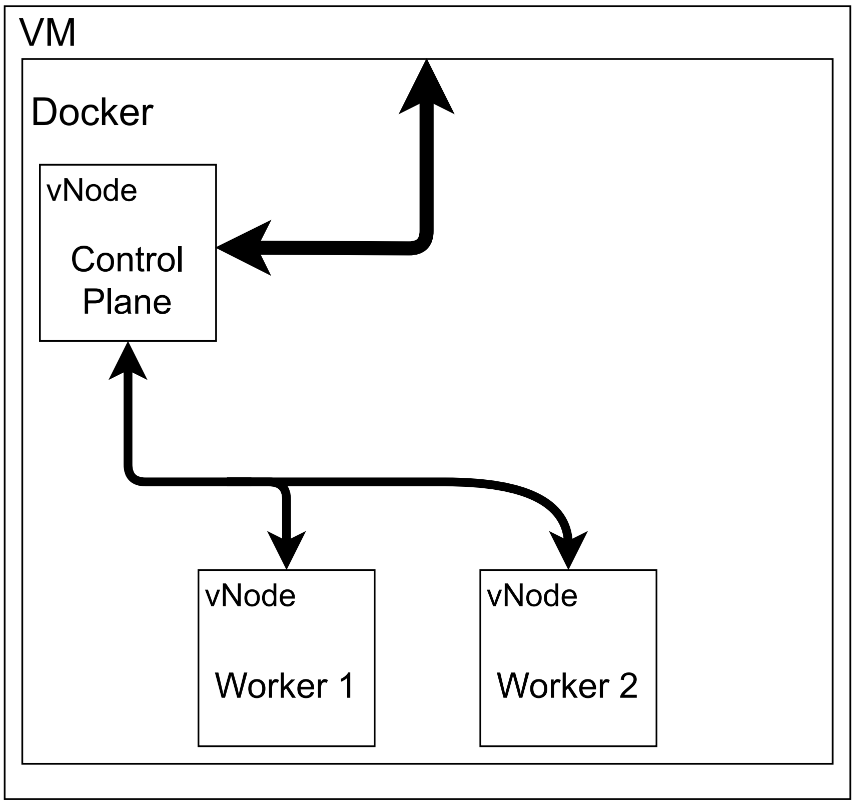


Figure 3.2: Data exchange on the kind cluster of vNodes used for testing.

address are defined here. Line 39 enables pass-through for headers with underscores in names which are used by the ICN. The certificates needed for serving HTTPS traffic are included in the ConfigMap too and later mounted to the correct path. The configuration for RMApp is similar.

The second part of the ConfigMap shown in Listing 3.3 declares parameters for promtail. The configuration is the same for ICN and RMApp. First the web port on which metrics are published is declared in line three. Then the pipeline stages starting from line nine define various parsing stages in which the input is parsed. The input consists of one line of the logfile. The metrics section from line 18 onwards defines the metrics generation. The stage in line 36 and 37 dropps the log to avoid sending it to the log management system which we do not use. The following static configs define which log file or log files are monitored for new input.

The storage needed for the LSDB and RMDB pods is provided by two persistent volumes since the database needs to be preserved. There are dedicated systems to deploy specific databases into the cloud while allowing for a horizontal scaling like Vitess[12] for MySQL. But the deployment of databases into the cloud is still researched heavily and not generally advisable. Our prototype does therefore currently not replicate the database instances. To keep the data local and the setup simple a local NFS server is used as the storage resource. The locality also helps in eliminating bottlenecks

