Orchestrating Information Governance Workloads as Stateful Services Using Kubernetes Operator Framework

Cataldo Mega(✉)

University of Stuttgart, Universitätsstraße 38, 56095 Stuttgart, Germany
cataldo.mega@ipvs.uni-stuttgart.de

Abstract. Regulatory compliance is forcing organizations to implement an information governance (IG) strategy, but many are struggling to evolve their IG solutions due to their legacy architecture, as they are not designed to adapt to new business models and for the growing amount of unstructured data produced by a potentially worldwide audience. One of the biggest problems faced is continuously determining data value and adaptation of measures to keep risks and operational costs under control. One way to solve this issue is to leverage cloud technology and find an affordable approach to migrate legacy solutions to a cloud environment. In most cases, this means de-composing monolithic applications, refactoring components and replacing outdated homegrown deployment technologies with cloud-native, automated deployment and orchestration services. Our goal is to show how operational costs can be reduced by running refactored versions of IG solutions in clouds with a minimum of human intervention. This paper discusses the steps to evolve a legacy multi-tier IG solutions from physical to containerized environments by encapsulating human operator knowledge in cloud topology and orchestration artifacts, with the goal of enabling automated deployment and operation in Kubernetes (K8s) managed execution environments.

Keywords: Information governance · IG workloads · cloud · stateful services

1 Introduction

Every company is subject to three basic business metrics; Value, cost and risk. They form the basis of any Enterprise Information Management (EIM) system. IG adds governance controls to information lifecycles and becomes the control authority for Information Lifecycle Governance (ILG). ILG starts with the creation and extends to the disposition of data. Data sets in the IG context represent governance metadata needed to control how data is processed and to create an appropriate governance context derived from applicable company policies, regulations and standards through the use of Records Lifecycle Management (RLM). This means that governance records relate to the security, classification, retention, and disposition of data. In practical terms, IG consists of implementing an Information Governance Program (IGP) that helps to steer information lifecycles based on actual data value. As a result, ILG workflows through their processes implement three key activities: 1) Use of analytics to determine and maximize data value as context erodes; 2) Enforce archiving of data onto tiered storage to ensure storage cost declines as value declines; 3) Trigger disposal of obsolete data to avoid cost and eliminate risk. As a result, in addition to actual business workloads, these activities also produce typical ILG workloads that an EIM system must handle.

1.1 Problem Statement and Requirements

Today, legacy IG solutions operating in a global open market have to deal with an increasing workload caused by international regulation pushing them to its operational and financial limits. The root cause of these shortcomings is a monolithic solution design and a production system running on a static IT infrastructure. These factors prevent flexibility at component level and elasticity at IT resource level, and are therefore costly to operate and maintain. One way out of this situation is to migrate these solutions to cloud environments and take advantage of the economies of scale where the sharing of IT resources makes it possible to minimize operational costs and optimize resource consumption through automation. Unlike traditional IT systems, clouds automate operational cost control by monitoring key performance indicators that report on cloud resource consumption, and more important make changes to the used infrastructure through dynamic provisioning and de-provisioning requests. This paper proposes steps to evolve and adapt the legacy architecture of IG solutions designed for bare metal production environments to modern cloud environments. To prove the feasibility of our approach, we implemented a prototype of an IG solution running on a Kubernetes-managed (K8s) platform using the operator pattern promoted by the Cloud Native Computing Foundation (CNCF) [1].

1.2 Contributions and Outline of this Paper

Contribution 1: We decomposed our IG solution, reworked its legacy design, and made the necessary changes to automatically deploy and operate it in a K8s execution environment. Major focus has been put into refactoring component and deployment models and the consolidation of the tier-based high availability (HA) design before moving from a bare-metal to a containerized on virtualized deployment model, shown on Fig. 3. Contribution 2: We formalized the knowledge of human operators and implemented a resilient IG solution that models HA, disaster recovery (DR) and scale-out by incorporating infrastructure operational logic into the design and implementation of stateless and stateful cloud services running under the control of the K8s orchestrator.

The remainder of this paper is structured as follows: Sect.: 2 presents a blue-print for IG solutions and an associated component model that we derived from a representative set of IG use cases. Some background on the benefits that the cloud offers for IG workloads is also provided. Section: 3 introduces the fundamental aspects of deployment topologies for IG solutions and discusses traditional versus cloud-native deployment models. It also briefly explains how K8s based workload orchestration works in the cloud. Section: 4 presents our solution approach. Section: 5 introduces the stateful IG solution prototype and its services. Section: 6 details the prototype development and the system under test (SUT) used. Section: 7 discusses the evaluation performed and the test results produced; Sect.: 8 presents our conclusion and provides an outlook on future work.

2 Background

In order to bring together IG solutions and the cloud we need to look at the requirements and workloads that regulations add to typical production systems.

IG requirements are mainly derived from corporate policies, regulations and standards. They influence the solutions design and define RLM control structures required for EIM and RLM lifecycles processes as described by the following use cases (UC) out of the EIM, RLM application domains:

- UC1 (EIM): Collect and classify enterprise data from known sources.
- UC2 (EIM): Load, store, index and secure data in enterprise repositories.
- UC3 (EIM): Search, access and retrieve information from the repositories.
- UC4 (RLM): Apply regulatory security, classification, retention, hold, and disposition policies.
- UC5 (RLM): Support legal cases through e-discover, aggregate and transfer case data on hold.

2.1 ILG Workload Models

By definition, a workload is defined as a representative mix of primitive operations performed against a system. The workloads implied by the UC1 – UC5 use cases fall into the following categories (details are discussed in Mega [5]):

- WL1: This workload is created by interactive users and external agents using web-requests through Https/REST issued against the IG services APIs.
- WL2: Is an interactive- and bulk workload, using lower-level application logic performing database operations consisting of a representative mix of primitive operations like: Create, Retrieve, Update, Delete and Search (CRUDS).
- WL3: Is an interactive- and bulk workloads using low-level file system functions against persisted files, consisting of digital objects of any type, format and size.

Together, use cases, workloads and real-world experience helped define an IG solution and blueprint as shown in Fig. 1 below.

The blueprint consists of seven key solution components, listed as CM1 to CM7.

Going left to right there is: CM1: Aggregates the subcomponents Data Collection, Classification, Assessment and Ingest. CM2: Content Services: Providing Access, Index, Search, Retrieval, Security and Management functions.

CM3: Records Services: These are, Classification, Retention, Disposition and Compliance. CM4: Case Management Services: Consisting of e-Discovery, Legal Data Requests, and Holds. CM5: Content Analytics: Related to, business Classification, Statistics, Reporting. CM6: Repository Services: Provide Information Retrieval, Catalog and Archive functions. CM7: Platform Services: Address Compute, Storage, and Network needs.

