Institut für Parallele und Verteilte Systeme

Abteilung Anwendersoftware

Universität Stuttgart
Universitätsstraße 38
70569 Stuttgart

Masterarbeit

About the Design Changes Required for Enabling ECM Systems to Exploit Cloud Technology

Gang Shao

Studiengang: Master of Science Information Technology

Prüfer: Prof. Dr.-Ing. Bernhard Mitschang

Betreuer: Dipl.-Phys. Cataldo Mega

begonnen am: 15.06.2020

beendet am: 15.12.2020

Contents

1 Introduction 7

1.1 Components of ECM 7

1.2 Typical End User Tasks Performed Using the ECM CRUD-S APIs 10

1.3 Cloud Technologies 12

2 Challenges of Monolithic ECM systems 16

2.1 Lack of Cost-effective Scalability 17

2.2 Lack of Support for Continuous Integration and Continuous Delivery (CI/CD) 19

2.3 Lack of Support for Automated Deployment and Dynamic Topological Change 20

2.4 Summary of the Past and Current Business Requirements 21

3 Design Changes of ECM Systems for the Cloud 23

3.1 Overview 23

3.1.1 Business Goals 23

3.1.2 Technical Requirements 24

3.1.3 Design Changes Required 24

3.2 Decomposing the ECM System 26

3.3 Decomposing ECM Data Storage 30

3.4 Dynamic Topology and Orchestration 31

4 Prototype of Proposed Design Changes 33

4.1 Operating System 33

4.2 Choosing ECM Products 33

4.3 Choosing Container Platform 36

4.4 Implementation Process 37

4.4.1 Analysis of IBM ECM Components & Planning 37

4.4.2 Containerize Components 39

4.4.3 Installation and Configuration of ECM Components 43

4.5 Resulting Prototype 47

4.6 Commands for Reference 50

4.7 Summary 52

5 Conclusion and Outlook 53

6 Bibliography 54

List of Figures

Figure 1-1 Search Sequence Diagram 11

Figure 1-2 Kubernetes in OpenStack 15

Figure 3-1 Overview of Design Changes 26

Figure 3-2 ECM Component Image Registry 26

Figure 3-3 Decomposed ECM Data Storage 31

Figure 3-4 Collaboration Diagram of Starting Another Resource Manager Container 32

Figure 4-1 Process of Analyzing IBM ECM Components 38

Figure 4-2 Process of Containerization 41

Figure 4-3 Process of Installing ECM Components 44

Figure 4-4 Prototype Overview 47

Figure 4-5 Docker Local ECM Image Registry 48

Abstract

Since the late 1980s, Enterprise Content Management Systems (ECM systems) have been used to store, manage, distribute all kinds of documents, media content, and information in enterprises. ECM systems also enable enterprises to integrate their business processes with contents, employing corporate information lifecycle and governance as well as automation of contents processing. The ever-changing business models and increasing demands have pushed ECM systems to evolve into a very active content repository with expectations such as high availability, high scalability, high customizability. These expectations soon became a costly financial burden for enterprises. The on-going hype around cloud computing has raised attention with its claims on improved manageability, less maintenance, and cost-effectiveness. Embracing the cloud might be a good solution for the next high-performance ECM system at an affordable price. To achieve such a goal, the designs of ECM systems must be changed before deployment into the cloud. Thus, this thesis aims to analyze the architecture design of legacy ECM systems, determine its shortcomings, and propose design changes required for embracing cloud technologies. The main proposal to design changes are i) decomposing an ECM system to its constituent components, ii) containerizing those components and create standard images, iii) decoupling the physical link between the data storage device from the applications container by utilizing docker volumes in dedicated persistent data containers instead, iv) utilizing software-defined network infrastructure where possible. These design changes then were tested with a proof-of-concept prototype, where an ECM product was successfully deployed and tested using Docker in a cloud environment backed by OpenStack.

Zusammenfassung

Seit Ende der 1980er Jahre werden Enterprise-Content-Management-Systeme (ECM-Systeme) eingesetzt, um alle Arten von Dokumenten, Medieninhalten und Informationen in Unternehmen zu speichern, zu verwalten und zu verteilen. ECM-Systeme ermöglichen es Unternehmen auch, ihre Geschäftsprozesse mit Inhalten zu integrieren, den Lebenszyklus von Unternehmensinformationen zu nutzen und zu steuern sowie die Verarbeitung von Inhalten zu automatisieren. Mit den sich ständig ändernden Geschäftsmodellen und steigenden Anforderungen haben sich ECM-Systeme zu einem sehr aktiven Content-Repository entwickelt, mit Erwartungen wie hohe Verfügbarkeit, hohe Skalierbarkeit, hohe Anpassbarkeit. Diese Erwartungen wurden bald zu einer kostspieligen finanziellen Belastung für Unternehmen. Der anhaltende Hype um Cloud Computing hat die Aufmerksamkeit mit seinen Behauptungen über verbesserte Verwaltbarkeit, weniger Wartung und Kosteneffizienz erhöht. Die Umarmung der Cloud könnte eine gute Lösung für das nächste leistungsstarke ECM-System zu einem erschwinglichen Preis sein. Um ein solches Ziel zu erreichen, müssen die Designs von ECM-Systemen vor dem Einsatz in der Cloud geändert werden. Daher zielt diese Arbeit darauf ab, das Architekturdesign von älteren ECM-Systemen zu analysieren, seine Mängel festzustellen und Designänderungen vorzuschlagen, die für die Nutzung von Cloud-Technologien erforderlich sind. Die wichtigsten Vorschläge für Designänderungen sind i) die Aufspaltung des ECM-Systems in den einzelnen Komponenten, ii) die Kontainerisierung dieser Komponenten und die Erstellung von standard Docker-Images, iii) die Entkopplung der physischen Anbindung des Datenspeichersystem von den Applications-Kontainer hin zur Verwendung von Docker Datenvolumen in Form von persistente Daten-Kontainer, iv) die Verwendung von softwaredefinierter Netzwerkinfrastruktur. Diese Design-Änderungen wurden dann mit einem Proof-of-Concept-Prototyp getestet, bei dem ein ECM-Produkt erfolgreich mit Docker in einer von OpenStack unterstützten Cloud-Umgebung implementiert und getestet wurde.

Acknowledgments

I would first like to thank my thesis advisor Dipl.-Phys. Cataldo Mega, who intrigued me and worked closely with me on this topic. He was always available for questions, which he answered helpfully. I would also like to thank Manuel Fritz, who helped me find this thesis topic and supervised me on the practical course, during which I learned a lot. I would also like to thank Dr. rer. nat. Pascal Hirmer, who helped me understand the IaaS administration practically.

Moreover, I want to say thank you to my friends Yuwei Jin and Zhenghao Fu. Without you guys, I would not have smoothly moved to another apartment under the stress of the master's thesis. Special thanks go to Sijie Chen, who enlightened me with planning and execution tips from her two-year experience as a project manager.

I am very grateful for my proofreader Sisi Miao and Yue Wang who proved magnificent endurance in reading through my thesis. They helped me to improve my writing significantly.

Finally, I must express my very profound gratitude to my parents for supporting and caring for me, in particular, reminding me to take good care of myself.

This accomplishment would not have been possible without them. Thank you.

I sincerely wish everyone stays safe and healthy, especially during the global COVID-19 pandemic.

1 Introduction

Enterprises have to deal with a large amount of information during their daily business operations. Depending on the fields of business, the types of information, the sizes of information, and the usages of information vary significantly. To efficiently store and manage such information, Enterprise Content Management Systems (ECM systems) were devised. Not only do ECM systems focus on the capture, management, storage, preservation, and delivery of content in the business, ECM systems also empower enterprises to integrate their contents with their business processes. They facilitate access to all relevant contents needed by complex business transactions, and more importantly, ECM systems support corporate information lifecycle, and governance strategy, and legal e-discovery. For example, during the process of a claim in an insurance company, a clerk needs to gather a number of documents from different sources, then to fill out forms and forward them to related parties. With the help of the ECM system, the clerk can 1) use predefined templates for an insurance claim, 2) send and receive documents to/from other departments within the system, 3) track progress of the claim, 4) generates letters and forms specific to the client, 5) archive the claim, 6) give the client access to the online portal for the claim documents. After this short introduction of ECM systems from an end user's perspective, we will introduce ECM systems from a technical perspective.

The Association for Information and Image Management (AIIM), a nonprofit membership organization in the field of information management technologies, defines ECM as "a dynamic combination of strategies, methods, and tools used to capture, manage, store, preserve, and deliver information supporting key organizational processes through its entire lifecycle" [1].

With the definition settled, next in this chapter, we will first introduce components of a typical ECM system and typical end-user tasks performed in an ECM system. Then we are going to introduce the cloud and which cloud technologies will be utilized.

Since "content" is a widely used umbrella term more suited in the field, in this thesis, we will use the term "content" to refer to data and information which is considered important enough to be stored and managed in a company. The word "content" covers any kind of data and information of any type and format that is captured, created, changed, or used inside a company.

1.1 Components of ECM

From the business model point of view, an ECM system is comprised of five components according to the definition by AIIM [2]:

- **Capture**: The component responsible for getting the content into ECM systems. During this process, the content is firstly captured, then pre-processed, and finally, a corresponding entry is created in ECM systems.
- **Manage**: Once the content is in ECM systems, it is controlled by the ECM systems. ECM systems provide functionalities such as controlled access, version management, deletion, data extraction, forwarding, and redaction.
- **Store**: Contents, along with their catalog information, are saved in persistent storage by the Store Component such that saved data is not lost after a system reboot. Store typically consists of two function categories: the catalog and the content repository. The catalog stores primarily the metadata and indexes of actual documents, while the content repository is responsible for storing the actual documents for retrieval.
- **Preserve**: Preserve is also responsible for storing content, but it puts the focus on long-term storage. For example, system and business relevant information and contents should and must be preserved as mandated by corporate and legal retention policies.
- **Deliver**: Deliver is responsible for delivering contents to end users. Delivery channels depend on the business processes, and they typically include web portals, emails, mailing of printed copies/CDs or streaming in case of audio or video content.

The current design of ECM systems aims to satisfy those five business requirements, such that the design usually is comprised of six and more components. From the technical point of view, these components are:

- **Data Catalog** (or Central Catalog): The Data Catalog component owns and manages the central catalog of all stored contents in the ECM system. It is comprised of different data models, relational schemas, stored procedures needed to meet all kinds of business and legal requirements. It supports indexing services and functions as the central coordinator for all distributed transactions across all system components. The Data Catalog stores no actual documents or files, it only contains metadata/indexes of the contents, like logical path, date created, tags, owner, keyword, and more. The catalog is stored in a relational database using a schema that consists of a system predefined part which is complemented by a company business specific part.
- **Resource Manager**: The Resource Manager is responsible for managing the storage of original contents (documents, multi-media contents, etc.) via filesystem APIs. The Resource Manager receives requests from the Client

Application and sends the responses back, commonly via HTTPS, a secure socket layer. During the process of handling requests, the Resource Manager also communicates with the Object Catalog, which stores the metadata (such as the locations, the size) about stored original contents. During the retrieval of a document, the impact of long physical distance between the end user and the deployment location of the Resource Manager is not negligible. Thus, it is wise to deploy the Resource Manager to locations or IT infrastructures near the end users, while the Data Catalog can be deployed around the world as long as it is in a highly connected network since the Data Catalog handles search requests and search results that are usually small in size.

- **Object Catalog**: The Object Catalog supports functionalities of Resource Manager by managing the catalog of the stored contents, so it is called "object catalog". It stores such a catalog in a relational database. The Object Catalog is separated from the Data Catalog due to the logical separation of functionalities and also due to the nature of operations related to actual stored documents. Those operations often require more computing resources. Take a full-text search for example, during a full-text search operation, the indexes of the contents will be continuously accessed, requiring more computing resources. Due to such separation, the metadata in Object Catalog and the entries and references in Data Catalog must be kept always in sync to avoid dangling pointers and orphaned objects. The referential integrity between the entries in the Data Catalog and the metadata in the Object Catalog must always be assured.

- **Client Application**: The Client Application is responsible for providing the functionalities to end users with graphical user interfaces. It often utilizes web technologies, such as HTML/CSS/JavaScript to display and parse user requests and HTTP/HTTPS protocol to communicate with other services. Internally and technically, the Client Application provides a standard set of interfaces to access contents in an agnostic way, regardless of the actual storage locations. This layer of abstraction is often called Repository Abstraction Layer (RAL) or Data Abstraction Layer (DAL). With such an abstraction layer, it enables users to use basic CRUD-S operations to contents, for example, to create/read/update/search a document from the system. The Client Application is also responsible for parsing the users' requests to queries that are understandable by the underlying services, such as the Data Catalog and the Resource Manager.

