Postgres Lifecycle Management Operators in Kubernetes

Miroslav Šiřina

Bakalářská práce
2023

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2022/2023

ZADÁNÍ BAKALÁŘSKÉ PRÁCE
(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Miroslav Šiřina**
Osobní číslo:	**A20079**
Studijní program:	**B0613A140020 Softwarové inženýrství**
Forma studia:	**Kombinovaná**
Téma práce:	**Správa životního cyklu Postgres v Kubernetes pomocí operátorů**
Téma práce anglicky:	**Postgres Lifecycle Management Operators in Kubernetes**

Zásady pro vypracování

1. Seznamte se s operátory v Kubernetes a Postgres databázovým systémem.
2. Definujte životní cyklus databázového serveru.
3. Vyberte vhodné operátory a popište je.
4. Sestavte metodiku testování jednotlivých operátorů.
5. Na základě sestavené metodiky otestujte vybrané operátory.
6. Proveďte zhodnocení.

Forma zpracování bakalářské práce: **tištěná/elektronická**
Jazyk zpracování: **Angličtina**

Seznam doporučené literatury:

1. SAYFAN, Gigi, 2020. Mastering Kubernetes: level up your container orchestration skills with Kubernetes to build, run, secure, and observe large-scale distributed apps. Third edition. Birmingham: Packt Publishing. ISBN 9781839211256.
2. RIGGS, Simon a Gianni CIOLLI, 2022. PostgreSQL 14 Administration Cookbook. Birmingham: Packt publishing. ISBN 9781803248974.
3. DOBIES, Jason a Joshua WOOD. Kubernetes operators: automating the container orchestration platform. Sebastopol, CA: O'Reilly Media, 2020. ISBN 9781492048046.
4. FARLEY, David, 2021. Modern software engineering: doing what really works to build better software faster. Boston: Addison-Wesley. ISBN 9780137314911.
5. DAME, Michael, 2022. The Kubernetes Operator Framework Book. Birmingham: Packt Publishing. ISBN 9781803232850.

Vedoucí bakalářské práce: **Ing. Peter Janků, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání bakalářské práce: **2. prosince 2022**
Termín odevzdání bakalářské práce: **26. května 2023**

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky

L.S.

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 7. prosince 2022

Prohlašuji, že

- beru na vědomí, že odevzdáním bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že bakalářské práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky. Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má Univerzita Tomáše Bati ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – bakalářskou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne ....

podpis studenta

ABSTRAKT

Text abstraktu česky

Klíčová slova: Přehled klíčových slov

ABSTRACT

Text of the abstract

Keywords: Some keywords

Zde je místo pro případné poděkování, motto, úryvky knih, básní atp.

TABLE OF CONTENTS

INTRODUCTION

It is essential for the database server to be as close as possible to the applications that are using it. This reduces the number of men in the middle between the database and the application, which reduces database access latency and thus reduces overall application latency and increases security. The mass migration of applications to Kubernetes clusters implies a necessary shift of Postgres to Kubernetes. This thesis defines Postgres, Kubernetes, and their operators. It then further describes the lifecycle of a Postgres cluster, and searches for operators capable of managing this lifecycle. It establishes metrics by which it tests and evaluates these operators. The result of this thesis is the recommendation of a suitable operator based on the defined metrics.

TBD - switch to Czech layout

TBD - Motto

TBD - Czech abstract, English abstract

TBD - Poděkování, motto

TBD - adjust the test theory to test process

TBD - describe tools

I. THEORY

1 BACKGROUND

This chapter introduces the key technologies used in this thesis including Postgres, Kubernetes, and Kubernetes operators.

1.1 Postgres

PostgreSQL is a powerful object-relational database management system (ORDBMS) derived from the POSTGRES package written at the University of California at Berkeley. [1] [2] The first version of POSTGRES was released in June 1989. POSTGRES has been used in many applications, including financial data analysis systems, asteroid tracking databases, medical information database, and several geographic information systems. The size of external community users has nearly doubled by 1993. [3]

POSTGRES was using its POSTQUEL query language from version, until Andrew Yu and Jolly Chen introduced SQL to POSTGRES in 1995. The name has changed to Postgres95. Postgres95 was completely ANSI C code reduced by 25 % and was 30 – 50 % faster than Postgres 4.2. [3]

It was clear by 1996 that the name would not stand the test of time therefore it has been renamed to PostgreSQL. As stated by PostgreSQL documentation [3]: "Many people continue to refer to PostgreSQL as "Postgres" (now rarely in all capital letters) because of tradition or because it is easier to pronounce. This usage is widely accepted as a nickname or alias." This thesis will use Postgres as an alias for PostgreSQL as well.

More than 30 years after the first version Postgres has been considered the most used ORDBMS for professional developers by Stack Overflow survey [4]. According to Riggs and Ciolli [2]: "The PostgreSQL feature set attracts serious users who have serious applications. Financial services companies may be PostgreSQL's largest user group, although governments, telecommunication companies, and many other segments are strong users as well." It is fully ACID compliant [5] and supports many kinds of data models such as relational, document, and key/value. [2]

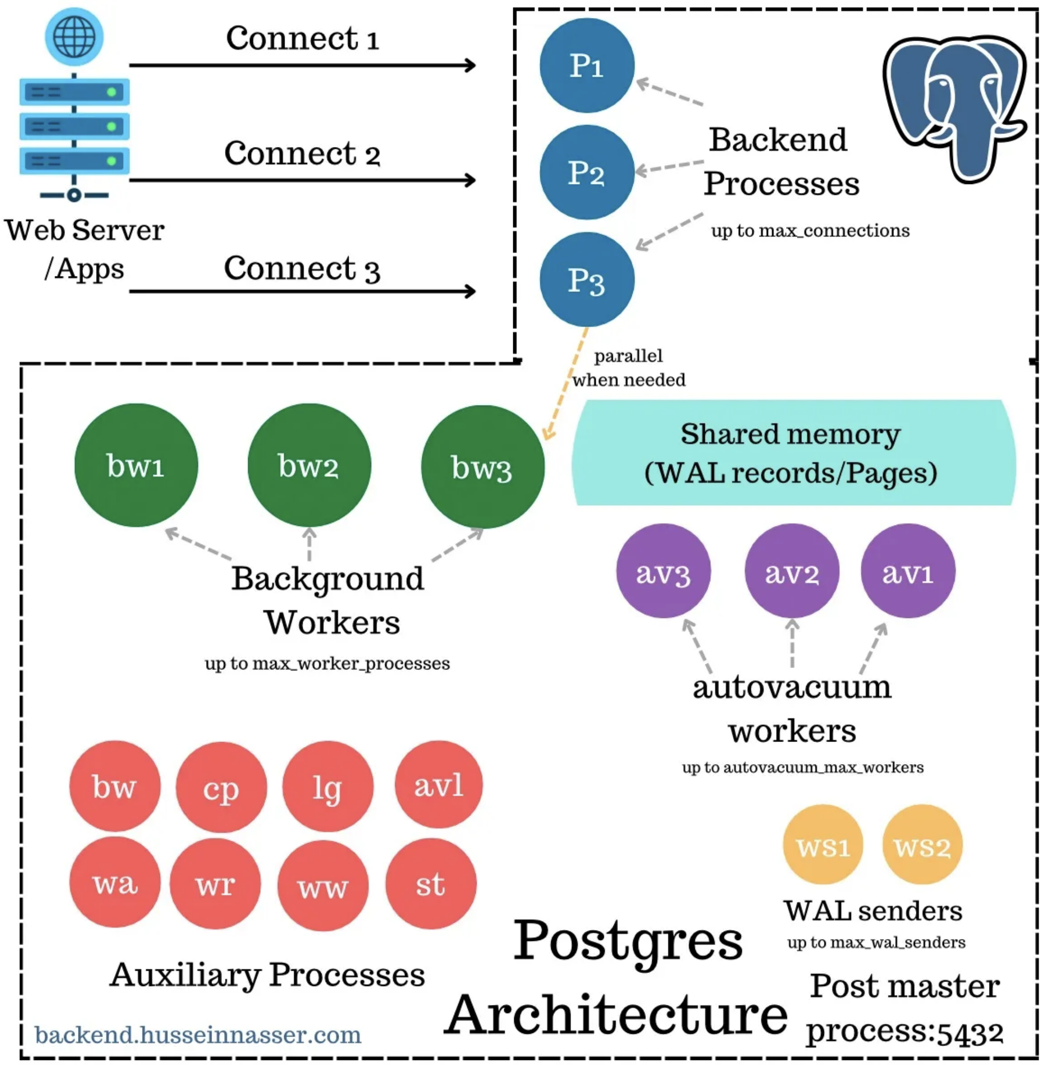

Figure 1.1 Postgres Architecture [6]

1.1.1 Write Ahead Log

Write-ahead Logging (WAL) is esential Postgres technique to ensure data integrity. Its main concept is that changes in data files (where tables and indexes are stored) must only be written after they are logged (saved to a log file). That means the database is updated after the changes are written to disk. In the event of a system crash, all transactions will be recovered from the disk. [7]

Although WAL is primarily designed for recovery after a database server crash, its design also allows any changes to the database server state to be replayed backward. A copy of the log is also a form of backup. Thus, for recovery to a point in time, only logs that have been saved to that point in time can be restored. This technique is called Point-In-Time Recovery (PITR). [8] These log files can also be streamed to other nodes to serve as a replica or remote backup. [9]

1.1.2 Backup and restore

A full set of backup commands is included in Postgres. Among the simple backup commands are pg_dump and pg_dumpall, which enable one or more databases to be saved in SQL format. A wide range of configuration options are available for these commands, including compression for large databases or exporting only the database schema. To restore a database from a file at a later time, the psql command can be used, which is capable of restoring a database from its dump. [10] These commands are also helpful with migration from one major Postges version to another because the dumped files are plain SQL commands.

However the backup options in Postgres are quite limited. Postgres allows to set up of a backup command that runs after the next log file is created, database dumps, and log streaming. For more advanced backup techniques, like scheduled backup or cloud backup, additional software such as PgBackRest must be utilized. [8]

PgBackRest PgBackRest is a reliable and simple backup and restore solution that provides many features on top of classic Postgres backup and restore tools like parallel backup options with compression, local or remote backups, cloud backup (S3. Azure and Google Cloud), or backup encryption. Full, incremental, or differential backup is also supported. [11]

1.1.3 High Availability

The basic structure of a database cluster consists of one or more database servers, which can be called nodes. In Postgres there are two types of nodes, Primary node and Standby node. A Primary node is such a node that allows reading and writing information. The newly written information is then streamed to the Standby nodes. Standby nodes are read-only, they do not allow writing. [9]

Achieving high availability with Postgres is possible by using more than one node in the cluster. Two options are possible here. A single Primary node option, where the Primary node is read and write enabled, and the other nodes are Standby nodes. If the Primary node is unavailable, then the Standby node is promoted to the Primary node.

If this event is planned it is called a switchover event, otherwise it is a failover. In this variant, the Primary node streams the logs to the Standby nodes. The second option is to use multiple Primary nodes. However, conflicts can occur because all Primary nodes allow concurrent writes. [12]

Patroni Since Postgres does not provide any software that can detect that a node is unavailable, it is necessary to use software outside of Postgres [13], such as Patroni. Patroni is a popular open-source tool created by Zalando to achieve high availability of Postgres clusters. Patroni uses a distributed configuration source such as ZooKeeper, Etcd, Consul, or Kubernetes for its operation. Patroni can automatically adjust the settings of all managed nodes, therefore it can automate failover and make it seamless. [14] [15]

1.1.4 Load Balancing and Connection Pooling

Using more than one node allows to direct traffic to a node that is less busy and thus achieve load balancing. Postgres doesn't come with any software that allows splitting the load on different nodes, so it is necessary to use an external load balancer such as HA Proxy or pgBouncer. The load balancer then acts as an intermediary between the database and the client and directs the traffic to the available nodes according to the set rules. These load balancers also enable connection pooling which is a technique for managing and reusing database connections to increase performance and reduce overhead. Connection pooling involves creating a pool of pre-created connections that can be shared and reused by multiple client requests, instead of creating a new connection for each request. This removes the overhead of creating a new process each time a client connects to Postgres and allows the client to use resources that would otherwise be used to service multiple requests (or complete them faster). [16]

1.2 Kubernetes

Kubernetes, also known as K8s, is an open-source platform for automating deployment, scaling, and management of containerized applications. It provides a way to manage and orchestrate containers, which are units of software that package up an application and its dependencies into a single, isolated package that can run consistently on any infrastructure. [17]

As described by Kubernetes Documentation [18] Kubernetes provides several key features, including:

- **Service discovery:** A container can be exposed by Kubernetes either through its DNS name or its own IP address.
- **Load balancing:** In the case of high traffic to a container, stability of the deployment can be ensured by Kubernetes load balancing and distributing the network traffic.
- **Storage Orchestration:** Storage orchestration in Kubernetes allows for the automatic mounting of a storage system of choice, including local storage, public cloud providers, and others.
- **Automated rollouts and rollbacks:** The desired state of deployed containers can be described using Kubernetes, and the actual state can be changed to the desired state at a controlled rate. For instance, the automation of Kubernetes can be utilized to create new containers for the deployment, remove existing containers, and transfer all their resources to the newly created container.
- **Automatic bin packing:** A cluster of nodes for running containerized tasks is provided to Kubernetes. The amount of CPU and memory required by each container is specified to Kubernetes. The optimal utilization of resources can be achieved by Kubernetes fitting the containers onto the nodes.
- **Self healing:** Containers that fail are restarted by Kubernetes, those that do not respond to the user-defined health check are replaced or killed, and they are not advertised to clients until they are deemed ready to serve.
- **Secret and configuration management:** Sensitive information, such as passwords, OAuth tokens, and SSH keys, can be stored and managed by Kubernetes. The deployment and updating of secrets and application configuration can be done without the need to rebuild container images and without the exposure of secrets in the stack configuration.

1.2.1 Kubernetes Components

Kubernetes cluster is composed of a set of worker machines that run containerized applications called nodes. Each cluster must have at least one node. [18]

The Kubernetes control plane is the management system of a Kubernetes cluster, responsible for maintaining the desired state of the cluster. It consists of multiple

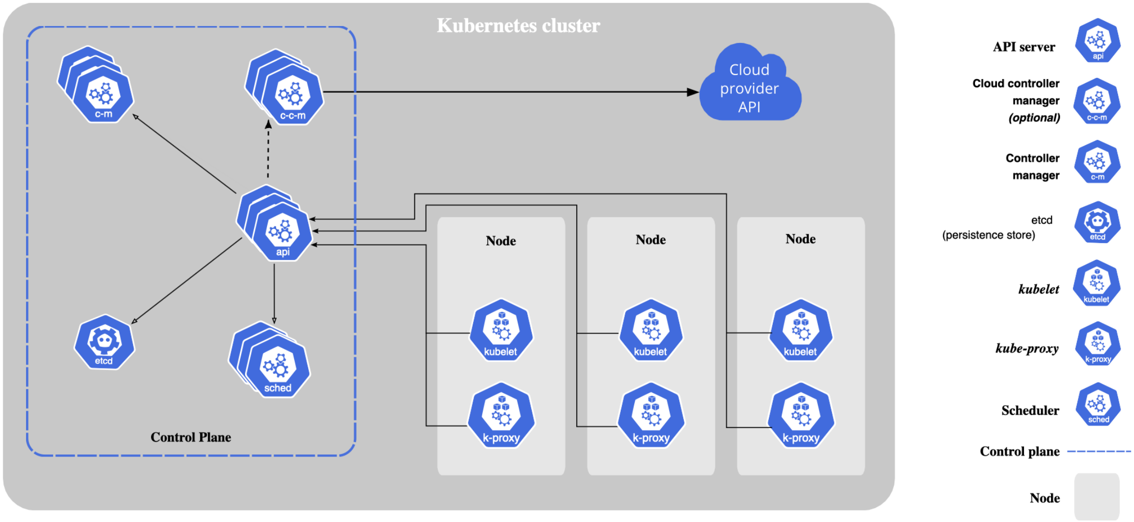

Figure 1.2 The components of a Kubernetes cluster [18]

components that work together to manage the cluster and its resources, including pods, services, and volumes. The key components of control plane are [19]:

- **kube-APIserver:** Acts as the front-end for the Kubernetes API and exposes the API to other components. [18]
- **Etcd:** Highly available distributed key-value store that serves as the backing store for the cluster's configuration data. [20]
- **kube-scheduler:** Assigns work to nodes in the cluster, such as scheduling pods to run on nodes. [21]
- **kube-controller-manager:** Monitors the cluster's state and makes adjustments as necessary to maintain the desired state. [19]
- **cloud-controller-manager:** Manages cloud-related tasks such as node creation and management, volume management, and load balancing, allowing the other components of the control plane to focus on their specific responsibilities. Cloud manager is optional. Can be avoided when Kubernetes not used in cloud. [18]

Node components: Node components in a Kubernetes cluster run on each node and provide crucial functionality for the operation of containers on that node. [18]

- **kubelet:** Is responsible for communicating with the control plane and ensuring that containers are running and healthy. [22]

- **kube-proxy:** Is responsible for maintaining network rules on the nodes, allowing network communication to the containers. It enables the containers in a pod to communicate with other containers and the outside world, and performs tasks such as load balancing and traffic routing. [22]
- **container runtime:** Is responsible for running containers. [18]

1.2.2 Kubernetes Concepts

Pod is the smallest deployable unit that can be created in Kubernetes. [23] A Pod in Kubernetes is comprised of multiple containers and storage volumes that are run together within the same execution environment. As a result, all containers included in a single Pod will always run on the same machine. [21] A Pod's specifications are outlined in a Pod manifest, which is simply a JSON or YAML text file that represents the Kubernetes API object. Kubernetes follows a declarative configuration approach, where the system's desired state is defined in a configuration file, and the service then implements the necessary changes to make the desired state a reality. [24]

ReplicaSet's purpose is to ensure a consistent number of replica Pods are running at all times. It is commonly used to guarantee a specified number of identical Pods are available. However, a Deployment is a more advanced concept that oversees ReplicaSets and provides a more streamlined way to make updates to Pods. It also offers additional features. As a result, it's advisable to use Deployments instead of directly utilizing ReplicaSets, unless there is a need for specific update requirements or no need for updates at all. [25]

Service is an abstraction layer and defines a group of Pods and the method to access them (often referred to as a micro-service). The group of Pods targeted by a Service is usually specified through a selector. The Service abstraction makes this possible by enabling the decoupling of components. [26] Kubernetes includes built-in service discovery mechanisms. When a service is created in Kubernetes, it is automatically assigned an IP address and DNS name. Clients and other services can use this name or address to access the service within the Kubernetes cluster. [26]

Containers and pods in Kubernetes are ephemeral. When a container is terminated, any data it has written to its own filesystem is lost. In Kubernetes, storage is represented by a basic abstraction called "volumes". Containers use these volumes by binding them to their respective pods, and can then access the storage regardless of its physical location as if it were a part of their local filesystem. [27]

Kubernetes version 1.5 came with a new object called StatefulSet that allows a set of stateful pods to be deployed and managed. Each pod has a unique, stable network identity and a persistent storage volume. This enables stateful applications like databases to be run on Kubernetes. Advantages of using StatefulSets include predictable naming schemes, ordered pod creation and deletion, and unique persistent storage. [28] [29]

In version 1.7, Kubernetes introduced the Custom Resources extension to its API. [30] This extension allows Kubernetes to use user-defined resources that are not native to Kubernetes as if they were native. [31] Custom resources (CR) is an extension to the Kubernetes API that extends the deployment with additional parameters that are not part of it. CR stores these parameters and allows the API server to access them just like the native Kubernetes parts. CR is created in the Kubernetes cluster using a definition called Custom Resource Definition (CRD). [32]

Kuberentes controllers are control loops[1)] that constantly check the state of their controlled objects. If the controlled objects are not in the desired state, the controller performs actions to get the controlled objects into that state. For example, restart a crashed node, add a new replica, modify settings, etc. [33] However, to work with CR, custom controllers that can work with these resources must be created, these controllers are called Custom Controllers. [34]

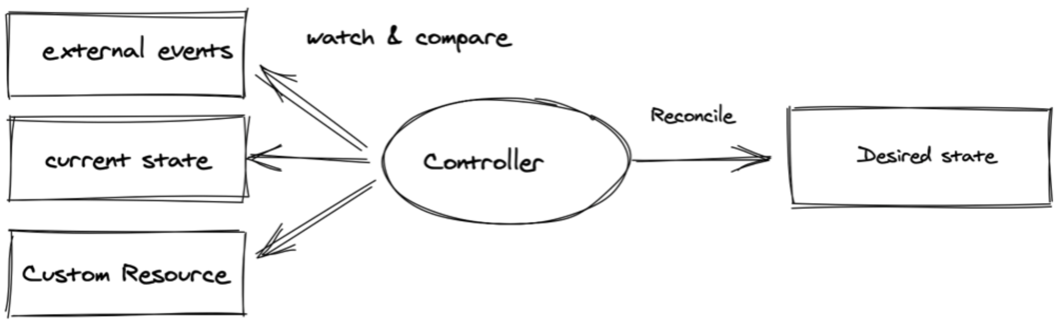

Figure 1.3 Kubernetes controller [35]

1.3 Running Postgres in Kubernetes

Due to its universality Kubernetes cannot know all complex stateful applications, which can contain a large number of nodes and have a wide range of uses while remaining general-purpose. The goal of Kubernetes is to provide an abstraction covering basic application concepts and providing options for extensions for more complex applications

[1)] A control loop is a process that continuously monitors the state of a system, compares it to a desired state, and makes adjustments to bring the system closer to the desired state.

and their specific operations. Kubernetes cannot and should not know all the possible settings and operations that, for example, a Postgres cluster needs to run. [36]

The easiest way to run Postgres in Kubernetes is through the StatefulSet just mentioned. This StatefulSet can start a Postgres pod, create a persistent volume, and connect this volume to the pod. A stateful set can do this for all replicas set in its configuration. It can also scale up or down. Unfortunately, however, all independent Postgres instances created by StatefulSet controller are not synchronized in any manner.