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

Listing 3.1 kind configuration file snippet. Creating three 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: 443
        hostPort: 9442
        protocol: TCP
    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
    - |
      kind: InitConfiguration
      nodeRegistration:
        kubeletExtraArgs:
          node-labels: "ingress-ready=true"
  - role: worker
  - role: worker
```

through communication over the internet. This also resembles a production deployment on a cloud cluster where the storage is frequently provided by the cloud computing provider. Therefore, the storage is optimized for usage with the computing cluster and probably 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.

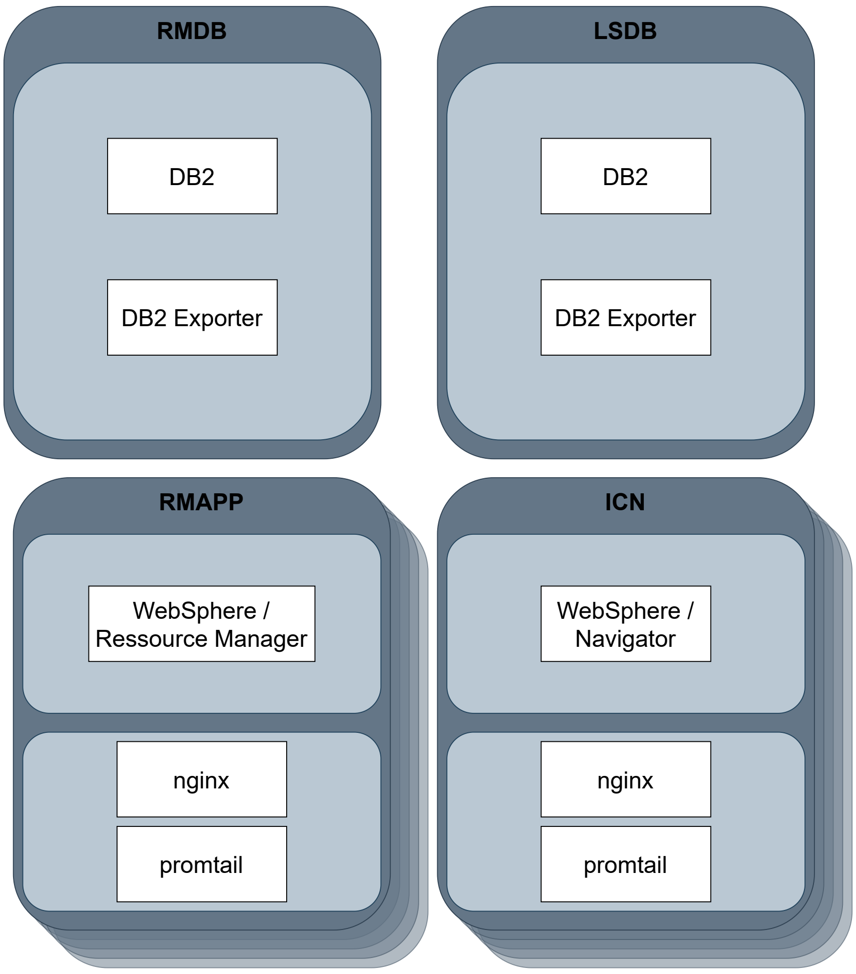


Figure 3.3: ECM components structure in K8s. Dark grey boxes represent pods, light gray boxes represent containers, white boxes represent processes.

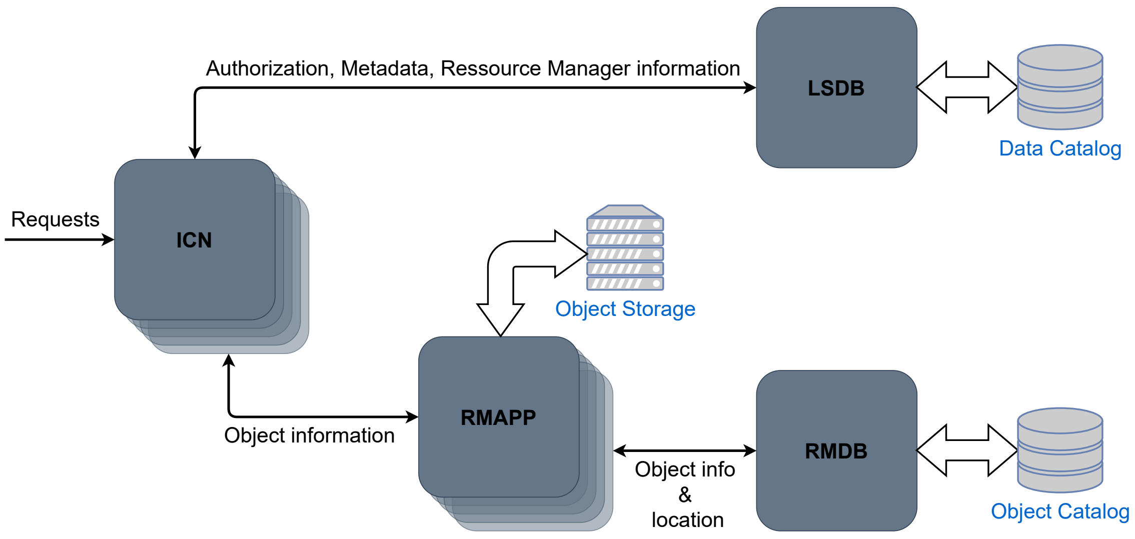


Figure 3.4: Data flow between ECM components.

Listing 3.2 K8s ConfigMap snippet for the NGINX sidecar container.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-conf-icn
  namespace: ecm
data:
  nginx.conf: |
    user nginx;
    worker_processes  auto;
    error_log  /var/log/nginx/error.log;
    events {
      worker_connections  10240;
    }
    http {
      ...
      log_format metrics escape=json '{'
              '"time_local":"$time_local",'
              '"method":"$request_method",'
              '"uri":"$request_uri",'
              '"status":$status,'
              '"bytes_sent":$body_bytes_sent,'
              '"request_length":$request_length,'
              '"request_time":$request_time'
              '}';

      access_log  /var/log/nginx/access.log main;
      access_log    /var/log/nginx/access_metrics.log metrics;
      ...
      include /etc/nginx/virtualhost.conf;
    }
  virtualhost.conf: |
    server {
      listen 9444 ssl default_server;
      server_name _;

      ssl_certificate /etc/nginx/selfsigned.crt;
      ssl_certificate_key /etc/nginx/selfsigned.key;

      underscores_in_headers on;

      location / {
        proxy_pass https://localhost:9443;
        ...
      }
    }
  selfsigned.key: |
    ...
  selfsigned.crt: |
    ...
```