- **Other components** such as Authentication Service and Search Engine are absolutely essential in an ECM system. However, their roles are out of the scope of this thesis.

1.2 Typical End User Tasks Performed Using the ECM CRUD-S APIs

ECM core operations, from the end user's point of view, are Create, Retrieve, Update, Delete, and Search. Based on these operations, the ECM CRUD-S APIs that are similar to database CRUD operations in concept are devised. In this section, we will introduce how Search, Create, and Delete are handled.

Search is one of the most common tasks in an ECM system. An ECM system is supposed to provide easy access to any kind of contents stored in the system everywhere in the company at any time. Thus, quick and efficient searching and retrieving of information is an essential part of daily business. All inquiries from employees, customers, and clients on documents, information, and statistics are processed by the Data Catalog and corresponding Resource Managers and Object Catalog, where all contents are properly indexed and classified to the requirements of these two catalogs. It is not a trivial task to implement the search functionality in ECM. It involves a number of components, including Catalog Services, Full-text search Engine, Authentication Service, and client application. Typical steps of an ECM system processing a search query can be summarized as below and are also illustrated in Figure 1-1.

1. The end user first formulates a search query in the Client Application (usually in a web page provided by the client application). Once the end user submits their query, it will be sent to the ECM Catalog Service. For instance, the query submitted is "search 'thesis' in home directory", which is made of a relational part and a full-text search part.

2. The Catalog Service then splits the query into two parts. In our example, "in home directory" is the relational part while "file(s) containing 'thesis'" is the full-text search part. It sends the former as a SQL query to the Catalog Database and the latter to the Full-Text Engine.

3. Independently the Catalog Database and the Full-Text Engine will produce two result lists and send them back to the Catalog Service.

4. The Catalog Service consolidates those two result lists. Before any data is returned to the end user, the Catalog Service must verify the authority of the user and remove unauthorized parts of the result lists. The Catalog Service sends a SQL query to the Authorization Service, asking if the user has the permission to get the results (or partial results).

5. The Catalog Service filters out the subset of the results for which the user has no permission. In our example, assuming the user only has the permission for

`/home/gang` but not `/home/centos`, only results found in `/home/gang` will be preserved.

6. The Catalog Service returns the final result list to the Client Application, which then presents the results to the end user (often as a web page in user's browser).

7. If the end user choose to view and download contents in the result list, the Client Application will retrieve the requested document from the Resource Manager.

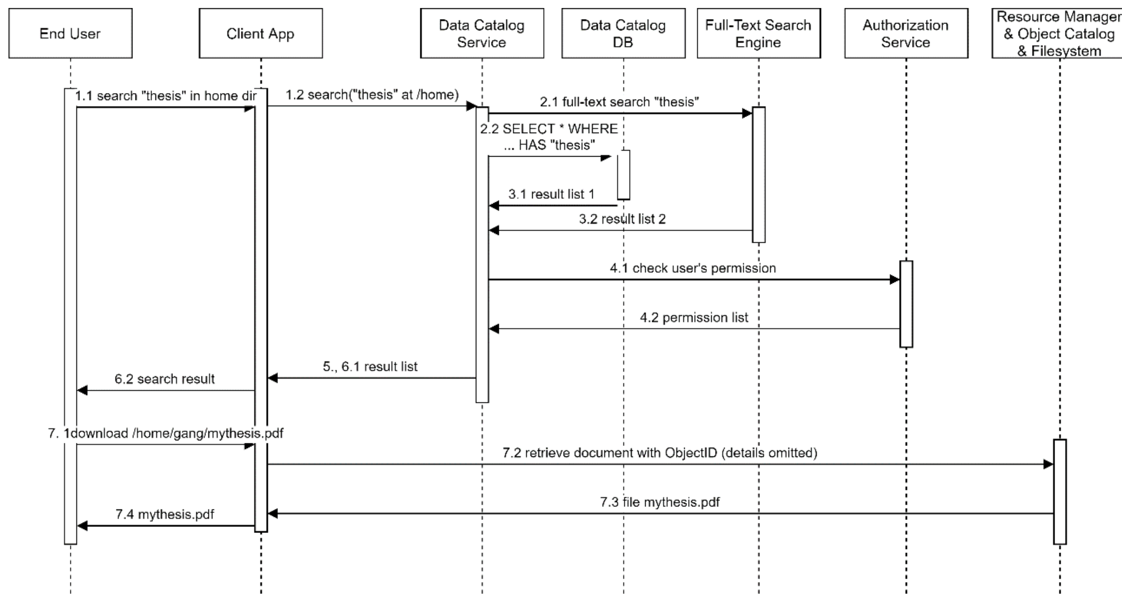

Figure 1-1 Search Sequence Diagram

Create task is one of the most important kinds of tasks in ECM systems. A common create task is described as following.

1. The end user selects a file, adds some custom metadata to it and specifies a (logical) destination directory in the Client Application. The Client Application forms the data input above to a query and sends it to the Catalog Service.

2. The Catalog Service parses the query. Before taking any action, it checks the user's permission by sending a SQL query to the Authorization Service. If the user has no permission, error messages will be returned, and the procedure terminates. Otherwise, the Catalog Service will send a quest to the Catalog Database to start a transaction, where a row of data is to be inserted, without actually committing it.

3. The Catalog Service then will try to upload the file to the ECM system's storage. If any of the steps above are unsuccessful, the transaction will be aborted, an

error message will be returned to the user and the Catalog Database remains unchanged. Otherwise, the Catalog Service will notify the Catalog Database to commit the transaction.

4. The Catalog Service returns the final result to the Client Application, which then presents the results to the end user (often as a web page in user's browser).

Deletion: If there is creation, then there will be deletion. However, deletion tasks are handled differently than Create takes. It is not done by simply replacing "create" to "delete" in queries. Of course, it depends on the implementation but usually, only a logical delete is done by the Data Catalog Service when the user sends a delete request for a specific file. The Object Catalog Service has a periodic batch job to query the Catalog Service of files to be deleted and then actually does a batch delete all documents marked as "to be deleted". This kind of deletion task is detailed below.

1. Again, the end user sends a document delete request via the Client Application to the Data Catalog Service.
2. The Catalog Service will start a transaction in the Catalog Database, where a document is marked for deletion.
3. The Catalog Service checks the user permission by sending a SQL query to the Authorization Service.
4. If any of the above fails, the transaction will be aborted, and error messages will be returned to the user. Otherwise, the Catalog Service will commit the delete transaction. The marked file then cannot be retrieved or be updated.
5. A batch job in the Object Catalog Service will periodically scan the Data Catalog Service for any document marked for deletion and do the deletion. This batch job runs independently and asynchronously.

1.3 Cloud Technologies

The exact origin of the cloud is quite a mystery but it was said that the concept of "Cloud" was first coined by George Favaloro in an internal Compaq business plan in 1996 [3]. The cloud often refers to servers, software, and database that are accessible over the Internet [4]. Main service models of cloud computing are Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS), Infrastructure-as-a-Service (IaaS), and Function-as-a-Service (FaaS). This thesis focuses on i) how the design change of ECM systems can benefit from the SaaS and PaaS aspects of cloud computing, ii) how to change the design of ECM systems to utilize features of PaaS in an environment where IaaS is integrated with PaaS (for example, the integration of Kubernetes and OpenStack [5]).

One of the most important technology in PaaS and IaaS of cloud computing is virtualization. Virtual machine (VM) technology has played an important role in computing history, making it possible to emulating multiple computer systems on a single set of hardware. This technology requires a hypervisor, sitting between the hardware and VMs, to manage and monitor VMs. A typical VM generally consists of three parts: the operating system, the supporting binaries and libraries, and the applications running within [6]. Since VMs lift the needs for actual computer hardware, they are often used as a way to reduce costs and increase efficiencies. However, there are some tradeoffs. Not only does a VM runs a full copy of an operating system, but also a virtual copy of all the hardware resources the operating system needs, which puts an extra workload, such as more RAM consumption and more CPU usage, to the hardware or the host machine. This extra cost is often negligible in the case of a sophisticated application deployed in a VM, but in the case of a small or simple application in a VM, it becomes a waste of system resources.

To mitigate the problem above, the concept of Container was brought up. Unlike VM Technology virtualizing hardware, container technology virtualizes operating systems, meaning a full copy of the operating system is no longer required. Each container shares the host OS kernel and usually some of the binaries and libraries as well. This significantly reduces the cost of duplicating a fully-fledged OS. Further, containers take the advantages of modern operating systems' features to provide similar process isolation features and resource allocation (control of the amount of CPU, memory, and file system access) as virtual machines. In the case of Linux as the operating system, kernel primitives, such as control groups (`cgroups`) and namespaces, are leveraged by containers. In this way, containers offer a solution based on virtualization that has much less overhead than that of virtual machines.

Container orchestration is to manage the lifecycles of containers, especially in dynamic and distributed environments [7], which is often managed by IaaS applications. Containers are a good way to run and bundle applications and using orchestration introduces more benefits. A container orchestrator typically provides a framework to "run distributed systems resiliently" [8]. It features functionalities such as:

- Load balancing: Orchestrators can expose containers with DNS name and monitors the traffic throughput. If the throughput is high, orchestrators can distribute the traffic to multiple containers.
- Storage orchestration: Orchestrators generally support a variety of storage solutions in different cloud computing platforms so that containers deployments are portable across platforms.

- Automatic rollouts and rollback: Some orchestrators use human readable configuration file (such as `.yaml` file in Kubernetes) to let developers describe the desired state of a container deployment. A deployment change will be carried out by orchestrators at a controlled rate so that the new containers are only visible when they are ready. All resources from the previous containers will be adopted to the new ones.

Moreover, the integration of IaaS and PaaS is desired for a higher-level functionality such as container orchestration. This cooperation offers many benefits such as

- Easy deployment with preconfigured, for example, the orchestrator conf and container images
- Dynamically managed topology based on configuration, for example, Kubernetes object (a .yaml file describing the deployment topology and the services to be configured).
- Human readable configuration
- Fine-grained control over managing VMs by managing containers (which are indeed processes)
- No additional application logics for custom watchdogs
- Unified log system

If IaaS and PaaS can cooperate with each other, it is possible to not only manage the containers and but also manages the topology/network of the hosts where the containers run. For example, Kubernetes (K8s) is an orchestrator that can be integrated with IaaS application OpenStack [5]. The architecture of Kubernetes within OpenStack is shown below in Figure 1-2. OpenStack manages the VMs for Kubernetes master and Kubernetes nodes where containers run. The containers are controlled by Kubernetes API Server, Controller Manager, and Scheduler via Kublet and an adapter called `dockershim`.

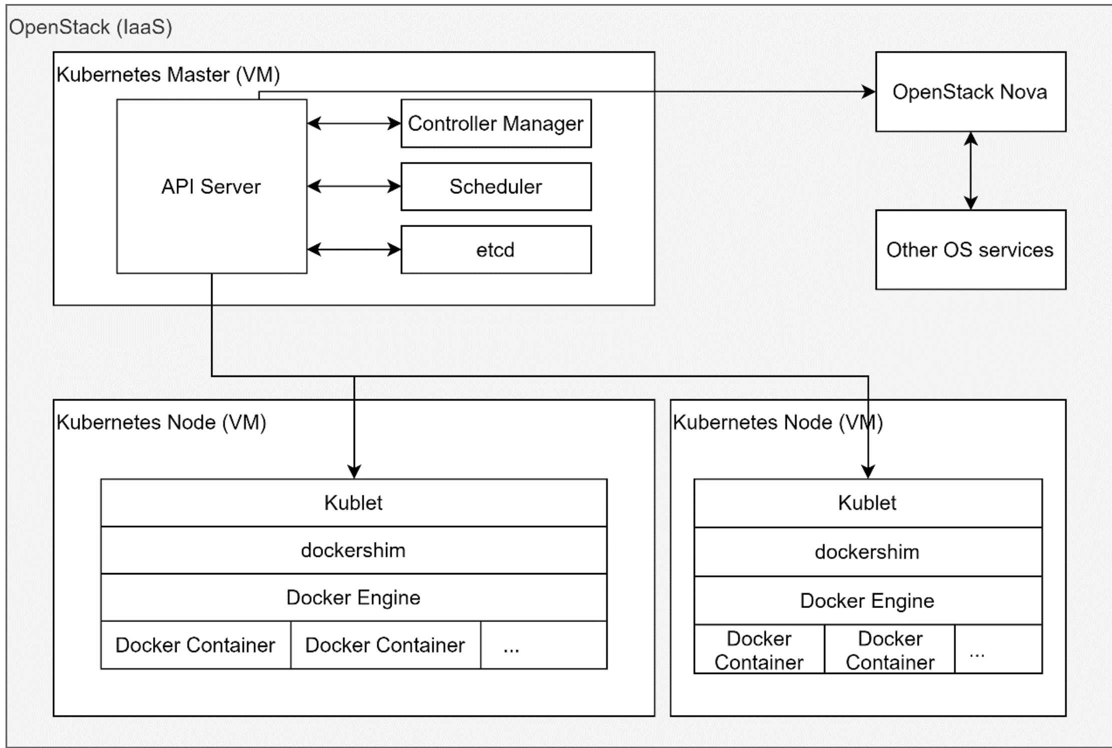

Figure 1-2 Kubernetes in OpenStack

2 Challenges of Monolithic ECM systems

As described in Chapter 1, ECM systems deal with a very large amount of data. The stored original documents are the largest, while the metadata and indexes can be large as well (full-text indexes can grow as big as twice the raw data). As time goes by, not only the amount of contents the ECM system manages grows, the number of end users accessing the service online, interactively, and from different locations grows as well. It then becomes increasingly challenging to perform maintenance tasks, like upgrading, applying security patches, and so on, to the existing ECM system.