This basic setup may be sufficient for running a single node, but it is no longer sufficient fro managing the whole Postgres lifecycle. For managing whole Postgres lifecycle it is necessary to install other applications in the Kubernetes cluster and then configure the entire Postgres cluster to work with them. This represents a large amount of work and subsequent maintenance that Kubernetes operators can facilitate.

1.4 Database System Lifecycle

The database system itself is a software like any other. It is therefore also subject to the same life cycle as software. As depicted in figure 1.4 application lifecycle consists of three main parts. It is the governance part, development, and operations. For this thesis, only the operations part is relevant because it is the only part we are able to control.

Operation is the process of running and managing the application, which starts with deployment and continues until the application is taken out of service. This aspect of the application lifecycle management covers the release of the application into production, ongoing monitoring, and other related tasks. [37]

Therefore the complete database life cycle can be outlined by following events:

- System installation
- System upgrade to a newer version (major and minor)
- System backup
- System restore
- System monitoring

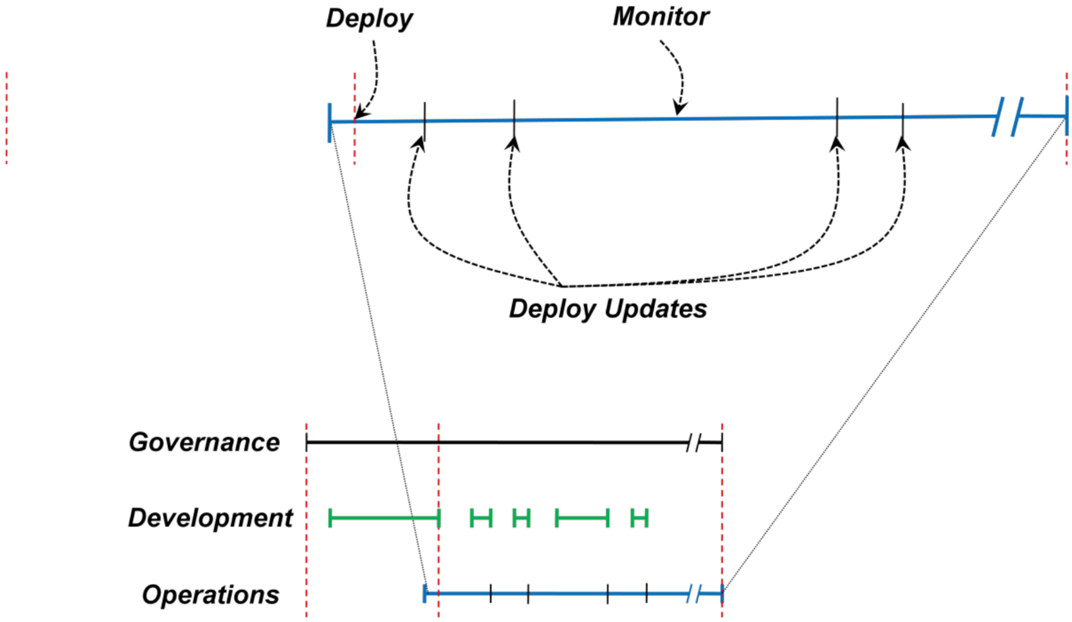

Figure 1.4 Application Life Cycle [37]

- System scaling (vertical and horizontal)
- System configuration update
- System uninstall

1.5 Operators

As described in chapter 1.3, Kubernetes can run stateless applications very well. But its general purpose makes running complex stateful applications on top of it quite challenging.

This has changed in 2016 when CoreOS came up with operators as a way to deploy complex applications with state such as databases, caches, or monitoring systems. [38]

An operator is a special kind of software that extends the Kubernetes API and has a particular knowledge of managed resource that Kubernetes does not have. The operator also serves as a packaging mechanism for distributing applications including their dependencies in Kubernetes. The operator can manage, restore, update or monitor the resource. It can also manage very complex applications. The Kubernetes operator thus replaces the human operator after which it is named, who would otherwise take care of these tasks. [39] [38]

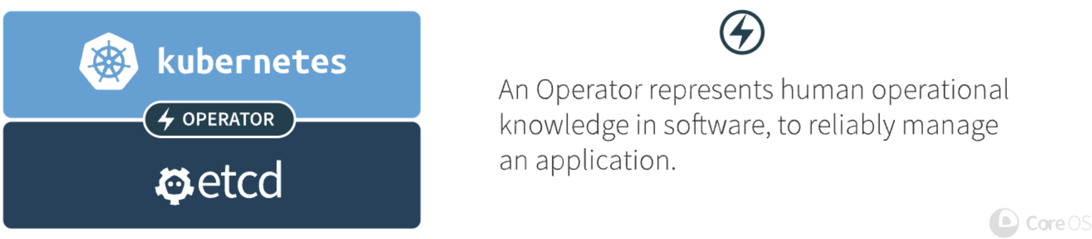

Figure 1.5 Definition of Kubernetes operator [38]

CoreOS demonstrated the use of its operator on Etcd (described in the Kubernetes Components chapter). When new Etcd nodes are created, it is necessary to give them a DNS names and use the Etcd cluster management tools to add the new nodes to an existing cluster. CoreOS has automated these tasks with the Etcd operator so that all that is required is to increase the number of replicas in the operator CRD and the Etcd operator will perform these tasks instead of a human operator. [38] By embedding the human operator's operational knowledge into the code, this ensures that these tasks are repeatable, testable and upgradable. It also ensures that the necessary operations are always performed, executed in the order in which they are supposed to be performed, and none are skipped. This reduces the number of hours spent on dull but essential work such as backups. [35]

As described by operator White Paper [35] and depicted in figure 1.6, operator consists of the following parts

- The managed application or infrastructure
- Software that has some specific knowledge of the managed application or infrastructure and allows the user to declaratively set the desired state
- Custom Controller, which is responsible for achieving the desired state

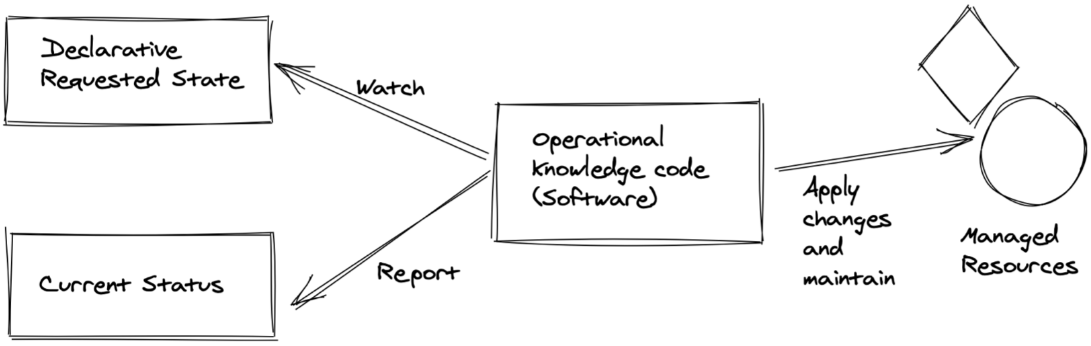

Figure 1.6 Operator pattern [35]

Like human operators, Kubernetes operators can have a level of manual skill ranging from basic software installation and setup skills to a high level where they can scale software vertically or horizontally to automatically change the configuration or detect abnormalities. All operator maturity levels are depicted in the figure 1.7. The highest level can only be reached by programming the operator in the GO programming language or by using the Ansible automation tool. [40]

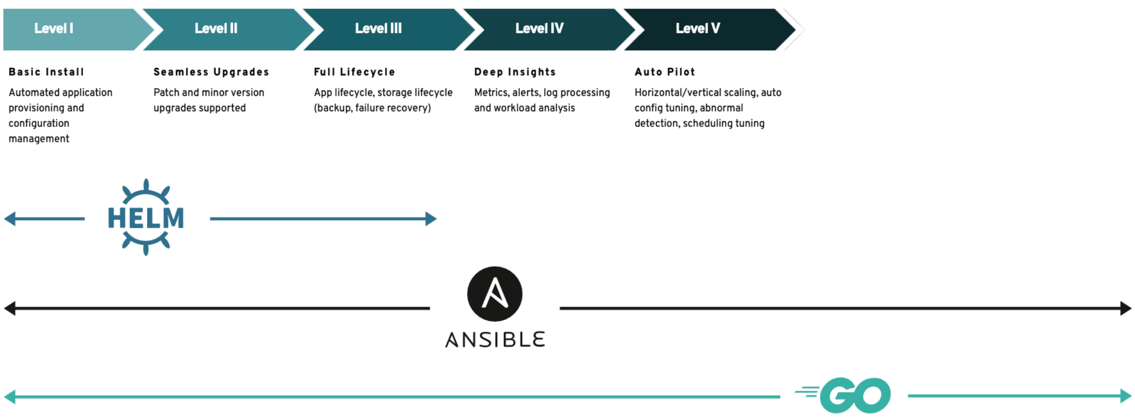

Figure 1.7 Operator maturity levels described by Operator Framework [41]

As stated in the Operator white paper, [35] the operator should be able to cover the complete life cycle of the managed resource as defined in the previous chapter without the need for external installation or upgrade intervention. Specifically as follows:

- Install or take ownership of the controlled application.
- Upgrade the managed application, including the monitoring of the upgrade process. It should also be able to roll back in case of failure. He should record the status of the upgrade.
- Back up the managed application and log when the application was last backed up and the status of that backup.
- Restore the application from the backup.
- Provide monitoring of the managed application.
- Scale the application.
- Automatically adapt the configuration of the application.
- Uninstall or disconnect from the application.

These are all capabilities that an operator should have at the highest level No. 5 - Autopilot. For lifecycle management described in previous chapter, the minimum level of operator capabilities must be at least level No. 4 - Deep Insights with an option to scale.

The Kubernetes cluster is divided into individual namespaces that separate the objects and names in the cluster and can have constraints applied to them. This partitioning makes it easier to share the cluster between users or entire teams. The object name must be unique within a namespace, but not between namespaces. An operator usually operates in its own namespace so it has a Namespace Scope, but it can also operate in the whole cluster in which case it will be a Cluster Scope operator. Namespace Scope operators are more flexible and easier to upgrade due to their independence from the rest of the cluster. Operator rights are further restricted by the so-called Role-Based Acceess Control (RBAC), which grants the rights assigned to the operator. [32]

The following options are advised by the Operator white paper [35] in case the operator is to be used for controlling the resource:

- Consultation with the creator of the resource to be controlled about the possibilities of using the operator.
- The search for public operator registries that provide a platform for publishing operators and the underlying documentation.

- The creation of own operator.

2 THESIS OBJECTIVE

The objective of this thesis is to conduct an extensive evaluation of various Kubernetes operators available for Postgres lifecycle management.

The ultimate aim is to provide clear and comprehensive recommendations on which operators are best suited to meet different stakeholder needs and preferences. This thesis intends to serve as a valuable resource that can guide stakeholders in making informed decisions about choosing the right operator for their specific operational context and requirements.

Furthermore, it aims to contribute to the broader knowledge base about Kubernetes operators and their application in managing Postgres databases in a cloud-native environment.

3 RESOURCE QUESTIONS

Having defined the thesis objective and established a thorough understanding of PostgreSQL, Kubernetes, the PostgreSQL lifecycle, and operators in previous chapters, we are now in a position to formulate research questions. These questions are designed to enable a deeper exploration into the complexities of managing Postgres in a Kubernetes environment via operators.

1. What operators exist for lifecycle management of Postgres in Kubernetes?
2. What metrics are suitable for comparing Opereators for lifecycle management in Kubernetes?
3. What approach should be taken to determine the degree to which the metrics are met?
4. How do the operators perform when evaluated according to the chosen metrics?

4 OPERATORS FOR LIFECYCLE MANAGEMENT IN KUBERNETES

This chapter aims to answer the first research question: 'What operators exist for lifecycle management of Postgres in Kubernetes?'" As recommended in chapter 1.5 the selection of the operator should first be consulted with the manufacturer of the controlled source. Postgres offers the following Kubernetes operators in its software catalog [42]: CloudNativePG, EDB Postgres for Kubernetes a Kubegres.

The next step involved a search of the operators' registers. In particular the Operator Hub. [43] Operator Hub presents nine operators with varying levels of capabilities, including Crunchy Postgres for Kubernetes by Crunchy Data, EDB Postgres for Kubernetes by EnterpriseDB Corporation, Ext Postgres Operator by movetokube.com, Percona Operator for PostgreSQL by Percona, Postgres-Operator by Zalando SE, Postgresql Operator by Openlabs, PostgreSQL Operator by Dev4Ddevs.com and StackGres by OnGres.

A deeper internet search revealed Stolon operator. [44]

Of the thirteen operators available, only five meet the minimum capability requirement of Deep Insight, namely: Crunchy Postgres for Kubernetes, EDB Postgres for Kubernetes, Percona operator for PostgreSQL, CloudNativePG operator, and StackGres operator. As a result, only these five will be subjected to deeper research, testing, and evaluation.

4.1 Crunchy Postgres for Kubernetes

Crunchy Postgres for Kubernetes (PGO) is a Postgres operator provided by Crunchy Data, which offers a declarative solution for the management of PostgreSQL clusters, with a focus on automation. Crunchy Data is a company that specializes in providing open-source software solutions for Postgres. The company also provides a range of support, consulting, and training services to help organizations implement and optimize their Postgres deployment. [45]

PGO's capabilities are the following:

- **Postgres Cluster Provisioning:** PGO is able to create [46], update [47] or delete Postgres cluster [48]
- **High Availability:** High availability is achieved by adding additional nodes. PGO uses a synchronous replication technique with Primary and Standby architecture. [49]
- **Postgres updates:** PGO is able to apply minor patches [50], and major upgrades since version 5.1. [51]
- **Backups:** PGO backup capabilities features: automatic backup schedules, backup to multiple locations, backup to cloud providers (AWS S3, Google Cloud Storage, Azure Blob), ad hoc backups, backup compression, and backup encryption. [52]
- **Disaster Recovery:** PGO is capable of Point In Time recovery, in place Point in Time Recovery, restore of an individual database. [53]
- **Cloning:** PGO is able to clone cluster. [53]
- **Monitoring:** Monitoring is provided by Prometheus, Grafana, and Alertmanager. [54]
- **Connection Pooling:** PgBouncer connection pooler from Postgres is part of PGO. [55]
- **Customization:** PGO provides a wide area of Postgres customization. [56]

PGO consists of the following key components [57]:

- High Availability: Patroni

- Backups: PgBackRest
- Connection Pooler: PgBouncer
- Monitoring: PgMonitor, Prometheus, Grafana, and Alertmanager

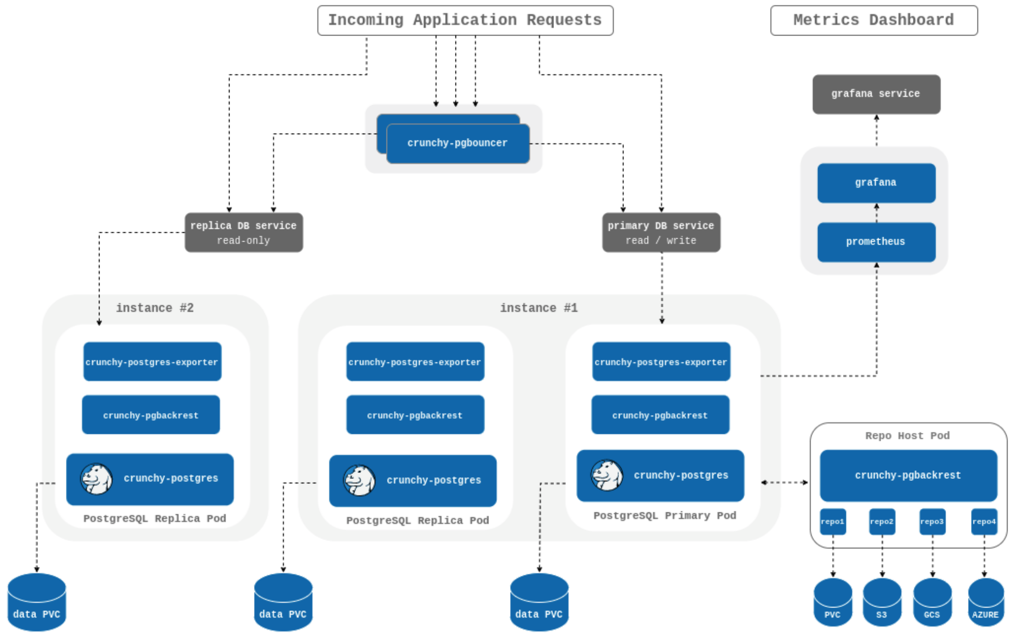

Figure 4.1 PGO's architecture [58]

The current stable version of PGO is 5.3.1 was released on 17th February 2023. [59]

PGO is distributed under the Apache License 2.0, an open-source license that allows for both commercial and non-commercial use. With regards to capability, PGO is considered to have the highest capability level, labeled as Autopilot. [60]

4.2 EDB Postgres for Kubernetes

The EDB Postgres for Kubernetes (EDBO) is a fully supported operator that has been designed, developed, and maintained by EnterpriseDB Corporation. It provides comprehensive coverage of the entire lifecycle of highly available PostgreSQL database clusters with a Primary/Standby architecture, utilizing native streaming replication. The operator is based on the open-source CloudNativePG operator and offers additional benefits. [61]

EDBO is distributed under the EDB Limited Usage License Agreement, a proprietary license that is specific to software provided by EnterpriseDB Corporation. A license key is always required for the operator to work longer than 30 days. [62] Due to the restrictive nature of the license EDBO will no longer be subject to testing and evaluation but its key component CloudNativePG will.