Listing 3.3 K8s ConfigMap snippet for promtail to export metrics to be fetched by Prometheus.

```
promtail.yaml: |
  server:
    http_listen_port: 9113
    grpc_listen_port: 0
    log_level: debug
  ...
  scrape_configs:
  - job_name: nginx
    pipeline_stages:
      - json:
         expressions:
          status: status
          sent: bytes_sent
          recv: request_length
          duration: request_time
          path: uri
      ...
      - metrics:
         sent_total:
          type: Counter
          description: "Total bytes sent"
          source: sent
          prefix: nginx_
          max_idle_duration: 24h
          config:
            action: add
         ...
         requests_total:
          type: Counter
          description: "Total number of requests"
          prefix: nginx_
          max_idle_duration: 24h
          config:
            match_all: true
            action: inc
      - drop:
         longer_than: 0
    static_configs:
    - targets:
      - localhost
      labels:
        __path__: /var/log/nginx/access_metrics.log
```

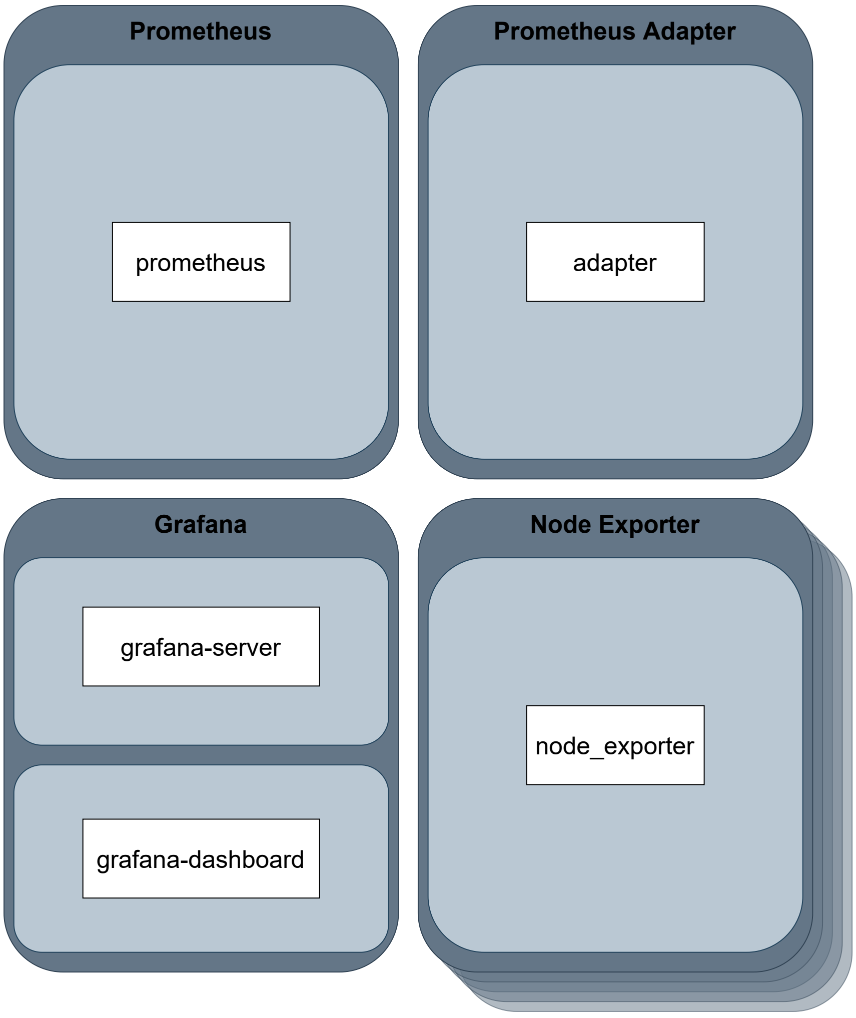


Figure 3.5: Monitoring components structure in K8s. Dark grey boxes represent pods, light gray boxes represent containers, white boxes represent processes.