Take online banking for example. Decades ago, if users would have liked to check their account balance, they did not have such convenient options as nowadays. They needed to call the bank's call center, make a trip to an ATM machine, or visit the bank in person. In this scenario, the software managing the bank accounts had very limited functionalities, and so were the client users' demands. A monolithic ECM system could handle it with ease. At that time, the deployment topology of the system could be defined in advance as the size of the incoming data, the number of clients, the access locations of the service, and even the time when the service would be available were known upfront. The reason behind it was that the system was only accessible by the employees at the call center, bank clerks, and ATM machines. ECM services needed only to support basic transaction functionalities and would run during the daytime only.

This is no longer the case today. The number of online users has increased in such a way that one can no longer predict accurately the actual system workload. A modern bank must be able to provide its service to a larger area and to a larger population. The number of accounts grows as the economics boosts and banks never stop advertising to gain more clients. The locations where the service is provided have changed as well. One may check their account balance of a German bank everywhere in Germany, rather than going to the city where the bank account was first created. The tunnels through which the service is provided have changed as well. Tasks like checking account balance and making a transaction are not chores anymore. One can simply check their account balance with a few mouse clicks on their computer, or by a few taps on their smartphones, regardless of where they are geographically. Services are expected to be always available so that clients can make fund transfers during the day and check the balance during the night.

All of these requirements force banks to use a more sophisticated architecture design of their ECM system. Today the goal to achieve is to load and store a constantly increasing data volume, support various access methods. In addition, the system must be highly available and have basically zero downtime even during maintenance and

upgrade. This means it is expected to support a Continuous Integration / Continuous Deployment paradigm (i.e. CI/CD). Thus, in this chapter, we will discuss the challenges a traditional ECM system faces. We are going to focus on three aspects, scalability, continuous integration & continuous delivery, and automated deployment.

Scalability roughly means that a system is able to utilize its resources as the system workload and computing resources increase and keeping the required performance levels. That means, given twice the amount of CPU power, RAM and storage, a scalable ECM system should be able to handle twice the number of requests. Application systems deployed in VMs have been able to solve various scalability issues. The details will be discussed in the following section.

Continuous Integration & Continuous Delivery is a development practice where developers continuously integrate their codes into a shared repository where the latest codes are automatically compiled and tested. With further configuration, newly compiled packages may be delivered directly to production. This practice enables quick identification of errors and keeps the application deployable at any point, enabling engineers to upgrade the application seamlessly. This is hard to achieve for traditional applications.

Automated Deployment is desired in elastic cloud environments since it helps with dynamic deployment and configuration of components. The traditional way of deployment on a bare-metal IT infrastructure is complex, costly, and requires well-trained personals. The personals must be experienced with the IT infrastructure, topology, and various application-specific installation methods and configurations. The deployment of applications is also highly subject to the existing IT infrastructures, meaning that it would take extra effort to change the deployment once the IT infrastructure changes. Traditional deployment approaches are not suited for applications in an elastic deployment.

2.1 Lack of Cost-effective Scalability

In the previous overview of scalability, we have only talked about one dimension of scalability in the definition of scalability by Neuman in 1994 [9]. Neuman defined "scalability" as the ability to handle the increased number of users and resources without suffering obvious performance loss or a significant increase in administrative effort. In this definition, the scalability is measured over three dimensions [9]:

- Size scalability: the ability to perform effectively as the resources and users begin to grow.

- Administrative scalability: the ability to manage increasing resources without a significant increase in administrative overhead.
- Geographical scalability: the ability to perform effectively regardless of the distance between the users and the resources.

Any monolithic application that can only be deployed in a single machine scales only in the limits of that specific machine. In the exemplary case about bank services above, if the ECM system were installed on a physical machine, serving as the only and central place of banking services. As the demands grow, it is only possible to scale vertically. In other words, we can of course mount new hard disks, upgrade the CPUs, and add more RAM. However, at some point, the application will hit the performance maximum of that single machine thus becoming the bottleneck.

Ten years ago, Intel's fastest enterprise CPU Xeon E7-8895 had 15 cores and a base clock of 2.8 GHz; however, in 2020, their fastest enterprise CPU Xeon Platinum 8380HL has 28 cores and a base clock of 2.9GHz, indicating the fact that increasing a single machine's computing power is not ideal. The key solution to application scaling is horizontal cross-machine scaling where a single component of an ECM system can be deployed and operated independently.

Around the year 2000 up to around 2010 [10] [11], the design of ECM systems adopted the virtualization technology. This adoption has helped to solve the system scalability issue, as the components of an ECM system deployed in virtual machines can be moved and duplicated at runtime to more powerful hardware. This solves the size scalability issue with the ability of horizontal scaling within the given IT infrastructure.

The virtualization approach solves partially the administrative scalability issue by enabling the deployment of new virtual machines heterogeneously on all kinds of hardware from different manufacturers. After the deployment in a static IT infrastructure, the ECM systems in VMs scales well in it; however, when the new requirements exceed the capacity of the infrastructure, the company then needs to change the topology of the infrastructure, which is often very costly. Not only does the change in IT infrastructure requires professionals and planning, but it also means the services are put offline during the IT infrastructure change. In summary, administrative scalability is good as long as the IT infrastructure does not change, but the administrative scalability decreases if the IT infrastructure fails to meet the requirements.

Further, as the users access an ECM system from a variety of locations, an ECM system is supposed to provide services to more locations, satisfying the geographical scalability requirements. The virtualization approach is a good solution to

geographical scalability requirements since the company can start or migrate the VMs to IT infrastructures located elsewhere. For example, an international enterprise may want to have a cluster of resource managers and object catalogs deployed in a country to provide users in that country a better experience. One thing to notice is that the cloud offers another solution to this and has its own advantages and disadvantages.

In the former solution, if there is no server room in the new deployment location, administrators must first configure the hardware in designated locations. Not only does this requires new sets of hardware, but also additional personal taking care of the servers, and rent and utility bills of the server room, before creating or migrating any virtual machines. This is quite an investment for a company at first, but once the system is up and running, the costs are averaged by the number of days and are no longer high. The "pay-as-you-go" model in the cloud gives enterprises another solution by offering deployments across the globe as long as cloud providers have support for them. The initial cost is much lower compared to self-maintained IT infrastructure; however, the cost could increase as time goes. The solution to cost-effective geographic scalability purely depends on the corporative models. For example, in a case where the business covers a very large number of countries and would like to provide fast services to them as soon as possible, deployment in the cloud might be a better solution.

2.2 Lack of Support for Continuous Integration and Continuous Delivery (CI/CD)

Continuous Integration and Continuous Delivery (CI/CD) is a popular software development practice that applies automation during development. It is created to reduce bugs and conflicts because conflicts are more likely to appear when the interval between code integrations is longer. In this practice, the code is regularly (often several times a day) integrated into a repository shared by the team. Then the code will automatically go through several pre-defined steps, such as compilation and test. This helps to find incompatibilities and errors as soon as it appears, accelerating the development process. CD is an approach that focuses on producing software in short cycles, ensuring that the software can be released safely at any time if desired. With the help of CI, developers can achieve higher development efficiency with prompt integration, automatic testing, and quick identification of bugs. With the help of CD, developers can keep the development environment always deployable, shipping updates safely to customers as soon as possible. With them combined, developers can incrementally and efficiently create reliable software and then ensures that the software can be safely deployed to production [12].

CI/CD cannot be simply applied to the development process of traditional, monolithic systems because the whole system is tightly coupled. An update of code to one component leads to an update to the whole complex software and a change to one component could lead to erroneous behavior in another component. As a result, it will take much more time for compilation, testing, and debugging. Moreover, each update in a traditional ECM system acts as an additional installation package to the existing components. During the update, the system will be shut down, meaning no service can be provided during the maintenance hours. It will be much better if we can simply replace one component with a newer version merely by a software switch, directly deliver the updates to existing deployment without any downtime.

If the ECM system is decomposed into smaller components, CI/CD can play a more important role. The CI /CD approach can be applied to two levels: One is at the team level, where it regulates the development of a single ECM component; the other one is at the whole project level, where it governs the whole software. In this way, the workload of development is distributed over teams, where a team cares mainly about its own component. By using the CI/CD approach, the team updates their code repository continuously and release only verified updates to the whole project. Then the CI/CD of the whole project will verify if the whole project runs error-free. If not, it will help to locate the problematic section by reporting conflicts and/or last working releases.

2.3 Lack of Support for Automated Deployment and Dynamic Topological Change

Many existing ECM systems are deployed with VMs, where the deployment tasks are solved by manual work of administrators or by using custom made software. For example, before adding another virtual machine, the administrator must consider several things, such as:

- When exactly to deploy new VMs?
- Do we need another fiber channel adapter / FC-cable for networking, or to configure it manually in the infrastructure management application?
- Is the existing hardware cluster enough for new VMs?
- What custom application logic should be created for managing more VMs?
- Documentation of the changed topology

All these considerations lead to an increase in administrative costs and slow down the reaction towards changes. It might take days or weeks to add a new Web Client

Application to the existing deployment. Such an approach introduces more cost for the deployment with which cloud technologies can help. In a cloud-enabled environment, administrators can define the threshold of creating another VM or container and define the desired topology. Then, the container orchestrator will try to achieve such topology using preconfigured container images of ECM components and its integration with infrastructure management applications. For example, Kubernetes is able to be integrated with OpenStack so that Kubernetes can start and stop VMs by sending requests to OpenStack.

Moreover, let's take the automatic failover as an example. Traditionally, a failover application is customized for specific usages, such as the failover application for ECM Web Application. The more failovers we'd like to add to our system, the more custom logics we must write, which in turn, increases the development costs and administration costs as we have to monitor the failover applications as well. This problem can be solved by utilizing the cloud orchestration tools if the ECM is already modified to be used in the cloud. For example, we first define the desired topology in a cloud provider, such as one load balancer and two HTTP servers. Then the cloud orchestration tool will always try to make sure there are one load balancer and two HTTP servers running. If any of them is down, the orchestration tool will try to start a new one and configure it as previously defined. The same can be applied to other components that require failover protections without additional costs.

2.4 Summary of the Past and Current Business Requirements

The previous sections have detailed how the current ECM system design has not fully met the current business requirements. Here, Table 2-1 is given for clearer comparisons between the past and current business requirements.

Table 2-1 Comparison of Past and Current ECM Requirements

Past Requirements	Current Requirements
• Service was provided to specific locations(s)	• Service is provided across countries or continents
• The workload was computed upfront	• Workload capacity (de-)increases dynamically
• Service was available only during office hours	• Always available, favorable to a 24/7/365 pattern
• Service was put offline during the maintenance	• Continuous Integration/Continuous Delivery
• Was deployed using a static topology	• Fast reaction to demand changes

• Deployment of an ECM production system took weeks/months	• Fast deployment of new ECM systems
• Scalability was only possible within the given system resources	• Scalable and cost-effective

3 Design Changes of ECM Systems for the Cloud

In the last chapter, we have detailed the challenges a typical monolithic ECM system faces and also illustrated what is expected from modern ECM systems. Cloud computing claiming to be the "faster, scalable, flexible" approach and its "pay-as-you-go" model could be a better cost-effective solution to the ECM system deployment. Thus, we will now focus on solving the design changes needed for current business requirements and for utilizing cloud capabilities. In this chapter, we will first give a concise overview regarding the design changes, then we will illustrate design changes in detail in order to mitigate the above problems and to enable traditional ECM systems to exploit the benefits of cloud technologies. It is worth noting that one of our principles is to improve the design evolutionarily instead of revolutionarily. The design changes must be non-disruptive.

3.1 Overview

In this section, we give an overview of everything regarding the design changes: the business goal we are attempting to achieve, the technical requirements to satisfy such goals, and the design changes required.