4.3 CloudNativePG

The CloudNativePG operator (CNPGO) is an operator that is available as an open-source solution and aims to manage PostgreSQL workloads across various Kubernetes clusters running in private, public, hybrid, or multi-cloud environments. The operator aligns with DevOps principles and concepts like immutable infrastructure and declarative configuration. [63]

Initially developed by EDB, CNPGO was later made available to the public as an open-source software under the Apache License 2.0. In April 2022, the project was submitted to CNCF Sandbox for further development and community engagement. [63]

CNPGO's capabilities are the following:

- **Postgres Cluster Provisioning:** CNPGO is able to create, update or delete Postgres cluster. [64]
- **High Availability:** High availability is achieved by adding additional nodes. PGO uses a synchronous replication technique with Primary and Standby architecture. [65]
- **Direct database imports:** CNPGO provides direct database import from remote Postgres server by using pg_dump and pg_restore even on different Postgres versions. [66]
- **Postgres updates:** CNPGO is able to apply minor patches. [67] Major updates are possible by Direct database imports[2)].
- **Backups:** CNPGO backup capabilities features: automatic backup schedules, backup to multiple locations, backup to cloud providers (AWS S3, Google Cloud Storage, Azure Blob), on-demand backups, and backup encryption [68][69]. Due to EDB's backup software Barman backup compression is available also. [68]

[2)] Due to its nature Direct database imports cannot be considered as major upgrade option.

- **Disaster Recovery:** CNPGO is capable of Point In Time recovery. [68]
- **Cloning:** CNPGO is able to create cluster replicas. [65]
- **Monitoring:** Monitoring can be provided by the additional installation of Prometheus, and Grafana, and Alertmanager. [70]
- **Connection Pooling:** Provided by native Postgres pooler PgBouncer. [71]
- **Customization:** CNPGO provides a wide area of Postgres customization such as max parallel workers tuning or WAL configuration [72]

CNPGO consists of the following key components [73] [70]:

- High Availability: Postgres instance manager
- Backups: Barman
- Connection Pooler: PgBouncer
- Monitoring: Prometheus, Grafana, and Alertmanager

The current major stable version of CNPGO is 1.20.0 was released on 27th April 2023. [74] CNPGO is distributed under the Apache License 2.0 open-source license. CNPGO is considered to have the highest capability level, labeled as Autopilot. [63]

4.4 StackGres operator

StackGres (SPGO) is a comprehensive distribution of Postgres for Kubernetes, delivered in a user-friendly deployment package. The distribution includes a set of Postgres components that have been carefully selected and optimized to work seamlessly with each other. [75]

SPGO is developed by OnGres that was established as a result of years of experience in working with and creating products based on Postgres and supporting clients with their Postgres infrastructures. Postgres databases are at the heart of the company's business, as the name suggests. [76]

SPGO's capabilities are the following [76]:

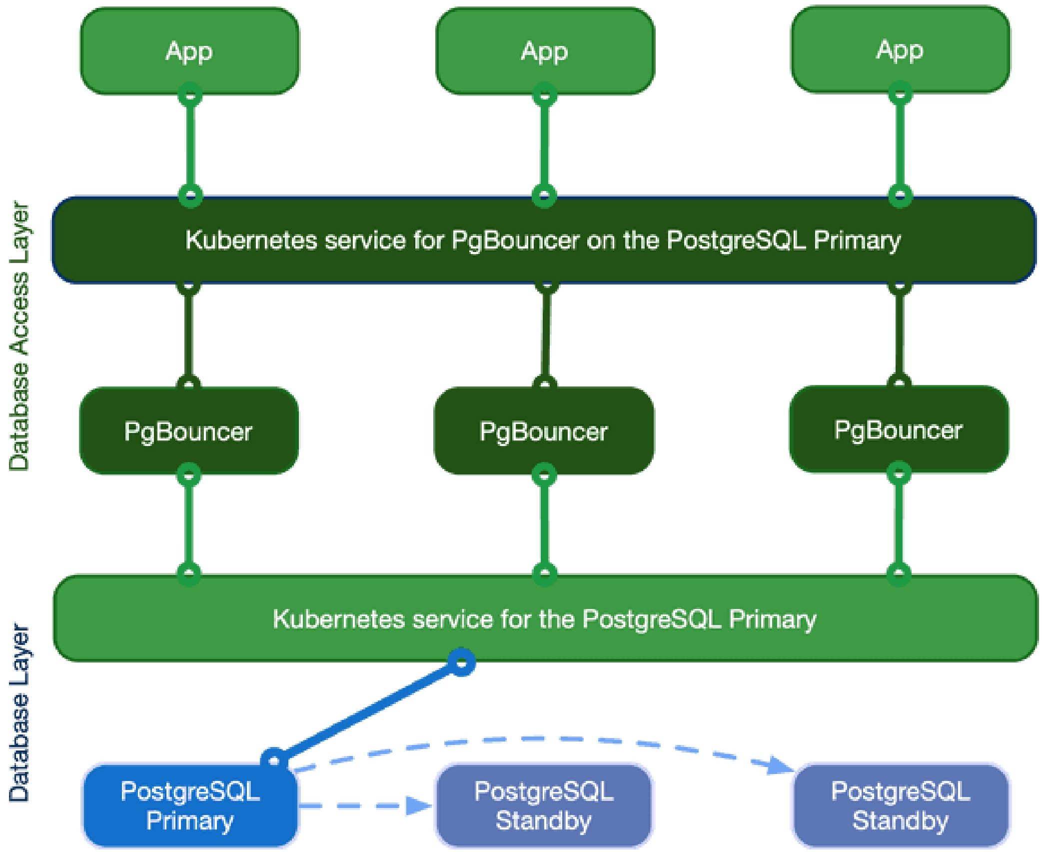

Figure 4.2 CNPGO's architecture [71]

- **Postgres Cluster Provisioning:** SPGO is able to create, update or delete Postgres cluster.
- **High Availability:** High availability is achieved by adding additional nodes with Primary and Standby architecture.
- **Postgres updates:** SPGO is able to apply minor patches. Major updates are possible by SGDbOps [77].
- **Backups:** SPGO backup capabilities features: automatic backup schedules, backup to multiple locations, backup to cloud providers (AWS S3, Google Cloud Storage, Azure Blob)
- **Disaster Recovery:** SPGO is capable of Point In Time recovery.
- **Cloning:** SPGO is able to create cluster replicas.
- **Monitoring:** Monitoring is provided by Prometheus, Grafana, and Alertmanager.

- **Connection Pooling:** Is provided by PgBouncer.
- **Customization:** SPGO provides a wide area of Postgres customization such as WAL configuration, archive mode, vacuum, etc. [78]
- **Mamagement Console:** SPGO provides a fully featured management web console.

SPGO consists of the following key components [73]:

- High Availability: Patroni
- Backups: WAL-G
- Connection Pooler: PgBouncer
- Monitoring: Prometheus, Grafana, and Alertmanager.

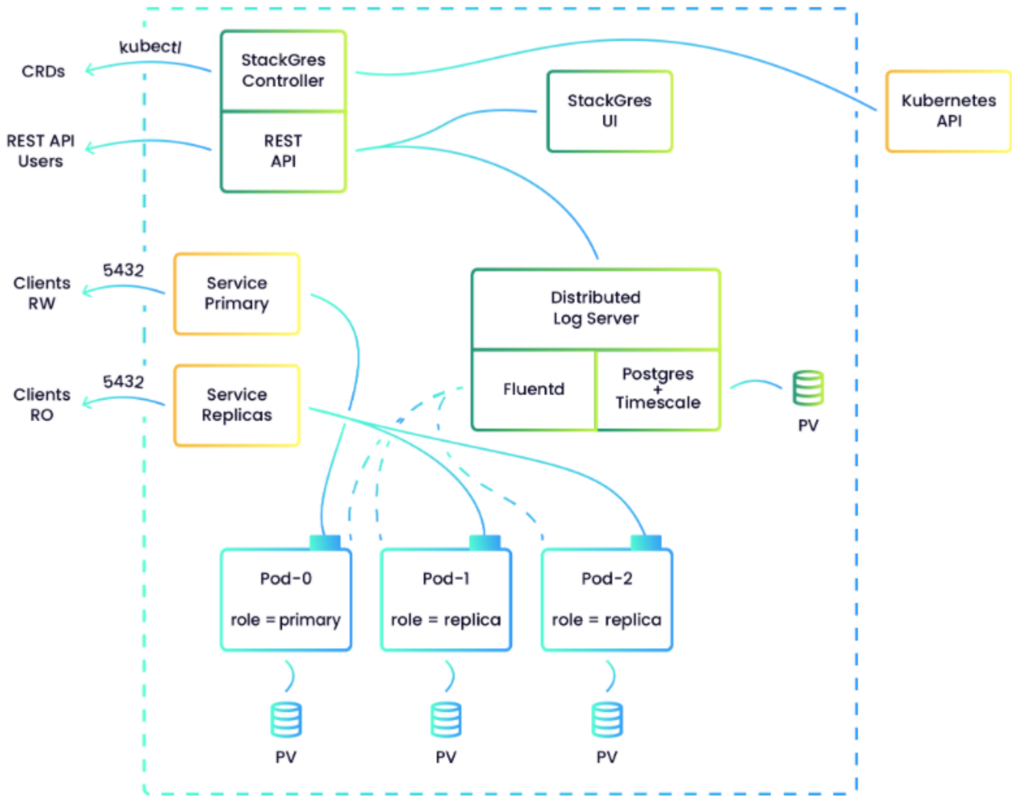

Figure 4.3 SPGO's architecture [79]

The current stable version of SPGO is 1.4.3[3)] was released on 24th January 2022. [80] SPGO is distributed under the AGPL3 open-source license. [81] With regards to capability, SPGO is considered to have the second highest capability level, labeled as Deep Insights. [82]

4.5 Percona operator for PostgreSQL

Percona is a company that provides services and solutions for open-source database technologies. It offers expertise, support, and software for MySQL, MongoDB, and PostgreSQL. The company's offerings help organizations manage their open-source databases and ensure optimal performance, security, and scalability. [83]

Percona operator for PostgreSQL (PPO) is based on Crunchy Postgres for Kubernetes. Percona forked PGO v 4.7 and has added enhancements for monitoring, upgradability, and flexibility. [84]

Differences between PGO and PPO are the following:

- **Postgres updates:** PPO provides automatic Postgres updates for minor and major versions of Postgres. [85]
- **Backups:** PPO is not able to back up to Azure. [86] Although it uses Patroni, which has this ability.
- **Disaster Recovery:** PPO documentation does not mention the possibility of restoring a single database from a backup. [87]
- **Monitoring:** PPO is not using the usual monitoring stack consisting of Prometheus and Grafana but their own Percona Monitoring and Management. [88]

The current stable version of PPO is 1.4.0 was released on 31st March 2023[4)]. [89] PPO is distributed under the Apache License 2.0, an open-source license that allows for both commercial and non-commercial use. With regards to capability, PPO is considered to have the second highest capability level, labeled as Deep Insights. [90]

[3)] Version 1.5.0 has also been released, it is not yet production-ready yet. Therefore will not be tested or evaluated.

[4)] Version 2.0.0 has also been released, it is not yet production-ready yet. Therefore will not be tested or evaluated.

4.6 Summary

Table 4.1 Summary of selected 0perators

operator	Maturity level	Current production version	Release date
PGO	Autopilot	5.3.1	17th February 2023
CNPGO	Autopilot	1.20.0	27th April 2023
SPGO	Deep Insights	1.4.3	23th February 2022
PPO	Deep Insights	1.4.0	31st March 2023

4.7 Key differences

Table 4.2 Key differences between selected operators

Feature	PGO	CNPGO	SPGO	PPO
In place Point in time recovery	Yes	No	No	No
Individual database restore	Yes	No	No	No
Operator user interface	No	No	Yes	No
Major version upgrade	Yes	No	Yes	Yes
Supported Postgres versions	v11 - v15	v11 - v15	v12 - v15	v12 - v14

5 METRICS

This chapter aims to answer the second research question: 'What metrics are suitable for comparing Opereators for lifecycle management in Kubernetes?'" According to Tom Gilb [91], the main issue in software attribute requirements is identified not in their functionality, but in their quality. Gilb differentiates these attribute requirements into two categories: Resources (people, time, money), which are always finite, and qualities or benefits, which are always fewer than desired. Knowledge about the functionality that an operator must provide to achieve a certain level of capabilities is obtained from Chapter 1.5. The most significant functional properties of operators have been detailed in Chapter 4. With the lifecycle of Postgres and the capabilities of operators now understood, what remains to be examined are their qualitative properties. The upcoming testing will be focused on the proposed qualitative metrics.

5.1 Performance

Performance is a qualitative parameter of a system, defined by the efficiency with which the system utilizes allocated resources. In the case of Postgres, performance can be expressed as the number of transactions executed per unit of time. A higher transaction rate is indicative of superior performance.

5.2 Reliability

Reliability is a qualitative parameter that determines the degree of system dependability. If a system cannot be relied upon, it cannot be utilized effectively. Therefore, reliability is a critical parameter of any system.

5.3 Usability

Usability is a key aspect of software design that focuses on user experience. It refers to the ability of a system to achieve certain goals for certain users in a particular context of use.

5.4 Maintenance

Activities that are performed after the software is deployed to ensure its correct functionality and performance. Maintenance may include bug fixes, adding new features, performance optimization, updates for compatibility with new systems, etc. Ignoring software maintenance can lead to increased repair costs and reduced system performance over time. Additionally, it can cause system instability, increased vulnerability to security threats, and eventually, potential system failure.

5.5 Security

Security refers to the measures, practices, and technologies employed to protect the system and its data from threats and attacks.

6 TESTING METHODOLOGY

This chapter aims to answer the third research question: 'What approach should be taken to determine the degree to which the metrics are met?'" and presents a high-level overview of the testing methodology. The goal of the methodology is to deliver rules and guidance for test process that produces test reports forming the basis of this evaluation.

6.1 Notice

It is important to notice at the beginning of this chapter that testing as described in [92] has following seven testing principles:

1. Testing shows the presence of defects, not their absence
2. Exhausting testing is impossible
3. Early testing saves time and money
4. Defects clusters together
5. Beware of pesticide paradox
6. Testing is context dependent
7. Absence of errors is fallacy

Therefore, the test process derived from this methodology as every test process will not exhaustively test the operators and will be depended on thesis context, author bias, and author skills. Because The objective of this thesis is to conduct an extensive evaluation of various Kubernetes Operators available for Postgres lifecycle management the main scope of this methodology is to deliver test process that will produce test reports that will form the base for this evaluation.

6.2 Criteria

Some criteria are well measurable on their own, while others require further breakdown into multiple sub-criteria in order to obtain data. The criteria have been divided according to Figure 6.1.

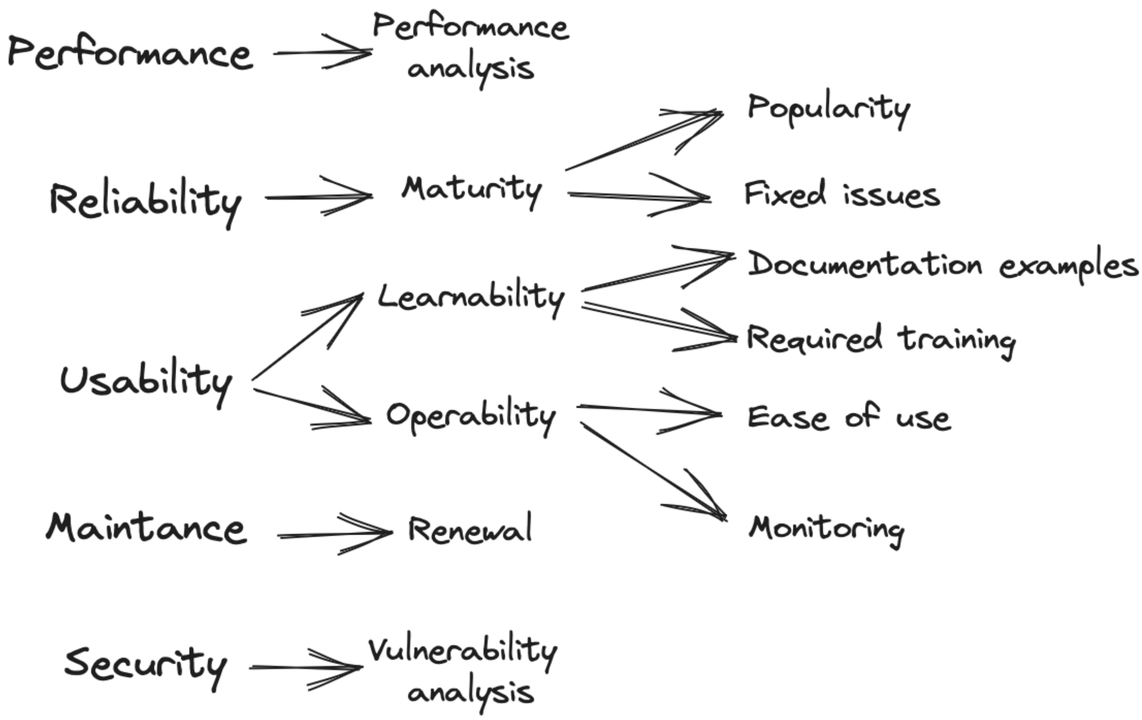

Figure 6.1 Criteria division

To keep track of these criteria, each one has been assigned an ID. The list of identified criteria is as follows:

- CP: Performance
  - CP1: Performance analysis
- CR: Reliability
  - CR1: Maturity
    - CR1A: Popularity
    - CR1B: Fixed issues
- CU: Usability
  - CU1: Learnability
    - CU1A: Documentation examples
    - CU1B: Required training
  - CU2: Operability
    - CU2A: Ease of use
    - CU2B: Monitoring

- CM: Maintenance
  - CM1: Renewal
- CS: Security
  - CS1: Vulnerability analysis

6.2.1 Performance testing

Performance testing must be conducted using performance analysis. To test the performance, an environment must be set up in which the deployed operator creates a high-availability (HA) Postgres cluster consisting of three nodes. The test environment must be recreated for each operator being evaluated. Additionally, an equal number of pooler nodes must be created. It is essential that the settings of all Postgres clusters are the same, and only the HA service cluster must be tested. The performance tests must be carried out at least three times to ensure accurate and reliable results.

6.2.2 Reliability testing

Reliability must be tested through operator maturity testing.

Maturity testing Determining the maturity of operators must be done by assessing their popularity. Popularity can serve as an indicator of maturity because widespread recognition and usage of a product often imply reliability, efficiency, and the ability to meet user requirements. Popularity will be determined by considering the number of stars in the operators' repositories.

Furthermore, the ratio of the number of open issues to the number of resolved issues will be examined. This data will also be gathered from the operators' repositories.

By considering both popularity and the ratio of resolved issues, valuable insights can be gained regarding the maturity level of the operators.

6.2.3 Usability testing

Usability must be tested through operator learnability testing and operability testing.

Learnability testing To determine the learnability of operators, a list of Postgres lifecycle events must be created, and the operator documentation must be examined to ensure that it provides examples for each Postgres lifecycle event. Throughout this process, the number of tools required to successfully use the operator must be recorded to determine the level of training needed.