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. Depending on the load prometheus replicates itself to keep up with demand. But due to the small 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*

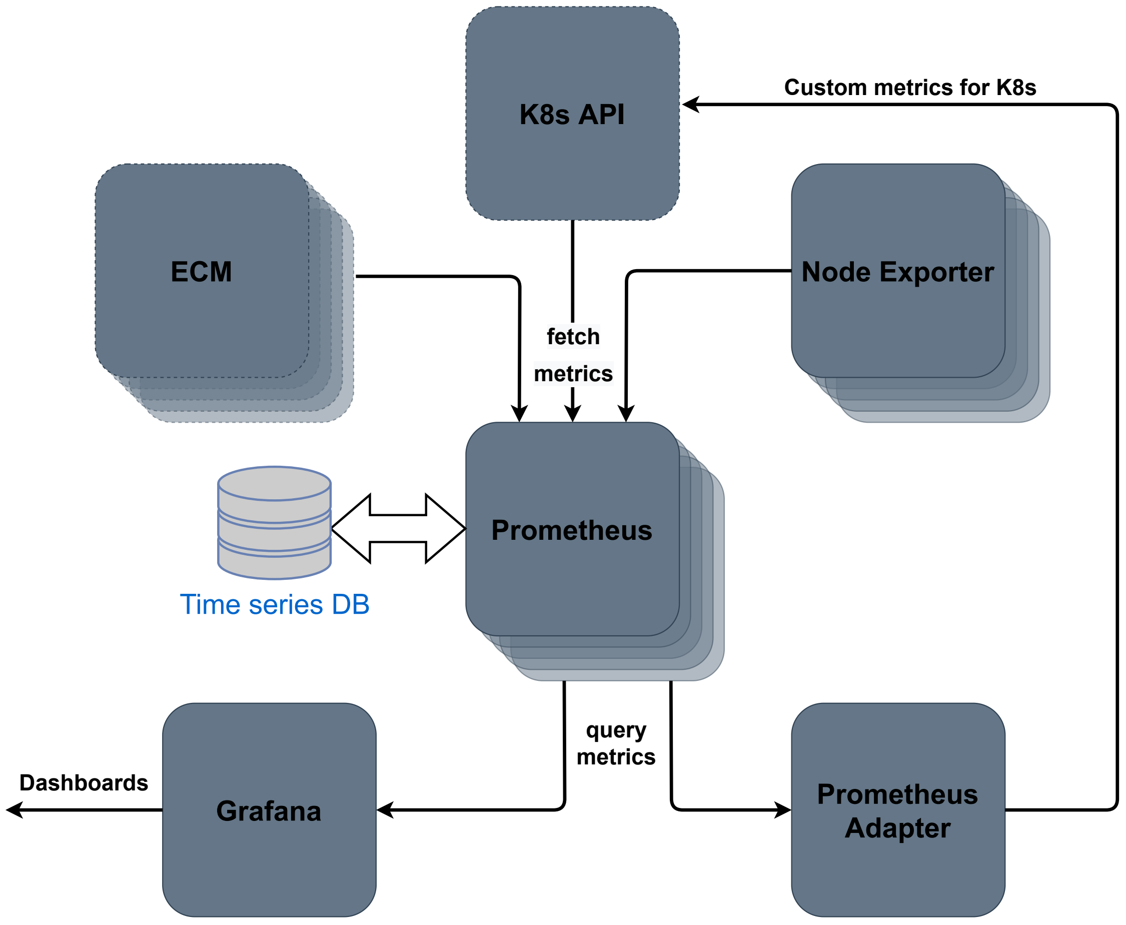


Figure 3.6: Data flow between monitoring components.

Monitors. These check the K8s API for Services with a specific set of labels. If they find a service who matches the specific set of labels, they add them to the pull configuration of Prometheus. This way it is unnecessary to specify each Service to pull metrics from but instead defines 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 configuration of the service monitor for the ICN is shown in Listing 3.4. The *spec* section describes the selector for services and the metrics port. There are two selections to restrict the services matched. The first defines to search only in the namespace ”ecm”. The other restricts the results to services with the value of the label ”app” set to ”icm86-icn”. On pods behind services matching these restrictions metrics are fetched on a port defined in the service as ”icm86-icn-metrics” every 10 seconds. RMApp uses the same configuration with only labels and names changed to match RMApp components.

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

Listing 3.4 Prometheus ServiceMonitor configuration.

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: icm86-icn
  namespace: monitoring
  labels:
    release: prometheus
spec:
  namespaceSelector:
    matchNames:
    - ecm
  selector:
    matchLabels:
      app: icm86-icn
  endpoints:
  - port: icm86-icn-metrics
    interval: 10s
```

Listing 3.5 Prometheus adapter configuration snippet to create K8s metrics with PromQL.