For a reference and comparison we looked at architectures published by California Department of Technology [6], Alfresco [7], IBM Cloud Design Center [3], IBM Content Manager Enterprise Edition [4], IBM FileNet Content Manager [8], and other major players in this domain. Workloads, similar to the one defined before, are discussed in Mega [5] and in Lebutsch [9].

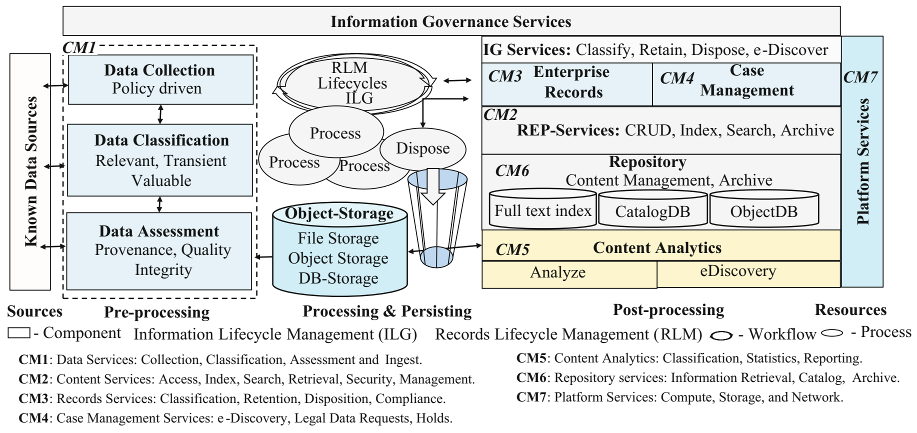
Information Governance Services
CM1
Data Collection
Policy driven
Data Classification
Relevant, Transient
Valuable
Data Assessment
Provenance, Quality
Integrity
Known Data Sources
RLM
Lifecycles
ILG
Process
Process
Process
Dispose
Object-Storage
File Storage
Object Storage
DB-Storage
IG Services: Classify, Retain, Dispose, e-Discover
CM3 Enterprise Records
CM4 Case Management
CM2
REP-Services: CRUD, Index, Search, Archive
CM6 Repository
Content Management, Archive
Full text index
CatalogDB
ObjectDB
CM5 Content Analytics
Analyze
eDiscovery
CM7 Platform Services
Sources
Pre-processing
Processing & Persisting
Post-processing
Resources
- Component
Information Lifecycle Management (ILG)
Records Lifecycle Management (RLM)
- Workflow
- Process
CM1: Data Services: Collection, Classification, Assessment and Ingest.
CM2: Content Services: Access, Index, Search, Retrieval, Security, Management.
CM3: Records Services: Classification, Retention, Disposition, Compliance.
CM4: Case Management Services: e-Discovery, Legal Data Requests, Holds.
CM5: Content Analytics: Classification, Statistics, Reporting.
CM6: Repository services: Information Retrieval, Catalog, Archive.
CM7: Platform Services: Compute, Storage, and Network.

Fig. 1. ILG Solution blueprint and component.

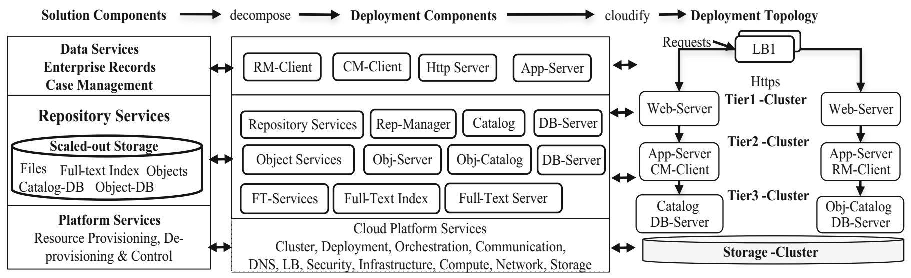

Fig. 2. Steps to create an ILG solution component model and deployment topology.

Both performed similar tests and used a deployment topology similar to that on the far right of Fig. 2. Moving from left to right, we sketched the steps in which the IG solution on the left is broken down into individual, self-sufficient components then assembled into a deployment package along with platform components and arranged as a deployable topology graph using a multi-tier application pattern, shown on the far left.

2.2 The Benefit of Clouds

Today's cloud platforms offer dynamic resource provisioning, scalability and efficiency to applications that are both containerized and virtualized - characteristics that legacy IG solutions lack. Virtualization affects physical production environments; it transforms physical infrastructure into purely virtual infrastructure through a Soft-ware Defined Infrastructure (SDI) approach. Containerization is done at the solution level by breaking down monolithic solutions into independent components that are suitable for running inside containers. Our approach follows the concept of a composable solution that runs on top of a composable infrastructure as coined by Gartner [2]. This approach suggests that IT resources are dynamically allocated through APIs based on policies. Composable in this context means striving for fully automated IT resource lifecycle management, where application workload pattern and Service Level Agreements (SLA) trigger resource provisioning and de-provisioning events. To prove this approach, we implemented a prototype using the IBM Content Services Reference Architecture [3] guidelines and a subset of IBM Content Management [4] family of products.

3 Foundation

Before cloud, there was a gap between cluster and cluster management. The topology graph of Fig. 2 emphasizes this aspect were each tier is designed as a cluster of applications/resources pair configured to address the need for service resiliency and scale using component-specific cluster management logic. IG solutions typically consists of multiple tiers. Examples are a web server tier, an application server tier hosting a content repository for managing unstructured content, a database server tier for storing meta data and a storage tier to persist digital content. Service high availability mandates that every tier withstands component failure therefore a high availability solution requires a high availability configuration for every tier. The complexity of configuring high availability holistically stems from the fact that different tier and server types use different approaches to high availability, consisting of specific operational logic, to holistically maintain a defined application state and meet established service level agreements (SLA). SLAs are measured through key performance indicators like: health (alive, dead), response time and throughput. On clouds, cluster operations are consolidated, centralized and application agnostic. Cloud applications are deployed in container together with their runtime environments, in units called Pod. Pod cluster management is an integral part of the cloud platform and independent of application type. Pods are the smallest deployable units in Kubernetes [10]. Cluster of Pods are centrally managed by the K8s control plane, which acts as a replacement for the legacy, tier-specific cluster management. This feature is the biggest advantage for a traditional multi-tier solution. By migrating legacy applications from bare-metal to the cloud, it is possible to close the gap between clusters and cluster management, simplifying and consolidating the operation of an IG production system.

3.1 Virtualizing and Componentizing a Monolithic IG Solution

Figure 3 is a visual of the platform related migration steps necessary for moving IG solutions from bare metal (left) through virtualization to containerized on virtualized (right) cloud execution environments, as suggested by the CNCF [1].