Learnability testing, which is a crucial aspect of usability testing, must be conducted to assess the ease with which new users can understand and use the operator. The testing aims to achieve two primary objectives. Firstly, it must identify the presence of examples in the documentation, as learning facilitated by examples is generally more effective. Secondly, it must gauge the level of learning required to operate the system effectively.

By conducting thorough learnability testing, valuable insights can be obtained regarding the user-friendliness and accessibility of the operators, contributing to the overall evaluation of their usability.

Operability testing To test the operability of the operator, the testing methodology involves creating use cases derived from the Postgres lifecycle, developing corresponding test cases for these use cases, and determining the number of commands necessary to accomplish the desired functionality. Additionally, a comprehensive list of required monitoring items must be compiled, and the test case incorporating monitoring activities assesses whether the operator's monitoring adequately encompasses these items. This comprehensive approach ensures that the operability of the operator is thoroughly assessed, taking into account its functionality, and monitoring capabilities.

6.2.4 Maintenance testing

Maintenance testing must be performed by measuring the number of commits in the operator's repository. The ratio of these commits to the number of days since the repository was created must be determined. This approach is necessary to calculate the number of commits per unit of time, which is crucial in assessing the rate at which the software is being renewed.

6.2.5 Security testing

Security testing must be conducted by analyzing the vulnerability of the docker image for each individual operator. As the overall cluster comprises multiple docker containers with various dependencies such as Postgres, pooling, backups, etc., to streamline the testing process, only the operator image will be subjected to vulnerability analysis. This approach allows focusing on the specific security aspects of the operator without testing the dependencies separately.

6.3 Test management process

According to the IEEE Standard for Software Test Documentation [93], test management processes have three main test processes: test strategy and planning, test monitoring and control, and test completion. As depicted in Figure 6.2 testing has more than one management test process. The main process is the Organizational process which is further divided into Test management processes that are then devided into Dynamic test processes.

Thesis test process will consist of one management process that will create two managed subprocesses, one for static testing and one for dynamic testing. This test management process will monitor and control subprocesses. Subprocesses will deliver all their deliverables to this main process.

The main management consist of following tasks:

- High-level test strategy
- High-level test planning

Subprocess consist of following tasks:

- Test strategy
- Test planning

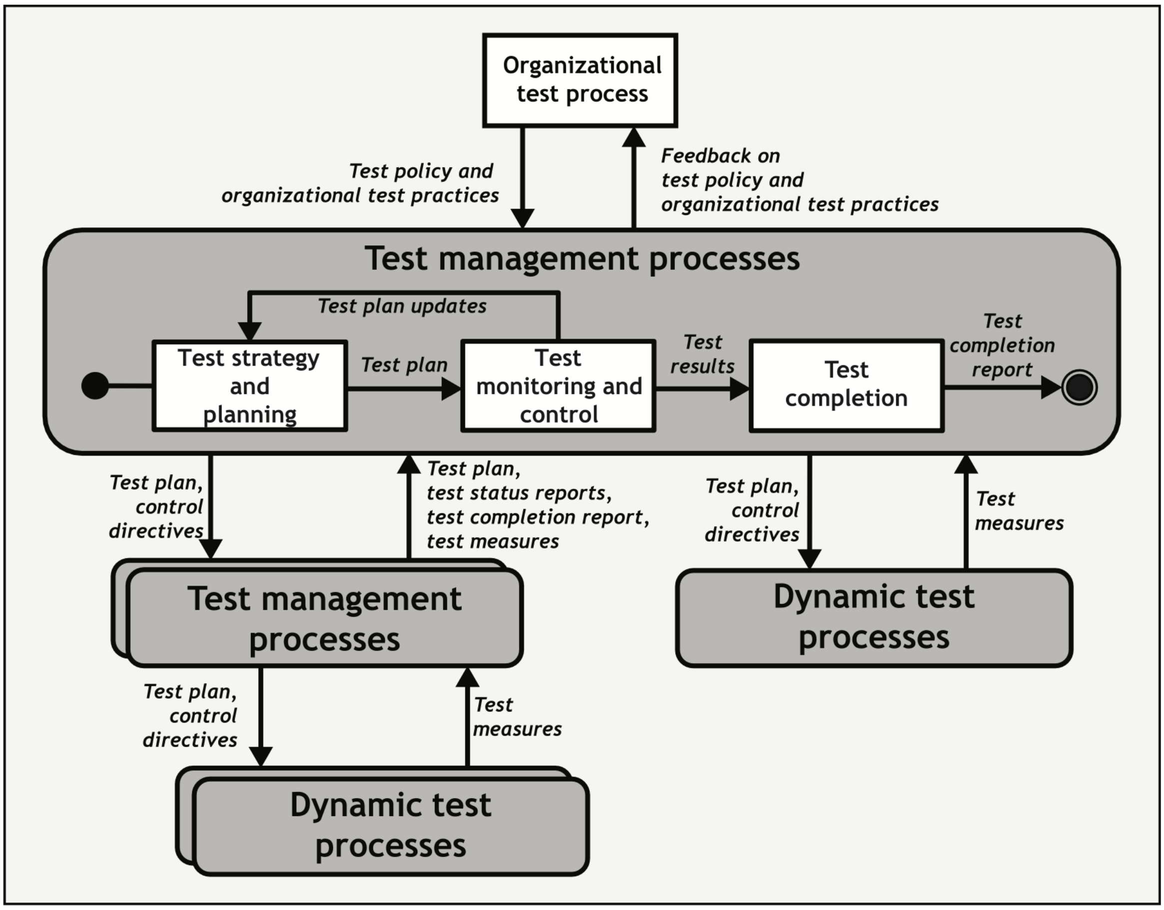

Figure 6.2 Test management process relationships [93]

6.4 Test strategy and planning

The output of the test strategy and planing will be the test plan, as the basis for its creation will be the requirements created earlier. Details about activities are described by ISO/IEC/IEE 29119-3.

As depicted on 6.2 test plan is not static but it changes according to monitoring.

6.5 Test plan

In order to create a test plan, IEEE proposes the following procedure shown in the figure 6.3 with the idea that some activities can be repeated.

The result of a properly designed test plan should be:

- Scope of testing

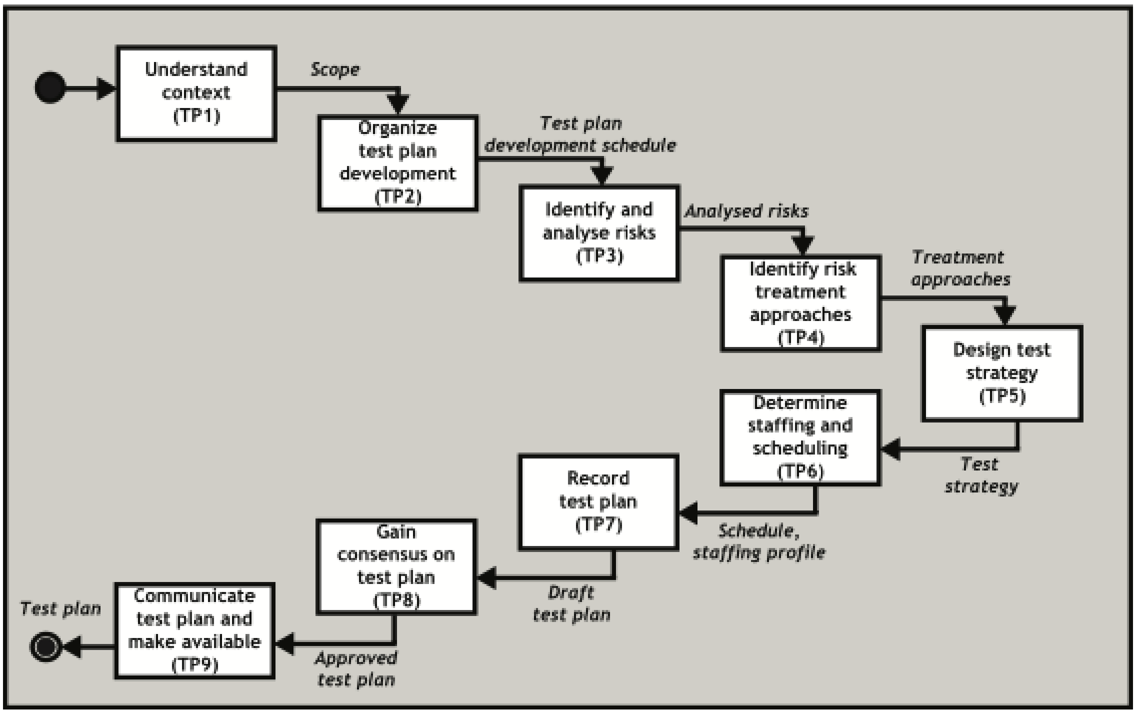

Figure 6.3 Test plan creation activities [93]

- List of identified risks
- Testing strategy
- Test environment
- Test tools
- Test data
- Staffing
- Scheduling
- Required training
- Estimates of time and resources
- Compliance with all stakeholders

6.6 Test monitoring and control process

The role of the test monitoring and control process is to observe the test process and detect deviations from the plan. This process controls the test process throughout its duration. The findings are then used to modify the test plan.

To avoid unnecessary bureaucracy, where the manager and tester are the same person, and consequently the testing progress would be reported to the person who is also filing it, there will be no status reports during this process.

6.7 Test completion process

The test completion process as depicted in the figure 6.4 will be used in testing after each test the test competition report will be created and delivered to a higher level.

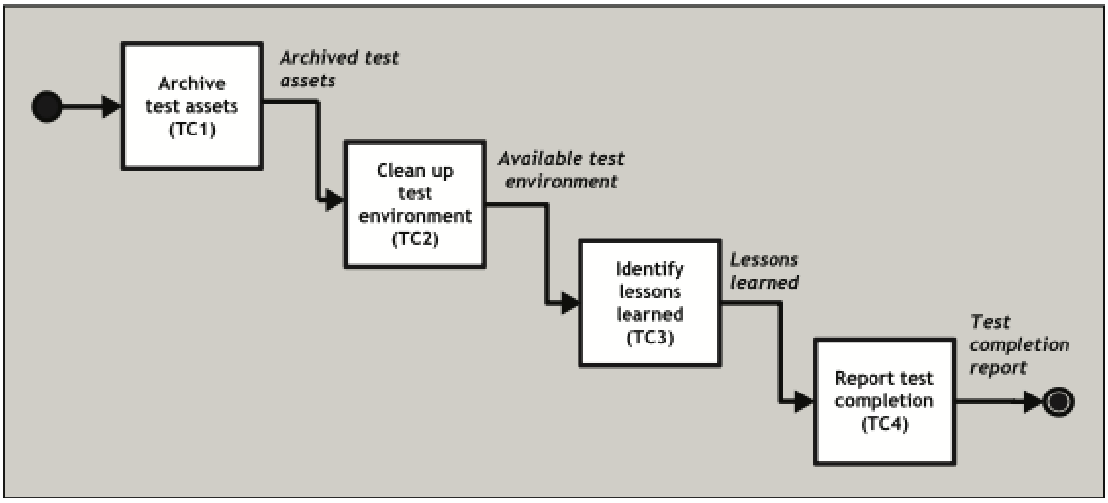

Figure 6.4 Test completion process [93]

6.8 Dynamic and static test processes

According to ISTQB [92] there are two types of tests. Static and dynamic. The main difference is that the static technique does not execute the tested software, but the dynamic does. The testing process will utilize both techniques using the dynamic testing process depicted in figure 6.5.

6.9 Test design and implementation processes

Test design and implementation process must follow the process depicted in the figure 6.6. Test design techniques should be used to derive test cases. Test cases must be traceable to requirements and must meet the ISO/IEC/IEE 29119-3 requirements. This process can be reentered multiple times and must meet the completion criteria specified in the test plan.

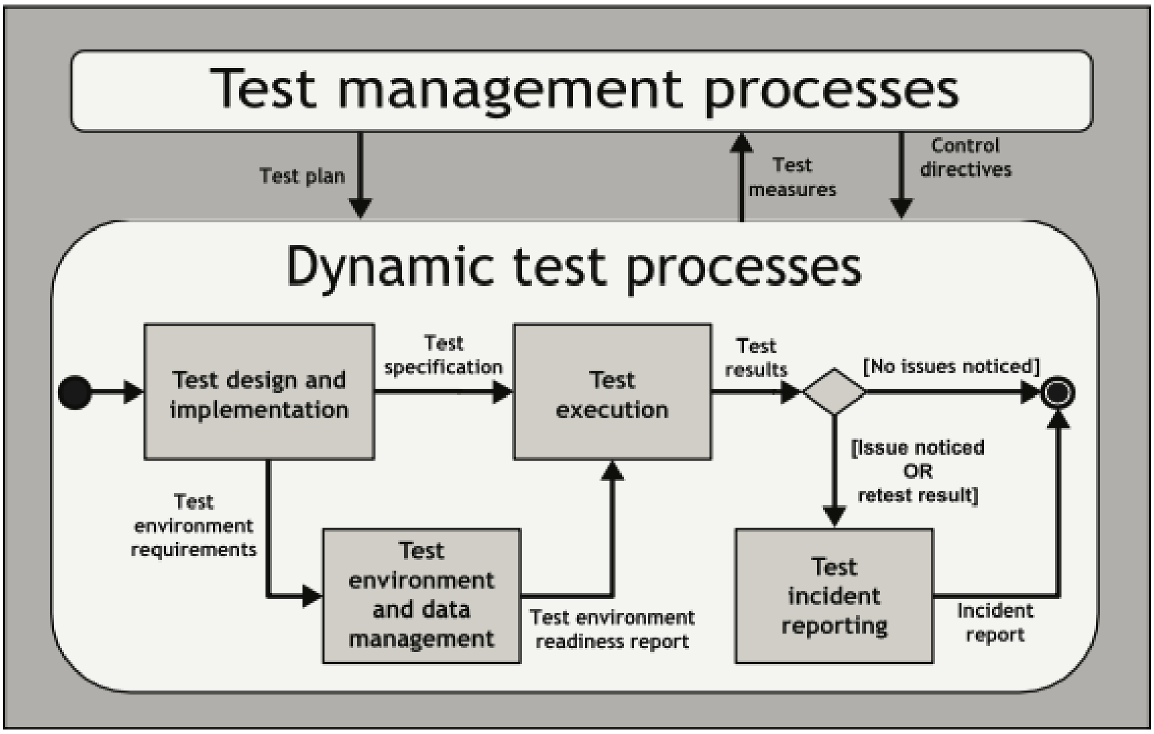

Figure 6.5 Dynamic test processes [93]

6.10 Test environment and data management processes

Based on the test plan all the environments must be established and well mainteained.

6.11 Test execution process

Test execution process depicted in 6.7 must be followed. After the test execution, the execution log must be delivered. Details about activities are described by ISO/IEC/IEE 29119-3.

6.12 Test incident report process

The process for reporting test incidents, as depicted in Figure 6.7, must be followed. It is important to note that the purpose of testing is not to simply find incidents, but rather to test the capabilities of the system. As such, the term "finding" will be used in cases where it is more appropriate.

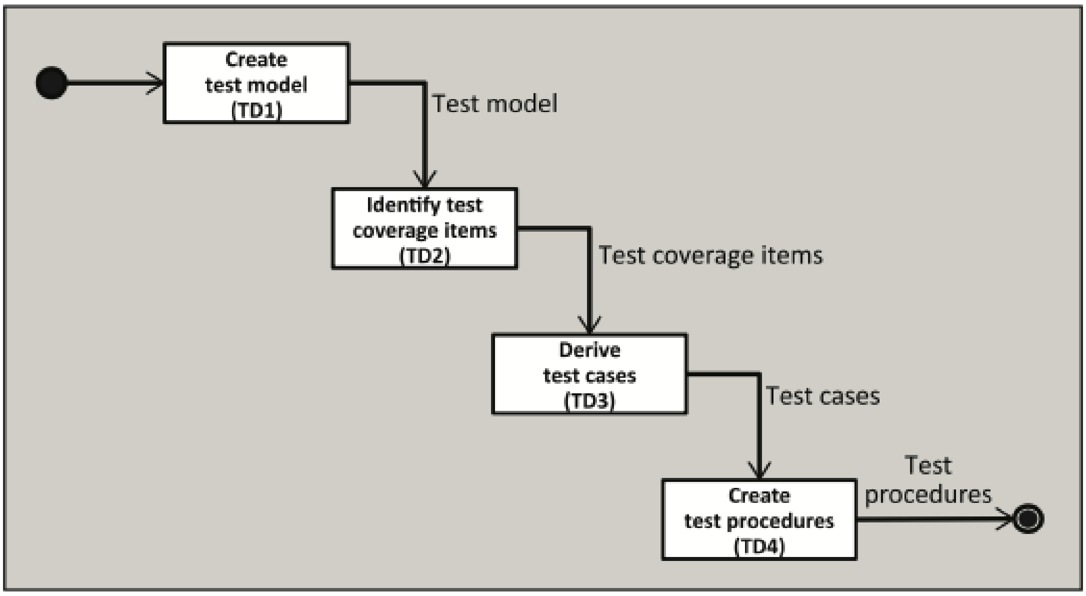

Figure 6.6 Test design and implementation [93]

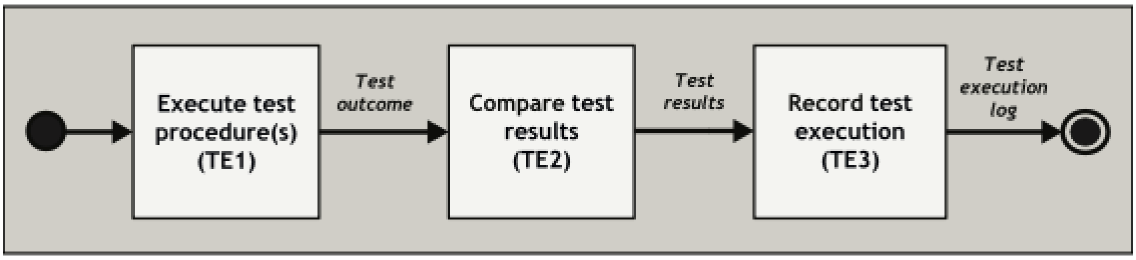

Figure 6.7 Test execution process [93]

6.13 Level of detail

Please note that this chapter provides only a high-level overview of the testing methodology. More detailed information can be found in ISO/IEC/IEEE 29119-2[5)]. If there are any doubts regarding the testing process, this standard should be used as a guide to ensure that all necessary aspects of testing are properly addressed.

5) https://standards.ieee.org/ieee/29119-2/7498/

II. APPLICATION OF THEORY

7 TEST PROCESS

As stated in the methodology, the main test process with a general test plan was created. The general test plan can be seen in Appendix AI. This process was divided into two subprocesses: one for static testing and one for dynamic testing.

According to the priorities stated in the general test plan, the first test process to be conducted was the static test process.