3.1.1 Business Goals

Before we list any potential changes, we must review the goals we are trying to achieve. We would like to build a system that can offer content management system as a service (CMaaS), providing such services to business that needs CM at an affordable price. The service can either be used as-is or be customized according to the customer's needs. The service should be always available in highly connected networks and so as the underlying system. The goals from a business point of view are summarized here.

- Provide Content Management as a Service, i.e. CMaaS
- Facilitate new business with easy access to CM via CMaaS, where the clients and CM are always connected
- Enable ECM to be built and configured to the customer's requirements
- Support for massive multi-tenancy
- System should be available 24/7/365
- Service should be available 24/7/365 in highly connected networks

- Support managing and processing of massive amount of electronic data at any instant
- Support electronic data of any kind
- Scale at an affordable cost

3.1.2 Technical Requirements

The business goals above pose a number of technical requirements. To achieve content management as a service (CMaaS), we must decompose the content management system to its constituent components and virtualize those components, such that CMaaS is able to be easily duplicated and to utilize hardware in a scale-up & scale-out way. Since CMaaS can be customized besides being used as-is, CMaaS should only use software-defined infrastructure (SDI) such that its infrastructure is entirely under the control of software with customizable configuration. For example, OpenStack is an open source cloud infrastructure deployed in many organizations and companies. It supports SDI and the integration of container orchestrators, such that it is able to manage the infrastructure of ECM deployments automatically provided that configurations are given. Further, to support massive multi-tenancy and to achieve high availability, it must have dynamic workload management capability and reliable network connectivity. Cloud technology claims to meet almost all the requirements above – virtualization, scale-out utilization, high availability, connectivity, and dynamic workload management. Thus, it is key to support cloud technology. The following summarizes the technical requirements to be satisfied.

- Decompose the CM to constituent components, virtualize them and create container images
- Utilize hardware in a scale-out way using container orchestrator technology
- Software-defined infrastructure (e.g., OpenStack as an IaaS IT platform) and enables dynamic workload management together with Docker/Docker-compose
- Runs efficiently in the cloud and have support for cloud technologies like Kubernetes/OpenShift or the like

3.1.3 Design Changes Required

Our priority was to decompose an ECM system to as small as possible components such that they can run independently as a sort of service independently in containers. The data storage shall also be strictly separated from the application itself because, in

a cloud environment, applications in containers are often ephemeral while the data must remain persistent.

Secondly, to achieve software-defined infrastructure, we should not use bare-iron hardware appliances nor should we use inner application logics to manage the topological deployment structure of the ECM application system. Instead, this job should be delegated/mapped to a chosen orchestrator (such as Kubernetes) and ECM specific deployment templates.

Thirdly, the network should also be defined programmatically, which has become a common practice named "Software-defined Network" (SDN). For example, OpenStack supports an easy configuration of the network for the virtualized machines it manages. Another example is Docker Engine, it supports SDN for containers in a single host.

Thus, all these changes are summarized below and also are illustrated in Figure 3-1, where a traditionally deployed ECM system is shown on the left and an ECM system with the proposed design changes is shown on the right.

- Split an ECM system to as small as possible components such that each functions as a stand-alone service in a container (marked in the figure with ①).
- Create a container image repository for accessing and storing above ECM components (shown in Figure 3-2)
- Separate data storage from its application to keep data persistent (marked as ② in the figure)
- Use orchestrators (e.g. Kubernetes) to manage the topological structure of containers, rather than using inner application logics (marked as ③ in the figure)
- Use software-defined network instead of hardware-defined load balancing and communication between components (marked as ④ in the figure)
- Add component/application-specific configuration scripts that will finalize the required configurations, such as hostname, public IP-address, data sources, filesystem/storage path inside a volume, instance/privileged users and startup/stop scripts, and trace and log settings.

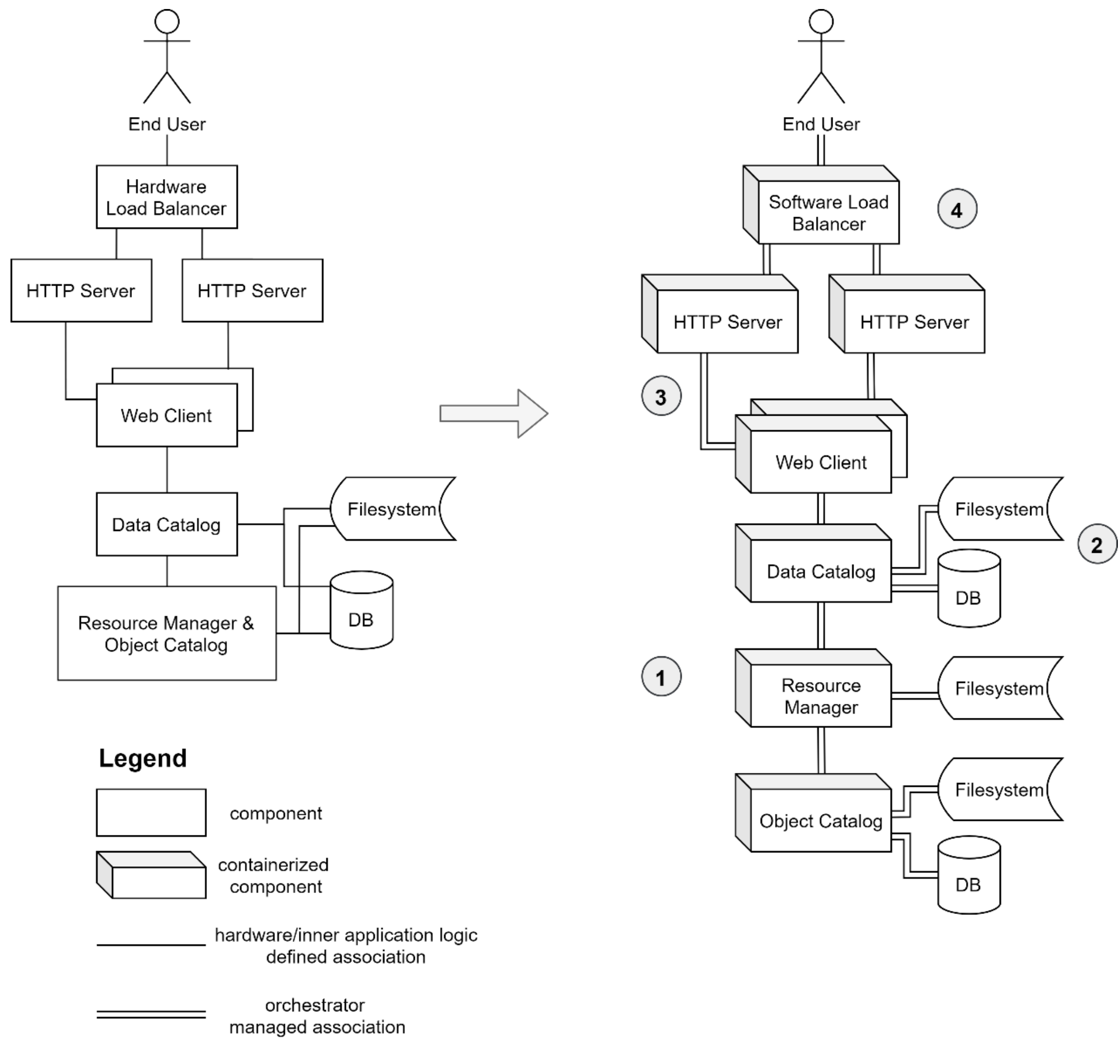

Figure 3-1 Overview of Design Changes

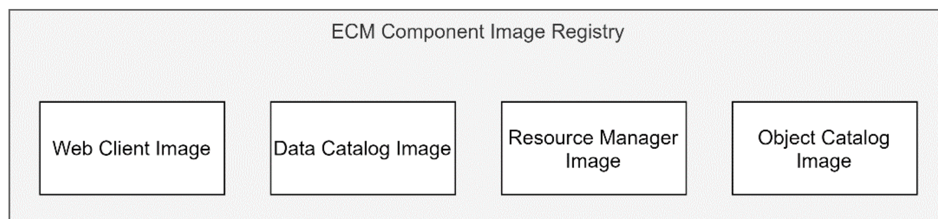

Figure 3-2 ECM Component Image Registry

3.2 Decomposing the ECM System

As detailed in Chapter 2, traditional ECM systems are often deployed as monolithic systems on physical compute servers, or on a mix of bare metal servers and virtual machines. This approach has no uniformly defined topological structure, meaning any of the components might be deployed together in a single VM in one situation while

they are deployed separately in other situations. Thus, it often holds assumptions about the infrastructure that might not be true in a cloud environment. These assumptions often include:

- Access to shared systems
- Deployed in the same cluster where direct access is possible
- Shared libraries/dependencies
- Complete knowledge of the topological structure

Unfortunately, these assumptions do not perform well with the concept of elasticity in a cloud infrastructure. If the application is configured to run on several machines (or virtual machines), it might also have assumptions such as:

- The machines (virtual machine) belong to a static network shared among a dedicated ECM cluster of VMs and will be long lasting
- All operating system and application functionalities are provided
- Certain OS system privileges such as root, instance owner, and the like are given

Hence, the first step of the design change is to analyze and decompose the application into smaller components that can function on their own and can cooperate with other components through unified communication channels. Components that are tightly coupled are not worth separating and they should share one container, for example, the synonym matching module might be built within a full-text search engine, then it should be preserved.

We investigated each component that poses special requirements on the system, like access to a shared system and certain versions of libraries/dependencies. Since we are changing the design non-disruptively, we need to mark them, define the correct topology, and alter no code. We also will discover components that can run as standalone containers and map them to their places in the business model.

Next, we will apply the aforementioned principle to traditional ECM systems. As shown in Figure 3-1 above, the design of a monolithic ECM is shown on the left, and the proposed changed design is shown on the right.

Data Catalog: Data Catalog component hosts metadata and indexes of all the stored documents. The indexes and metadata are later used to manage the actual documents. When a client request arrives at the Data Catalog, it will look for the

metadata of the document, and returns a reference of the document (including its URI) and an authorization token to the Client Application API. Afterward, the Client Application communicates with the Resource Manager to fetch the actual file with the reference and the token.

In the proposed changed design, the Data Catalog is separated and containerized. It now has its own file system and database for the storage of metadata and relational indexes. In this case, the Data Catalog communicates with the client and Resource Manager through predefined communication channels, e.g. `REST`, `HTTP`, and `JDBC` protocols. The Data Catalog component has no access to the actual files and has no knowledge of the document's physical address, but only the metadata and indexes which are enough to identify a document among others. The job of store and fetch the actual file is delegated to the Resource Manager(s).

Resource manager: The Resource Manager is responsible for storing, retrieving, updating, and deleting the actual document stored in the ECM system with the necessary information from the Object Catalog. In an exemplary monolithic design, Resource Manager is implemented in 3 components: i) the resource manager J2EE application, ii) the resource manager database that holds the object catalog and iii) the resource manager physical file storage which can be a filesystem, a tape library or cloud object storage.

An implementation of the resource manager application is that it runs in a J2EE Web-Application Container like IBM WebSphere and communicates with the Object Catalog directly via a `JDBC` connection. It communicates to the object store using FS-APIs and to the ECM users via HTTPS, a secure socket layer.

The Resource Manager is changed to be isolated from the application into a container in the new design for faster startup and easier orchestration. Thus, its functionality and position in the whole application become more distinguishable. It acts solely as the middle application for fetching the actual document in storage by accepting requests from the Resource Manager and by gathering needed metadata about the actual document from Object Catalog. Since it is containerized and a corresponding image is created in the proposed design, another Resource Manager can be started somewhere in the cloud environment (also in different geographical locations) and configured to accept requests from Data Catalog and have the same access to the filesystem of the previously deployed Resource Manager and Object Catalog, dynamic increase its processing capability.

Object Catalog: Object Catalog, backed up by a relational database, holds the metadata of the actual documents in the ECM. It is the backbone of the Resource Manager because it provides the necessary information about documents (such as the physical location, and the size) to the Resource Manager. In the monolithic design, Object Catalog is coupled with Resource Manager. In our proof of concept, we used the IBM DB2 RDBMS to prove our approach.

In the proposed design, Object Catalog is in its own container, acting as a service providing catalog functionalities to the Resource Manager. This design change unbundles the process of create/delete/get/update the actual document to two parts: Resource Manager APP for the actual file manipulation and communication, and Object Catalog (or called RMDB in IBM Content Manager Enterprise Edition) for indexing stored files and storing file locator records. Since it is containerized to a service with a simple yet clear topology, when a new Resource Manager is created, a corresponding new Object Catalog can be easily spawned as required. The communication between the Resource Manager and Object Catalog uses a `JDBC` / socket channel.

