BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

CHAOS TESTING OF THE STRIMZI PROJECT USING THE LITMUS PLATFORM

TESTOVANIE PROJEKTU STRIMZI S VYUŽITÍM CHAOSU A PLATFORMY LITMUS

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. HENRICH ZRNČÍK**
AUTOR PRÁCE

SUPERVISOR **prof. Ing. TOMÁŠ VOJNÁR, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2022

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2021/2022

Master's Thesis Specification

Student: **Zrnčík Henrich, Bc.**
Programme: Information Technology and Artificial Intelligence
Specialization: Information Systems and Databases
Title: **Chaos Testing of the Strimzi Project Using the Litmus Platform**
Category: Software analysis and testing
Assignment:

1. Familiarise yourself with principles of Chaos engineering and tools supporting it, especially the Litmus platform.
2. Get acquainted with the Kubernetes, Apache Kafka, and Strimzi projects.
3. Design scenarios for chaos testing of the Strimzi project, including both already known possible experiments (such as deleting pods or network failures) as well as new ones designed specifically for Strimzi/Kafka.
4. Implement the proposed scenarios.
5. Using the implemented scenarios, perform experiments with a deployment of the Strimzi project that will simulate a production environment of its users.
6. Discuss the obtained results and their possible improvements.

Recommended literature:

1. Poulton, N.: The Kubernetes Book. Independently published, 2017. ISBN 978-1521823637.
2. Narkhede, N., Shapira, G., Palino, T.: Kafka, The Definitive Guide. O'Reilly Media, 2017. ISBN 978-1491936160.
3. Rosenthal, C., Jones, N.: Chaos Engineering: System Resiliency in Practice. O'Reilly Media, 2020. ISBN 978-1492043867.
4. LitmusChaos Authors: Litmus 2021, available online at https://docs.litmuschaos.io/. [checked Oct. 29, 2021]
5. Strimzi Authors: Strimzi -- Apache Kafka on Kubernetes, available online at https://strimzi.io/. [checked Oct. 29, 2021]
6. Red Hat, Inc.: Red Hat OpenShift, available online at https://www.openshift.com/. [checked Oct. 29, 2021]

Requirements for the semestral defence:

- The first two items of the assignment and at least some work on the third item.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Vojnar Tomáš, prof. Ing., Ph.D.**
Consultant: Stejskal Jakub, Ing., RedHatCZ
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2021
Submission deadline: May 18, 2022
Approval date: November 3, 2021

Abstract

The last decade in software engineering has seen a trend towards automation and abstraction with increasing use of micro-service architecture. The trend towards micro-service architecture has brought with it a need to rethink how we implement software quality assurance. Running micro-services in the cloud with multiple distributed components requires additional management of shared and inter dependent components. This in turn requires additional testing of the system's resilience. A possible answer is chaos engineering, which is often considered the next logical step after end-to-end and integration testing.

This thesis will focus on the gaps in testing created by the move to micro-service architecture and how chaos engineering can fill them. In particular it will focus on Apache Kafka deployed onto a kubernetes platform (Strimzi) and how the Litmus framework can be used to implement Chaos testing against this deployment. As our use-case was to have long running Kafkas deployed on kubernetes we had to adapt and extend the Litmus framework and build experiments that could test both long running kafkas and long running kubernetes. This thesis will demonstrate how we did this.

Abstrakt

Posledná dekáda v poli softwarového inžinierstva sa niesla v duchu automatizácie a abstrakcie. Vzostup nového spôsobu písania a menežovania softwaru (taktiež známeho ako architektúra mikroslužieb) so sebou taktiež priniesol nové výzvy v rámci zaručovania kvality softwaru. Beh systému v cloudovom prostredí s množstvom komponentov, ktoré sú roztrúsene po rôznych uzloch vyžaduje uvažovanie o závislostiach medzi týmito komponentami a dodatočné testovanie ktoré potvrdí odolnosť systému. Riešením je chaos inžinierstvo, často považované za logický krok po testovaní systému ako celku.

Táto práca sa zaoberá riešením problému nedostatočných možností pre aplikáciu chaosu (a to prostredníctvom projektu Litmus) do produktu Apache Kafka, ktorý je nasadený na Kubernetes platforme ako súčasť projektu Strimzi. Inými slovami, aby sme mohli aplikovať chaos na projekte Strimzi, či iných systémoch ktoré ho používajú, musíme vytvoriť úplne nové časti Litmusu. Čo sa samotnej aplikácie chaosu týka, fakt že Strimzi je systém sám o sebe, avšak často súčasť iných systémov, znamená že budeme potrebovať vytvoriť rozšírenejšie riešenia. Práca je zavŕšená výslednými experimentami a potvrdením odolnosťi projektu v reálnom nasadení.

Keywords

Apache Kafka, Kubernetes, container Orchestration, Kubernetes operators, Strimzi, Openshift, Distributed systems, Chaos engineering, observability, Litmus

Klíčová slova

Apache Kafka, Kubernetes, Orchestrácia kontajnerov, Kubernetes operátori, Strimyi, Openshift, Distribuované systémi, Chaos inžinierstvo, Pozorovatelnosť, Litmus

Reference

ZRNČÍK, Henrich. *Chaos Testing of the Strimzi Project Using the Litmus Platform.* Brno, 2022. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Ing. Tomáš Vojnár, Ph.D.