```
1    ...
2    - seriesQuery: 'nginx_duration{namespace!="",pod!=""}'
3      resources:
4          overrides:
5            namespace:
6              resource: namespace
7            pod:
8              resource: pod
9      name:
10       matches: ".*"
11       as: "response_time_avg"
12     metricsQuery: sum(rate(<<.Series>>{<<.LabelMatchers>>}[1m]) * rate(nginx_requests_total{<
   <.LabelMatchers>>}[1m])) by (<<.GroupBy>>) / sum(rate(nginx_requests_total{<<.LabelMatchers>
   >}[1m])) by (<<.GroupBy>>)
13     ...
```

create dashboards representing different aspects of the system. Because Grafana is not an active part of the monitoring system, it is an optional component. The main purpose is for development and administration where it can help to find and isolate problems or detect long term changes.

In Listing 3.5 a snippet of the Prometheus adapter configuration is shown. This snippet shows the generation of a custom metric from a PromQL query in the adapter configuration. The Prometheus metric to use is defined in line two. Then resources are associated with different labels of the metric. On line 11 the name of the resulting custom metric is defined. Line 12 describes the query template used to fetch metrics from Prometheus. Template parameters in double lesser and greater signs are replaced with appropriate values for the query by the adapter.

Listing 3.6 HPA definition for the ICN deployment.