Client Application: Client is the application providing functionalities to end users, such as presentation, management, and delivery of stored contents. In a traditionally designed ECM system, the client would be deployed in the same system as the rest of the components. It communicates with the Data Catalog via a standard channel such as HTTP/HTTPS and returns the results to the user.

With the proposed design, the Client Application is containerized. Its functionality does not change, since it still uses standard protocols (like HTTP and HTTPS) to provide access to end users and communicates with other components. The key benefit is with the container image of the Client Application, it is possible to spawn more Client Applications when the number of queries is high and vice versa.

Summary: In the decomposing process, we first analyzed the existing ECM design and decomposed it into its smallest components. These components are then containerized to include the necessary libraries and dependencies, such that they can run as a stand-alone service container once a correct configuration is provided. Containerization looks similar to creating virtual machines, but it is different in the following ways:

- Smaller footprint of each component
- Faster startup time of containers as they share the operating system

- Computing resource controlled to the process level
- Easier orchestration of the single components

3.3 Decomposing ECM Data Storage

Data is the key for many applications, and it is of exceptional importance in ECM systems. ECM systems are made to manage documents. In order to manage documents, ECM comes with predefined data models related to management, administration, operations, authentication, authorization, and access control, as well as the ability to accommodate business application-specific data model extensions like document management, archiving, and information lifecycle and governance.

To complement the data model, the management of physical data storage must be considered. In a cloud infrastructure where applications in containers are ephemeral, we must separate the storage of data from the application and put these data in persistent storage volumes. The pattern "database per service" [13] is often used in cloud environments. This pattern advises each service should encapsulate, govern, and protect its own domain model and persistent storage. For security reasons, only one service should have access rights to data storage. We can adopt this pattern in our service containers as well. The Object Catalog should manage its own domain models and its database, in a way to make it easy to identify from the topology, to migrate to another physical storage, to protect its integrity, and to secure from attacks. The same applies to Data Catalog.

Hence, a storage volume should be created and mounted to a container. This is shown in Figure 3-1, marked with ②. And it is also shown below in Figure 3-3.

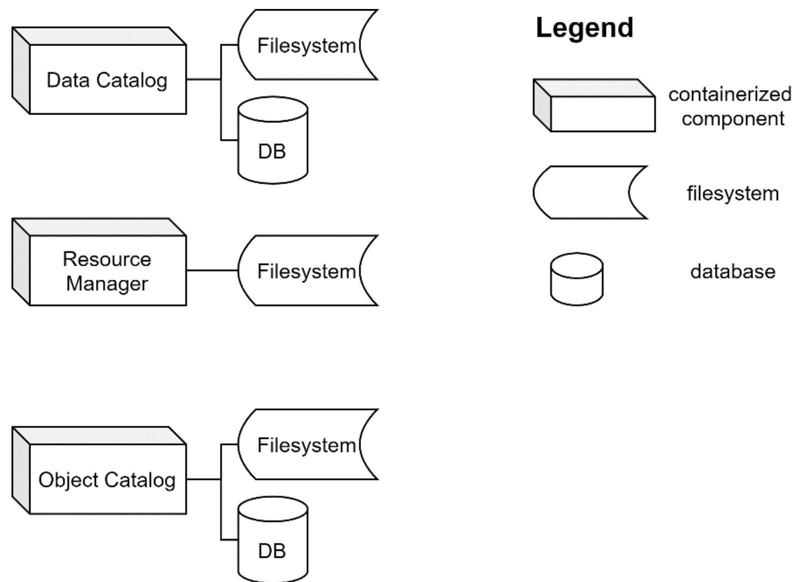

Figure 3-3 Decomposed ECM Data Storage

When a container is shut down or maintained, it is always possible to just replace the application itself while preserving the data. A side bonus is the separate data volume can be easily packaged and duplicated like normal files.

3.4 Dynamic Topology and Orchestration

Software-defined networking (SDN) is a network management strategy that configures network programmatically and dynamically, aiming to improve network performance and monitoring [14]. In an IaaS managed cloud environment, the infrastructure and network are managed by the IaaS software. Since the container orchestrator can work together with IaaS software [5], it is possible to achieve a dynamically managed topology. For example, if we set in Kubernetes that it should start another Resource Manager when the system resources such as CPU and RAM utilization is high, it will communicate with OpenStack Nova (an OpenStack component that manages VM instances) to start another VM where Kubernetes can start and manage another Resource Manager based on the container image of Resource Manager. Since the Resource Manager image is preconfigured with the existing Object Catalog, it will function as the previous one. This process is demonstrated below in Figure 3-4.

1. Kubernetes (“K8s”) monitoring service in the AIP Server monitors the criterion of starting a new RM is met

2. K8s sends a request to OpenStack Nova to start a new VM. Nova then contacts with other OpenStack services such as authentication service Keystone, image management service Glance, and block storage management service Cinder (not shown in the figure). After successful authentication, image pulling, and volume attachment, a new VM is started to be used by K8s. K8s API Server then starts a pod there.
3. The pod will pull the RM image from the private image registry and start a new container of RM.

Further, to balance the requests, Kubernetes should also start a software load balancer for the incoming requests from client applications (not shown in the figure).

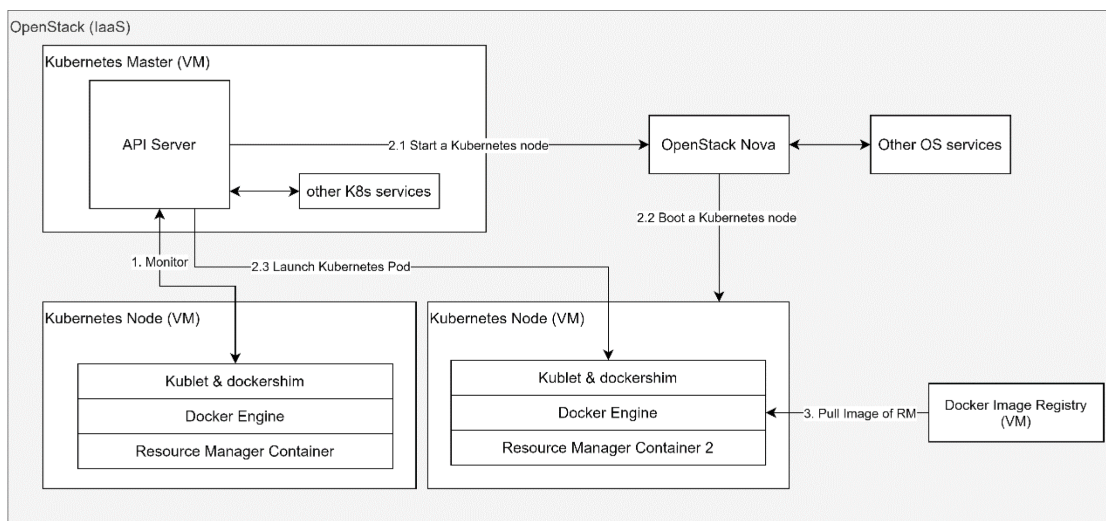

Figure 3-4 Collaboration Diagram of Starting Another Resource Manager Container

4 Prototype of Proposed Design Changes

Previous Chapter 3 has detailed the design changes for ECM systems to embrace cloud technology. In this chapter, we will illustrate the process of creating a prototype of the proposed design, which should serve as a proof of concept. In this process, we first chose the operating system, container platform, and ECM product, then analyzed how the chosen ECM product's implementation matched the generic ECM design, and finally, we decoupled the ECM product into constituent components and put those components in containers.

4.1 Operating System

The operating system in the prototype setup was CentOS 7.4. CentOS is a free and community-supported Linux distribution that "focuses on delivering a robust open-source ecosystem" [15]. It was chosen because of mainly two reasons. Firstly, CentOS is free of charge. Secondly, CentOS 7 is very similar to Red Hat Enterprise Linux 7 (RHEL), and the latter is officially supported by the IBM ECM product family. Thus, choosing CentOS 7.4 resulted in maximum compatibility during the prototyping process at zero monetary cost.

4.2 Choosing ECM Products

Several ECM products from a variety of companies exist in the current market, such as Alfresco Content Services (ACS) from Alfresco Software Inc., OpenText Content Suite from OpenText Corporation, and IBM Content Manager Enterprise Edition from IBM Inc. The design changes to legacy ECM systems are meant to be in general, instead of focusing on one specific product from a certain company, such that our prototype does depend on which product is chosen. Some notable ECM products are briefly introduced below [2].

- Alfresco Content Services (ACS) [16] is an ECM system from Alfresco Software, which has played an important role in the business of Alfresco Software Inc. since its foundation. ACS is built with the Java Spring framework, such that it is able to support various operating systems and allows modularized functionalities. ACS adopts the client-server model and it consists of a web-based client named Alfresco Share and a server-side application named "content application server". ACS offers some value-added services that extend the content application server.

- OpenText Corporation (hereinafter referred to as "OpenText") offers several ECM solutions, one of which is OpenText Content Suite ("OTCS"). According

to OpenText [17], the design of OTCS focuses on Service Oriented Architecture (SOA) with three core services, namely Enterprise Service Bus, Business Services, and Master Data Management. On top of those services, lie the Busines Process Management Suite (BPMS), which includes four components such as business process management and case management. OTCS adopts the client-server model and offers user interfaces through web technologies.

We have chosen to use IBM Content Manager Enterprise Edition (ICM) to demonstrate the proposed design changes of the ECM system. The foremost rationale behind this decision was that the IPVS department in the University of Stuttgart, where this thesis was conducted, has had a tradition of a cooperative partnership with IBM in Böblingen. IBM also hosts a publicly accessible knowledge center for its ECM products, which came in handy during the prototype process. IBM ECM suite represents a typical ECM system, and the conclusion should hold for other ECM products as well.

IBM ECM suite claims to offer a "complete set of enterprise content management capabilities to capture, analyze, and engage with content" for business [18]. This software suite includes a variety of IBM proprietary software for ECM, including front-end and back-end since they all adopt the client-server model. For the back-end, IBM offers three different server products with their own focus.

- **IBM Content Manager Enterprise Edition (ICM)** is a traditional ECM software to manage almost every kind of business documents. It supports multiple operating systems and includes administration tools and user management. It also offers a set of web service APIs to be leveraged by a company's own application or the front-end products from IBM.
- **IBM FileNet Content Manager** is an ECM suite released after IBM's acquisition of a company named FileNet, along with its flagship product FileNet P8, in 2007. FileNet P8 could be used as-is, while it could also act as a framework for developing custom enterprise systems. IBM claims that FileNet Content Manager focuses more on the cloud, however, it remains to be verified since its history of development is as old as ICM.
- **IBM Content Manager OnDemand** is yet another ECM application offered by IBM. It is specialized for well-defined structural documents, such as forms. Users can use predefined processes to generate documents where each field is filled with custom data. Then those generated documents are sent to clients. In this way, it is sufficient to store only structured information/data and templates. This saves much storage space since generated documents are

considerably larger than original data, especially in cases like monthly reports for all bank accounts.

For the front-end, IBM offers IBM Content Navigator as described below.

- **IBM Content Navigator (ICN)** is a web client that enables end users to work with content from multiple content servers via a graphical interface. Users can browse, search, edit, and add documents with ICN. Engineers can also utilize various extensions and APIs to extend the ICN with custom actions, menus, and layouts. It is compatible with IBM Content Manager Enterprise Edition, IBM Content Manager OnDemand, IBM FileNet P8 repositories, and OASIS Content Management Interoperability Services.

In the following chapters, we will take a deeper look at ICM. Besides, IBM products will be referred to by their abbreviations.

Content Management System: IBM ICM is the CMS product we have chosen. As explained in the introductory chapter, CMS consist of Data Catalog, Resource Manager, and Object catalog. The general design holds for most ECM products in the market, but the components are named differently. In IBM ICM, Resource Manager is named Resource Manager Application (RMApp), Object Catalog is named Resource Manager Database (RMDB), and the Data Catalog manager application and the Data Catalog database is named Library Server (LS) and Library Server Database (LSDB). The RMApp is a J2EE application and can communicate with the RMDB directly via a JDBC connection. RMApp communicates to the object storage using FS-APIs and to ICN via HTTPS.

Web Application Server: Web Application Server is the server that hosts web applications, provides both facilities to create web applications and a server environment to run them. In our case, this is IBM WebSphere Application Server (WAS). WAS is a J2EE Web Application Container and it is needed to support the runtime of RMApp and ICN.

Relational Database System: It is responsible to host all kinds of metadata, configuration, indexes, and other data required by CMS. In the prototype, we use IBM DB2, because it is one of the two supported DB by ICM (the other is Oracle Database). In the implementation, both LSDB and RMDB are backed up by two separate DB2 installations.

Content Web Application Client: It presents the functionalities to end users and also parses the queries from end users. Here we use ICN, which is a J2EE application that needs to run in a J2EE Web Application Container (in our case, WAS). ICN must be

configured to be connected to ICM and RMApp before usage. ICN also requires a relational database to store its configurations, which are small in size. Thus, the configuration of ICN also should be backed up by DB2.