Rozšířený abstrakt

Posledné roky sa nesú v znamení veľkého posunu od monolitickej architektúry smerom k mikroslužbám. Tento prechod v architektúre a uvažovanie o software ide ruka v ruke s prechodom z datacentier na cloud. Okrem jasných výhod ktoré tato nová architektúra nesie (a ktoré väčšinou plynú z dekompozície) prináša táto nová architektúra aj nové výzvy. Cloud je okrem iného veľmi známy tým, že komponenty občas zlyhajú, čo môže byť zapríčinené čímkoľvek od mechanickej poruchy až po distribuovaný útok. Po prechode na cloud, skôr či neskôr začneme zažívať výpadky jednotlivých častí systému. Keďže sa tento problém naskytol v datacentrách pri monolitických aplikáciách len zriedkavo, a obvykle znamenal výpadok celého systému, väčšina systémov nemala praktickú skúsenosť s budovaním odolnosti voči týmto novým čiastočným a frekventovaným výpadkom. Jednou z odpovedí na tieto výpadky a nezvyčajné podmienky je práve chaos inžinierstvo. To môže byť definované ako úmyselné vytváranie turbulentných podmienok a sledovanie toho, ako na ne systém reaguje.

Každý spomenutý postup a prechod bol umožnený vďaka sérii technologických pokrokov v oblasti virtualizácie softwaru. Od inštalácie systémov na fyzické počítače sme sa cez virtuálne stroje a kontainerizované aplikácie dostali až k používaniu orchestračných nástrojov. Najznámejší z nich je Kubernetes, ktorý sa stal praktickým štandardom v oblasti orchestrácie softwaru. Toto globálne prijatie bolo hlavne vďaka dvom faktorom, a teda tomu, že Kubernetes je open source a takisto jednoduchému spôsobu ktorým ho je možné rozšíriť. Jedným z týchto rozšírení je aj projekt Strimzi. Ten umožňuje jednoduché nasadenie Apache Kafka v cloudovom prostredí. Samotne Apache Kafka si kladie za úlohu zjednodušovanie komunikácie medzi entitami, kde slúži ako prostredník pre prijímanie a posielanie správ (tzv. Event Streaming).

Cieľom tejto práce je poskytnúť možnosť jednoduchého aplikovania chaos inžinierstva pre ostatné systémy, ktoré Strimzi používajú, rovnako ako aj aplikovať tento Chaos na produkčný systém. Toto je docielené vďaka rozšíreniu ďalšieho Open Source projektu, ktorý sa nazýva Litmus. Litmus obsahuje viac ako päťdesiat existujúcich vzorov, ktoré môžu byť použité na vytvorenie konkrétnych druhov chaos experimentov.

Kedže Strimzi predstavuje sériu operátorov, jeho správne fungovanie zahrňuje množstvo udalostí a komponent. Tieto situácie su pomerne špecifické a preto vzory poskytnuté Litmusom nebudú dostačovať pre všetky prípady. Práca teda začína návrhom a implementáciou vhodných vzorov, z ktorých je následne možné vytvárať konkrétne experimenty. Tieto vzory budú zverejnené a voľne používané pre aplikáciu chaosu na projekte Strimzi. V momente kedy sú tieto vzory naimplementované, je možné ich použiť rovnako ako všetky ostatné už existujúce vzory, ktoré sú poskytnuté Litmusom. Tieto vzory sú použité pre implementácie sady testov, ktoré môžu byť použité voči akejkoľvek generickej konfigurácii projektu Strimzi. Poslednou časťou implementácie je príprava špecifických experimentov pre produkčné prostredie, v ktorom je Strimzi nasadené ako súčasť systému. Všetky tieto nasadania následne ukazujú odolnosť projektu Strimzi napriek turbulentným podmienkam, ktoré môžu nastať v prostredí ich nasadenia.

Chaos Testing of the Strimzi Project Using the Litmus Platform

Declaration

Hereby I declare that this Master's thesis was prepared as an original author's work under the supervision of professor Ing. Tomáš Vojnár Ph.D. The supplementary information was provided by Ing. Jakub Stejskal from Red Hat Czech s.r.o. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Henrich Zrnčík
May 20, 2022

Acknowledgements

I thank my supervisors, professor Ing. Tomáš Vojnár PhD, for his time and help, and to Ing. Jakub Stejskal for all reviews, materials, and guidance.

Contents

Chapter 1

Introduction

Chaos engineering (also referred as *chaos experimenting*) is often considered to be a logical step after end-to-end testing. The whole idea behind it is relatively simple; identify the steady-state in which the system works the way we expect it to, inject fault (Chaos), and verify regaining of the steady-state within a reasonable amount of time. Depending on this simple process, we either gain confidence that our system works correctly even under turbulent conditions or discover weaknesses. In other words, we want to make sure our system will withstand turbulent conditions, which naturally occur when we are running systems in production. Not every system requires the application of chaos engineering. The need for it has roots in new ways of running our systems, which are becoming more distributed than ever before, running on machines scattered around the globe. Distributed applications now resemble complex systems, mostly known for their unpredictable behaviour under specific conditions. Projects that had moved to the cloud from data centres (i.e., from infrastructure where the fantastic mean time between failure gave the impression that failure is indeed a rare thing) had to face *outages* and other unpredictable conditions on an almost daily basis. These traits of a system are the direct result of having numerous components which depend on one another, with multiple direct or transitive dependencies, as visible in Figure 1.1.