```
1  apiVersion: autoscaling/v2beta2
2  kind: HorizontalPodAutoscaler
3  metadata:
4    name: icm86-icm
5    namespace: ecm
6  spec:
7    scaleTargetRef:
8      apiVersion: apps/v1
9      kind: Deployment
10     name: icm86-icn
11   minReplicas: 1
12   maxReplicas: 5
13   metrics:
14   - type: Pods
15     pods:
16       metric:
17         name: response_time_avg
18       target:
19         type: AverageValue
20         averageValue: 1
```

The ICN HPA configuration shown in Listing 3.6 defines the elasticity based on the metric created by the adapter. Lines seven to 10 define the target which should be scaled. Then in lines 11 and 12, the number of replicas is limited. Lines 13 and following define the metric to scale on based on an desired average value. The metric is referenced by name as defined in the adapter configuration. This configuration for RMApp differs in the target, the allowed number of replicas as well as the desired value. The relation between the number of replicas and the metric is adjusted by the average value.

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 network layers 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 port space 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 unused and instead an 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.

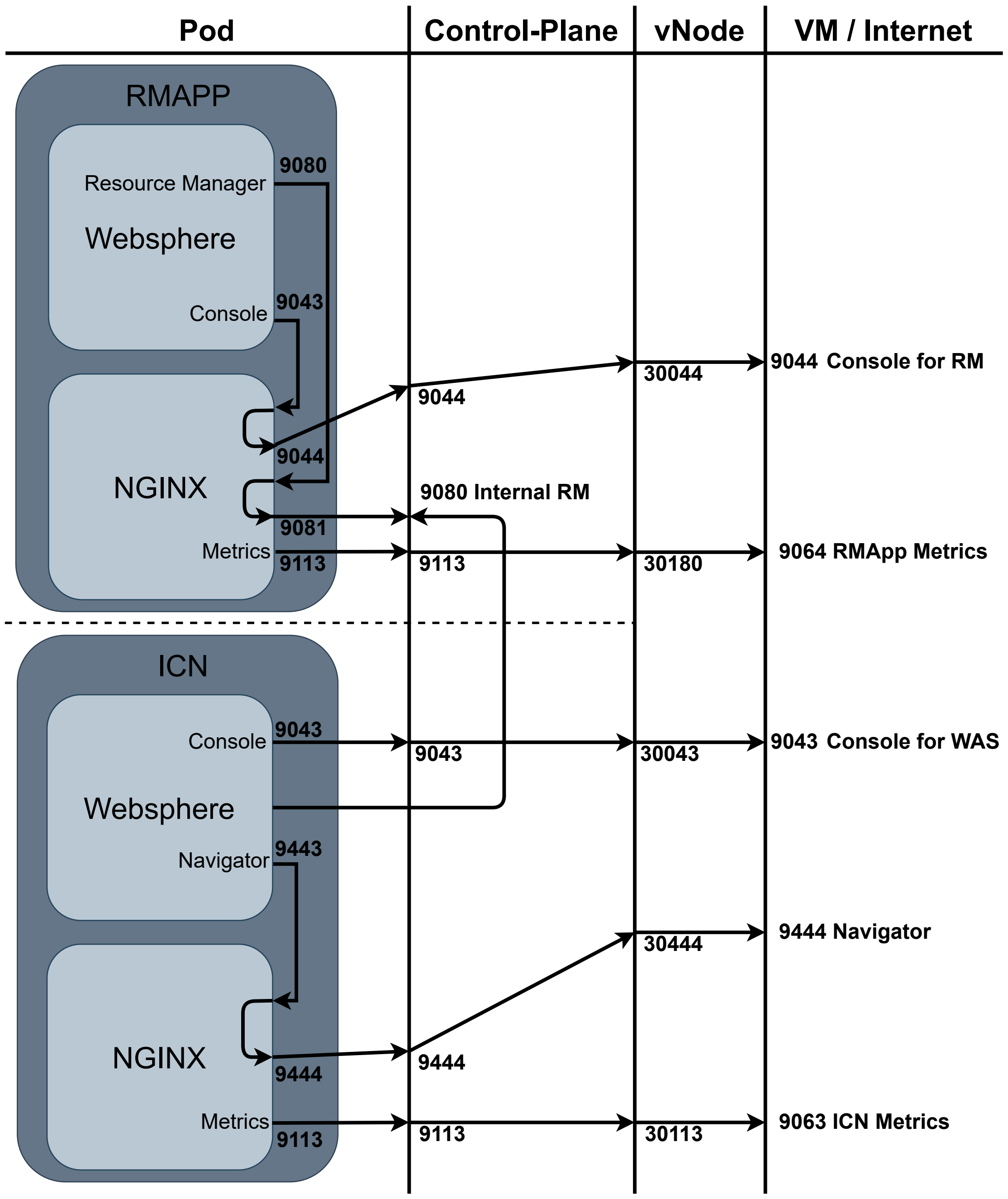


Figure 3.7: Network layers and port allocations from container level up to the VM hosting the cluster.

The port mappings on each layer allow for a fine-grained structure 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 separately. To avoid failures by accessing nodes which are in the startup or shutdown phase additional readiness checks are performed by the pods. These checks verify that the pod is in a state where it can process requests. If a pod is not ready, the service excludes this pod from the request forwarding.

3.2.5 Metrics

The system state is provided by several custom metrics introduced by the ECM application. These are provided 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 utilized 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 metrics 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 intermediate group the Sidecar metrics (2) are metrics dedicated specifically to an application. They are gathered by observation of the application from the outside often achieved 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 more traditional 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) are the most sophisticated which are directly integrated into the application. As such they provide the most insight and fine-grained information. Most applications developed with cloud deployment in mind provide this 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. Additional metrics are provided 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 occurring like creating or restarting a pod.

Variable Name	Description	Example
$time_local	Timestamp of the request (ISO-8601)	2021-08-28T17:23:20+00:00
$request_method	HTTP request method	GET, POST, HEAD, PUT, ...
$request_uri	Path of the request URL	app/v1/index.html
$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.

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

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. A more detailed description of all available variables is available in 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.

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

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 impossible 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 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 most executed methods and how they relate to other metrics like CPU and memory usage.

3.3 Testing

To test the functionality of the application scenarios with different workloads are tested. This validates the system in terms of appropriate fetching of metrics and scaling of the application. As a first step the performance of a single instance is examined without scaling enabled. From here the point of maximum efficiency for each scenario is extracted. Then scaling is enabled, and the same scenarios are simulated again to validate the functionality.

3.3.1 Setup

The setup of the monitoring environment is based on the prototype developed in [Try21]. After the initial setup the script seen in Listing 3.7 is executed to initialize the required components. Before all else the monitoring components are initialized. Then port forwarding is enabled to access the monitoring system from outside the cluster. After that the ECM application deployment is updated. This adds the NGINX webserver and configuration to the deployments and enables scaling through the HPA. When all pods are started the ECM application is deployed with a starting set of replicas. The complete setup process on the prototype system takes around 15 minutes. Ten minutes of which are used to copy the container images for the ECM application into the vNodes. As soon as the monitoring system starts fetching metrics the HPA starts to scale the application accordingly.

3.3.2 Workloads & Scenarios

We define two simple workloads for three scenarios. The first workload ”light” simulates a higher load on the database and computing performance. This workload user performs an extensive search for documents then looks at the thumbnail and information of an object from the search results. The second workload ”heavy” puts the main load on the network. The workload user opens a folder then chooses an object and looks at the thumbnail. As last action two downloads are performed. One for a small file of $\approx 100kB$ size and for a bigger file of $\approx 3MB$. As load generator for these scenarios

Listing 3.7 Shell script to setup the monitoring environment and add monitoring functionality to the ECM environment.