The refactored IG solution design which we used to develop the prototype required the following migration steps: 1) We decomposed the IG solution design in to smaller independent components; 2) We then virtualized the production environment, selecting OpenStack and KVM as the cloud platform/hypervisor technology (Gang [11]); 3) The third step was to containerize the chosen components using Docker for the container and Kubernetes for the cluster and orchestration technology (Trybek [12], Hagemann [13]) and applied it to the stateless application-tier components; 4) The last step included developing the stateful services based on Kubernetes StatefulSets and the operator framework (Wang [14]). Throughout development our focus was on the re-design but were

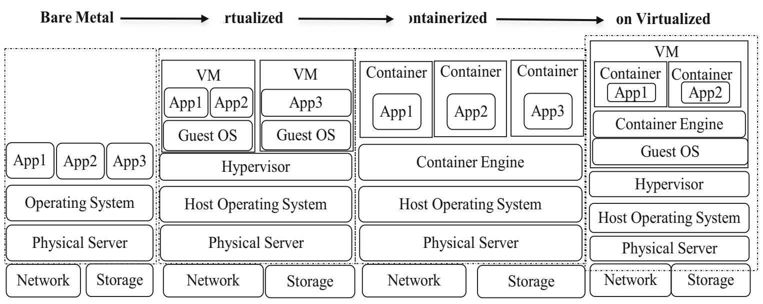

Fig. 3. Migrating from bare metal to containerized on virtualized.

possible also replacement of old components with new cloud-ready technology. As an example, physical components like load balancer (LB), compute server and some networks were replaced with virtual resources provisioned by the cloud platform. Web[1] and application[2] tiers-specific cluster management was replaced with K8s built-in Pod cluster management. Only the management of the database cluster required a custom developed database operator for the DB2 HA-reconciliation and cluster administration logic.

3.2 Comparing Physical vs Virtual Infrastructure Models

Figure 4 shows the deployment topologies of both the original physical production system versus the new virtual, cloud-based production platforms. On the left, you see the legacy system deployed on bare metal servers, in a static, pre-configured production environment. This configuration does not support dynamic topology changes as physical resources are provisioned manually and on request. In these environments software triggered dynamic pro(de)visioning events are not an option. In addition, tier-specific cluster management requires more complex planning and labor intensive operator interventions.

The three clusters (Cluster1–3) on the left of Fig. 4 relate to the three tiers (T1 -T3), web, application and database in a physical environment. The right side shows the same configuration but with a K8s assisted deployment topology optimized for managing the container on virtualized infrastructure. The benefit gained is a consolidated platform built-in cluster management, including a centralized service orchestration facility. In addition, the database specific cluster management is controlled alongside through the K8s APIs using a custom database operator.

[1] https://www.ibm.com/docs/en/ibm.

[2] https://www.ibm.com/docs/en/was/9.0.5?topic=websphere-application-server-overview.

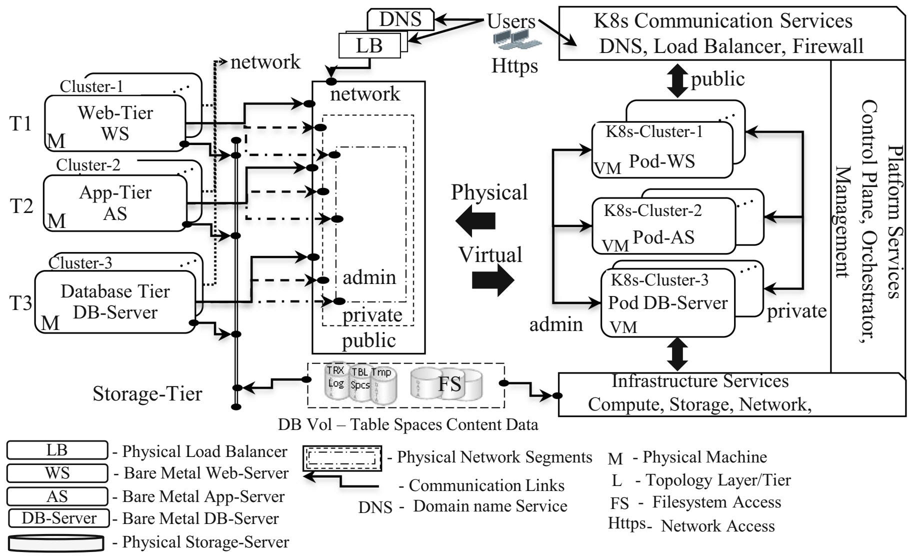

Fig. 4. Migrating solutions from physical to virtual infrastructures.

3.3 Kubernetes Stateful Architecture and its Entities

For a better understanding of our solution approach, we introduce Kubernetes, its components, resources and the operator framework at the high-level. The most important components of K8s are: Controller, Scheduler, Configuration Database (ETCD), a Node (VM), and the actual Operator.

The Deployment, Service and StatefulSets are K8s script resources that are required to define deployment topology and runtime context using YAML grammar.

More specifically their definition is as follows:

- A Deployment is a declarative description of PODs, who carry stateless services.
- A StatefulSets[3] is a declarative description of PODs, carrying stateful services.
- A Service is a declarative way to expose PODs to the external world. The Service defines network access and load-balancing policies to PODs hosting applications that provide the actual service.
- A Custom Resource Definition (CRD) is a declarative description representing a resource known to, but not managed by K8s.
- A Custom Resource (CR) is a component implementing a custom control loop used to manage a custom resource throughout its entire lifecycle. A CR carries the human operator knowledge in form of resource specific implementation artifact.
- An Operator is a K8s extension that allows custom software to be management from within Kubernetes using a Custom Resource Definition (CRD) and the corresponding Custom Resource (CR) component via K8s APIs.

By definition, an IG solution consists of components that provides both stateless and stateful services. This means that the following 3 K8s resources must be used to

[3] https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/.

bring stateless and stateful services under the control of K8s: Deployments for stateless services; StatefulSets for modeling stateful services and operators that use application-specific management logic to control topology changes via APIs. Figure 5 shows the control flow of an operator for managing the lifecycle based on state changes of a custom resource.

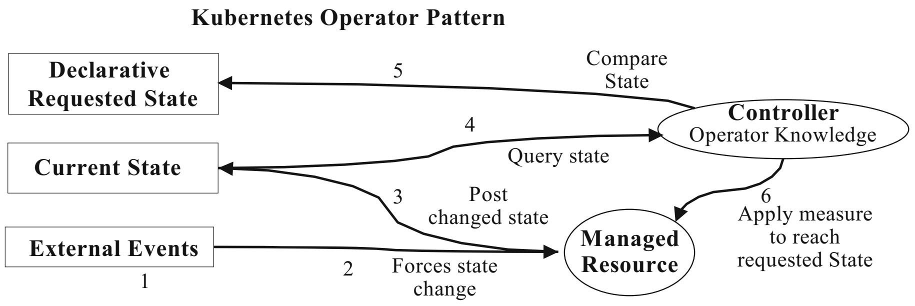

Fig. 5. K8s control loop of the operator pattern.