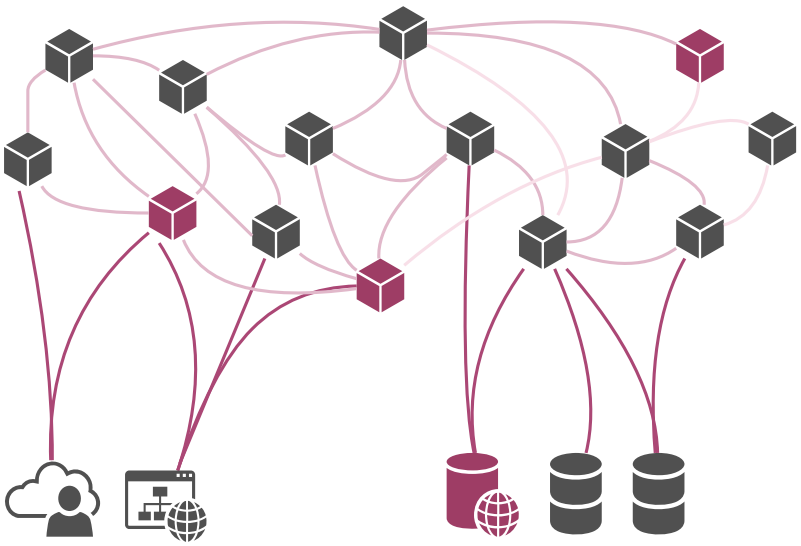
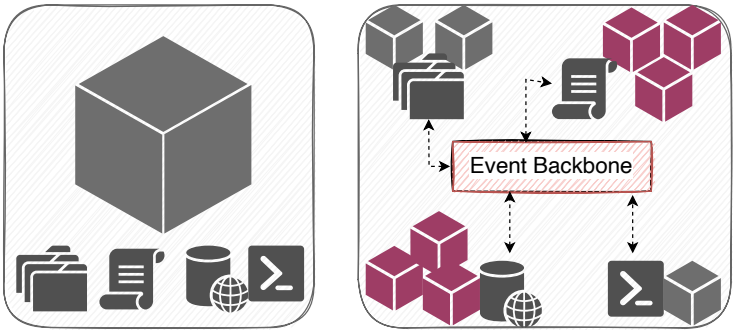


Figure 1.1: A realistic architecture of a microservice. Inspired by [22].

The mission statement for this thesis is; Provide a consistent way to apply chaos engineering principles to the Strimzi project. This is accomplished by creating experiments to target Kafka instances and their dependencies deployed with Strimzi. The first step to achieving our objectives is to understand the technology stack Kafka depends on. The project itself provides a way to run Apache Kafka on a Kubernetes cluster. Strimzi additionally incorporates Kafka and all manipulation necessary for its correct functioning. The trade-off for this functionality is abstraction and dependencies. As a result, the number of components that need to work and interact correctly increases exponentially.

The rising level of abstraction and components is rather a global trend. Projects that used to be monolithic, as illustrated in Figure 1.2 (a), undergo a shift to *microservices*, illustrated in Figure 1.1. Microservices represent an architectural paradigm where we build software from small components that interact with each other. These systems also require additional inputs to keep running smoothly (e.g., performing updates, scaling, deploying), exactly where tools for containerization and orchestration came in. Because of the growing number of components that need to communicate with each other, systems often incorporate entities that will simplify the communication. In most cases, this is an *Event Backbone* (e.g., Apache Kafka), also visible in Figure 1.2 (b).



(a) Monolithic application. (b) Microservices application

Figure 1.2: Different examples of application's architecture

The Mentioned shifts in architecture and deployment, especially if connected with moving to the cloud, bear one crucial trait; i.e., an application or service which runs there suddenly depends on a whole set of components, all of which have the potential to fail. Running an application in the Kubernetes cluster means that we have to think about countless additional cloud problems (e.g., network latencies, CPU hogs), Kubernetes events (e.g., node restarts), and other application-related problems. Despite our application being without a single bug, it may still behave unexpectedly in case of these turbulent conditions. Kafka and Kubernetes address the issue of failure and *disaster* recovery in their own way. After examination of relations between components and identification possible weakspots, we either implement new way how to simulate these conditions, or use already existing templates to do so. This step includes proposing chaos experiments, i.e., formulating hypothesis about our system's behavior, injecting chaos, and ensuring observability. When the chaos experiments are concluded, we will either have confidence in our deployment or have identified areas for additional engineering to improve resilience. Solution's suitable design and implementation require addressing many issues (e.g., configurability of underlying technologies, diversity of infrastructure), which may lead us to compromise in implementation and experimentation.

The implementation proved to be more extensive than initially expected. Originally two templates were proposed for future chaos experiments; however, we identified two additional needed templates, bringing the total to four. We implemented our four templates and afterwards used them alongside additional Litmus templates targeting generic and production systems. The application of these experiments confirmed that our Kafka deployments were resilient to its dependant underlying platforms and components being subjected to the application of faults. In addition to the feedback from the Litmus experiments, independent monitoring of the cluster verified the outcomes

The key contributions of related work lies in the pioneering implementation of chaos engineering principles in new levels of abstraction by applying chaos (and also providing an easy way for others to apply chaos) into a system which is to be used as part of other systems. The first and most crucial part is providing a generic way to apply chaos by nothing more than specifying a few key parameters for all other projects that use Strimzi. This is accomplished by combining implementation and know how concerning Strimzi and concepts and necessary steps in Litmus chaos. The given code is open sourced, and after merging to the main branch[1] templates will be usable also from the Litmus user interface. The second part is implementing the test suite, which can inject chaos into generic Strimzi deployment, implemented in such a manner that targeting different environments takes no effort and incorporating new chaos templates requires only a few lines of new code. The last contribution lies in applying chaos to the production environment while monitoring its impacts on the system's normal functioning. The combination of all these three contributions can then be summarized as the provision of an easy way to apply chaos to the project Strimzi for the general public, create new templates which could later serve for the creation of chaos experiments, and finally, the actual application of chaos into all sort of Strimzi deployment, including production environment.