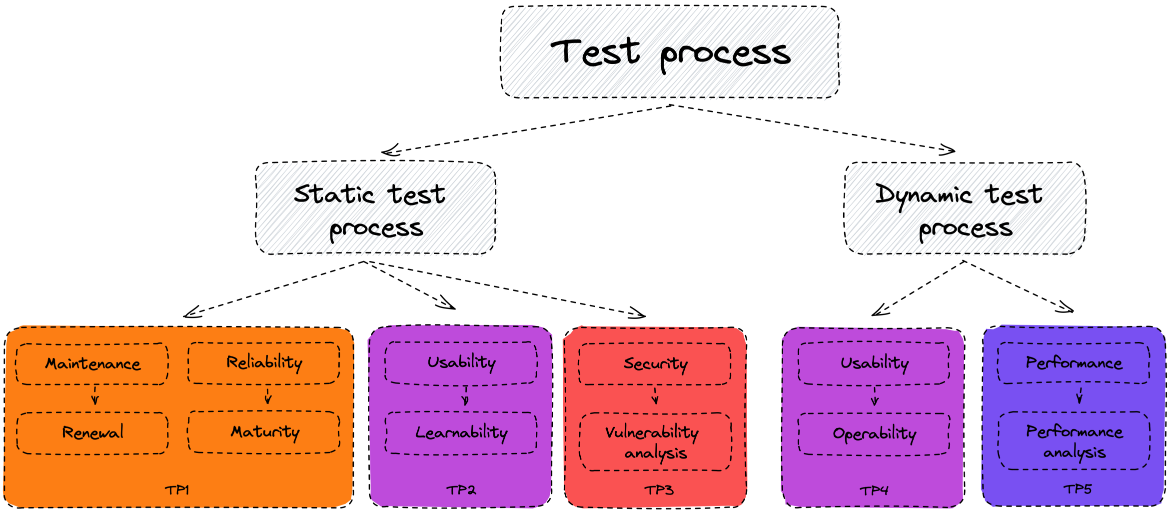

Figure 7.1 Test process

7.1 Tools

1. **Bash**: Most of the test cases during the testing process are scripted using the Bourne Again Shell (Bash).
2. **Git**: One of the test procedures is employing Git for repository management and version control.
3. **Kubectl**: The Kubectl tool is employed for managing the Kubernetes cluster.
4. **Kustomize**: Tool for customizing Kubernetes configurations without changing base resources.
5. **Terraform**: Terraform is leveraged to simplify the deployment and replication of the GKE cluster.
6. **Trivy**: The Trivy security scanner is utilized for vulnerability analysis of the operators' images.
7. **Snyk**: Snyk is also utilized for vulnerability analysis of the operator's images.

7.2 Static test process

The static test process was divided into three separate subprocesses, each with its own test plan.

7.2.1 Reliability and maintenance

Reliability and maintenance testing consists of two parts: renewal and maturity. Both of these parts form the first static testing process because they share the same test items and therefore dividing them into separate processes doesn't make sense. This process was designed to provide the necessary information for decision-making regarding the maintenance quality and maturity level of the operators. A detailed test plan and further specifics of this process can be found in Appendix A II.

During the test process, it was observed that PPO does not use the repository to track issues, instead it uses Jira. Another challenge arose when trying to determine the repository creation date and getting the number of commits in repository. Retrieving this data from Gitlab or Github proved to be quite difficult. As a workaround, all the repositories were cloned, and the date of the initial commit was obtained using the command mentioned in Listing 2. Likewise, the total number of commits was determined using the command mentioned in Listing 1. Due to these modifications, the test plan was subsequently revised.

Listing 1 Commits count

```
$ git rev-list --all -count
```

Listing 2 Reverse git log

```
$ git log --reverse
```

The results of this repository analysis are presented in Table 7.1. As can be seen, PGO and PPO share the same creation date. This is due to the fact that PPO is a fork of PGO, as metioned earlier in Chapter 4.5.

7.2.2 Usability: Learnability

The learnability testing process was divided into two distinct aspects: the first one being the training required to operate with the operators, and the second one being

Table 7.1 Operator repository analysis

	PGO	CNPGO	SPGO	PPO
Repo creation	27th Feb 2017	18th Feb 2020	29th May 2019	27th Feb 2017
Test date	1st May 2023	1st May 2023	1st May 2023	1st May 2023
Stars	3258	1198	84	149
Issues	1884	764	1959	317
Issues fixed	1755	691	1514	282
Commits	5582	3362	7208	4689

the presence of examples concerning Postgres life cycle events in the operator's documentation. Following the guidelines of the test plan (available in Appendix A III), a checklist was created. The documentation for each operator was thoroughly reviewed, resulting in the findings presented in Tables 7.2 and 7.3.

Table 7.2 Training needed

	PGO	CNPGO	SPGO	PPO
1st training	Kubectl	Kubectl	Kubectl	Kubectl
2nd training	Kustomize	Helm	Helm	Helm
3rd training		Cnpg		

7.2.3 Security

To proceed with the security testing, a vulnerability analysis test plan was created (which can be found in Appendix A IV), and the process was carried out according to this plan.

According to test plan, the test items for this process were identified, the test tool was downloaded, and a test procedure was developed which consisted of four test cases - one for each operator (details in Appendix A IV). Each operator was then tested according to this procedure, with the overall vulnerability results presented in Table 7.4.

During this test rocess the security scanner Trivy was unable to detect any vulnerabilities in CNPG's container image utilizing Debian 11.6. The rest of the operators are using the Red Hat 8.7 container image, which resulted in almost identical vulnerability scores. Unfortunately, the implementation of Red Hat 8.7 in both PGO and SPGO has

Table 7.3 Documentation examples

	PGO	CNPGO	SPGO	PPO
Cluster creation	Yes	Yes	Yes	Yes
Minor upgrade	Yes	Yes	No	Yes
Major upgrade	Yes	No	No	Yes
Backup	Yes	Yes	Yes	Yes
Restore	Yes	Yes	Yes	Yes
Monitoring	Yes	Yes	Yes	Yes
Vertical scaling	Yes	Yes	Yes	No
Horizontal scaling	Yes	Yes	Yes	Yes
Configuration Update	Yes	Yes	Yes	Yes
Uninstall	Yes	Yes	Yes	No

a high vulnerability with openssl-libs (CVE-2023-0286). More details about this vulnerability can be found here: `https://avd.aquasec.com/nvd/2023/cve-2023-0286/`.

The absence of any detected vulnerabilities in CNPGO by Trivy raised a question: Is Trivy accurately scanning this image? To address this concern, the test plan was updated to include an additional test case. This test involved scanning the base image of CNPGO to ascertain if Trivy could detect any vulnerabilities in Debian 11.6.

Debian image was scanned with the results presented in Table 7.4 which can be interpreted in two ways. The first interpretation suggests that CNPG might be using Debian but has effectively removed or mitigated the vulnerable parts. The second interpretation considers the possibility that Trivy may not be able to accurately identify Debian vulnerabilities within CNPGO.

To eliminate the second interpretation an additional vulnerability analysis tool, Snyk, was incorporated into the testing process. The test plan was subsequently adjusted, and the images were scanned using Snyk. This alternative method produced results similar to those from the Trivy scanning for each operator. However, there were exceptions in terms of High severity issues; Snyk identified three additional vulnerabilities in the openssl-libs of PGO and SPGO and provided different results for Debian.

Complete results of Trivy and Snyk scans can be located in the thesis repository[6)]

[6)] https://github.com/Ovec/Bachelors-thesis folder at `tests/vulnerability_analysis`. The overall results are presented in Table 7.4 and 7.5

Table 7.4 Trivy vulnerability analysis results

	PGO	CNPGO	SPGO	PPO	Debian 11.6
Critical	0	0	0	0	1
High	1	0	1	0	17
Medium	40	0	41	35	6
Low	36	0	36	36	59
Unkown	0	0	0	0	0

Table 7.5 Snyk vulnerability analysis results

	PGO	CNPGO	SPGO	PPO	Debian 11.6
Critical	0	0	0	0	0
High	4	0	4	0	1
Medium	38	0	39	36	2
Low	39	0	39	39	48
Unkown	0	0	0	0	0

7.3 Dynamic test process

The dynamic test process was divided into two separate subprocesses, each with its own test plan.

7.3.1 Environments

Two environments were used for the dynamic test process, each designed for specific test types based on resource requirements:

1. **Kind Kubernetes Cluster**: Kind is a tool that creates Kubernetes clusters within Docker, was used for less resource-intensive tests. This setup was run on a first-generation M1 MacBook Air equipped with 8GB of RAM and a 500GB disk.

2. **Google Kubernetes Engine (GKE)**: More resource-demanding tests, such as performance testing, were conducted on a robust Google Kubernetes Engine cluster, consisting of three e2-standard-2 nodes. Each node with 2 virtual CPUs and 8GB of RAM. Additionally, a standalone Postgres node with pgbench was deployed to this cluster for performance measurements.

The configurations for Kind and the Terraform plans used for deploying the GKE cluster, as well as the commands used for both deployments, can be found in the repository directory located at `tests/environments`.

7.3.2 Usability: Operability

The operability testing process was divided into two distinct aspects: the first one being ease of use, and the second one being the quality of provided monitoring. Following the guidelines of the test plan, use cases derived from Postgres lifecycle and testing procedure was created (all available in Appendix A V).

Ease of use Testing the ease of use involved testing each operator for the functionality outlined by the use cases and quantifying the number of commands required to implement these functions. This form of testing tested the functional aspects of the operator in conjunction with its ease of use. To achieve this, three types of shell scripts were utilized: before.sh, test.sh, and cleanup.sh.

The before.sh script prepared the environment for the test, the test.sh script executed the test itself, and the cleanup.sh script restored the environment to its state prior to the test. The precondition for testing the operators was a functioning Kubernetes cluster with kubectl configured for this cluster. Additionally, to perform tests on the Postgres clusters, it was necessary for the operator to be installed in the cluster.

In cases where it was necessary to verify functionality, this verification was made manually.

SPGO The reason why SPGO has a zero value in most test cases is due to its user-friendly interface. Most tasks can be accomplished directly within this interface, eliminating the need for using the terminal. Even complex tasks, such as performing a major version upgrade, can be easily carried out via this Graphical User Interface (GUI). An example of this user interface is shown in Figure 7.2. Additional images can be found in the repository folder at `doc/graphics/monitoring/SPGO`.

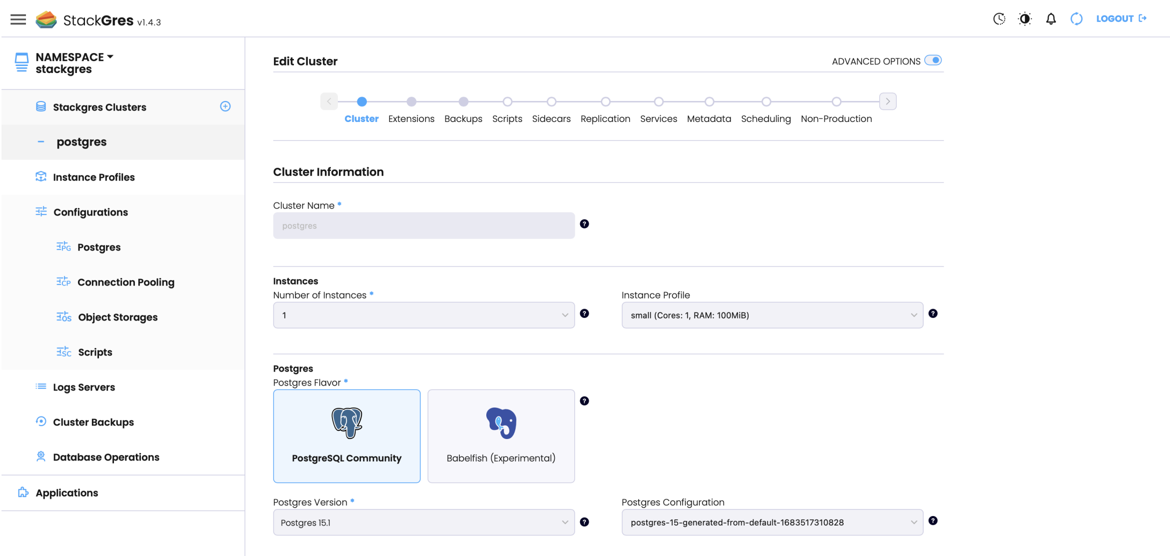

Figure 7.2 SPGO's user interface

Cluster major version upgrade The upgrade to a new major version appears to be the most challenging task for each operator. While SPGO handled the cluster upgrade seamlessly, PGO required four steps to proceed with the major version upgrade. On the other hand, CNPGO claimed to be capable of an "Offline import of existing PostgreSQL databases, including major upgrades of PostgreSQL". This process involves dumping the database and restoring it to a new cluster, which can be done with any cluster and is not considered a major upgrade. PPO declared in their documentation that they are capable of automatic updates even between versions. However, this Jira issue (`https://jira.percona.com/projects/K8SPG/issues/K8SPG-254?filter=allopenissues`) suggests otherwise.

PPO During the cluster update test case, specifically the update of max_wal_size, PPO experienced a failure. Despite updating the cluster configuration, the max_wal_size of Postgres remained unchanged. PPO also was the only operator that did not support the PostGis extension for PostgreSQL.

CNPGO During the testing procedure, it was noted that CNPGO didn't require the CNPG plugin, as initially mentioned in Chapter 7.2.2, to perform any operation. Consequently, the necessity for training related to this plugin was reevaluated, leading to its removal. The updated results are presented in Table 7.7.

Table 7.6 Ease of use

	PGO	CNPGO	SPGO	PPO
Operator installation	2	1	2	3
Cluster installation	1	2	0	1
Cluster monitoring	2	4	3	5
Cluster vertical scaling	1	1	0	1
Cluster horizontal scaling	1	1	0	1
Cluster connection pooling	1	1	0	0
Cluster extension install	1	1	0	-
Cluster number of connections increase	1	1	0	2
Cluster max_wall_size increase	1	1	0	-
Cluster scheduled backup	1	1	0	1
Cluster ad-hoc backup	2	1	0	1
Cluster restore	2	1	1	1
Cluster minor version upgrade	1	1	0	1
Cluster major version upgrade	4	-	0	-
Operator uninstall	1	1	1	1
Cluster uninstall	1	1	0	1

Monitoring During the monitoring deployment test case, screenshots of each monitoring system were taken. The results are presented in Table 7.8. These screenshots can be found in the repository folder at `doc/graphics/monitoring`.

All of the operators, except for PPO, use the traditional Grafana Prometheus monitoring stack, while PPO uses the Percona Monitoring and Management solution. This Percona monitoring system is quite extensive, but its coverage of parameters is comparable to that of the less extensive CNPGO monitoring system.

7.3.3 Performance

To measure the performance of the Postgres cluster, an operator was deployed to the GKE cluster. Subsequently, a high availability Postgres cluster was established, consisting of three nodes (one primary and two replicas), along with three connection pooler instances. A simplified representation of this cluster configuration is depicted

Table 7.7 Training needed: udpated

	PGO	CNPGO	SPGO	PPO
1st training	Kubectl	Kubectl	Kubectl	Kubectl
2nd training	Kustomize	Helm	Helm	Helm

Table 7.8 Monitoring

	PGO	CNPGO	SPGO	PPO
Health	Yes	Yes	Yes	Yes
Query performance	Yes	Yes	No	Yes
Number of connections	Yes	Yes	Yes	Yes
Locks	Yes	Yes	Yes	Yes
Index hit	No	No	No	No
Cache hit	Yes	No	Yes	Yes
Disk space usage	Yes	Yes	No	Yes
CPU and memory usage	Yes	Yes	Yes	Yes
WAL generation rate	Yes	Yes	No	Yes
Replication lag	Yes	Yes	No	No
Errors and logs	No	No	No	Yes
Backup and recovery	Yes	Yes	Yes	No

in Figure 7.3.

For each operator test, the Kubernetes cluster was recreated to ensure a clean starting point. A Postgres cluster with three nodes (one primary and two replicas) was deployed, along with three connection pooler instances.

To standardize the setup across each cluster, configurations were calibrated in line with recommendations provided by the pgTune[7)] website. The adjusted settings were as follows:

- max_connections: 200
- shared_buffers: 1536MB
- effective_cache_size: 4608MB

[7)] https://pgtune.leopard.in.ua

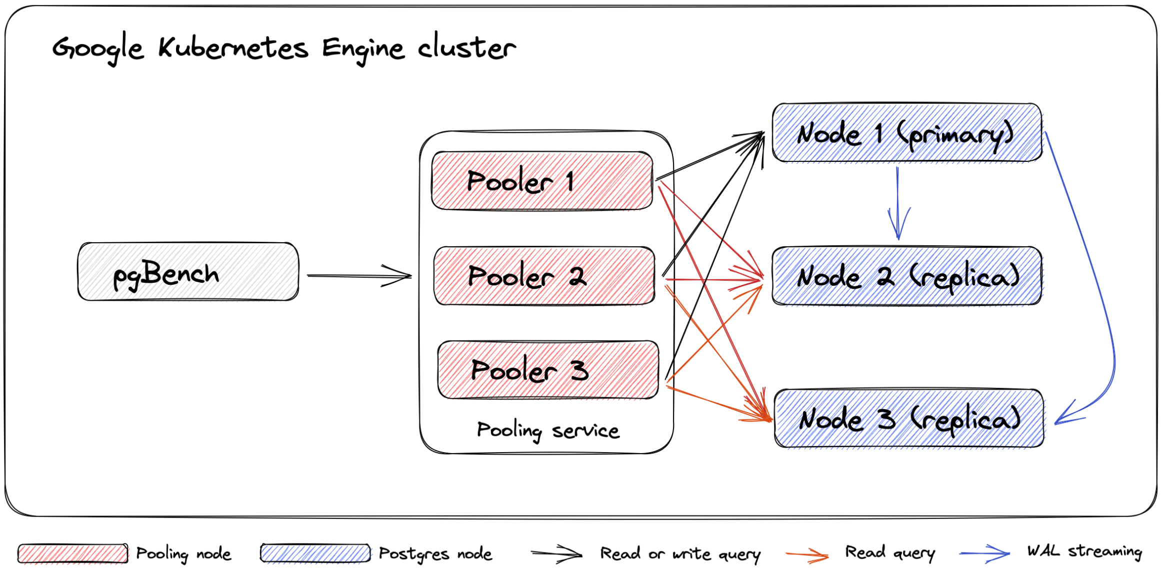

Figure 7.3 Kubernetes performance setup

- maintenance_work_mem: 384MB
- checkpoint_completion_target: 0.9
- wal_buffers: 16MB
- default_statistics_target: 100
- random_page_cost: 4
- effective_io_concurrency: 2
- work_mem: 3932kB
- min_wal_size: 1GB
- max_wal_size: 4GB

After the creation of the Postgres cluster, each cluster was tested using the pgBench tool, a PostgreSQL benchmarking utility. This tool was configured to execute 10,000 transactions across 25 concurrent clients and utilizing 10 threads. This benchmark was directed towards the Postgres cluster's pooler service. This procedure was replicated twice more for thoroughness.

The results presented in Table 8.13 only display the transactions per second, as these offer a sufficient indication of the operator's performance. For a comprehensive view of the results from this benchmark, along with the test process details, please refer to Appendix A VI.

The commands executed throughout this procedure, along with the complete configuration of the operators, can be located in the `tests/performance` directory in the repository.

7.3.4 Issues with CNPGO

CNPGO was the only one unable to execute 250,000 transactions in each run due to an error (client 6 script 0 aborted in command 4 query 0: FATAL: query wait timeout, SSL connection has been closed unexpectedly). This error usually occurs when a query takes too long to execute, leading to a timeout. The SSL connection is then closed unexpectedly, causing the transaction to fail. This might suggest that CNPGO is struggling with performance or network stability in this particular scenario.

7.3.5 Issues with SPGO

The possible reason for SPGO's low transactions per second score is that a cluster profile is needed to deploy an SPGO cluster. When this profile was correctly set, Google Kubernetes Engine was unable to deploy the cluster. By gradually reducing these values, the available resources were eventually found, but these settings were probably too low for optimal performance (500m CPU and 2Gi RAM). Despite the cluster showing that it had more memory and CPU allocable, as may be seen in Figure 7.4, the reduced resource allocation might have constrained SPGO's performance.

7.3.6 Issues with PPO

As mentioned in Chapter 7.3.2, changes to PPO's configuration do not affect the cluster, therefore, the cluster was not modified during this test. This means that the performance results for PPO are based on its default configuration settings.