4.3 Choosing Container Platform

Docker is a powerful and open platform for "developing, shipping and running applications" [19], because of the ability provided to package and run applications in a container. We choose Docker and Docker Engine as the container platform in the prototype. There are several reasons for this decision:

- Docker is the most popular container platform to date. It shared 79% of the container market in 2019 and 83% in 2018, according to two reports by Sysdig [20] [21].
- Docker has the most comprehensive documentation and community QA due to it being the number one container platform.
- Docker has a higher chance of being compatible with IBM ECM products because IBM has released 2939 containerized applications with Docker, like DB2 in Docker container and WebSphere Liberty in Docker container [22].
- Functionalities provided by Docker are sufficient for our purpose, which are detailed below.

Base Images: The prototype relies on products from IBM other than IBM ECM suites, such as IBM DB2 and IBM WebSphere. These two then pose requirements on the operating system, such as CentOS (RHEL) version 7 or later. When prototyping, we started from several base images, such as CentOS 7 and IBM DB2 Community version on Docker. The huge collection of base images was very helpful.

Volume: It is incredibly easy to mount volumes to Docker containers. Docker offers two types of storage, Docker volume, and bind mount. Volumes are located in a part of the host filesystem that is managed by Docker. Bind mounts can be stored anywhere on the host filesystem, which can be modified by both Docker and non-Docker processes. Both options are very easy to use and are designed for different purposes. Docker advises the usage of volumes in most of the case. Volumes are good for sharing data among multiple running containers, storing data on cloud providers, backing up and migrating data for a Docker container, and so on. Bind mounts are good in the cases of host sharing files with containers, such as sharing a binary built by the host to containers for testing.

In our case, we used the so-called bind mounts. First of all, bind mounts and volumes do not make a noticeable difference during the prototype. Secondly, a bind mount might be a good choice in some cases. The default docker volume path in Linux is `/var/lib/docker/volumes/<volume-name>`, which might not be desirable in a system that has limited storage by itself but has mounted external storage under other directories.

Network: Docker enables containers and services to be connected together, as well as connected with non-Docker workloads. Docker can manage the network mapping in a platform-independent manner, such that applications and services running in Docker containers have no knowledge if they are deployed in Docker containers, nor do they need to have.

From a technical point of view, Docker Engine manages the networks outside of containers. Developers can create a number of networks in Docker Engine, and assign containers to a certain network. Containers assigned to the same network can communicate with each other with hostnames (which can be user-defined), which is also dynamically added to each container by Docker Engine.

Orchestration: Docker containers are compatible with a couple of orchestrators, such as Amazon ECS container orchestration service, Kubernetes, and Docker Swarm. While Docker's own component Docker Compose offers management of containers in one host, those orchestrators have the capability to orchestrate containers in multiple hosts. Docker's compatibility with many orchestrators allows us to adopt SDI and SDN as well as dynamic topology management.

4.4 Implementation Process

In this section, we will illustrate the details of the implementation process. We first analyzed the existing IBM ECM products. Each component has a different design and it needs to be handled differently. Then we attempted to containerize components according to the analysis and installed each component into its container. A flow chart is added in each subsection for clearer illustration.

Some important commands are listed in this section as well, such that these processes are reproducible.

4.4.1 Analysis of IBM ECM Components & Planning

Figure 4-1 shows the process of analyzing existing ECM components from the IBM ECM suite. We investigate the components of the IBM ECM suite one by one, to see if it can be decoupled to stand-alone services. If a component can be containerized, we

then containerize it; otherwise, we attempt to identify with which it is tightly coupled. After every component is analyzed, we then look at them as a whole, to verify whether it will operate correctly or not.

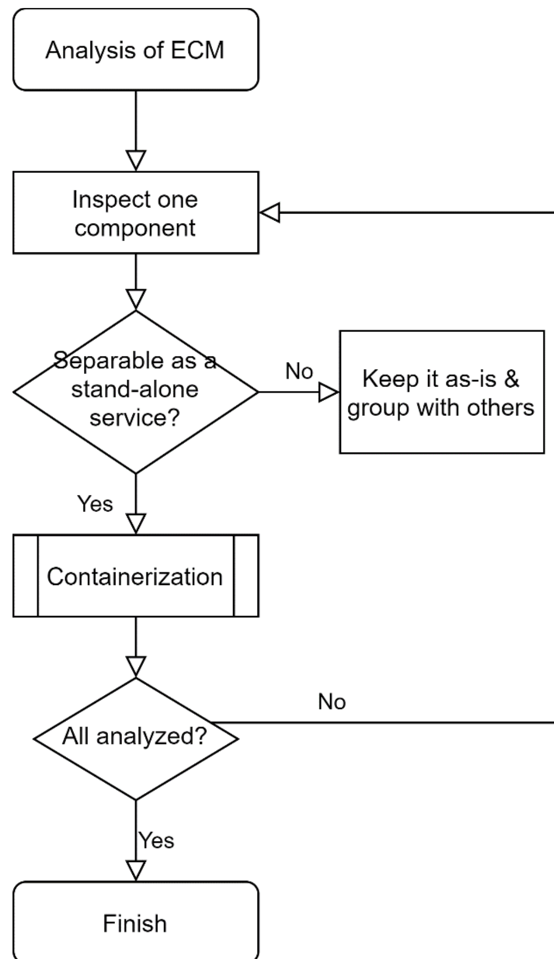

Figure 4-1 Process of Analyzing IBM ECM Components

Table 4-1 concludes the analysis results. Most of the components can be decomposed to stand-alone services, while some need to be coupled with others. For example, Library Server and Library Server Database should be combined as one component, because Library Server itself is rather light featured, whose most features are predefined procedures that run on the Library Server Database. Since they can run only when both are active, it is better to combine them together into one container. Another worth noting component is that both the Resource Manager and ECM Web Client require Web Application Servers (which is WAS in our case) to support their communication with other ECM components, the connection to databases (e.g. via `JDBC`), authentication, and display of webpages. Apparently, a Web Application Server alone in a container is useful to be used as the base image during development. However, in our case, it makes no sense to have a blank Web Application Server

container running, other than serving as a reference installation of Web Application Server. Thus, the Web Application Server, i.e. IBM WebSphere Application Server must be combined with the Resource Manager and the ECM Web Client. By the design of IBM, ICN (i.e. the Web Client Application) stores its configurations in a database, which is called ICN Config DB (i.e. Web Client Config DB). This database is small and will usually not grow considerably, hence, we put it together with ICN, making it more portable.

As analyzed theoretically as a whole, this setup should function just fine. Hence, in the next section, we will combine those tightly coupled components as one container while leaving those loosely coupled components in their own container.

Table 4-1 Analysis of IBM ECM Components

Component	**Corresponding IBM product**	**Separable as a stand-alone service?**
Data Catalog	ICM (Library Server)	Yes, if with Library Server DB.
Data Catalog DB	ICM (Library Server DB), DB2	No, the Library Server is tightly coupled with its DB
Resource Manager	ICM (Resource Manager App), WAS	Yes
Object Catalog	ICM (Resource Manager DB), DB2	Yes
Web Application Server	WAS	No, must be combined with components requiring an application server
Web Client	ICN, WAS	Yes
Web Client Config DB	ICN, DB2	Yes, but not necessary

4.4.2 Containerize Components

Figure 4-2 shows the process of containerization of the IBM ECM components analyzed above. Several points must be considered before creating a container in Docker, since once a container is up and running, it is no longer allows such administrative changes to containers in Docker. These points include the base image, the network & port specifications, persistent storage requirements, and privileged rights. Each of them is explained below.

- **The Base Image** determines the base and the entry point of a container. Using a CentOS Docker image as the base image will result in a container in the environment where CentOS is the operating system. Using an Apache Web Server Docker image (called "`httpd`") will result in a container with an Apache Web Server installed and preconfigured and as well as the base image Apache Web Server image used. This decision is made based on the purpose of the containers.

- **Network & port specification** is important for communication between containers. Two containers should be added to the same network to enable their direct communication without the necessity of port forwarding. The port we wish to expose for access (such as port `:80` and `:443`) outside containers (e.g. from the host or outside host) must be mapped. The mapping can be explicit or implicit. In the explicit case, we assign a port of the host to the port of the container, while in the implicit case, the port of the host mapped to the port of the container is generated by Docker. We choose the explicit mapping since we would like to have the result reproducible. Further, we should also assign hostnames to containers, so that they can communicate with each other via the hostname, rather than IP (subject to change). Docker achieved this by dynamically generating the `/etc/hosts` file.

- **Persistent storage** must be added for applications with the need for data storage. It preserves the data for containers if they are shut down or are migrated. Persistent storage also helps to debug the containers as we can modify the application(s) in a container at will without corrupting the data.

- **Privileged** containers have the right to access devices connected to the host and system functionalities, such as `systemctl` on CentOS. There is potentially more risk of using a privileged container, but sometimes it is required. An application may need to utilize some systematic features to be able to function properly, such as a database server application registering system events and automatic startups.

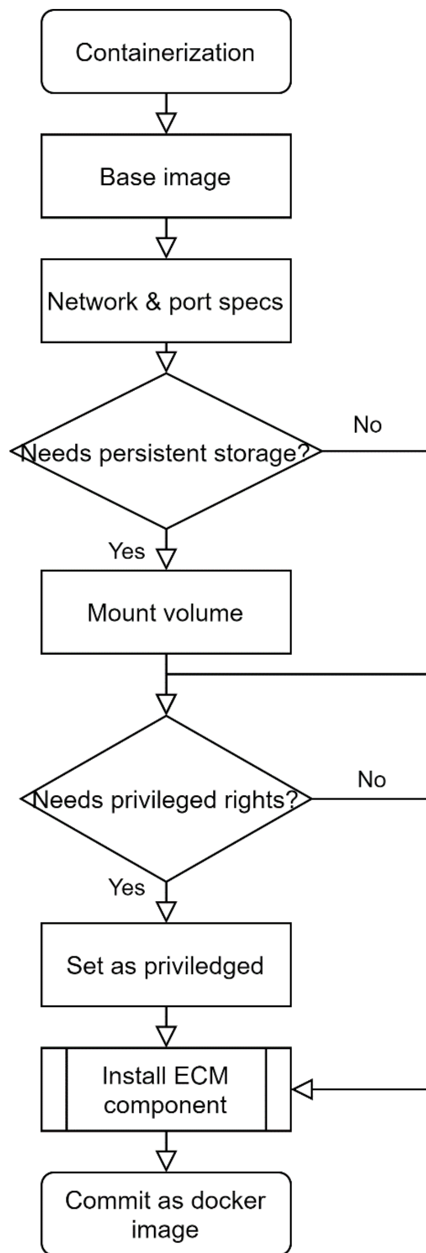

Figure 4-2 Process of Containerization

Table 4-3 and Table 4-4 below summarize the container configurations during this phase. Based on Table 4-1 above, we have created four containers. The containers are named based on the IBM-specific components in them, for example, “lsdbsrv” stands for “Library Server & Library Server Database Service” and “wasrm” stands for “Resource Manager in IBM WebSphere Application Server”. This abbreviation is presented in Table 4-2.

Table 4-2 Abbreviated Container Name Explanation

Abbreviated Name	**Explanation**
lsdbsrv	Library Server & Library Server Database Service
rmdbsrv	Resource Manager Database Service
wasrm	Resource Manager Application in WAS
wasicn	IBM Content Navigator in WAS

The base images were determined by the components. In container "lsdbsrv", the main application is the database (i.e. IBM DB2), so we used the `ibmcom/db2:latest` from Docker Hub. In containers "wasrm" and "wasicn", we simply used `centos:7` and installed the components by ourselves, because none of those components' images is provided publicly. Since "lsdbsrv" and "rmdbsrv" are both hosting databases, persistent storage for them is a must. "wasrm" also requires persistent storage since Resource Manager Application manages the raw data in the ECM system. Containers with DB2 must be set as privileged because DB2 uses some administrative system calls.

Table 4-3 Container Configuration (a)

Container Name	**Containerized Components**	**Base Image**	**Persistent Storage?**	**Privileged?**
lsdbsrv	Data Catalog	`ibmcom/db2:latest`	Yes	Yes
rmdbsrv	Object Catalog	`ibmcom/db2:latest`	Yes	Yes
wasrm	Resource Manager, HTTP Server	`centos:7`	Yes	No
wasicn	Web Client, Web Client Config DB, HTTP Server	`centos:7`	No	No

Further, our ECM system will not function properly unless all components (i.e. containers) are connected, thus, they were all added in a custom network named "ecm-net". Port `50000` is for DB2, and others are for the HTTP Server's port. The application access port is for accessing the installed application in WAS, for example, `https://wasicn:9444/navigator` is for accessing the ICN installed in WAS via `HTTPS`.