The structure of the thesis is the following. Chapter 2 describes all preliminaries for this thesis. The purpose of this is to understand project Strimzi (described in Section 2.3). This project simplifies running Apache Kafka (an event-streaming platform) on Kubernetes. Kafka and Kubernetes are described in Sections 2.2 and 2.1, respectively. Kubernetes and its components are repeatedly used until the end of this thesis. Kafka represents underlying software which is the core of the validated system. Chapter 3 describes Chaos Engineering, its origin, and formalization. Afterwards, Section 3.2 contains details about the chaos framework Litmus. Chapter 4 contains all details about the design of experiments and proposed integration of application of chaos engineering on project Strimzi using the Litmus chaos framework. Afterwards, Chapter 5 covers all details about implementation, changes, and additions necessary made to the solution. Finally, Chapters 6 and 7 discuss evaluation, monitoring, and future ideas.

[1]Github repository with the code – https://github.com/henryZrncik/litmus-go/tree/xzrnci00

Chapter 2

Preliminaries

This chapter provides essential information about the technologies used in this thesis. A correct understanding of these concepts is the key factor to comprehend later chapters as they build on these preliminaries.

The first section of this chapter describes *Kubernetes*[1], which we can define as a platform for orchestration of container-based applications. Running applications on Kubernetes simplifies the shift towards microservices, as it solves countless problems (e.g., deployment, scaling) that have discouraged organizations from this shift.

The second section describes Apache Kafka[2], which is distributed event-streaming platform. It addresses several problems (e.g., tightly coupled services, different communication protocols, loss of information due to lack of possibility to replay events) of applications with multiple services and growing volume of data by providing an *event backbone* with durable event storage. The section is based on Kafka documentation [6], several books [28, 17, 19], and other resources.

The last part of this chapter describes project Strimzi[3]. Running Kafka correctly for a specific use-case is still quite a tedious task, and running it correctly on Kubernetes bears even more complications. Information provided in this section describes the motivation and functioning behind these technologies but is only deep enough for a reader to understand the concept and implementation of this thesis.

2.1 Kubernetes

The following pages describe why there was even a need for a platform such as *Kubernetes*, technological concepts, and paradigms that stand behind it. Lastly, its extendability, as it allows the creation of projects built on top of Kubernetes.

2.1.1 Early Stages

Every application runs on top of the operatin -system. Several new approaches were developed since running applications directly on a physical server. Each new one primarily focused on solving most emergent problems that the previous solution had failed to resolve:

[1] Kubernetes – https://kubernetes.io
[2] Apache Kafka – https://kafka.apache.org
[3] Strimzi – https://strimzi.io/

Physical servers

Developers and users knew exactly what software was running on each machine. Each server used to bear names after composers or gods from Greek mythology. Engineers treated servers like beloved pets[4]. When an application runs on a physical server, spawning more than one instance means that the admin has to take care of each instance separately, assign different ports and estimate memory and CPU consumption ahead [32]. Furthermore, there is no isolation between applications, and a crash of one may eventually bring down even the whole server. Most of the applications were large monolith[5] which are easy to deploy but hard to manage as project growth. In time, progress on work often started to stagnate, as no one could understand the whole repository, and the distance between developers and operation teams was enormous due to the need to deploy and manage every new dependency manually.

Virtualization

The next stage in the evolution of running applications was *virtualization*, i.e., the process of virtualizing the operating system on top of the running one. Software that creates and runs virtual machines is called a *hypervisor*. Thanks to it, we can host multiple virtual machines on a single physical server, each having separated resources, e.g., memory, CPU, and ports. Virtualization resolves security problems (in terms of isolation) and price, but virtualizing the whole operating system burdens memory and booting time.

Containerization

Containerization is considered to be *lightweight virtualization*. Unlike hypervisor virtualization, where one or more independent machines run virtually on physical hardware via an intermediation layer, *containers* instead run in userspace on top of an operating system's kernel. As a result, container virtualization is often called *operating-system-level virtualization*. Same as virtualization, it involves encapsulating or packaging up software code and its dependencies to run uniformly and consistently on any infrastructure. Nevertheless, there are a few key differences, described in the following lines and also visible in Figure 2.1, where we can see a solution for the deployment of a simple Java application that needs to communicate with the Redis database[6]. Containers come with the following advantages:

- **Container networks** – It is straightforward for containers to communicate, so there is no need to keep all parties that need to communicate within a single unit (container and virtual machine, respectively).
- **Unit of scale** – We can start to scale only parts which truly require scaling (instead of the whole virtual machine).
- **Size** – As there is no need to virtualize the whole operating system, we inherently save most of the space the operating system would otherwise need. Regardless, every container still has its filesystem and ports.
- **Fast booting** – As there is no longer a need to boot up the operating system.

[4]Cattle vs Pet is the term used to describe a new approach we should take towards servers. Instead of unreplaceable pets, which require much care, we should not depend on a specific server and instead, replace it automatically with a sign of the first problem.