Table 7.9 Performance analysis

	PGO	CNPGO	SPGO	PPO
First run	544.91 tps	403.70 tps	284.39 tps	401.46 tps
Second run	543.29 tps	402.54 tps	279.89 tps	392.04 tps
Third run	538.51 tps	392.63 tps	309.16 tps	387.79 tps
Mean	542.24 tps	399.62 tps	291.15 tps	393.77 tps

Nodes

Filter Filter nodes

Name	Status	CPU requested	CPU allocatable	Memory requested	Memory allocatable	Storage requested	Storag
gke-operators-e2-standard-2-8a273e24-1vhq	Ready	289 mCPU	1.93 CPU	560.99 MB	6.33 GB	0 B	
gke-operators-e2-standard-2-8a273e24-d2mn	Ready	481 mCPU	1.93 CPU	503.32 MB	6.33 GB	0 B	
gke-operators-e2-standard-2-8a273e24-j33v	Ready	531 mCPU	1.93 CPU	576.23 MB	6.33 GB	0 B	

Figure 7.4 GKE nodes details

8 EVALUATION

This part of the thesis will address Research Question No. 4, as outlined in Chapter 3. It will utilize the findings presented in Chapter 7 to conduct a comprehensive evaluation of each quality attribute associated with the operators. Each attribute will be quantified and discussed in detail. The cumulative results and final assessment will be presented in the concluding chapter.

8.1 Measuring rule

In this chapter, the operators will be evaluated. When a dedicated measuring tool is not available, the operators will be compared against each other to provide a relative measure of their attributes. This comparative analysis will help illuminate their relative strengths and weaknesses.

8.2 Reliability

8.2.1 Maturity

In order to determine the maturity of the system, data was statically collected from the repositories of each operator (Chapter 7.2.1). This included the popularity of the operators, the number of issues they had, and the number of issues that were resolved. The maturity was then determined based on the popularity of the operators and the ratio of resolved issues.

Popularity After examining the popularity by the number of stars, it is clear that PGO far exceeds the others (results presented in Table 8.3). In contrast, CNPGO shows much lower popularity. The remaining operators show marginal levels of popularity to the level of popularity that PGO or CNPGO have received. If the ranking were based solely on a popularity contest, PGO would clearly win. However, the evaluation requires a more comprehensive examination beyond simply measuring popularity.

Fixed issues As presented in Table 8.3, PGO, CNPGO, and PPO are shown to have resolved a significant number of issues, with PGO registering the highest ratio of fixed issues. On the other hand, despite being less popular and more recent than PGO,

Table 8.1 Popularity of operators

	PGO	CNPGO	SPGO	PPO
Popularity (stars)	3258	1198	84	149
Popularity ratio	100%	37%	3%	5%

SPGO has a larger number of reported issues, many of which remain unresolved (445 in total). This might suggest that the development of SPGO is still in progress.

Table 8.2 Operators issues

	PGO	CNPGO	SPGO	PPO
Total issues	1884	764	1959	317
Issues fixed	1755	691	1514	282
Fixed issues ratio	93%	90%	77%	89%

8.2.2 Overall maturity / reliability

The overall maturity of the operators was determined by averaging the maturity and the ratio of fixed issues. These results are presented in Table 8.3. As maturity is the only subtest within the reliability category, it is thus used as the measure of reliability.

Due to its high popularity and impressive ratio of fixed issues, PGO's results are outstanding, suggesting that it can be considered the most reliable operator among all. The results for CNPGO are good, while those for SPGO and PPO can be regarded as fair.

Table 8.3 Operators maturity / reliability

	PGO	CNPGO	SPGO	PPO
Overall maturity / reliability	97%	64%	40%	47%

8.2.3 Maintenance

Renewal The overall renewal rate of the operators, as presented in Table 8.4, was calculated based on the number of commits since the repository's creation date. SPGO achieved the highest average ratio of commits per day, at 5.03.

Interestingly, this contrasts with the operator Fixed Issues, where SPGO scored the lowest, suggesting that it is still in the development stage. A potential explanation for this could be that SPGO utilizes the 'issues' function for internal project management purposes, rather than exclusively for issue tracking, or for the frequent updating of SPGO's user interface.

PGO and PPO share the same lifetime since PPO was forked from PGO. However, PGO has a higher rate of commits per day, indicating that it is updating more rapidly than PPO.

The overall results for this attribute are high, with values greater than 2 indicating that all of the operators are frequently updated. When comparatively analyzed, SPGO is updated more than twice as frequently as PGO and PPO, and almost 1.8 times more frequently than CNPGO.

Table 8.4 Operators renewal

	PGO	CNPGO	SPGO	PPO
Lifetime (days)	2254	1168	1433	2254
Sum of commits	5582	3362	7208	4689
Commits/day	2.48	2.88	5.03	2.08
Renewal / maintenance	49%	57%	100%	41%

8.3 Usability

Usability testing was conducted in two separate test processes: the first being the static test process described in Chapter 7.2.2, which aimed to review the documentation, and the second being the dynamic test process outlined in Chapter 7.3.2, which aimed to test the operator's operability. The results from both processes will be evaluated in this chapter.

8.3.1 Learnability

TBD - (start) move this to methodology

Learnability testing, which is one part of usability, aims to assess the ease with which new users can understand and use a product. This particular testing was conducted with two primary objectives. The first objective was to identify the presence of examples in the documentation, as learning facilitated by examples is generally more effective than without. The second objective was to gauge the level of learning required to operate the system effectively.

TBD - (end) move this to methodology

Learnability was divided into two parts: one evaluating the required training for each operator, and the other assessing the presence of examples in the documentation. Both aspects are tested in Chapter 7.2.2.

Required training The training required presented in Table 8.5 was calculated as the additional training needed to work with each operator, over and above knowledge of Kubectl (the Kubernetes command-line tool). Each additional tool required to work successfully with the operator resulted in a 5% score decrease.

Overall, the required training to work with operators is minimal. For PGO, the only additional software needed, apart from Kubectl—which is essential for working with Kubernetes clusters—is Kustomize. For the rest of the operators, Helm is required to install the monitoring stack. It can therefore be asserted that proficiency with Kubernetes equates to the ability to work with operators.

Table 8.5 Training needed

	PGO	CNPGO	SPGO	PPO
Required Training	95%	95%	95%	95%

Documentation examples As indicated in Table 7.3, examples in the documentation are prevalent across all operators. SPGO, however, is at a slight disadvantage. Despite having a Graphical User Interface capable of managing the entire cluster and efficiently handling all cluster operations, it does not provide examples for major and minor updates. These could be easily configured via the GUI, making it beneficial if the provided examples indicate that the respective functionality can be conveniently implemented using the GUI. Given that there are ten examples, each one has been assigned a value contributing 10% towards the total score.

Examples in the documentation were widely presented in the operators' documentation. PGO achieved the highest rate, presenting all the examples in its documentation and thus offering the most helpful guidance. Although CNPGO has extensive documentation, it lacks some examples, rendering it slightly less helpful than PGO's. Both SPGO and PPO were missing even more examples, but despite this, the level of detail in the available examples was still high.

Table 8.6 Documentation examples

	PGO	CNPGO	SPGO	PPO
Documentation examples	100%	90%	80%	80%

Overall learnability The overall learnability rating of the operators, as presented in Table 8.7, was calculated as the average of the scores from the required training and the documentation examples.

Overall, the learnability levels are quite high, suggesting that all of the operators are relatively easy to learn.

Table 8.7 Learnability of operators

	PGO	CNPGO	SPGO	PPO
Learnability	98%	93%	88%	88%

8.3.2 Operability

Operability was divided into two parts: one evaluating the ease of use for each operator, and the other assessing the quality of monitoring. Both aspects are tested in Chapter 7.3.2.

Ease of use The number of commands executed to achieve the objective was counted for each test case. In instances where an operator did not provide the necessary functionality, or the functionality was malfunctioning, the number of steps required was designated as 10. This allocation is due to the significant effort that would be required to realize this functionality, or the possibility that it might not be achievable at all.

Due to its GUI, SPGO is the easiest operator to use. In comparison to SPGO's ease of use, the ease of use level of the remaining operators is relatively poor.

Table 8.8 Ease of use

	PGO	CNPGO	SPGO	PPO
Sum of commands	23	29	7	49
Ease of use	30%	24%	100%	14%

Monitoring The quality of monitoring was tested in Chapter 7.3.2. Each operator was evaluated based on the number of necessary attributes covered in the monitoring.

PGO has the highest monitoring capabilities, followed closely by CNPGO and PPO, both of which also demonstrate good monitoring abilities. Although SPGO is equipped with a GUI, its monitoring performance is only fair.

Table 8.9 Monitoring

	PGO	CNPGO	SPGO	PPO
Monitoring	83%	75%	50%	75%

Overall operability The overall operability rating of the operators, as presented in Table 8.10, was calculated as the average of the scores from the ease of use and the monitoring.

Owing to its high score in ease of use, SPGO has the highest operability among the operators. PGO and CNPGO exhibit similar levels of operability, while PPO, with the lowest score, can be considered the least operable.

Table 8.10 Operability

	PGO	CNPGO	SPGO	PPO
Ease of use	30%	24%	100%	14%
Monitoring	83%	75%	50%	75%
Operability	57%	50%	75%	45%

8.4 Overall usability

The overall usability was calculated as the average of learnability, as presented in Table 8.7, and operability, as presented in Table 8.10.

The results, as shown in Table 8.11, indicate that SPGO is the most usable operator among all, followed by PGO and then CNPGO, with PPO being the least usable. Nevertheless, the overall usability of the operators is at a good level.

Table 8.11 Usability

	PGO	CNPGO	SPGO	PPO
Learnability	98%	93%	88%	88%
Operability	57%	50%	75%	45%
Usability	78%	73%	82%	67%

8.5 Security

The results of the vulnerability analysis, as presented in Table 7.4 and 7.5, were quantified as follows: the presence of a critical vulnerability was scored as 0%, high severity vulnerability as 20%, medium as 40%, low as 60%, unknown as 80%, and the absence of vulnerabilities as 100%.

CNPGO was the only operator with no detected vulnerabilities, thus achieving a full score and can be regarded as secure. PPO, having only medium and lower severity vulnerabilities, can also be considered relatively secure. However, both PGO and SPGO, which were found to have high-severity vulnerabilities, can be deemed less secure compared to CNPGO and PPO.

Table 8.12 Vulnerability analysis

	PGO	CNPGO	SPGO	PPO
Security	20%	100%	20%	40%

8.6 Performance

According to the results presented in Table 7.9 the most performant operator PGO received a score of 100% in this test, while the other operators were assigned scores proportionally based on their performance.

Table 8.13 Operators performance

	PGO	CNPGO	SPGO	PPO
Performance	100%	74%	54%	73%

8.7 Overall quality of the operators

The overall quality, according to the metric defined in Chapter 5, was calculated as the average of these metrics and is presented in Table 8.14. According to these results, the CNPGO operator can be considered the highest quality among the evaluated operators, followed by PGO, SPGO, and finally PPO.

Table 8.14 Overall quality of operators

	PGO	CNPGO	SPGO	PPO
Performance	100%	74%	54%	73%
Reliability	97%	64%	40%	47%
Usability	78%	73%	82%	67%
Maintenance	49%	57%	100%	41%
Security	20%	100%	20%	40%
Average	68.8%	73.6%	59.2%	53.6%

Table 8.15 Overall quality without security

	PGO	CNPGO	SPGO	PPO
Performance	100%	74%	54%	73%
Reliability	97%	64%	40%	47%
Usability	78%	73%	82%	67%
Maintenance	49%	57%	100%	41%
Average	81%	67%	69%	57%

CONCLUSION

In this final chapter of the thesis, the objectives outlined in Chapter 2 are aimed to be met by delivering clear and comprehensive recommendations for stakeholders on which operators are best suited to meet varying needs and preferences.

The metrics for comparing operators were established in the previous chapters, and the operators were tested against these metrics, with the measured values being evaluated.

According to these results, the best choice is the CloudNativePG (CNPGO) operator, as it received the highest overall score and demonstrated a standout performance in the security category, where other operators received significantly lower scores.

The evaluation considers security as a crucial parameter due to its considerable impact on the operator scores.

If the security parameter is disregarded, the most notable performer is the Crunchy Postgres for Kubernetes (PGO), which achieved the highest scores in performance and reliability. Interestingly, without the security factor, the StackGres operator would surpass the CNPGO, as it scored highest in the maintenance attribute and was rated as the most usable operator, largely due to its Graphical User Interface.

The Percona Operator for PostgreSQL (PPO) cannot be recommended due to its underperformance across all evaluated categories. With the lowest scores across the board, its overall performance falls short in comparison to the other operators, making it the least suitable option based on our evaluation criteria.

REFERENCES

[1] Group, T. P. G. D.: PostgreSQL 15.1 Documentation - What Is PostgreSQL? [visited 2023-02-08].
URL `https://www.postgresql.org/docs/15/intro-whatis.html`

[2] Riggs, S.; Ciolli, G.: *PostgreSQL 14 Administration Cookbook*, chapter Introducing PostgreSQL 14. Birmingham: Packt publishing, 1st edition, 2022, ISBN 9781803248974, pp. 2–8.

[3] Group, T. P. G. D.: PostgreSQL 15.1 Documentation - History. [visited 2023-02-08].
URL `https://www.postgresql.org/docs/15/history.html`

[4] Inc., S. E.: Stack Overflow 2022 Developer Survey. [visited 2023-02-07].
URL `https://survey.stackoverflow.co/2022/`

[5] Juba, S.; Vannahme, A.; Volkov, A.: *Learning PostgreSQL*, chapter Relational databases. Packt Publishing Ltd, 1st edition, 2015, ISBN 9781783989188, pp. 1–4.

[6] Nasser, H.: PostgreSQL Process Architecture. Creating a listener on the backend. . . . 2023, [visited 2023-03-16].
URL `https://medium.com/@hnasr/postgresql-process-architecture-f21e16459907`

[7] Group, T. P. G. D.: PostgreSQL: Documentation: 15: 30.3. Write-Ahead Logging (WAL). [visited 2023-02-21].
URL `https://www.postgresql.org/docs/current/wal-intro.html`

[8] Group, T. P. G. D.: PostgreSQL: Documentation: 15: 26.3. Continuous Archiving and Point-in-Time Recovery (PITR). [visited 2023-02-23].
URL `https://www.postgresql.org/docs/15/continuous-archiving.html`

[9] Riggs, S.; Ciolli, G.: *PostgreSQL 14 Administration Cookbook*, chapter Replication and Upgrades. Birmingham: Packt publishing, 1st edition, 2022, ISBN 9781803248974, pp. 499–557.

[10] Group, T. P. G. D.: PostgreSQL: Documentation: 15: 26.1. SQL Dump. [visited 2023-02-23].
URL `https://www.postgresql.org/docs/15/backup-dump.html`

[11] Group, T. P. G. D.: Reliable PostgreSQL Backup and Restore. [visited 2023-02-23].
URL `https://pgbackrest.org`

[12] Group, T. P. G. D.: PostgreSQL: Documentation: 15: 27. High Availability, Load Balancing, and Replication. [visited 2023-02-23].
URL `https://www.postgresql.org/docs/15/high-availability.html`

[13] Group, T. P. G. D.: PostgreSQL: Documentation: 15: 27.3. Failover. [visited 2023-02-23].
URL `https://www.postgresql.org/docs/15/warm-standby-failover.html`

[14] Stratnev, P.: Migrating a PostgreSQL database cluster managed by Patroni. [visited 2023-02-24].
URL `https://blog.palark.com/migrating-a-postgresql-cluster-managed-by-patron`

[15] SE, Z.: Introduction — Patroni 3.0.1 documentation. [visited 2023-02-24].
URL `https://patroni.readthedocs.io/en/latest/`

[16] Avinash Vallarapu, F. L. C., Jobin Augustine: Scaling PostgreSQL using Connection Poolers and Load Balancers for an Enterprise Grade environment. [visited 2023-02-24].
URL `https://www.percona.com/blog/scaling-postgresql-using-connection-poolers`

[17] Vayghan, L. A.; Saied, M. A.; Toeroe, M.; et al.: Kubernetes as an availability manager for microservice applications. *arXiv preprint arXiv:1901.04946*, 2019.

[18] Authors, T. K.: Kubernetes Components. [visited 2023-02-07].
URL `https://kubernetes.io/docs/concepts/overview/components/`

[19] Sayfan, G.: *Mastering Kubernetes*, chapter Kubernetes concepts. Birmingham: Packt Publishing, third edition edition, [2020]., ISBN 978-183-9211-256.

[20] Dobies, J.; Wood, J.: *Kubernetes operators*. Sebastopol, CA: O'Reilly Media, 2020, ISBN 978-149-2048-046.

[21] Burns, B.; Beda, J.; Hightower, K.; et al.: *Kubernetes: up and running*, chapter Pods. " O'Reilly Media, Inc.", third edition edition, 2022.

[22] Burns, B.; Beda, J.; Hightower, K.; et al.: *Kubernetes: up and running*. " O'Reilly Media, Inc.", third edition edition, 2022.

[23] Authors, T. K.: Pods. [visited 2023-02-07].
URL `https://kubernetes.io/docs/concepts/workloads/pods/`

[24] Authors, T. K.: Create static Pods. [visited 2023-02-07].
URL `https://kubernetes.io/docs/tasks/configure-pod-container/static-pod/`

[25] Authors, T. K.: ReplicaSet. [visited 2023-02-07].
URL https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/

[26] Authors, T. K.: Service. [visited 2023-02-07].
URL https://kubernetes.io/docs/concepts/services-networking/service/

[27] Sayfan, G.: *Mastering Kubernetes*, chapter Managing Storage. Birmingham: Packt Publishing, third edition edition, [2020]., ISBN 978-183-9211-256, pp. 175–219.

[28] Authors, T. K.: StatefulSets. [visited 2023-02-24].
URL https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/

[29] Foundaution, C. N.: kubernetes/CHANGELOG-1.5.md at master · a0x8o/kubernetes. [visited 2023-02-24].
URL https://github.com/a0x8o/kubernetes/blob/master/CHANGELOG-1.5.md

[30] Foundaution, C. N.: Custom Resource (CR). [visited 2023-03-02].
URL https://ibm.github.io/kubernetes-operators/lab1/

[31] Ziolkowski, D.: Kubernetes CRDs: What They Are and Why They Are Useful. [visited 2023-03-01].
URL https://thenewstack.io/kubernetes-crds-what-they-are-and-why-they-are-us

[32] Dobies, J.; Wood, J.: *Kubernetes operators*. Sebastopol, CA: O'Reilly Media, 2020, ISBN 978-149-2048-046, pp. 27–32.

[33] Authors, T. K.: Controllers. [visited 2023-03-02].
URL https://kubernetes.io/docs/concepts/architecture/controller/

[34] Authors, T. K.: Custom Resources. [visited 2023-03-02].
URL https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/

[35] Kahani, O.; Strejevitch, J.; Schuetz, T.; et al.: CNCF Operator White Paper - Final Version. 2023.
URL https://github.com/cncf/tag-app-delivery/blob/main/operator-wg/whitepaper/Operator-WhitePaper_v1-0.md

[36] Dobies, J.; Wood, J.: *Kubernetes operators*. Sebastopol, CA: O'Reilly Media, 2020, ISBN 978-149-2048-046, pp. 4–8.

[37] David, C.: WHAT IS APPLICATION LIFECYCLE MANAGEMENT? [visited 2023-02-07].
URL `http://davidchappell.com/writing/white_papers/What_is_ALM_v2.0--Chappell.pdf`

[38] Philips, B.: Introducing Operators: Putting Operational Knowledge into Software. [visited 2023-03-01].
URL `https://web.archive.org/web/20170129131616/https://coreos.com/blog/introducing-operators.html`

[39] Dobies, J.; Wood, J.: *Kubernetes operators*. Sebastopol, CA: O'Reilly Media, 2020, ISBN 978-149-2048-046, pp. XIII–XVI.