Table 4-4 Container Configuration (b)

Container Name	Network	Opened Ports	Port Remarks
lsdbsrv	ecm-net	`50000`	DB2 port
rmdbsrv	ecm-net	`50000`	DB2 port
wasrm	ecm-net	`9043, 9443, 9080`	WAS administration port, secured application access port, insecure application access port
wasicn	ecm-net	`9044, 9444, 9081`	(ditto)

4.4.3 Installation and Configuration of ECM Components

Figure 4-3 below depicts the process of installing IBM ECM components into containers. This process seems incredibly simple, but in reality, there were a few things that one must sort out: the dependencies, different ways of installation, and application-specific configurations. Products from IBM ECM were often written by different teams and hence, they are very likely to have different dependencies. The official knowledge center does list some dependencies, however, they were not enough during the prototype process, as one cannot simply assume all of them have been pre-installed in the operating system. In particular, the CentOS Docker image is a very light and GUI-less version of CentOS which does not many libraries and utilities pre-installed. Thus, dependencies are listed in Table 4-5.

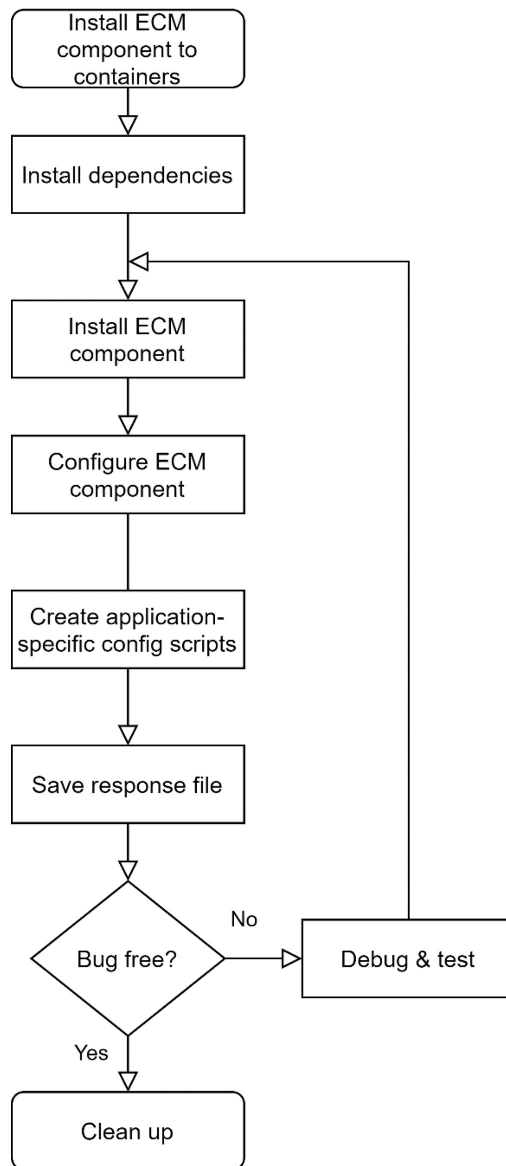

Figure 4-3 Process of Installing ECM Components

Most IBM ECM products can be installed using command line commands but often they provide more user-friendly applications for installation and configuration with GUI. To be able to use them, we have libraries such as `gtk3` and `xauth` installed. The installer of ICN will complain that it fails to open the display even it is executed in silent mode (GUI-less), please be aware of that. Another interesting example of dependencies is that the configuration manager of ICM complained about the operating system being 32-bit even though it was clearly 64-bit. After debugging, we realized that it was because the utility `file` was not installed so that the check of operating system bitness fell back to the default - 32-bit.

There is a short explanation of the dependencies. Most of the installers and some of the configuration tools from IBM required GUI support, thus, almost all of them required the installation of `xauth` for `Xming`, `gtk3` for GUI support, and so on. `xclock` is a small but effective application to test whether `Xming` works or not. Some configurations were done by `ssh` to the remote containers; thus, `openssh-server` and `openssh-clients` were also needed.

Table 4-5 ECM Dependencies

ECM Component	Dependencies Installation Command
All	`yum install -y unzip java-1.8.0 xclock xauth openssh-server openssh-clients file sudo`
DB2	`yum install -y libaio binutils zlib perl-Sys-Syslog pam pam.i686 libstdc++ kernel-devel numactl-libs gcc gcc-c++ libXtst PackageKit-gtk3-module.i686`
WAS	`yum install -y gtk3 mesa-libGLw`

Planning was necessary before the installation. We realized that the container of Resource Manager (i.e. RMApp) and the container of Web Client Application (i.e. ICN) both required a Web Application Server (i.e. IBM WAS). We also found out that the container of Data Catalog (i.e. LS and LSDB) and the container of Object Catalog (i.e. RMDB) both required an instance of the database server (i.e. DB2). Based on the planning above, we first created images of a fresh installation of WAS and DB2, and then we proceeded to install and configure the rest components.

It is important to know that the installation of IBM ECM has usually three phases. In phase one, the installer is downloaded and then unpacked. In phase two, we need to find the executable for installing the unpacked data to the system. The executables of ICN, WAS and DB2 install the actual application along with respective configuration managers, but the executable of ICM installs a configuration manager for deploying the actual ICM. The reason for such behavior is that the system administrator can install the ICM configuration manager on a single machine and use it to install the components (Library Server, Library Server Database, Resource Manager, and Resource Manager Database) to machines hosting those components. In phase three, we need to use the configuration manager to configure (and deploy) the actual component.

Let's examine the installation of ICM in detail. In phase one, we unpacked the ICM from the provided `GA8.6.00.000_installer_lnx.tar` file. In phase two, we ran the command `bash <unpacked_dir>/install` to start the installer and followed the instructions to install the ICM configuration manager to the host machine. The installer

installed it in the directory `/opt/IBM/cmrepository/8.6.00.000`. Then we proceeded to phase three.

Phase three was a bit more complicated because we need to use the configuration manager ("ICMCM") to install and configure Library Server DB, Resource Manager, and Resource Manager Application. We can find ICMCM at `/opt/IBM/cmcrepository/8.6.00.000/bin/cmcfgmgr_CM`. We switched there and ran `bash ./cmcfgmgr_CM` to start the ICMCM (make sure you have already configured `Xming`). In the GUI of the ICMCM, we created a new profile named `wasrmProfile` to install the Resource Manager Application in the container `wasrm`. For remote configuration like this, ICMCM needs to `ssh` to the remote host (in our case, `wasrm`) with an administrative user. `ssh` is not enabled by default in containers due to security reasons and the behavior of direct `ssh` to containers is discourage by Docker, but it is only needed for installation, so we still enabled it and turned it off afterward. To enable it, we first installed `openssh` client and server applications inside `wasrm`, then we started the `sshd` service and created an administrative user named `dev`. Commands used are shown below [23].

```
# Install openssh client and server
yum install -y openssh-server openssh-clients sudo
# Generate default keys
/usr/bin/ssh-keygen -A
# Start sshd daemon
/usr/sbin/sshd
# Add user named dev
useradd --create-home --shell /bin/bash --groups root dev
# set the password of dev, we used 'passw0rd' as the password
passwd dev
# Give user dev with sudo rights
usermod -aG wheel dev
# Open sshd settings and change "UseDNS" to "no"
vi /etc/ssh/sshd_config
```

Next, ICMCM verified if ICM is compatible with `wasrm` by using the `dev` user we just created. We installed dependencies and changed the `/etc/redhat-release` in `wasrm` to make them compatible. After a successful verification, we chose to install the Resource Manager Application and we filled out all required fields for configuration. The configurations were included in a response file generated at the end; hence, we will not demonstrate it in detail here. Please remember that before installing RMApp, we must install WAS in `wasrm` first, as RMApp needs to be run in a web application server.

After the installation and configuration of IBM ECM products, they will offer to save all the variables entered during the installation to a file named response file, which can be used for installation next time in silence mode. This is a neat feature and all response files were saved.

Last but not least, application-specific scripts were created. These scripts include configuration of data sources (e.g. `JDBC`), filesystem/storage path inside a volume, start/stop scripts for WAS, and trace and log settings. Some of the commands are listed in section 0. For instance, to start/stop ICN in WAS in container `wasicn`, we used the following commands:

```
# Start WAS
bash /opt/IBM/WebSphere/AppServer/profiles/icnProfile/bin/startServer.sh \
server1 -username wasadmin -password passw0rd
# Stop WAS
bash /opt/IBM/WebSphere/AppServer/profiles/icnProfile/bin/stopServer.sh
server1 -username wasadmin -password passw0rd
```

4.5 Resulting Prototype

With everything installed and configured, we have the working prototype shown below in Figure 4-4. The prototype functions well as desired. There are four containers in the prototype and all of them are able to communicate with each other. To achieve our goal, only one component of the ECM system runs in each container. It is very clear that the end user interacts with the Web Client, while the Web Client sends data to/receives data from the Data Catalog, the Resource Manager, and the Object Catalog (some typical tasks were described in Section 1.2). Further, it is easy to find that there are two containers with a database connected respectively (and the database storage volume) and one container using persistent storage.

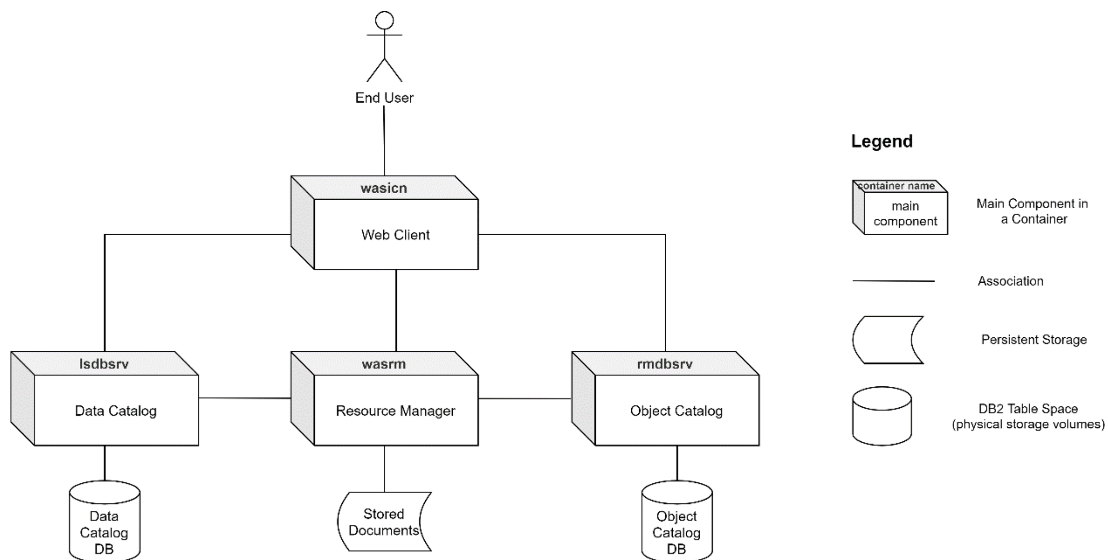

Figure 4-4 Prototype Overview

With this prototype in Docker containers, one can easily populate them and create new deployment. Since we isolated each component, we can achieve much fine-grained control over the deployment of an ECM system. In the case of populating another Resource Manager, we can simply run another Docker container based on "wasrm" and start a new Resource Manager to help with incoming workload; or if the system is short of storage volume, we can start another Docker container based on "wasrm" and attach new persistent storage to it. The new instance of RM will work as expected after some configuration. When we need another Web Client instance, we can as well start another container based on the image "wasicn" and connect both web clients to a load balancer (not shown in the figure). The prototype also enables an ECM system to be fast and easily deployed by using preconfigured container images.

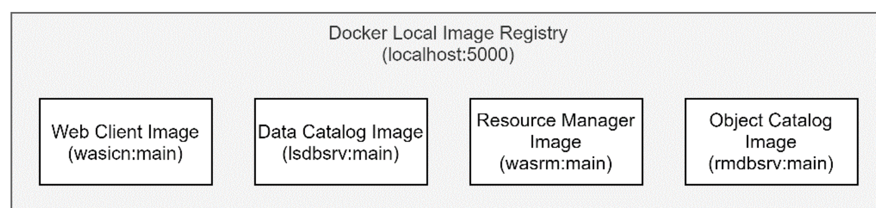

Figure 4-5 Docker Local ECM Image Registry

The following is a sample excerpt of Docker Compose file for deploying the prototype images using Docker Compose. The general guideline here is that we define four services for each component above and specify their ports, volumes, and images.

```
version: "0.1"
services:
  lsdbsrv:
    ports:
      - "50000:50000"
    volumes:
      - /opt/lsdb:/database
    image: localhost:5000/lsdbsrv:main
  rmdbsrv:
      - …
```

Using Docker Compose is a good idea but not the best, because Docker Compose is mainly used for orchestrating containers in a singles host. For a deployment that scales horizontally, we shall use container orchestrators (e.g. Kubernetes) which orchestrate containers over multiple VM nodes. A very simple deployment of the above topology can be achieved using Kubernetes. An excerpt from the prototype `.yaml` file is shown below.