[5]Entire app built into one executable/package

[6]Redis – https://redis.io/

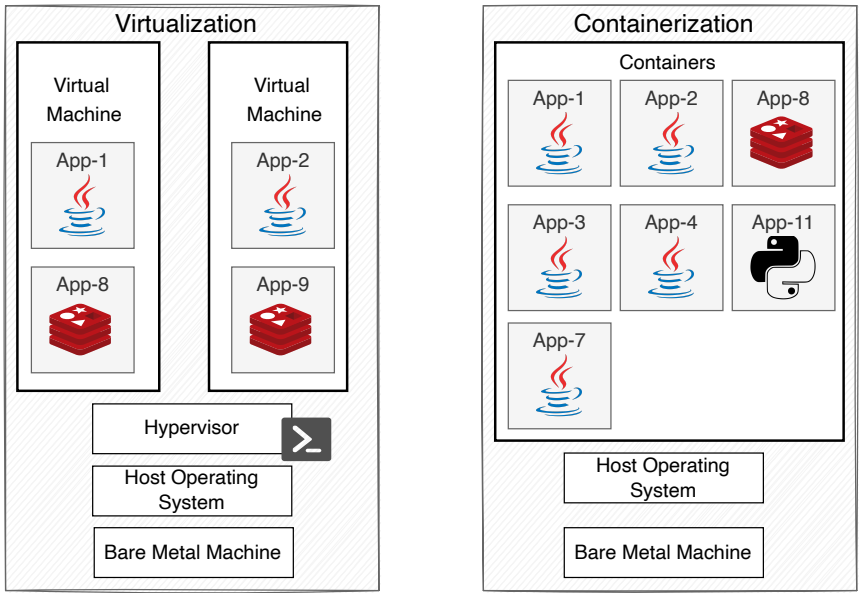


Figure 2.1: Virtualization (left side), containerization (right side)

One of the most used platforms for containerization is Docker[7]. The name comes from the British slang expression for a worker who unloads cargo from ships. Docker's simple architecture, shown in Figure 2.2. We create *images* which serve as a template for containers. We can define a container as a running instance of the image. Docker Architecture can be explained communication of following three entities:

- **Image registry** – Register of all available images, which can be downloaded and afterward used for the creation of containers or as a base images for new images.
- **Client**: Usually command-line interface client for interacting with the daemon.
- **Docker daemon** – Docker runs a process that waits for API commands from the client. Afterwards manipulates images and containers [34]. If it does not have the image for creating the container, it looks for it in the remote image registry.

Other containerization frameworks work in a very similar manner. For this work, it is not essential to understand Docker perfectly; instead, to be familiar with containerization principles because orchestration builds on top of it. A good example is a switch from incremental updates on one server with a growing stack of dependencies to a simple replacement of the image our application is built on. Instead of slowly fixing a single virtual machine or physical server, we create a new image. By doing so, we preserve history in case of need for *rollback* while considering the current version to be immutable.

2.1.2 Motivation for Orchestration

Google open-sourced the Kubernetes project in 2014, and it became part of CNCF[8]. It was described as a platform for managing containerized workloads and services. Today Kubernetes combines the best of ideas from the community and countless years of Google's

[7]Docker – https://www.docker.com
[8]CNCF - Cloud Native Computing Foundation, more at https://www.cncf.io/

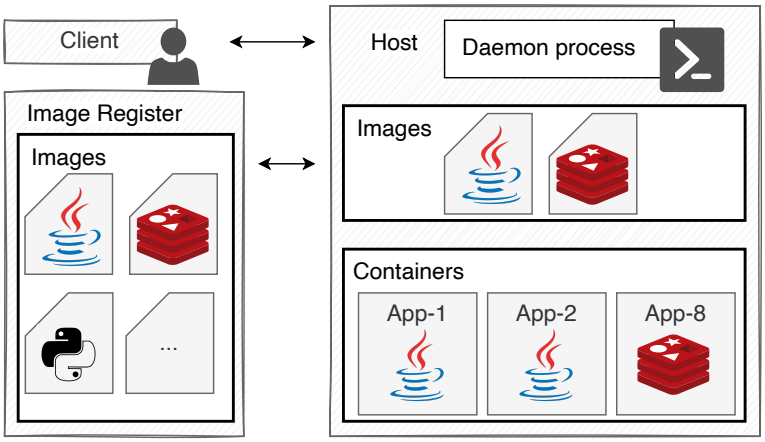


Figure 2.2: Docker architecture.

experience running production workloads at scale. The name originates from the Greek word, meaning pilot or helmsman[9] [7]. The following lines explain the reasons and the main benefits for the usage of *container orchestration* and specifically Kubernetes.

Declarative approach and abstraction

Kubernetes works in *declarative manner*, i.e., the user only specifies the desired state of the cluster. Kubernetes is cloud-agnostic; all underlying manipulating with infrastructure stays abstracted, as seen in Figure 2.3. This makes the deployment, scaling, and administration of components much more effortless. This serves as an alternative to the imperative configuration when we specify a set of steps that are supposed to lead to the desired state.

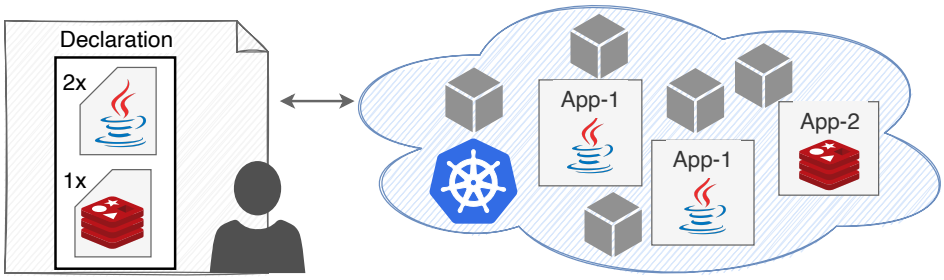


Figure 2.3: User only specifies the desired state, Kubernetes handles the rest

Uptime

With growing numbers of deployable components and larger data centres, it becomes increasingly challenging to configure, manage, and keep the whole system running smoothly. Furthermore, it is harder to figure out where to put each of those components to achieve high resource utilization and thereby maintain the hardware costs down [20]. In addition, the software industry has evolved to the level where services are updated hourly, and users

[9]Commonly used Abbreviation K8s instead, as there are eight letters between the first and last letter in word Kubernetes

expect constant uptime, even if the software they are running is changing constantly [11]. Using Kubernetes, there can be multiple updates within a single day while maintaining a highly available service.