In summary, Kubernetes manages the execution environment at and above the Pod level, but not the application within the containers. The operator[4]pattern is intended to close this gap. That is, human operator functions were made available to a K8s operator to manage sets of services in an automated way via K8s APIs. For example, the imitation of a human database operator through database-specific administration logic implemented with scripts or program modules that specify setup, configuration and management of the database in a production environment.

4 Solution Approach

For our IG solution design, we envisioned a 2-level hierarchy of five K8s operators. The first operator on the left of Fig. 6 represents the top level ILG service operator, who controls and monitors the four operators at the 2nd-level. These are the Repository service, the Client service, the ObjServer service, and the DB service, which together form the four-tiered deployment topology shown in Fig. 4. As can be seen, the web and application tiers are mapped to three stateless services implemented as K8s Deployments. The combined database and storage tier are implemented through a K8s StatefulSet, which is used to control and manage the DB service operator, as shown in Fig. 6, bottom right. The DB service operator contains the definition of the DB cluster and the logic required to support high availability, read-scalability and disaster recovery. We deployed and tested the prototype implementation in 2 phases. In the first phase, we focused on the stateless services of the web and application tier, which are shown as the upper part of the topology graph in Fig. 7. In the second phase, we developed and deployed the underlying stateful repository services, including the database and storage tiers shown in the image at the bottom of the topology graph.

[4] https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/.

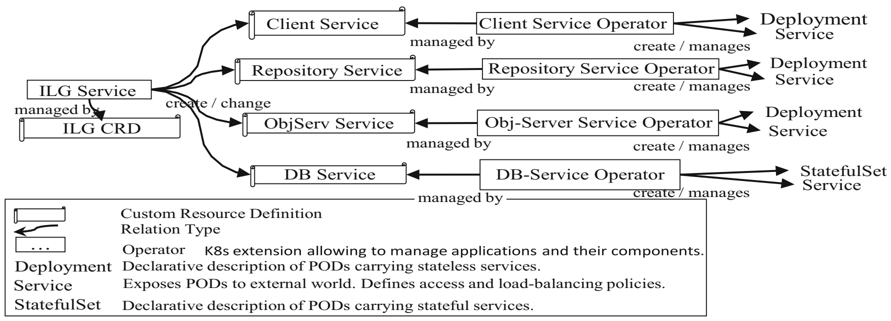
Client Service
Client Service Operator
managed by
create / manages
Deployment
Service
Repository Service
Repository Service Operator
managed by
create / manages
Deployment
Service
ILG Service
managed by
ILG CRD
create / change
ObjServ Service
Obj-Server Service Operator
managed by
create / manages
Deployment
Service
DB Service
DB-Service Operator
managed by
create / manages
StatefulSet
Service
Custom Resource Definition
Relation Type
Operator K8s extension allowing to manage applications and their components.
Deployment Declarative description of PODs carrying stateless services.
Service Exposes PODs to external world. Defines access and load-balancing policies.
StatefulSet Declarative description of PODs carrying stateful services.

Fig. 6. K8s operator hierarchy for managing ILG deployment topology.

By stateful database services we mean a service that is resilience to component failures. In our context this might be database instance, a storage or a network failure. The implemented solution is a shared-nothing database cluster with at least 3 independent database instances and a mechanism that replicates the database data using synchronous or asynchronous replication.

The rest of this papers focuses on the aspect of highly available stateful database services and the required orchestration logic used, which we derived from database product guidelines and our own expertise.

4.1 K8s Operator Extended Control Loop

To support a stateful database service by running a cluster of database instances in containers on a virtualized in environment, it was necessary to design database-specific cluster management using the components and a topology shown in Fig. 7. The integration of the database cluster and its execution environment is controlled by the StatefulSet complemented by the DB2 operator, together they control the database overall state and topology through the K8s control plane. The DB2 operator and respective custom control loop is shown in the lower left part of Fig. 7. It also shows the K8s and the DB2 control loops, so-called MAPE loops, a concept that is being discussed in Maurer [15]. MAPE stands for Monitor, Analyze, Plan and Execute, basically the chain of processes that, through decision logic determines what activities must follow after a change of the desired state of the stateful service. The MAPE process steps are: Monitor the target resource state; Analyze and compare current state with the desired state; In case of misalignments, Plan what activities to perform; Execute the reconciliation plan, taking the necessary actions to align current state with desired service state.

4.2 Related Work

The prototype implementation work was done in the course of 4 master thesis at the university of Stuttgart by Gang [11], Trybek [12], Hagemann [13] and Wang [14]. The concept design around dynamic topology was published by Mega [5], Börner [16], and contribution on how application might use the MAPE loop concept came from Ritter [17]. A concept model of an ECM system including governance services was provided

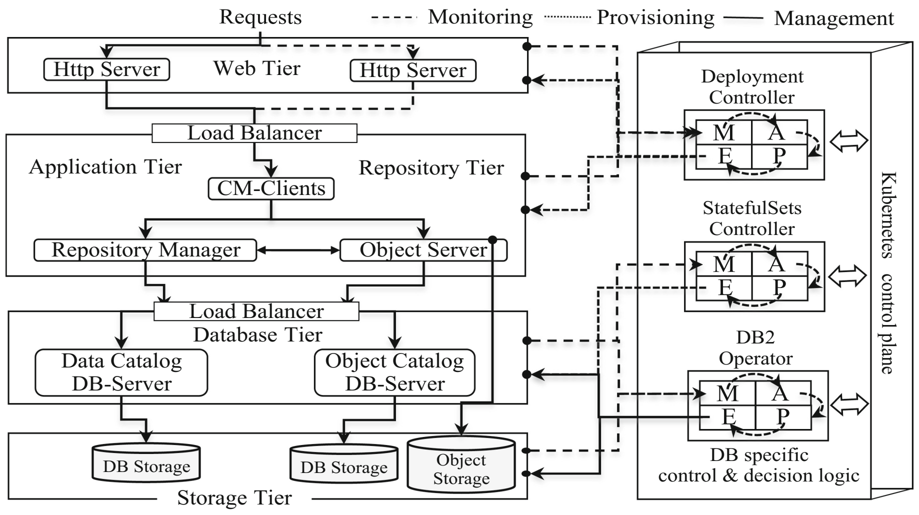

Fig. 7. ILG solution deployment topology and the K8s control loops.

by the IBM Cloud Architecture Center [3]. The CNCF [1] published white paper on the operator pattern provided the ground work for our migration approach. Andrikopoulos [18] in his paper outlines a generic introduction on how to adapt applications for the cloud. Kubernetes best practices, specific to StatefulSets and operators came from Palak [19] at Google, and aspects of EIM practices in companies from Chaki [20]. The California Department of Technology [6] published an ECM reference architecture, that was complemented by information management governance guidelines from Victoria State Government [21] and other agencies, which we used to align our blueprint with. Maurer et al. [15] elaborated on MAPE for autonomic management of cloud infrastructures. Overall, our research lead to several academic sources on stateful services on cloud, but none that address specifically the aspect of refactoring monolithic, legacy IG solutions and none how to move them on cloud execution platforms.