[40] Dobies, J.; Wood, J.: *Kubernetes operators*. Sebastopol, CA: O'Reilly Media, 2020, ISBN 978-149-2048-046, pp. 27–32.

[41] Framework, T. O.: Welcome to Operator framework. [visited 2023-03-03].
URL `https://operatorframework.io/operator-capabilities/`

[42] Group, T. P. G. D.: PostgreSQL: Software Catalogue - Clustering/replication. [visited 2023-03-10].
URL `https://www.postgresql.org/docs/15/warm-standby-failover.html`

[43] OperatorHub.io: OperatorHub.io | The registry for Kubernetes Operators. [visited 2023-03-10].
URL `https://operatorhub.io/?keyword=postgres`

[44] Bogdanov, N.: Comparing Kubernetes operators for PostgreSQL. [visited 2023-02-20].
URL `https://blog.palark.com/comparing-kubernetes-operators-for-postgresql/`

[45] Crunchy Data Solutions, I.: Trusted Open Source PostgreSQL and Commercial Support for the Enterprise. [visited 2023-02-07].
URL `https://www.crunchydata.com`

[46] Crunchy Data Solutions, I.: PGO, the Postgres Operator from Crunchy Data. [visited 2023-02-15].
URL `https://access.crunchydata.com/documentation/postgres-operator/5.3.0/tutorial/create-cluster/`

[47] Crunchy Data Solutions, I.: PGO, the Postgres Operator from Crunchy Data. [visited 2023-02-15].

URL https://access.crunchydata.com/documentation/postgres-operator/5.3.0/tutorial/customize-cluster/

[48] Crunchy Data Solutions, I.: PGO, the Postgres Operator from Crunchy Data. [visited 2023-02-15].
URL https://access.crunchydata.com/documentation/postgres-operator/5.3.0/tutorial/delete-cluster/

[49] Crunchy Data Solutions, I.: PGO, the Postgres Operator from Crunchy Data. [visited 2023-02-15].
URL https://access.crunchydata.com/documentation/postgres-operator/5.3.0/tutorial/high-availability/

[50] Crunchy Data Solutions, I.: PGO, the Postgres Operator from Crunchy Data. [visited 2023-02-15].
URL https://access.crunchydata.com/documentation/postgres-operator/5.3.0/tutorial/update-cluster/

[51] Crunchy Data Solutions, I.: PGO, the Postgres Operator from Crunchy Data. [visited 2023-02-17].
URL https://www.crunchydata.com/blog/easy-major-postgresql-upgrades-using-pg

[52] Crunchy Data Solutions, I.: PGO, the Postgres Operator from Crunchy Data. [visited 2023-02-15].
URL https://access.crunchydata.com/documentation/postgres-operator/5.3.0/tutorial/backups/

[53] Crunchy Data Solutions, I.: PGO, the Postgres Operator from Crunchy Data. [visited 2023-02-15].
URL https://access.crunchydata.com/documentation/postgres-operator/5.3.0/tutorial/disaster-recovery/

[54] Crunchy Data Solutions, I.: PGO, the Postgres Operator from Crunchy Data. [visited 2023-02-15].
URL https://access.crunchydata.com/documentation/postgres-operator/5.3.0/tutorial/monitoring/

[55] Crunchy Data Solutions, I.: PGO, the Postgres Operator from Crunchy Data. [visited 2023-02-15].
URL https://access.crunchydata.com/documentation/postgres-operator/5.3.0/tutorial/connection-pooling/

[56] Crunchy Data Solutions, I.: Customize a Postgres Cluster. [visited 2023-03-10]. URL `https://access.crunchydata.com/documentation/postgres-operator/v5/tutorial/customize-cluster/`

[57] Crunchy Data Solutions, I.: PGO, the Postgres Operator from Crunchy Data. [visited 2023-02-15]. URL `https://github.com/CrunchyData/postgres-operator`

[58] L'Ecuyer, A.: Easy Postgres Major Version Upgrades Using PGO v5.1. [visited 2023-02-15]. URL `https://access.crunchydata.com/documentation/postgres-operator/5.3.0/architecture/overview/`

[59] Crunchy Data Solutions, I.: Customize a Postgres Cluster. [visited 2023-04-15]. URL `https://access.crunchydata.com/documentation/postgres-operator/5.3.1/releases/5.3.1/`

[60] OperatorHub.io: OperatorHub.io | The registry for Kubernetes Operators. [visited 2023-02-07]. URL `https://operatorhub.io/operator/postgresql`

[61] OperatorHub.io: OperatorHub.io | The registry for Kubernetes Operators. [visited 2023-02-13]. URL `https://operatorhub.io/operator/cloud-native-postgresql`

[62] Corporation, E.: EDB Postgres for Kubernetes v1. [visited 2023-02-17]. URL `https://www.enterprisedb.com/docs/postgres_for_kubernetes/latest/license_keys`

[63] Contributors, T. C.: CloudNativePG. [visited 2023-02-17]. URL `https://cloudnative-pg.io/documentation/1.19/`

[64] Contributors, T. C.: CloudNativePG. [visited 2023-02-17]. URL `https://cloudnative-pg.io/documentation/1.19/postgres_for_kubernetes/latest/replica_cluster/`

[65] Contributors, T. C.: CloudNativePG. [visited 2023-02-17]. URL `https://cloudnative-pg.io/documentation/1.19/replication/`

[66] Contributors, T. C.: CloudNativePG. [visited 2023-02-17]. URL `https://cloudnative-pg.io/documentation/1.19/database_import/`

[67] Contributors, T. C.: CloudNativePG. [visited 2023-02-17].
URL https://cloudnative-pg.io/documentation/1.19/rolling_update/#rolling-updates

[68] Corporation, E.: EDB Postgres for Kubernetes v1. [visited 2023-02-16].
URL https://cloudnative-pg.io/documentation/1.19/postgres_for_kubernetes/latest/backup_recovery/

[69] Contributors, T. C.: CloudNativePG. [visited 2023-02-17].
URL https://cloudnative-pg.io/documentation/1.19/postgres_for_kubernetes/latest/tde/

[70] Contributors, T. C.: CloudNativePG. [visited 2023-02-17].
URL https://cloudnative-pg.io/documentation/1.19/quickstart/

[71] Contributors, T. C.: CloudNativePG. [visited 2023-02-17].
URL https://cloudnative-pg.io/documentation/1.19/postgres_for_kubernetes/latest/connection_pooling/

[72] Contributors, T. C.: CloudNativePG. [visited 2023-02-17].
URL https://cloudnative-pg.io/documentation/1.19/postgres_for_kubernetes/latest/postgresql_conf/

[73] Stölting, S. J.: PostgreSQL On Kubernetes Experiences. [visited 2023-02-20].
URL https://proopensource.it/blog/postgresql-on-k8s-experiences

[74] Contributors, T. C.: Release notes for CloudNativePG 1.19. [visited 2023-04-15].
URL https://cloudnative-pg.io/documentation/1.19/release_notes/v1.19/

[75] Inc., O.: OnGres Inc. / StackGres. [visited 2023-02-20].
URL https://gitlab.com/ongresinc/stackgres

[76] Inc., O.: About. [visited 2023-02-20].
URL https://www.ongres.com/about-us/

[77] Inc., O.: SGDbOps. [visited 2023-02-20].
URL https://stackgres.io/doc/latest/reference/crd/sgdbops/#major-version-upgrade

[78] Inc., O.: SGPostgresConfig. [visited 2023-02-20].
URL https://stackgres.io/doc/latest/reference/crd/sgpgconfig/

[79] Inc., O.: Architecture. [visited 2023-02-20].
URL https://stackgres.io/doc/latest/intro/architecture/

[80] Inc., O.: CHANGELOG.md · main · OnGres Inc. / StackGres. [visited 2023-02-20].
URL https://gitlab.com/ongresinc/stackgres/-/blob/main/CHANGELOG.md

[81] Inc., O.: Licensing. [visited 2023-02-20].
URL https://stackgres.io/doc/latest/intro/license/

[82] OperatorHub.io: OperatorHub.io | The registry for Kubernetes Operators. [visited 2023-02-14].
URL https://operatorhub.io/operator/stackgres

[83] LLC, P.: About Percona - Percona. [visited 2023-02-14].
URL https://www.percona.com/about

[84] Pronin, S.: Run PostgreSQL in Kubernetes: Solutions, Pros and Cons. [visited 2023-02-20].
URL https://www.percona.com/blog/run-postgresql-in-kubernetes-solutions-pros

[85] LLC, P.: Update Percona Operator for PostgreSQL. [visited 2023-02-21].
URL https://docs.percona.com/percona-operator-for-postgresql/update.html

[86] LLC, P.: Comparison with other solutions. [visited 2023-02-20].
URL https://docs.percona.com/percona-operator-for-postgresql/compare.html

[87] LLC, P.: Providing Backups. [visited 2023-02-21].
URL https://docs.percona.com/percona-operator-for-postgresql/backups.html

[88] LLC, P.: Monitor with Percona Monitoring and Management (PMM). [visited 2023-02-2o].
URL https://docs.percona.com/percona-operator-for-postgresql/monitoring.html

[89] LLC, P.: Percona Operator for PostgreSQL. [visited 2023-02-21].
URL https://docs.percona.com/percona-operator-for-postgresql/2.0/index.html

[90] OperatorHub.io: OperatorHub.io | The registry for Kubernetes Operators. [visited 2023-02-14].
URL `https://operatorhub.io/operator/percona-postgresql-operator`

[91] Gilb, T.; Finzi, S.; et al.: *Principles of software engineering management*, volume 11. Addison-wesley Reading, MA, 1988.

[92] Graham, D.; Black, R.; van Veenendaal, E.: *Foundations of software testing*. Australia: Cengage, 4th edition, 2020, ISBN 978-147-3764-798.

[93] ISO/IEC/IEEE International Standard - Software and systems engineering —Software testing —Part 2:Test processes. *ISO/IEC/IEEE 29119-2:2013(E)*, 2013: pp. 1–68, doi:10.1109/IEEESTD.2013.6588543.

LIST OF ABBREVIATIONS

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
CNPGO	CloudNativePG
CR	Custom Resource
CRD	Custom Resource Definition
EDBO	EDB Postgres for Kubernetes Operator
GKE	Google Kubernetes Engine
GUI	Graphical User Interface
HA	High Availability
K8s	Kubernetes
ORDBMS	Object-relational Database Management System
PGO	Crunchy Postgres for Kubernetes
PITR	Point-In-Time Recovery
PPO	Percona Operator for PostgreSQL
RBAC	Role-Based Acceess Control
SPGO	StackGres Operator
WAL	Write Ahead Log

LIST OF FIGURES

LIST OF TABLES

LIST OF APPENDICES

APPENDIX A I. GENERAL TEST PLAN

- Test plan ID: TP0
- Context of testing:
  - Project: Bachelor's thesis.
  - Test levels: Acceptance testing.
  - Test types: Static and dynamic.
  - Test items:
    * Crunchy Postgres for Kubernetes Operator v5.3.1.
    * CloudNativePG Operator v1.20.0.
    * StackGres Operator v1.4.3.
    * Percona Operator for Postgres 1.4.0.
  - Test scope: Operator, Operator's documentation, Operator's repository.
  - Test basis: Defined criteria.
- Risk register:
  - Limited staff and time might prevent thorough testing of all features and functionalities of the software during acceptance testing.
  - Inadequately trained staff might struggle to design effective test cases, which could result in missed defects and lower overall testing effectiveness.
  - Due to the lack of expertise among staff members, the software's readiness for production might be inaccurately assessed, leading to incorrect conclusions about its quality and suitability for release
- Test strategy:
  - General: The purpose of testing is to evaluate the ability of Operators to fulfill the desired criterias, and to provide information for making informed decisions on which Operator to select in last chapter. Non-functional requirements will be tested with static and dynamic test techniques.
  - Test levels: Acceptance testing
  - Test deliverables: Test plans, test model specification, test procedure specification, incident reports.
  - Test design techniques: Exploratory Testing, Use cases, Walkthroughs.
  - Entry criteria: Created environments.

- Exit criteria: Decision metrics were collected.
- Test competition criteria: All criteria covered by at least one test case.
- Degree of independence: No connection between tested Operators and tester. Tester is fully independent.
- Metrics to be collected:
  * Static testing: Vulnerability analysis (number of vulnerabilities and their severity), Repository review (sum of issues, sum of repaired issues, sum of stars, sum of commits, repository creation date), Documentation review (examples, training needed),
  * Dynamic testing: The sum of the commands required to achieve functionality. Covered monitoring. Performance described by transactions per second.
- Test data requirements:
  * Crunchy Postgres for Kubernetes Operator v5.3.1
    · PGO: https://github.com/CrunchyData/postgres-operator-examples
    · PGODOC: https://access.crunchydata.com/documentation/postgres-operator/v5/
    · PGOREPO: https://github.com/CrunchyData/postgres-operator
  * CloudNativePG Operator version 1.20.0
    · CNPGO: https://raw.githubusercontent.com/cloudnative-pg/cloudnative-pg/release-1.20/releases/cnpg-1.20.0.yaml
    · CNPGODOC: https://cloudnative-pg.io/documentation/1.20/
    · CNPGOREPO: https://github.com/cloudnative-pg/cloudnative-pg
  * StackGres Operator version 1.4.3
    · SPGO: https://stackgres.io/downloads/stackgres-k8s/stackgres/helm/
    · SPGODOC: https://stackgres.io/doc/1.4/
    · SPGOREPO: https://gitlab.com/ongresinc/stackgres
  * Percona Operator for PostgreSQL version 1.4.0
    · PPOO: https://raw.githubusercontent.com/percona/percona-postgresql-operator/v1.4.0/deploy/operator.yaml
    · PPODOC: https://docs.percona.com/percona-operator-for-postgresql/index.html

        - PPOREPO: https://github.com/percona/percona-postgresql-operator
- Test environment requirements:
    * Kind Kubernetes cluster with two worker nodes for all dynamic test except performance tests, installed on Unix/Linux compatible machine.
    * Google Kubernetes Engine with two worker nodes.
    * Terraform
    * Trivy security scanner.
    * Kubectl kubernetes controll tool.
    * EXCEL.
- Retesting: Retesting is not needed.
- Reggresion testing: Reggresion testing is not needed.
- Testing activities and estimates:
    * Environment setup – 30m.
    * Repository walkthrough – 2h/Operator.
    * Documentation walkthrough – 2h/Operator.
    * Deployment and configuration – 4h/Operator.
    * Performance – 4h/Operator.
    * Operabilitiy and documentation – 8h/Operator.
    * Test completion report – 1h/testing day.
- Staffing (roles and responsibilities)
    * Roles: Test architect, test manager, test designer, test automator, tester and test analyst
    * Staff: Miroslav Šiřina.
- Training needed
    * Test management.
    * Test design.
    * Test analyst.
    * Trivy and results interpretation skills.
- Test priorities
    * Static tests have higher priority to dynamic.
    * Critical features have higher priority.
- Schedule

- 1st May: repositories and documentations walkthroughs.
- 2nd May: vulnerability analysis.
- 3rd - 8th May: operability testing.
- 9th May: performance testing.
- 10th May: 11th Testing closure.

APPENDIX A II. TEST PROCESS NO. 1

Table 2.1 Test plan No. 1

Test plan ID	tp1
Revision	2
Introducton	Repositories walkthrough
Test items	Operator's repositories
Covered criteria	CR1, CM1
Test type	Static
Test approach	Repositories walkthrough
Exit criteria	All metrics gathered
Delivarables	Sum of commits, sum of stars, sum of issues, sum of fixed issues.
Duration	2 h for each Operator
Reviewer	Miroslav Šiřina
Start	1st May
Schedule	1st May: repositories walkthrough and test report.
Revisions	Rev No. 2 - added test cases MRC5 to MRC9.

2.1 Test items

Test items for the procedure were following:

- PGOREPO from General test plan
- CNPGOREPO from General test plan
- SPGOREPO from General test plan
- PPODREPO from General test plan
- PPOJIRA https://jira.percona.com/projects/DISTPG/issues/DISTPG-352?filter=allopenissues

2.2 Test tools

Excel sheet

2.3 Test procedure

- MRC1 - PGOREPO walkthrough
- MRC2 - CNPGOREPO walkthrough
- MRC3 - SPGOREPO walkthrough
- MRC4 - PPODREPO walkthrough
- MRC5 - PGOREPO cloning - retrieving the date of creation and number of commits
- MRC6 - CNPGOREPO cloning - retrieving the date of creation and number of commits
- MRC7 - SPGOREPO cloning - retrieving the date of creation and number of commits
- MRC8 - PPODREPO cloning - retrieving the date of creation and number of commits
- MRC9 - PPOJIRA walkthrough

Test completition report
Testing performed: Repositories walkthrough, repositories cloning, Percona Jira walkthrough
Deviations from planed testing: Percona is using Jira for tracking issues. To get issues Jira walkthrough was necessary. To count number of commits and get the date of first commit the repository cloning was necessary.
Test completion evaluation: The testing process was successful in gathering key data about the system despite deviations from the initial plan. The flexibility in testing procedures resulted in a more comprehensive evaluation and provided valuable insights into the system.
Factors that blocked progress: reposistory clonning, Jira walkthrough
Test Result Analysis: The tests provided valuable data about the state and history of repositories, as well as key insights into issue tracking.
Lessons Learned: The necessity to deviate from the initial test plan underlines the importance of flexibility in testing procedures. An adaptive approach can lead to a more thorough evaluation and better data collection.

APPENDIX A III. TEST PROCESS NO. 2

Table 3.1 Test plan No. 2

Test plan ID	tp2
Revision	1
Introducton	Checklist-based documentations review
Test items	Operator's documentations
Covered criteria	CU1
Test type	Static
Test approach	Checklist-based Testing
Exit criteria	All checklists completed
Delivarables	List of examples and checklist
Duration	2 h for each Operator
Reviewer	Miroslav Šiřina
Start	1st May
Schedule	1st May: documentations walkthrough and test report.

3.1 Checklist

- Instalation
- Minor upgrade to new version
- Major upgrade to new version
- Backup
- Restore
- Monitoring
- Vertical scaling
- Horizontal scaling
- Configuration Update
- Uninstall
- Training needed

3.2 Test procedure

- LC1 - PGODOC checklist-based review
- LC2 - CNPGODOC checklist-based review
- LC3 - SPGODOC checklist-based review
- LC4 - PPODOC checklist-based review

3.3 Test results

PGO Documentation `https://access.crunchydata.com/documentation/postgres-operator/v5/`
Cluster creation
`https://access.crunchydata.com/documentation/postgres-operator/v5/tutorial/create-cluster/`
Minor upgrade to new version
`https://access.crunchydata.com/documentation/postgres-operator/5.3.1/tutorial/update-cluster/`
Major upgrade to new version
`https://access.crunchydata.com/documentation/postgres-operator/5.3.1/guides/major-postgres-version-upgrade/`
Backup
`https://access.crunchydata.com/documentation/postgres-operator/v5/tutorial/backup-management/`
Restore
`https://access.crunchydata.com/documentation/postgres-operator/5.3.1/tutorial/disaster-recovery/`
Monitoring
`https://access.crunchydata.com/documentation/postgres-operator/5.3.1/tutorial/monitoring/`
Vertical scaling
`https://access.crunchydata.com/documentation/postgres-operator/5.3.1/tutorial/resize-cluster/`
Horizontal scaling
`https://access.crunchydata.com/documentation/postgres-operator/5.3.1/tutorial/resize-cluster/`
Configuration Update
`https://access.crunchydata.com/documentation/postgres-operator/5.3.1/tutorial/customize-cluster/`

Uninstall

https://access.crunchydata.com/documentation/postgres-operator/5.3.1/tutorial/delete-cluster/

Notes: PGO use kustomize for customization of yaml manifests.