Self-healing and scaling

When Kubernetes receives a desired state configuration, it does not simply take a set of actions to make the current state match the desired state a single time. Instead, it continuously takes steps to ensure that the current state matches the desired state [11]. Once there are more replicas than expected, it can also destroy them. The whole concept of self-healing and scaling goes even further with Operators, which are described in Section 2.1.6.

Others

Functional changes from embracing orchestration (with Kubernetes) imply organizational changes for different roles. For example, developers packaged the whole system and handed it to the operation team at the end of each long cycle. In case of failure, the operation team would migrate applications to a healthy server, often by hand [30]. However, today developers may deploy containerized applications themselves, migrations and scaling are done automatically, and administrators can work directly with the whole cluster. The trade-off for all of these improvements is that we need an understanding of orchestration and a dedicated cluster.

2.1.3 Architecture

Cluster structure

A Kubernetes cluster is a collection of hosts (nodes) that provide computing, storage, and networking resources. Kubernetes uses them to run the system's workloads [20]. The cluster comprises two types of nodes (namely, *master* and *worker*) as depicted in Figure 2.4.

- **Master node:** A Master node is the *control plane* of Kubernetes. It consists of 4 components. These components are often on the same node, but technically they can be on different nodes:
  - **API server** – API server exposes an HTTP API that lets end-users, different parts of your cluster, and external components communicate with one another [7]. Before this, each request first goes through authentication and authorization. API provides CRUD[10] operation for manipulation with all objects within the cluster.
  - **Scheduler** – Scheduler is component which finds a suitable worker node for the creation of the requested object, possibly also selecting the best one. Kubernetes generally tries to keep all nodes equally loaded regarding memory and CPU.
  - **Controller manager** – Cluster technically consists of multiple *Controllers* (specialized for concrete resources) such as services, deployments, stateful sets, namespaces, and other resources described in the following sections.
  - **Cluster store** – Cluster store is *cluster brain*, as the Scheduler and Controllers act based on data stored here. All communication with a store is possible only

[10] CRUD – is a commonly used acronym for Create, Read, Update and Delete operations on some resources.

via the *API server*. The store itself is based on etcd[11] database, which prefers consistency over availability and stores data as key-value pairs.

- **Worker node** – Node that provides resources, i.e., environment for running containers in the cluster is called Worker. To work properly it needs to run following three processes:
  - **Container runtime** – Container runtime is process needed by worker to pool images and create containers. Most often used options nowdays are docker and crio[12].
  - **Kubelet** – Kubelet is the process of Kubernetes itself. It *monitors* the API server for Pods that have been scheduled to the node, registers node as ready, and *reports back* all events which happen inside resources present in the concrete node.
  - **Proxy** – Originally, Kubernetes used a proxy process that accepted connections and afterwards directed communication to the Pod. Today, this process configures *iptables* to perform load balancing or more effective routing within the cluster.

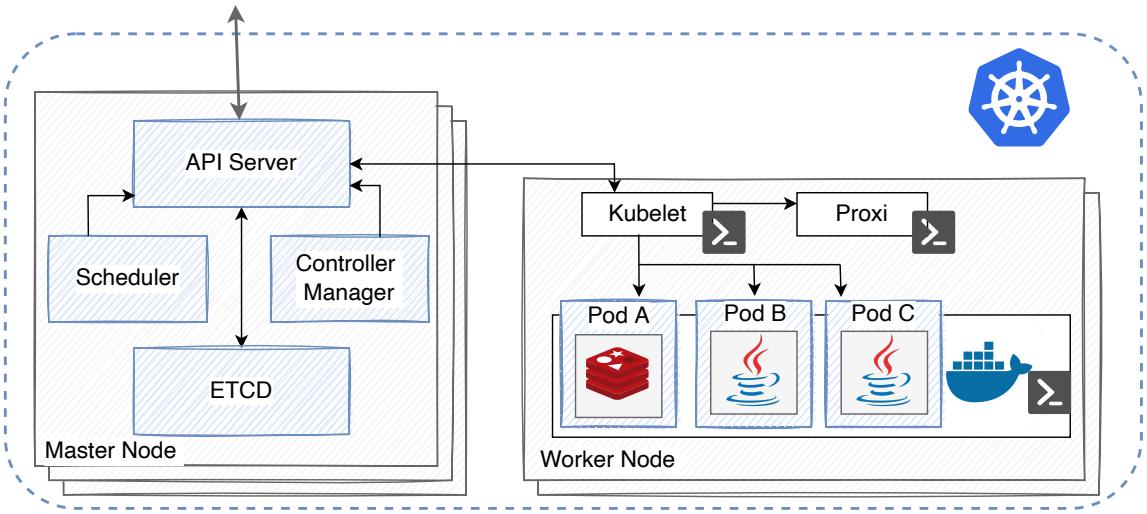


Figure 2.4: Architecture of Kubernetes nodes.

2.1.4 Objects

Almost every Kubernetes object includes four object fields that fully describe it. The object's `kind`, `metadata`, `spec`, and `status`. These fields describe what kind of object it is, its name, the object's specifications and its current status. The following lines describe most[13] of the objects that users either create or manipulate most often.

[11] Etcd – https://etcd.io/
[12] Cri-o – https://cri-o.io/
[13] Kubernetes ecosystem is much wider than is possible to capture within the scope of this work. Although the correct design and implementation will require a proper understanding of advanced concepts, it is not strictly necessary to be familiar with them to understand the implementation and ideas behind this thesis

Pods

The Pod is the *smallest unit of abstraction* and scaling. This allegory in name[14] is probably due to the representation of containerized applications as a whale. When we create a Pod, it contains one or many containers. IP is assigned per Pod, and containers within the same Pod can communicate using the localhost interface, visible in Figure 2.5.

```yaml
kind: Pod
metadata:
  name: pod-a
spec:
  containers:
   - name: redis
     image: redis
   - name: my-app
     image: java/app
     ports:
      - port: 5000
```

Listing 2.1: A yaml manifest of the Pod from Figure 2.5.