5 The ILG Repository Stateful Service Prototype

Compared to stateless services, stateful services are more complex to design and to implement because K8s was initially designed for stateless services only. Stateful services were introduced later for integrating custom resources. For the prototype we chose an IG solution based on IBM ECM [4] and other necessary components, consisting of IBM Content Navigator, IBM Content Manager, IBM WebSphere Application Server and the IBM DB2 database server. This decision was based on practical experience with these products, in building ECM production systems that provide information governance services. A knowledge we have acquired through several customer projects. The configuration of the prototype is designed to test scale-out, service availability and disaster recovery and was translated into a set of K8s stateful services. The DB2 service operator is used specifically to automate the management of the DB2 database cluster through K8s APIs.

5.1 Kubernetes Stateful Services Cluster Setup

Figure 8 shows the multi-tiered deployment model of the refactored IG solution which uses K8s automated operating concept. This setup, implements the web tier with Docker-compose and focused on the application and database tiers for our HA testing.

The application cluster at Tier-2 is managed by a K8s Deployment artefact not shown in Fig. 8. Instead, the database cluster uses a K8s StatefulSet together with the DB2 operator as Tier-3. The DB2 StatefulSet defines, creates and controls the Pod cluster, which consists of Pod1 - Pod4 running on Node1and Node2. It defines two service entry points SVC-Read/Write and SVC-Read-only, and ensures that the persistent volumes PV1 - PV4 are attached to the Pods. Each Pod consists of one container that hosts one database instance. The StatefulSet also ensures that each Pod has an ordered, stable identity, a unique network identifier and is bound to its persistent volume (PV), surviving deletions and recreations. If a Pod fails or dies, then the StatefulSet control loop will recreate the Pod with exactly the same identity and rebound them to the original PV, ensuring the Pod can access the previously owned database data.

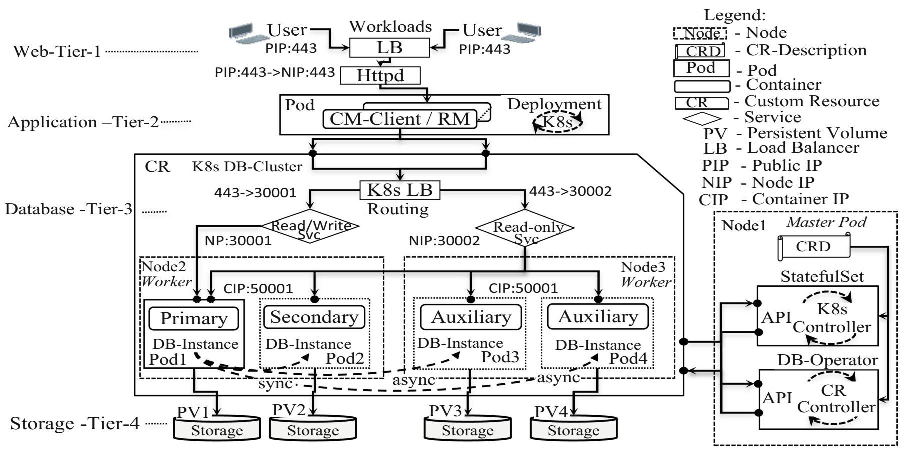

Fig. 8. Deployment model of a K8s cluster of DB2 instances.

The K8s DB2 operator complements the StatefulSet by creating and managing the cluster of DB2 instances using the CRD. The operator itself is deployed in another Pod. Its task is to create the DB2-CR using the DB2-CRD specification, once it is activated. Once active, the Governor, which represents the DB2 cluster control loop, begins monitoring the health of each database instance, continuously compares it to the desired state. If the current state deviates from the desired state, the control loop triggers a series of actions, to reconcile current state with the desired state using database-specific administration logic.

Figure 9 details the DB2 cluster setup in an HA and DR configuration. Primary and secondary instances have each a collocated Governor component.

All four DB-instances have a connection to the DB2 HADR component, which implements the DB2 cluster management logic. Figure 9 also shows the different roles assigned to each cluster members. The primary instance is the cluster leader and owns the reference database. The principal standby instance is attached to the first instance as a peer instance, and its database is the HA-synchronous replication target. Database service fail-over is between primary and standby (the secondary) database instance.

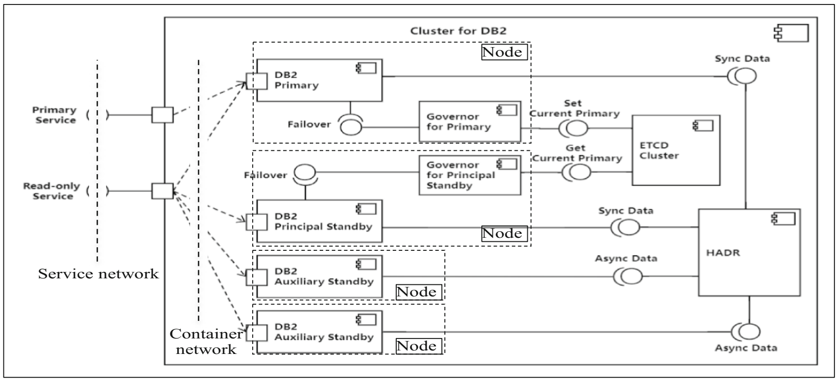

Fig. 9. Component model of a cluster of DB2 instances.

Optionally, there can be up to two auxiliary stand-by instances that can be used to mitigate a production site outage. In this setup, only the primary servers both read/write requests, while all others support read-only requests, forming a read-only scale-out farm. Database operations that modify data are redirected to the primary instance. All changes are propagated to all stand-by instances via log shipping using synchronous or asynchronous replication mode. The synchronization source though, is always the primary. These built-in DB2 capabilities enable HA and DR configurations to be realized, with the positive side effect of supporting scale-out of read-only workloads. The roles of primary and secondary are interchangeable. Fail-over and fallback is triggered by state change events, and state reconciliation is based on the logic implemented through the DB2 operator.