```
# monitoring configuration
kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/download/v0.5.0/components.yaml
kubectl patch deployment metrics-server -n kube-system --patch "$(cat monitoring/metric-server-patch.yaml)"
helm --kube-context=kind-ecm install prometheus prometheus-community/kube-prometheus-stack --namespace monitoring --create-namespace
helm --kube-context=kind-ecm install prometheus-adapter prometheus-community/prometheus-adapter --namespace monitoring -f monitoring/adapter-values.yaml
kubectl -n monitoring patch svc prometheus-grafana --type='json' -p '[{"op":"replace","path":"/spec/type","value":"NodePort"},{"op":"replace","path":"/spec/ports/0/nodePort","value":30440}]'
kubectl -n monitoring patch svc prometheus-kube-prometheus-prometheus --type='json' -p '[{"op":"replace","path":"/spec/type","value":"NodePort"},{"op":"replace","path":"/spec/ports/0/nodePort","value":30441}]'
kubectl -n monitoring patch svc prometheus-kube-prometheus-alertmanager --type='json' -p '[{"op":"replace","path":"/spec/type","value":"NodePort"},{"op":"replace","path":"/spec/ports/0/nodePort","value":30442}]'
# ecm configuration
kubectl apply -f icn/icm86-icn-config.yaml
kubectl apply -f icn/icm86-icn-deployment.yaml
kubectl apply -f icn/icm86-icn-scaling.yaml
kubectl apply -f rmapp/icm86-rmapp-config.yaml
kubectl apply -f rmapp/icm86-rmapp-deployment.yaml
kubectl apply -f rmapp/icm86-rmapp-scaling.yaml
```

the python application Locust[14] is used. It provides a quick setup and a simple web-based graphical interface for load generation management. The workloads are provided in a so called ”locustfile” which defines the work items and intervals as a python class. The work items are predefined HTTP requests which model the user interactions in the browser. The cross-platform scripting language allows for quick alteration and a wide range of host machines to run the generator from.

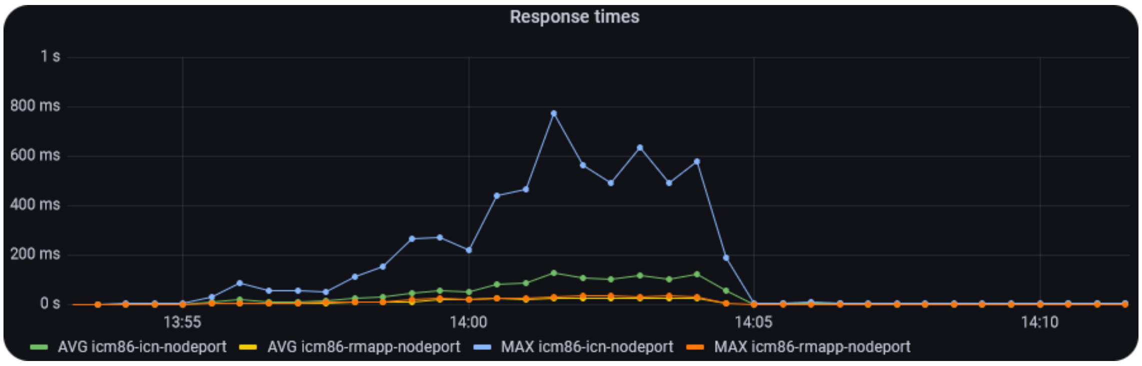
Three scenarios are tested to distinguish different patterns. The first scenario reflects many users accessing the system at the same time while not performing network-heavy tasks. This scenario will use only light workloads. The second scenario reflects high network load through simultaneous downloads. Only heavy workloads will be used but the concurrent user count is reduced to 10. The third scenario contains mixed workloads with 50% heavy and light workloads. It represents a more realistic workload pattern than the edge cases in the first two scenarios.

[14]Locust: https://locust.io/

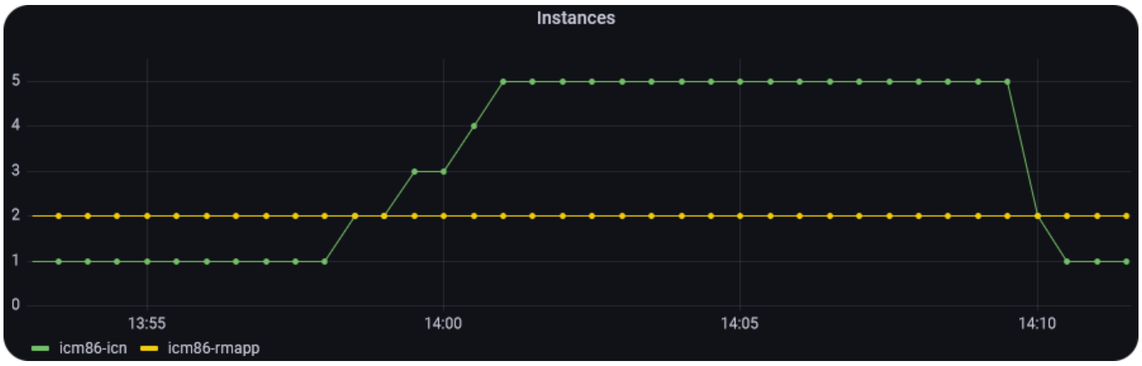
3.3.3 Results

To test the maximum number of concurrent users supported the number of users is raised in steps of five. The limit is reached if a steep increase in response times is observed indicating a struggling system. Simultaneously an increasing the Requests per Second (RPS) do not increase with an increased number of users. Testing with the heavy workload scenario indicated an approximate maximum of 10 concurrent users. The bottleneck is the network bandwidth which is at its peak of 100 MBit. To further investigate this scenario the bandwidth needs to be increased to discover additional bottlenecks.
Testing the light workload scenario revealed a maximum number of about 30 concurrent users. The bottleneck identified for this case is the RMApp response time. It is linked with the storage bandwidth limitations for the request of files. Although this workload does not explicitly download content from the system. The high storage demand is caused by files loaded to generate thumbnail images. A more effective caching of these thumbnail images needs to be applied to mitigate this bottleneck.

For the mixed workload scenario, there was a more slowly increase in response times. At about 20 users the network was once more the limiting factor. The average response time needs to be considered to meet SLAs. To verify the scaling of the components, we set the desired response time to 100 ms. This allowed the application to scale up to five instances of ICN and three of RMApp under load. Figure 3.8 shows the scaling behaviour of ICN and RMApp under load. The delay between the start of the load generation and the scaling is approximatly three minutes. A higher delay of approximately five minutes is measured between the stop of the load generation and the scale-down. A long startup and shutdown time of about two minutes for the ECM components was observed earlier. This is due to the size of the container as well as the start time of internal components. It explains the delay for the startup while an additional delay intentionally inserted by the HPA explains the delay for the scale-down. A further optimization in both aspects will increase the response time to sudden changes in the workload pattern.



(a)



(b)

Figure 3.8: Maximum and average response times (a) and number of instances (b) for ICN and RMApp from the Grafana dashboard.

4 Conclusion