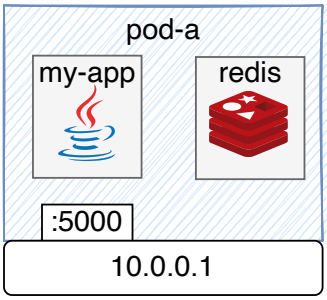


Figure 2.5: An example of a Pod with two containers and one open port.

The Pod itself is *ephemeral* and is considered healthy only in case all its containers work correctly. It is up to the Pod to execute its containers' readiness checks and health checks. These checks ensure that the application running inside the container still works as expected. Once the Pod dies, so do its IP address and data, which are the traits that the following two components try to resolve.

To deploy Pods (and all other Kubernetes resources) in a declarative manner, all the user has to do is specify traits of this Pod. Then, all it takes are a few lines of code (also shown in the Listing 2.1) and afterwards, apply it. Most of the resources are specified in YAML[15] format, which is most commonly used mainly due to its readability.

Services

The main trait of this component is its static IP. It does nothing else than accept and redirect traffic to selected[16] pods. We use Service whenever we need some static IP (e.g., connect to a database, provide IP outside of the cluster). As we can see in Figure 2.6, every Pod within the cluster may now communicate with the database or any other container under this Service.

Internally this works due to the Kube proxy, which adds rules to *iptable*; afterwards, the Service redirects traffic to a random Pod, which is the *Endpoint* of the Service. Figure 2.6 depicts a very simple type of service called *ClusterIP*. Other services which build on top of ClusterIp are *NodePort* and *LoadBalancer*. In a nutshell, the former type of Service allows external communication, and the latter adds load balancing.

[14] Pod's name comes from the pod of whales, which is like a herd of whales.
[15] YAML – https://yaml.org/
[16] Service knows which pods are of its concern thanks to the next Kubernetes concept called labels.

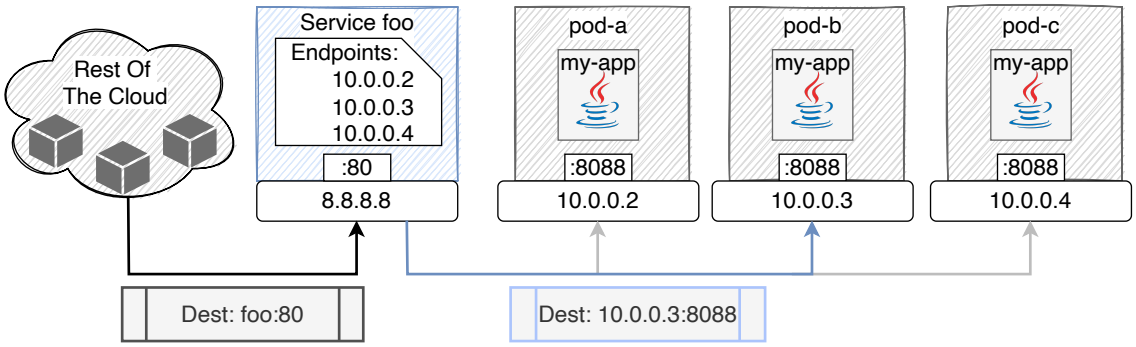


Figure 2.6: Relationship of Pods and Services.

Volumes

In Kubernetes, storing data permanently is something the user needs to take care of explicitly. Once Pod dies, all its data is gone. Some applications cannot afford to lose data, so they have to save data directly into some *persistent storage*, i.e., *Volume*. There are two types of volumes:

- **Local volumes** – An application running inside the Pod, *writes data directly into the node* it is running on, instead of virtual space. These kinds of Volumes are hostPath, gitRepo, and emptyDir. The user must keep in mind that if the Pod dies, data stays on the node, so if a new Pod is created on a different node, the application will be without original data.
- **Remote volumes** – These volumes may reside outside of the cluster; therefore, working with them is a bit trickier. Kubernetes provides a series of objects that help with mounting these volumes. A pod can use this kind of Volume with the help of *VolumeClaim*, which allows binding of the Pod's storage directly to Volume.

ConfigMaps

ConfigMap is an object for storing configuration data. These data are mostly the configuration of containers (pods). Although configuration can be passed as a *file*, *environment variables*, or *arguments*, all of these methods are *static*, and not very organized. Storing specific configurations as arguments or environment variables in manifest requires multiple images. Whereas configuration from ConfigMap can be referenced by key, so manifest does not need to change every time values of data do. We use ConfigMap for non-confidential data, but there is also another resource called *Secret*[17], for those that are confidential.

Namespaces

It is a resource that allows splitting and *categorizing of cluster* into multiple namespaces. In other words, it provides scope for the names of objects. However, not all resources are namespaced, e.g., Custom resource definitions 2.1.6, Nodes. Namespaces work well also for applying quotas for different parts and users of the cluster, with the help of additional objects such as RoleBindings and Roles. If a namespace is not provided, the default namespace is assigned.

[17] Work with them does not differ much from working with *ConfigMap*.

2.1.5 Controllers

Kubernetes is all about *Objects and Controllers*.*Objects* (e.g., Pod, Namespace, ConfigMap) are persistent entities in the Kubernetes system. They represent the state of the cluster. Controllers, on the other hand, are, in a nutshell, infinite loops that check the current state of the cluster, compare it with the desired state, and act upon it in case of any divergences [35]. Controllers' purpose is simplistically summarized in Figure 2.7.