5.2 K8s DB2 Stateful Service Design and Implementation

According to the K8s Operator framework, an operator consists of the following components: API, CRD, CR, a Controller and the resource specific management logic. The operator itself is defined through a K8s 'Deployment' that describes security, roles, accounts management and runs in its own Pod. Figure 10 shows the DB2 operator components and their relationships. By definition, the K8s DB2 operator manages the lifecycle of the DB2 resources, that is, creating and managing the cluster of DB2 database instances that are unknown to K8s and its native cluster management services. In our prototype, the custom resource administration logic is spread among the operator Pod, the DB2 instance Pods and the ETCD Pod. The constituent custom resource components are: Governor, DB cluster controller, DB2 APIs, HADR and ETCD components shown at the bottom of Fig. 10. The ETCD is a distributed key-value store that is used to store the DB-cluster topology information in a look-up table, like: host name, role lock, timestamp and other required configuration parameters.

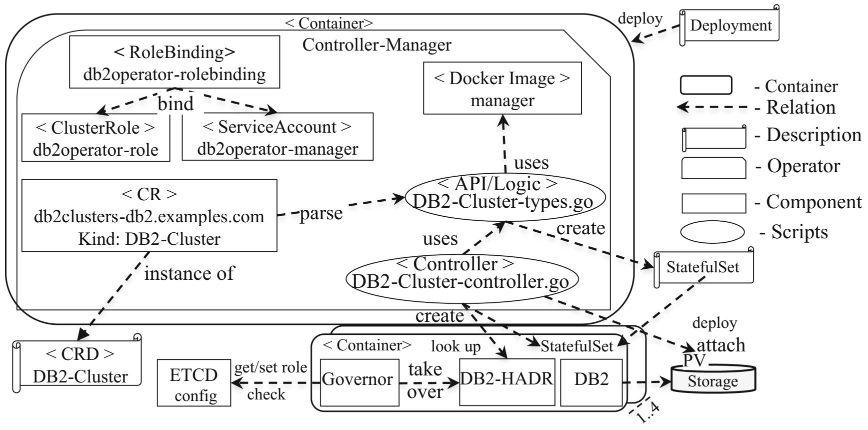

Fig. 10. Components of the Kubernetes Operator for DB2.

The operator component model of Fig. 10 shows the Governor and the HADR component as deployed collocated with the DB instance on every Pod, bottom tight. The DB2 Controller and API, on the other hand, are hosted on the operator Pod, upper part. A possible situation of “split-brain syndrome”, i.e. a situation in which both the primary and the secondary instance try to restart at the same time, resulting in duplicate services, is avoided using ETCD as an external reference point monitored by the Governor, as shown on the bottom left in Fig. 10, overseeing the automated fail-over/fallback process. Creating and managing the database topology is done by the DB2 Controller inside the DB2 Operator.

6 DB2-Operator Prototype Test System Setup

We implemented and evaluated the prototype on our department’s cloud infrastructure. The test environment consists of OpenStack, which is used to provision compute, storage, network and virtual machines (VM). The VMs run an Ubuntu server, configured with Docker, Compose and Kubernetes. The test infrastructure resources include 3 VMs labeled Node1- Node3, the public and internal networks and 4 physical storage volumes PV1-PV4, as shown in Fig. 11. This setup has one master and two worker K8s nodes. The test system includes: an ETCD cluster, the Google Operator template, the HAProxy load balancer and the DB2 Pod cluster. The actual database instances are loaded into the containers using docker images. The DB2 operator artifacts consist of as set of kubernetes YAML scripts and the custom database cluster management tools, implemented as Python, Go and Bash scripts. Figure 11 shows two aspects of our database test system setup, consisting of four Pods, the test client and the HAProxy used as the load balancer. The database instances are configured to start automatically with the Pod using startup shell scripts that start the DB2 database, the Governor component and the DB2 HADR component. The configuration on the left side of Fig. 11 highlights the routing path for the read/write workload, represented by the service selector leading to the primary instance. The right side shows the same configuration with the routing path for the read-only workload, represented by the service selector that leads to all instances, the scale-out farm. Instance state, roles and network configuration are stored and updated periodically in the ETCD key-value store and monitored by a watchdog.

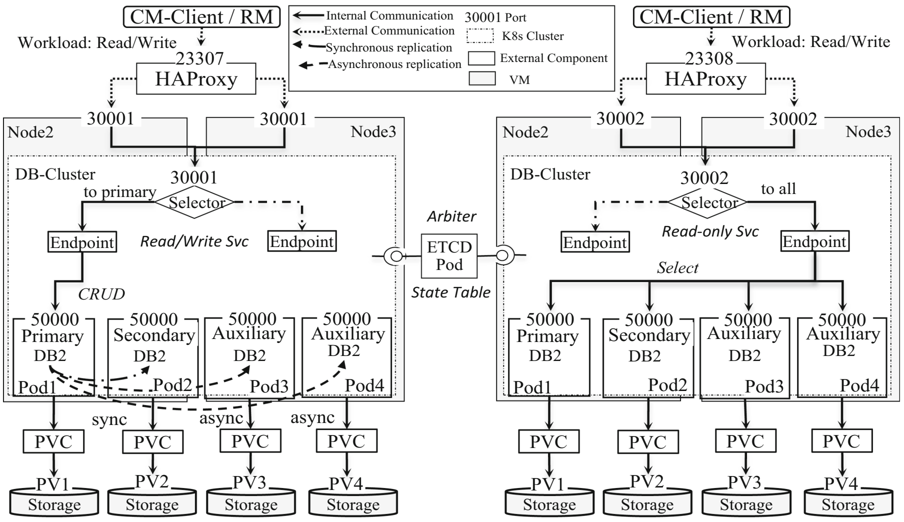

Fig. 11. System under test (SUT) Prototype.

The initialization routines ensure that the K8s service instances receive the correct labels and are associated with corresponding communication endpoints, i.e. their IP addresses and ports. Endpoints configurations are dynamically updated when Pods die or are recreated. Each database instance has a specific role and together represent the DB2 database HA-cluster. Figure 11 also outlines the external and internal communication endpoints and lists the flow of user request for the different workload types.

This test setup includes the HA-Proxy component that plays the role of the request dispatcher and load balancer. We have configured HA-Proxy to run outside of the K8s cluster and to forward incoming client requests to the two worker nodes. HA-Proxy provides a pair of public communications end-points that are linked to the DB service entry points inside the K8s cluster, shown in Fig. 11. K8s Service artifacts act as service proxies of the actual database service. In the case of a read-only workload, the K8s selector (a built-in K8s LB) forwards requests to all Pods across the VM worker nodes to ensure the request traffic is load balanced based on defined policies.

7 Tests, Results and Evaluation

Our test scenarios were created to evaluate the prototype in terms HA, DR and scale-out capabilities. The actual verification tests were developed using a Python client application that simulates an interactive multi-user database transaction workload. We ran several load and scalability tests against the database HA-cluster and collected the results. The external HA-Proxy server provided in-cluster response-time statistics, end-to-end response-times were generated by our own client application. Results include request response times, data throughput, the number of connections, as well as server status, reaction time to failures and service recovery times.