CNPGO

Documentation

https://cloudnative-pg.io/documentation/1.19/

Cluster creation

https://cloudnative-pg.io/documentation/1.19/quickstart/#part-3-deploy-a-postgresql-cluster

Minor upgrade to new version

https://cloudnative-pg.io/documentation/1.19/rolling_update/

Major upgrade to new version Not found

Backup

https://cloudnative-pg.io/documentation/1.19/backup_recovery/#scheduled-backups

Restore

https://cloudnative-pg.io/documentation/1.19/backup_recovery/#scheduled-backups

Monitoring

https://cloudnative-pg.io/documentation/1.19/monitoring/

Vertical scaling

https://cloudnative-pg.io/documentation/1.19/resource_management/#resource-management

Horizontal scaling

https://cloudnative-pg.io/documentation/1.19/resource_management/#resource-management

Configuration Update

https://cloudnative-pg.io/documentation/1.19/postgresql_conf/#postgresql-configuration

Uninstall

https://cloudnative-pg.io/documentation/1.19/cnpg-plugin/#destroy

Notes: Uninstall example use cnpg plugin. Helm is needed to install monitoring.

SPGO Documentation

https://stackgres.io/doc/1.4/

Cluster creation

https://stackgres.io/doc/1.4/demo/quickstart/

Minor upgrade to new version – It is mentioned in documentation but without example `https://stackgres.io/doc/1.4/reference/crd/sgdbops/#major-version-upgrade`

Major upgrade to new version - It is mentioned in documentation but without example

`https://stackgres.io/doc/1.4/reference/crd/sgdbops/#minor-version-upgrade`

Backup

`https://stackgres.io/doc/1.4/tutorial/complete-cluster/backup-configuration/`

Restore

`https://stackgres.io/doc/1.4/runbooks/restore-backup/`

Monitoring

`https://stackgres.io/doc/1.4/install/prerequisites/monitoring/`

Vertical scaling

`https://stackgres.io/doc/1.4/tutorial/complete-cluster/instance-profile/`

Horizontal scaling

`https://stackgres.io/doc/1.4/tutorial/complete-cluster/create-cluster/`

Configuration Update

`https://stackgres.io/doc/1.4/tutorial/complete-cluster/postgres-config/`

Uninstall

`https://stackgres.io/doc/1.4/administration/uninstall/`

Notes: Helm is needed to install monitoring.

PPO Documentation

`https://docs.percona.com/percona-operator-for-postgresql/index.html`

Cluster creation

`https://docs.percona.com/percona-operator-for-postgresql/gke.html#installing-the-operator`

Minor upgrade to new version

`https://docs.percona.com/percona-operator-for-postgresql/update.html?h=postgres+update#semi-automatic-upgrade`

Major upgrade to new version

`https://docs.percona.com/percona-operator-for-postgresql/update.html?h=postgres+update#semi-automatic-upgrade`

Backup

`https://docs.percona.com/percona-operator-for-postgresql/backups.html?h=backup#use-google-cloud-storage-for-backups`

Restore

`https://docs.percona.com/percona-operator-for-postgresql/backups.html?`

h=backup#use-google-cloud-storage-for-backups

Monitoring

https://docs.percona.com/percona-operator-for-postgresql/monitoring.html?h=version#installing-the-pmm-client

Vertical scaling Not found

Horizontal scaling

https://docs.percona.com/percona-operator-for-postgresql/scaling.html?h=scale

Configuration Update

https://docs.percona.com/percona-operator-for-postgresql/options.html#creating-a-cluster-with-custom-options

Uninstall Not found

Notes: Helm is needed to install monitoring.

Test completition report

Testing performed: Checklist-based Testing

Deviations from planed testing: None

Test completion evaluation: The testing process was successful.

Factors that blocked progress: None

Test Result Analysis: The tests provided valuable data about the state of documentations, as well as key insights into Operators operation.

Lessons Learned: Future projects should be prepared for a high level of diversity in documentations. This might involve allocating more time for research or including personnel with a broader range of expertise.

APPENDIX A IV. TEST PROCESS NO. 3

4.1 Test plan No. 3

Table 4.1 Test plan No. 3

Test plan ID	tp3
Revision	2
Introducton	Vulnerability analysis of operators
Test items	Operator's container images
Covered criteria	C5
Test type	Static
Test approach	Vulnerability analysis
Exit criteria	Completed analysis
Tools	Trivy security scanner, Snyk security scanner
Delivarables	Sum of vulnerabilities, vulnerability reports
Duration	4 h for each Operator
Tester	Miroslav Šiřina
Start	2nd May
End	18th May
Schedule	2nd May: analysis and test report.
Revisions	19th May: Rev No. 2 - added test case STC5.
	19th May: Rev No. 3 - added test cases STC6 - STC10.
	19th May: Rev No. 3 - Snyk introduced

4.2 Test items

Test items for the procedure were following:

- registry.developers.crunchydata.com/crunchydata/postgres-operator:ubi8-5.3.1-0
- ghcr.io/cloudnative-pg/cloudnative-pg:1.20.0
- stackgres/operator:1.4.3
- percona/percona-postgresql-operator:1.4.0-postgres-operator

4.3 Test tools

Trivy security scaner version: 0.39.0, Vulnerability DB for all test cases except STC5 from 2nd May. Vulnerability DB for STC5 from 18th May.

4.4 Test procedure

- STC1 - Vulnerability analysis of PGO
- STC2 - Vulnerability analysis of CNPGO
- STC3 - Vulnerability analysis of SPGO
- STC4 - Vulnerability analysis of PPO
- STC5 - Vulnerability analysis of Debian
- STC6 - Vulnerability analysis of PGO
- STC7 - Vulnerability analysis of CNPGO
- STC8 - Vulnerability analysis of SPGO
- STC9 - Vulnerability analysis of PPO
- STC10 - Vulnerability analysis of Debian

4.5 Test completition report

Testing performed: Vulnerability analysis
Deviations from planed testing: Described in revisions.
Test completion evaluation: The testing process was successful.
Factors that blocked progress: None
Test Result Analysis: The tests provided valuable data about vulnerabilities in Operators.

APPENDIX A V. TEST PROCESS NO. 4

Table 5.1 Test plan No. 4

Test plan ID	tp4
Revision	1
Introducton	This test should test the Operators usability
	and quality of their monitoring.
Test items	Operator deployed in the cluster
Covered criteria	CU2
Test type	Dynamic
Test approach	Use case based Blackbox testing
Exit criteria	Each use case covered with atleast one test case
Tools	Kind cluster, kubectl, helm, kustomize
Delivarables	Number of commands needed to perform required operation.
	List of covered monitoring topics. Print screens of monitoring.
Duration	8 h for each Operator
Tester	Miroslav Šiřina
Start	3rd May
End	8th May
Schedule	3rd May: Use cases creation
	4th May: PGO
	5th May: CNPGO
	6th May: SPGO
	7th May: PPO

5.1 Actors

K - Kubernetes cluster
U - User
O - Operator

5.2 Use cases

5.3 Test procedure

- TOA1 - Operator installation.

Table 5.2 Use case No. 1

Use case name	Operator installation	
Use case ID	UCA1	
Traceability	CU2A	
Precondition	Prepared Kubernetes cluster	
Scenario		
Step No.	Actor	Description
1	U	Use case starts with prepared kubernetes cluster.
2	U	The user initiates the installation of the Operator.
3	K	Kubernetes installs the Operator.
4	U	Use case ends.

- TOB1 - Cluster installation.
- TOC1 - Cluster monitoring.
- TOD1 - Cluster vertical scaling.
- TOE1 - Cluster horizontal scaling.
- TOF1 - Cluster connection pooling.
- TOG1 - Cluster extension install.
- TOG2 - Cluster number of connections update.
- TOG3 - Cluster max_wal_size change.
- TOH1 - Cluster scheduled backup.
- TOH2 - Cluster ad-hoc backup.
- TOI1 - Cluster restore.
- TOJ1 - Cluster minor update.
- TOK1 - Cluster major update.
- TOL1 - Operator uninstallation.
- TOM1 - Cluster uninstall.

Table 5.3 Use case No. 2

Use case name	Basic cluster creation	
Use case ID	UCA2	
Traceability	CU2A	
Precondition	Installed Operator	
Scenario		
Step No.	Actor	Description
1	U	Use case starts with the Operator installed.
2	U	The user initiates basic cluster install.
3	O	The Operator installs the cluster.
4	U	Use case ends.

Table 5.4 Use case No. 3

Use case name	Monitoring installation	
Use case ID	UCC1	
Traceability	CU2A, CU2B	
Precondition	Installed Operator and created cluster	
Scenario		
Step No.	Actor	Description
1	U	Use case starts with created cluster.
2	U	The user initiates monitoring installation.
3	O	The Operator installs cluster monitoring.
4	U	Use case ends.

Table 5.5 Use case No. 4

Use case name	Vertical scaling	
Use case ID	UCD1	
Traceability	CU2A	
Precondition	Installed Operator and created cluster	
Scenario		
Step No.	Actor	Description
1	U	Use case starts with created cluster.
2	U	The user initiates vertical scaling.
3	O	The Operator scales the cluster.
4	U	Use case ends.

Table 5.6 Use case No. 5

Use case name	Horizontal scaling	
Use case ID	UCE1	
Traceability	CU2A	
Precondition	Installed Operator and created cluster	
Scenario		
Step No.	Actor	Description
1	U	Use case starts with created cluster.
2	U	The user initiates horizontal scaling.
3	O	The Operator scales the cluster.
4	U	Use case ends.

Table 5.7 Use case No. 6

Use case name	Connection pooling	
Use case ID	UCF1	
Traceability	CU2A	
Precondition	Installed Operator and created cluster	
Scenario		
Step No.	Actor	Description
1	U	Use case starts with created cluster.
2	U	The user initiates connection pooling installation.
3	O	The Operator installs connection pooler.
4	U	Use case ends.

Table 5.8 Use case No. 7

Use case name	Configuration update - extension installation	
Use case ID	UCG1	
Traceability	CU2A	
Precondition	Installed Operator and created cluster	
Scenario		
Step No.	Actor	Description
1	U	Use case starts with created cluster.
2	U	The user initiates Postgis extension installation.
3	O	The Operator installs Postgis extension.
4	U	Use case ends.

Table 5.9 Use case No. 8

Use case name	Configuration update - connections increase	
Use case ID	UCG2	
Traceability	CU2A	
Precondition	Installed Operator, created cluster, installed pooler	
Scenario		
Step No.	Actor	Description
1	U	Use case starts with created cluster and installed pooler.
2	U	The user increase connection pooler connections.
3	O	The Operator updates connection pooler configuration.
4	U	Use case ends.

Table 5.10 Use case No. 9

Use case name	Configuration update - max wal size	
Use case ID	UCG3	
Traceability	CU2A	
Precondition	Installed Operator and created cluster	
Scenario		
Step No.	Actor	Description
1	U	Use case starts with created cluster.
2	U	The user initiates max_wal_size parameter update.
3	O	The Operator updates max_wal_size parameter.
4	U	Use case ends.

Table 5.11 Use case No. 10

Use case name	Cluster scheduled backup	
Use case ID	UCH1	
Traceability	CU2A	
Precondition	Installed Operator and created cluster	
Scenario		
Step No.	Actor	Description
1	U	Use case starts with created cluster.
2	U	The user creates cluster backup schedule.
3	O	The Operator applies the cluster backup schedule.
4	O	The Operator creates cluster backup.
5	U	Use case ends.

Table 5.12 Use case No. 11

Use case name	Cluster ad-hoc backup	
Use case ID	UCH2	
Traceability	CU2A	
Precondition	Installed Operator and created cluster	
Scenario		
Step No.	Actor	Description
1	U	Use case starts with created cluster.
2	U	The user initiates ad-hoc backup.
3	O	The Operator creates cluster backup.
4	U	Use case ends.

Table 5.13 Use case No. 12

Use case name	Cluster restore	
Use case ID	UCI1	
Traceability	CU2A	
Precondition	Installed Operator and created backup	
Scenario		
Step No.	Actor	Description
1	U	Use case starts with created backup.
2	U	The user initiates cluster restore.
3	O	The Operator restores the cluster.
4	U	Use case ends.

Table 5.14 Use case No. 13

Use case name	Minor upgrade	
Use case ID	UCJ1	
Traceability	CU2A	
Precondition	Installed Operator and created lower minor version cluster.	
Scenario		
Step No.	Actor	Description
1	U	The use case with the Operator installed
		and lower minor version cluster created.
2	U	The user initiates minor version upgrade.
3	O	The Operator performs minor version upgrade.
4	U	Use case ends.

Table 5.15 Use case No. 14

Use case name	Major upgrade	
Use case ID	UCK1	
Traceability	CU2A	
Precondition	Installed Operator and created lower major version cluster.	
Scenario		
Step No.	Actor	Description
1	U	The use case with the Operator installed
		and lower major version cluster created.
2	U	The user initiates major version upgrade.
3	O	The Operator performs major version upgrade.
4	U	Use case ends.

Table 5.16 Use case No. 15

Use case name	Operator uninstallation	
Use case ID	UCL1	
Traceability	CU2A	
Precondition	Installed Operator and created cluster	
Scenario		
Step No.	Actor	Description
1	U	Use case starts with created cluster.
2	U	The user initiates Operator uninstallation.
3	O	Operator uninstalls but keep the cluster
4	U	Use case ends.

Table 5.17 Use case No. 16

Use case name	Cluster uninstallation	
Use case ID	UCM1	
Traceability	CU2A	
Precondition	Installed Operator and created cluster	
Scenario		
Step No.	Actor	Description
1	U	Use case starts with created cluster.
2	U	The user initiates cluster uninstallation.
3	O	Operator uninstalls cluster.
4	U	Use case ends.

APPENDIX A VI. TEST PROCESS NO. 5

Table 6.1 Test plan No. 5

Test plan ID	tp5
Revision	1
Introducton	Performance analysis
Test items	Operator deployed GKE in the cluster
Covered criteria	CP
Test type	Dynamic
Test approach	Performance analysis
Exit criteria	Completed analysis
Tools	PgBench Postgres benchmark tool
Delivarables	Performance reports
Duration	4 h for each Operator
Tester	Miroslav Šiřina
Start	9th May
Schedule	9th May: analysis and test report.

6.1 Test completition report

Testing performed: Performance analysis
Deviations from planed testing: SPGO analysis took too long
Test completion evaluation: None
Factors that blocked progress: Deploying SPGO proved challenging due to the fact that SPGO requires a cluster profile to deploy the cluster, which will specify the allocated processor and memory. With the correct settings of these values, the cluster was unable to find suitable resources for SPGO. By gradually reducing these values, available resources were eventually found.
Test Result Analysis: The tests provided valuable data about performanc of Postgres deployed by Operators.

6.2 Test results

6.2.1 PGO

pgbench (15.2)
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>

scaling factor: 1
query mode: simple
number of clients: 25
number of threads: 10
maximum number of tries: 1
number of transactions per client: 10000
number of transactions actually processed: 250000/250000
number of failed transactions: 0 (0.000%)
latency average = 45.879 ms
initial connection time = 745.406 ms
tps = 544.913924 (without initial connection time)
pgbench (15.2)
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 25
number of threads: 10
maximum number of tries: 1
number of transactions per client: 10000
number of transactions actually processed: 250000/250000
number of failed transactions: 0 (0.000%)
latency average = 46.016 ms
initial connection time = 688.436 ms
tps = 543.289895 (without initial connection time)
pgbench (15.2)
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 25
number of threads: 10
maximum number of tries: 1
number of transactions per client: 10000
number of transactions actually processed: 250000/250000
number of failed transactions: 0 (0.000%)
latency average = 46.424 ms
initial connection time = 750.453 ms

tps = 538.511794 (without initial connection time)

6.2.2 CNPGO

pgbench (15.2)
starting vacuum...end.
pgbench: error: client 6 script 0 aborted in command 4 query 0: FATAL: query wait timeout
SSL connection has been closed unexpectedly
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 25
number of threads: 10
maximum number of tries: 1
number of transactions per client: 10000
number of transactions actually processed: 240000/250000
number of failed transactions: 0 (0.000%)
latency average = 61.927 ms
initial connection time = 729.483 ms
tps = 403.698126 (without initial connection time)
pgbench: error: Run was aborted; the above results are incomplete.
command terminated with exit code 2
pgbench (15.2)
starting vacuum...end.
pgbench: error: client 21 script 0 aborted in command 4 query 0: FATAL: query wait timeout
SSL connection has been closed unexpectedly
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 25
number of threads: 10
maximum number of tries: 1
number of transactions per client: 10000
number of transactions actually processed: 240000/250000
number of failed transactions: 0 (0.000%)
latency average = 62.105 ms
initial connection time = 710.013 ms

tps = 402.544415 (without initial connection time)
pgbench: error: Run was aborted; the above results are incomplete.
command terminated with exit code 2
pgbench (15.2)
starting vacuum...end.
pgbench: error: client 1 script 0 aborted in command 4 query 0: FATAL: query wait timeout
SSL connection has been closed unexpectedly
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 25
number of threads: 10
maximum number of tries: 1
number of transactions per client: 10000
number of transactions actually processed: 240000/250000
number of failed transactions: 0 (0.000%)
latency average = 63.673 ms
initial connection time = 706.203 ms
tps = 392.629101 (without initial connection time)
pgbench: error: Run was aborted; the above results are incomplete.
command terminated with exit code 2

6.2.3 PPO

pgbench (15.2, server 14.7 - Percona Distribution)
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 25
number of threads: 10
maximum number of tries: 1
number of transactions per client: 10000
number of transactions actually processed: 250000/250000
number of failed transactions: 0 (0.000%)
latency average = 62.272 ms
initial connection time = 707.856 ms
tps = 401.464378 (without initial connection time)

pgbench (15.2, server 14.7 - Percona Distribution)
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 25
number of threads: 10
maximum number of tries: 1
number of transactions per client: 10000
number of transactions actually processed: 250000/250000
number of failed transactions: 0 (0.000%)
latency average = 63.769 ms
initial connection time = 690.586 ms
tps = 392.039997 (without initial connection time)
pgbench (15.2, server 14.7 - Percona Distribution)
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 25
number of threads: 10
maximum number of tries: 1
number of transactions per client: 10000
number of transactions actually processed: 250000/250000
number of failed transactions: 0 (0.000%)
latency average = 64.467 ms
initial connection time = 574.644 ms
tps = 387.793393 (without initial connection time)

6.2.4 SPGO

pgbench (15.2, server 15.1 (OnGres 15.1-build-6.18))
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 25
number of threads: 10
maximum number of tries: 1

number of transactions per client: 10000
number of transactions actually processed: 250000/250000
number of failed transactions: 0 (0.000%)
latency average = 87.906 ms
initial connection time = 60.527 ms
tps = 284.394442 (without initial connection time)
pgbench (15.2, server 15.1 (OnGres 15.1-build-6.18))
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 25
number of threads: 10
maximum number of tries: 1
number of transactions per client: 10000
number of transactions actually processed: 250000/250000
number of failed transactions: 0 (0.000%)
latency average = 89.321 ms
initial connection time = 67.533 ms
tps = 279.890815 (without initial connection time)
pgbench (15.2, server 15.1 (OnGres 15.1-build-6.18))
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 25
number of threads: 10
maximum number of tries: 1
number of transactions per client: 10000
number of transactions actually processed: 250000/250000
number of failed transactions: 0 (0.000%)
latency average = 80.863 ms
initial connection time = 74.577 ms
tps = 309.164610 (without initial connection time)
pgbench (15.2, server 15.1 (OnGres 15.1-build-6.18))