In this `.yaml` excerpt, we define it as a K8s deployment where four containers are orchestrated. The containers' images are from the local registry and we specified the container names, ports and volume mounts similar to the previous analysis (tag names

are changed, e.g. `volumeMounts` and `containerPort`). Next, volumes are defined using `hostPath`, which mounts a directory of the host machine to K8s worker nodes. This `.yaml` file excerpt shows the readability and simplicity of managing container deployment, yet it is only a naïve version, and improvements shall be made in future researches. For example, how to use application-specific scripts in K8s deployment to achieve automatic high availability deployment and dynamic topology based on container traffic.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ecm-deployment
  labels:
    app: ecm
spec:
  replicas: 1
  selector:
    matchLabels:
      app: ecm
  template:
    metadata:
      labels:
        app: ecm
    spec:
      containers:
      - name: lsdbsrv
        image: localhost:5000/lsdbsrv:main
        ports:
        - containerPort: 50000
        volumeMounts:
        - mountPath: /database
          name: lsdb-volume
      - name: rmdbsrv
        image: localhost:5000/rmdbsrv:main
        ports:
        - containerPort: 50000
        volumeMounts:
        - mountPath: /database
          name: rmdb-volume
      - name: wasrm
        image: localhost:5000/wasrm:main
        ports:
        - containerPort: 9043
        - containerPort: 9443
        - containerPort: 9080
        volumeMounts:
        - mountPath: /var/rmdata
          name: rmapp-volume
      - name: wasicn
        image: localhost:5000/wasicn:main
        ports:
        - containerPort: 9044
        - containerPort: 9444
        - containerPort: 9081
      volumes:
      - name: lsdb-volume
        hostPath:
          # directory location on host
          path: /opt/lsdb
```

```
    - name: rmdb-volume
      hostPath:
        path: /opt/rmdb
    - name: rmapp-volume
      hostPath:
        path: /opt/rmdata
```

4.6 Commands for Reference

Some commands during the containerization process are listed below for reference.

To start a Docker container based on `ibmcom/db2`, we can use the following command. `docker run` is for starting a container based on a Docker image, in our case, we start a container named "lsdbsrv" based on the image "`ibmcom/db2`" from Docker Hub. We set this container to be privileged, and map the port `50000` of the container to the port `50000` of the host. Option `-itd` means that we'd like to keep `STDIN` open even if not attached, allocate a pseudo-tty, and start the container in detached mode.

```
# Start a container from db2 image
docker run -itd --name lsdbsrv --privileged -p 50000:50000 -e LICENSE=accept -
e DB2INST1_PASSWORD=passw0rd -v /home/lsdb:/database ibmcom/db2

# Start a container lsdbsrv from local image registry
# with hostname, network and volume mount options
docker run -td --name lsdbsrv --privileged -p 50000:50000 --hostname lsdbsrv -
-net-alias="lsdbsrv" --network="ecm-net" -v /opt/lsdb:/database
gang/lsdbsrv:main

# Start ICN in WAS
docker run -td --privileged=true --hostname wasicn --net-alias="wasicn" --
network="ecm-net" -p 9044:9044 -p 8081:80 -p 9444:9444 -p 9081:9081 -p
52023:22 --name wasicn gang/wasicn:main

# Start RMApp in WAS
docker run -td -p 9043:9043 -p 8080:80 -p 9443:9443 -p 52022:22 -p 9080:9080 -
-hostname wasrm --net-alias="wasrm" --network="ecm-net" -v
/opt/rmdata:/var/rmdata --name wasrm gang/wasrm:main
```

The following are some WAS-specific commands used to start and stop a WAS server. These commands were put into scripts so that it is possible to use a container orchestrator (such as Kubernetes) to manage them automatically.

```
# Create a WAS profile named rmappProfile
bash /opt/IBM/WebSphere/AppServer/bin/manageprofiles.sh -create -templatePath
/opt/IBM/WebSphere/AppServer/profileTemplates/default -profileName
rmappProfile -federateLater false -enableAdminSecurity true -adminUserName
wasadmin -adminPassword passw0rd

# View the details of created rmappProfile
less
/opt/IBM/WebSphere/AppServer/profiles/rmappProfile/logs/AboutThisProfile.txt

# Start a server instance in rmappProfile
```

```
/opt/IBM/WebSphere/AppServer/profiles/rmappProfile/bin/startServer.sh server1
-profileName rmappProfile -username wasadmin -password passw0rd

# Stop the server instance in rmappProfile
/opt/IBM/WebSphere/AppServer/profiles/rmappProfile/bin/stopServer.sh server1 -
profileName rmappProfile -username wasadmin -password passw0rd
```

The followings are some commands related to DB2. In particular, a useful command to change the hostname stored in DB2 is listed because the hostname change could happen often in a cloud environment.

```
# To change the hostname in a DB2 server
sudo su - db2inst1
db2stop
db2set -g DB2SYSTEM=<new_hostname>
echo "0 <new_hostname> 0" > acutal_db2_storage_location/sqllib/db2nodes.cfg

# To restart a DB2 server
sudo su - db2inst1
db2 force applications all
db2 terminate
db2stop
db2start
```

The followings are some commands that solved compatibility issues between ICM and CentOS 7.

```
# To generate a machine-id
dbus-uuidgen > /etc/machine-id

# To change the release info to RHEL
cp /etc/redhat-release /etc/redhat-release.centos
echo "Red Hat Enterprise Linux Server release 7.3 (Maipo)" > /etc/redhat-
release
```

The followings are some commands for the administration in Docker, in particular, the command to remove dangling images and dangling volumes.

```
# To remove dangling images
docker rmi $(docker images -f "dangling=true" -q)

# To remove dangling volumes
docker volume rm `docker volume ls -q -f dangling=true`

# To run a container in detached mode indefinitely (remove option -i)
docker run -td <image> /bin/bash

# To save a Docker image as a .tar file
docker save -o <filename> <repository>:<tag>

# To load a .tar image file to Docker (DO NOT use docker import, because that
is only for loading filesystem structures)
docker load -i <tar_filepath>

# To commit an image to local registry
docker tag <local_image_name>:<tag> localhost:5000/<new_image_name>:<tag>
```

```
docker push localhost:5000/<new_image_name>:<tag>
```

4.7 Summary

In this chapter, we have detailed the prototype process of the proposed design. The prototype functions well as a normal ECM system on a virtual machine (or bare metal). It is sufficient for proving that the proposed design changes for the ECM systems to exploit the cloud technology are viable.

5 Conclusion and Outlook

This thesis aims to propose design changes for legacy ECM system to exploit the cloud technology in a non-disruptive way. To achieve this goal, we first analyzed the challenges of monolithic ECM systems. Then, we summarized the technical requirements for ECM system in the current market. Next, we proposed those design changes, which were followed by a prototype based on IBM ICM and Docker that verified the design changes proposed were effective and achievable.

This thesis opened the possibility of the dynamically managed topology, replications, container orchestration capability, and customizable automatic deployment for an ECM system. Yet, due to the scope of this thesis, these aspects were not investigated. Future work can research on one of the above aspects to achieve an ECM system that fully exploits the potential of cloud computing.

6 Bibliography

[1] Association for Information and Image Management, "What is Enterprise Content Management (ECM)?," Association for Information and Image Management, 2020. [Online]. Available: https://www.aiim.org/What-is-ECM. [Accessed 1 December 2020].

[2] J. Mancini, "The Next Wave: Moving from ECM to Intelligent Information Management," 2017. [Online]. Available: https://cdn2.hubspot.net/hubfs/332414/AIIM_Blog/Intel-info-Next-Wave-2017-updated.pdf. [Accessed 1 December 2020].

[3] A. Regalado, "Who Coined "Cloud Computing"?," 31 October 2011. [Online]. Available: https://www.technologyreview.com/2011/10/31/257406/who-coined-cloud-computing.

[4] Cloudflare, Inc., "What Is the Cloud?," Cloudflare, Inc., 2020. [Online]. Available: https://www.cloudflare.com/learning/cloud/what-is-the-cloud/. [Accessed 12 December 2020].

[5] The Kubernetes Authors, "How We Architected and Run Kubernetes on OpenStack at Scale at Yahoo! JAPAN," The Linux Foundation, 24 October 2016. [Online]. Available: https://kubernetes.io/blog/2016/10/kubernetes-and-openstack-at-yahoo-japan/. [Accessed 1 December 2020].

[6] R. Bauer, "What's the Diff: VMs vs Containers," Backblaze, 28 June 2018. [Online]. Available: https://www.backblaze.com/blog/vm-vs-containers/. [Accessed 23 November 2020].

[7] I. Eldridge, "What Is Container Orchestration?," New Relic, Inc., 17 July 2018. [Online]. Available: https://blog.newrelic.com/engineering/container-orchestration-explained/. [Accessed 12 December 2020].

[8] The Kubernetes Authors, "What is Kubernetes?," The Linux Foundation, 22 October 2020. [Online]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. [Accessed 1 December 2020].

[9] B. C. Neuman, "Scale in Distributed Systems," in *Readings in Distributed Computing Systems*, IEEE Computer Society Press, 1994, pp. 463-489.

[10] P. Bandhyapadhya, J. Chowdhury and Y. Patil Sr, "Integration of “Virtualization with Enterprise Content Management System” which impose

“Green Computing”," *Journal of Engineering, Computers & Applied Sciences (JEC&AS)* , vol. 1, no. 3, pp. 21-24, 2021.

[11] F. Wagner, "A virtualization approach to scalable enterprise content management.," 7 March 2011. [Online]. Available: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIS-2011-04&mod=0&engl=0&inst=IPVS. [Accessed 1 10 2020].

[12] M. Stine, Migrating to Cloud-Native Application Architectures, Sebastopol: O'Reilly Media, 2015.

[13] C. Richardson, "Pattern: Database per service," Microservices.io, 2020. [Online]. Available: https://microservices.io/patterns/data/database-per-service.html. [Accessed 12 December 2020].

[14] K. Benzekki, A. El Fergougui and A. Elbelrhiti Elalaoui, "Software-defined networking (SDN): a survey," *Security Comm. Networks,* vol. 9, no. 18, pp. 5803-5833, 1 December 2016.

[15] The CentOS Project, "The CentOS Project," Red Hat, Inc., 11 December 2020. [Online]. Available: https://www.centos.org/. [Accessed 11 December 2020].

[16] Alfresco Software, Inc., "Alfresco Content Services architecture overview," Alfresco Software, Inc., 11 December 2020. [Online]. Available: https://docs.alfresco.com/5.2/concepts/alfresco-arch-about.html. [Accessed 11 December 2020].

[17] OpenText Corporation, "OpenText Process Suite Platform Architecture," 2016. [Online]. Available: https://www.opentext.com/file_source/OpenText/en_US/PDF/opentext-whitepaper-opentext-process-suite-platform-architecture-en.pdf. [Accessed 11 December 2020].

[18] IBM, "What is enterprise content management?," United States, 25 11 2020. [Online]. Available: https://www.ibm.com/cloud/automation-software/enterprise-content-management. [Accessed 25 11 2020].

[19] D. Inc., "Docker overview," Docker Inc., 1 May 2020. [Online]. Available: https://docs.docker.com/get-started/overview/. [Accessed 26 November 2020].

[20] E. Carter, "2018 Docker usage report," Sysdig Inc., 29 May 2018. [Online]. Available: https://sysdig.com/blog/2018-docker-usage-report/. [Accessed 26 November 2020].

[21] E. Carter, "Sysdig 2019 Container Usage Report: New Kubernetes and security insights," Sysdig Inc., 29 October 2019. [Online]. Available:

https://sysdig.com/blog/sysdig-2019-container-usage-report/. [Accessed 26 November 2020].

[22] A. Generated, "ibmcom's Profile - Docker Hub," Docker Inc., 26 November 2020. [Online]. Available: https://hub.docker.com/u/ibmcom. [Accessed 26 November 2020].

[23] crondog, "SSH: Could not load host key: /etc/ssh/ssh_host_rsa_key," 18 June 2013. [Online]. Available: https://bbs.archlinux.org/viewtopic.php?id=165382. [Accessed 1 October 2020].

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Datum und Unterschrift:

14.12.2020

Declaration

I hereby declare that the work presented in this thesis is entirely my own. I did not use any other sources and references that 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.

Date and Signature:

14.12.2020