Note: The tests carried out are only indicative and serve to verify the steps of platform migration, estimate approximate effort and prove the feasibility of our approach.

7.1 Service Availability and Failover Scenario

The first test scenario shown in Fig. 12, simulates a failure of the database service by simply deleting the Pod along with the primary database instance, see red lightning bolt at T1. We then measured the time it took until the outage was discovered, the time at which the database service was re-restored and verified the consistency of the database and its data. The relevant HA metrics used are: Reaction time Trec = T2-T1; Fail-over time Tfov = T3-T2; HA-service restore time Tha = T4-T3; and the auxiliary reconfiguration time Taux = T5-T4. The sum over the partial times is the overall configuration reset time. Using the interaction diagram of Fig. 12 we have following flow of events: At T1, the Pod of the primary server is deleted and the primary lock (a timestamp) in ETCD is no longer updated. T2 – is the case when the K8s Deployment control loop (C-loop in the diagram) detects that the primary Pod has died, and re-creates the Pod with the same data (PV1) but with new IP address and eventually on a different node; T3 - the Governor on the secondary server detects that the timestamp of the primary lock exceeds the time to live (TTL) and therefore declares it inactive. At this point the secondary takes on the role of the primary. This is done by starting the DB2 HA-specific take-over process and re-establishing the database service.

T4 is when the Governor on the new clone of the old primary Pod through the ETCD database detects that there is a new active primary server and assigns itself the role of the new secondary server, connects as peer, and starts the synchronous database replication. At T5, the two Auxiliary instances become aware of the role change and reconnect to the new primary instance, triggering the database replication, as shown in the interaction diagram of Fig. 12. We performed 10x test runs of the HA failover scenario and measured reaction-, failover- and service recovery times listed in Table 1. The test results are displayed in seconds. The average measured reaction time of the K8s control loop was about ~2 s, while the service recovery time was about ~14 s on average.

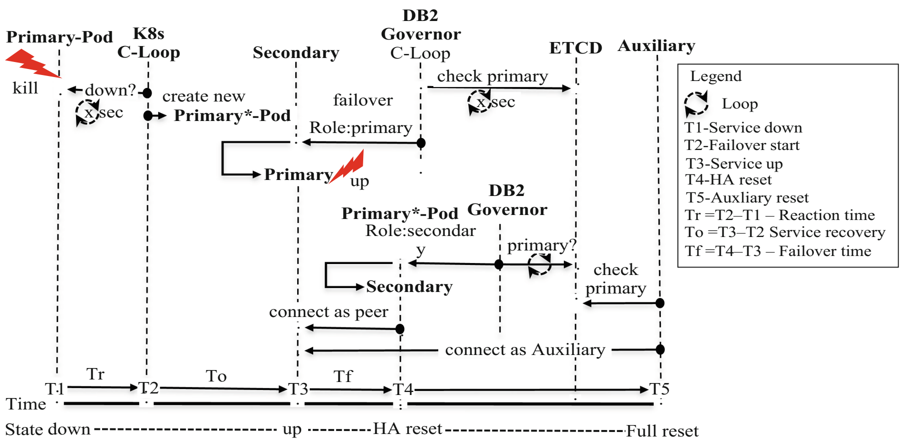

Fig. 12. HA fail-over flow of events and reaction times.

Table 1. DB availability (HA) failover response time test results.

#	Trec (s)	Tfov (s)	Tha (s)
1	2.859	3.621	6.425
2	1.995	4.322	6.839
3	2.403	9.655	18.180
4	2.066	5.856	13.793
5	2.022	36.309	41.639
6	2.051	9.632	14.555
7	1.720	4.839	9.570
8	2.058	29.886	32.728
9	2.624	9.679	18.111
10	2.059	33.286	34.581
Avg.	**2.186**	**14.709**	**19.642**

The variations in failover time, i.e. the time it takes for the new secondary database to restore HA-state, were approximately ~20 s, as this depends on the think time of the governor's control loop, which was configured to 30 s. In the worst case, this means a 30 s wait period before the role change happens, plus some time delay due to system load, which explains the magnitude of the fluctuations in the Tf response time. The results suggest that the reaction time of the K8s control loop is negligible compared to the service recovery time. K8s rebuilds Pods faster than the time it takes for the custom resource takes to restore service. In our test environment, service recovery seems to be in the of range of minutes, while Pod cloning is in the range of tenths of seconds. Overall though, the results obtained prove the feasibility of our approach.

7.2 Read-Only Workload Scalability Test

The first series of tests focuses on determining the response time characteristics of the DB2-cluster under different loads. The test is to create an increasing number of interactive (10–30) virtual users that send a series of SQL requests to a database table that is replicated to the four DB2-cluster members. The workload itself consists of simple select statements that are issued in a loop with a short think time in between. Each run is repeated with 10, 20 and 30 simulated users against 1, then 2, then 3 and finally 4 DB2-instances. Each user sends 1000 requests for a total workload of 10.000, 20.000 and in the third iteration 30.000 read requests, which equates to 30.000 database transactions at peak. Figure 13, shows the response time of Pod/ DB-instances (x-axis) and workloads (colored horizontal lines).

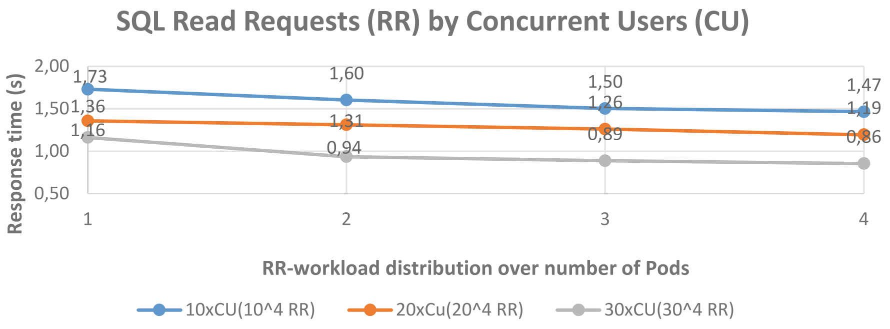

Fig. 13. Average read response time by number of users and DB-instances. (Color figure online)

As can be seen, the three response time graphs all show that time gradually decreases as Pods (instances) are added independent of the workload used: blue (10.000), orange (20.000), grey (30.000). This is Easily explainable, because the workload is distributed across all available instances. Instead, with a constant number of Pods, the response time increases as the workload increases demonstrating the scale-out behavior of the system.

In the second scenario, load balancing across the DB2-cluster is evaluated. With this test case, we investigated the distribution of work among the cluster members by constant load. The results of the three set of tests are shown in Fig. 14. The vertical axis shows the number of read requests generated by the virtual users in batches of 10.000, 20.000 and 30.000 transactions. For each batch, we repeated the test with 1,2,3 and 4 DB2-instances.