Cloud computing is being adopted at a steep rate. Additionally the trend to microservices and containerization further increases integration into the cloud. Many applications and companies are still in the transitioning phase to adapt cloud technologies. This leads to systems distributed into classic deployments and cloud deployments. Connecting these systems is an additional challenge for administrators to tackle. Industry reports suggest that the transition is speeding up fueled by open-source software. 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 some proprietary solutions offer additional functionality and reduced management effort. Decision for a particular approach are dependent on the capabilities of the individual companies. 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. The most significant fields are currently undergoing major progress and research. 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 applications are ported into the cloud. Statistical evaluation of the most relevant metrics allows insight into the ECM-specific application processes. From there the required porting strategies can be developed. The K8s eco-system offers a wide range of functionalities for most applications without relying on proprietary software. The main effort lays in the steep learning curve caused by the extensive configurability of each component as well as the transition of the ECM application. The transition is made easier by the big community to support all sorts of constellations with different software. The main problems identified from the port of our prototype into the cloud are mainly down to:

1. The separation of the application into loosely coupled or independent components and
2. the identification of stateless and stateful parts followed by the reduction and aggregation of states to keep the components stateless.

For monitoring and further dynamic load balancing additional challenges arise. We choose a legacy application 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 are supported. Components which are unsupported 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 unintended, 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 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 configuration, topology description and metrics should make it easier to choose from and switch between systems.
Another point of research is the integration of stateful applications. Special care is needed to ensure the integrity and efficient organization of data at the same time. Many stateful applications like databases do not benefit enough from an adoption into a dynamic cloud environment at the moment. While deployment into cloud computing clusters is certainly possible the usage of dynamic load balancing is difficult. For legacy applications that should be brought into the cloud, there are two possibilities to consider. Either switching to applications which are developed with cloud concepts in mind or adapting their applications into the cloud environment. Further development may lead to easier adoption techniques which open 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 to a great extend and thus improving accuracy on scaling. Further analysis should see the scaling behaviour based on a mix of metrics and thereby improving efficiency. Initial testing with a limited set of workloads to validate functionality were successful. Additional tests with more realistic workloads should be performed to test efficiency 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 p. 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. 17).

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

[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. 24).

[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).

[KHF+19] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, Y. Yarom. "Spectre Attacks: Exploiting Speculative Execution". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 1–19. DOI: 10.1109/SP.2019.00002 (cit. on p. 14).

[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).

[LSG+18] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, M. Hamburg. "Meltdown: Reading Kernel Memory from User Space". In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 973–990. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp (cit. on p. 14).

[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. 17).

[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. 20).

[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. 17).

[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, 17).

[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. 19).

[Try21] C. Trybek. “Investigating the Orchestration of Containerized Enterprise Content Management Workloads in Cloud Environments Using Open Source Cloud Technology Based on Kubernetes and Docker”. MA thesis. Germany: University of Stuttgart, 2021 (cit. on pp. 31, 49).

[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. 21–23).

[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.

Backnang, 31.09.2021

place, date, signature