The graph shows the system load as color-coded sections of the vertical bars. The blue area represents the primary, brown the secondary, grey the auxiliary-1 and yellow the auxiliary-2 database instance. The number of transactions served by each DB2-instance is proportional to the size of the section in the respective bar.

The result demonstrates, that using a constant workload with a growing number of DB2-instances, the individual load on each instance decreases as the overall workload is distributed across all cluster members.

Here, too, the test results confirm our claim. Therefore, migrating legacy IG solutions to cloud execution environments is feasibility and with an affordable effort. Typical IG solution capabilities such as HA, DR and scale-out are retained, benefiting automated service delivery, flexible resource allocation, and reduced operational costs.

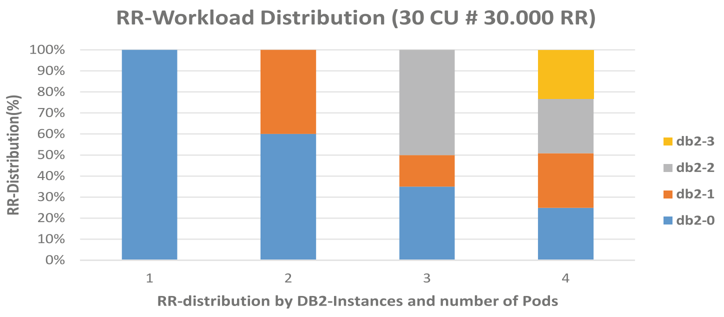

Fig. 14. Read-only workload. Response time and request

8 Conclusion and Outlooks

This paper explored the effort required to migrate a legacy IG solution designed to operate in a pre-configured, physical production environment to a dynamic software-defined cloud infrastructure (SDI). The focus of this work was on refactoring the legacy solution design and successfully moving from a physical to a cloud environment. The benefit gained from this is the ability to orchestrate ILG workloads using stateful services in K8s managed clusters. We have learned that cloud platforms provide cluster control mechanisms and resource topology management across all solution tiers, which can simplify and reduce the application-specific cluster management complexity. With Pod clusters managed by K8s, Pod, Node, Network and Storage management is kept out of application responsibility and centrally consolidated in the cloud platform. This makes application layer-specific cluster management obsolete and solution design leaner. In addition, built-in control loops with the cloud platform enables monitoring of resource health and automatic triggering of provisioning and de-provisioning requests. Component specific lifecycle management tasks are integrated as K8s extensions using the operator pattern. Operators make it possible to take advantage of the elasticity of the cloud infrastructure and react dynamically to changes in workload. The resulting effects are the avoidance of manual interventions, gain in flexibility and the reduction of associated operating costs.

Our prototype leveraged the IBM ECM product stack, consisting of IBM Content Navigator, IBM Content Manager Enterprise Edition, along with the required IBM WebSphere Application Server and IBM DB2 database server. We have developed an IG solution design as used by traditional ECM customers world-wide, most of whom still run their systems on-premise on a physical infrastructure. Currently, all products support virtualized environments, but not all support containerized in virtualized environments. We couldn't, find a customer story that holistically shows the migration of an IG solution from physical to cloud, but several blogs explaining cloud implementations of individual component. Future work could focus on real-world production deployments and repeat our tests with more realistic workloads and database sizes.

References

1. CNCF, CNCF operator white paper. https://github.com/cncf/tag-app-delivery/blob/main/operator-wg/whitepaper/Operator-WhitePaper_v1-0.md. Accessed 30 July 2023
2. Panetta, K.: Gartner keynote: the future of business is composable the future of business is composable, 19 October 2020 https://www.gartner.com/smarterwithgartner/gartner-keynote-the-future-of-business-is-composable. Accessed 30 July 2023
3. IBM, Content management: content services reference architecture. https://www.ibm.com/cloud/architecture/architectures/contentManagementdomain/reference-architecture/. Accessed1 Mar 2023
4. IBM Corporation, "IBM Content Manager Enterprise Edition components," IBM Corporation, Online (2023)
5. Mega, C., Waizenegger, T., Lebutsch, D., Schleipen, S., Barney, J.M.: Dynamic cloud service topology adaption for minimizing resources while meeting performance goals. IBM J. Res. Dev. **58**, 1–10 (2014)
6. California department of technology , enterprise content management reference architecture, California department of technology 1325 J Street, Suite 1600, Sacramento, CA 95814 (2014)
7. Alfresco, Alfresco Content Services (2021). https://www.alfresco.com/platform
8. IBM Corporation, "FileNet P8 baseline architecture," IBM Corporation, Online (2023)
9. Lebutsch, D., Bolik, C., Hennecke, M.: Content management as a service—financial archive cloud. Datenbank-Spektrum **10**, 131–142 (2010)
10. kubernetes.io, "Kubernetes Concepts," 30 July 2023. https://kubernetes.io/docs/concepts/. Accessed 30 July 2023
11. Shao, G.: About the design changes required for enabling ECM systems to exploit cloud technology (2020)
12. Trybek, C.: Investigating the orchestration of containerized enterprise content management worklaods in cloud environments using open source cloud technology based on kubernets and docker (2021)
13. Hagemann, P.: About the design changes required for enabling ECM systems to exploit cloud technology (2021)
14. Wang, X.: Orchestrating stateful database services in cloud environments using Kubernetes stateful services framework, OPUS - publication server of the University of Stuttgart (2022)
15. Maurer, M., Breskovic, I., Emeakaroha, V.C., Brandic, I.: Revealing the MAPE loop for the autonomic management of Cloud infrastructures. In: 2011 IEEE Symposium on Computers and Communications (ISCC) (2011)
16. Andreas, B.: Orchestration and provisioning of dynamic system topologies, Stuttgart (2011)
17. Ritter, T., Mitschang, B., Mega, C.: Dynamic provisioning of system topologies in the cloud. In: Enterprise Interoperability V, London (2012)
18. Andrikopoulos, V., Binz, T., Leymann, F., Strauch, S.: How to adapt applications for the cloud environment – challenges and solutions in migrating applications to the cloud. Computing **95**, 493–535 (2013)
19. Bhatia, P., Tee, J.X.: Best practices for building Kubernetes Operators and stateful apps. https://cloud.google.com/blog/products/containers-kubernetes/best-practices-for-building-kubernetes-operators-and-stateful-apps. Accessed 20 October 2018
20. Chaki, S.: Enterprise information management in practice (2015)
21. .StateGovernmentofVictoria, "Information Management Maturity Measurement" (2019)
22. IBM Corporation, "IBM Enterprise Content Management Performance Methodology," IBM Corporation, Online (2015)
23. IBM Corporation, "IBM FileNet Content Manager 5.2High Volume Scalability," IBM SWG, Online (2014)
24. IBM Corporation, "IBM Content Services," IBM Corporation, Online (2022)