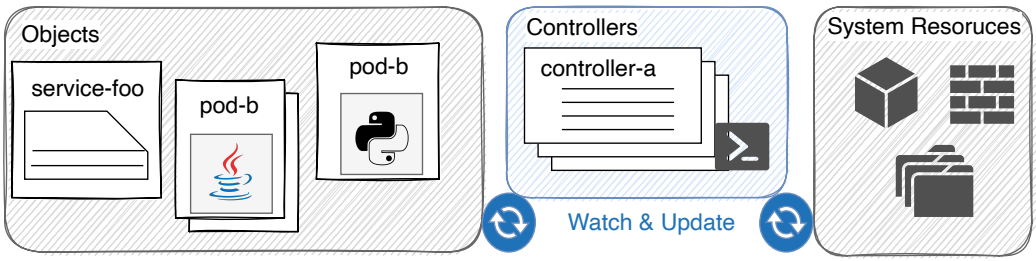


Figure 2.7: Controllers interacting with Objects and System resources.

The following lines describe some of the Controllers, their structure, and how they accomplish desired properties.

ReplicaSet

The role of *ReplicaSet* is to make sure that the exact number of healthy Pods that match its selector exist at a given moment, also depicted in Figure 2.8. When Pod stops working, ReplicaSet replaces it with a new one. This process is known as *self-healing*. There are two more Controllers of a similar kind. *ReplicaController*, which is the predecessor of ReplicaSet, but was marked as deprecated and slowly replaced by ReplicaSet, which has more advanced options regarding selectors. The next one is *DaemonSet*, which ensures the existence of precisely one Pod of the desired type per node. ReplicaSet, just like Pod, is not often used alone. Instead, as part of higher abstraction, specifically Deployment or Stateful.

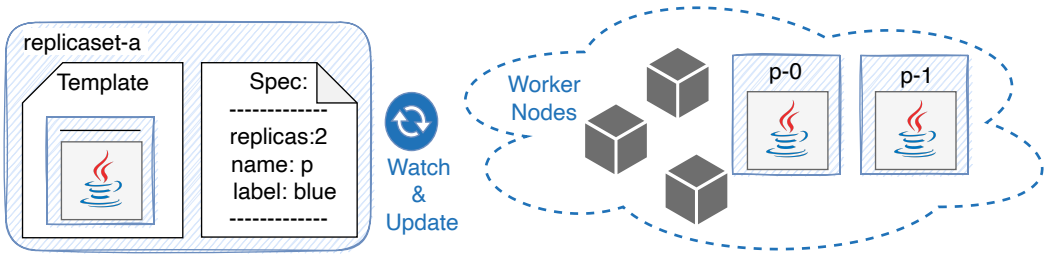


Figure 2.8: ReplicaSet with 2 instances

Deployment

The need for this component arose from the often-repeated task of updating applications. We would either need user interaction or a script to do that to accomplish this. Both of these solutions are prone to errors, the latter due to possible network problems. Deployment

provides all that ReplicaSet (it internally uses ReplicaSets), but also additional operations, i.e., *Rolling Update* and *Rollback*. The rolling update allows the user to update his application (using a new image). The exact behaviour of Deployment depends on the strategy we choose. Still, all it essentially does is that it creates a new ReplicaSet and updates the old one to hold zero instances (replicas). Old ReplicaSet will delete all its instances afterwards, while the new one creates its own with the desired version of the application. Deployment keeps track of its former ReplicaSet, as visible in Figure 2.9, which makes the switch to a previous version (i.e., *Rollback*) very simple.

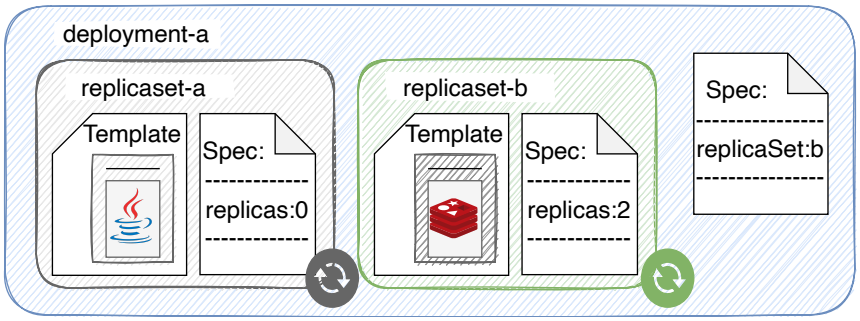


Figure 2.9: Deployment with its ReplicaSets.

Deployment is a simple option for many applications, but almost all are stateless. All pods within the same Deployment share the same Volume (claimed by the same VolumeClaim), which does not allow for the scale of stateful application. When ReplicaSet scales the number of replicas down, we do not know which instances it removes. These traits are not shortcomings, but this behaviour is insufficient for most larger stateful applications, which is also motivation for the last major Controller.

StatefulSet

StatefulSet works almost like deployment but with following key differences:

- **Pod identity** – Whereas pods managed by *Deployment* have a random hash appended as a suffix of their name, pods managed by StatefulSet are ordered with a zero-based index.
- **Scaling** – Pods are scaled up and down in order, based on their index. We always know which Pod will be deleted.
- **Volumes** – Each pod has its own VolumeClaim (and volume). Each Pod has its private persistent storage; if this Pod dies, a new one will take its claim. This storage is preserved even in case the number of Pods is scaled down, so data are once again available once this number goes back.

Job

The Job creates the specified number of pods and continues to retry the execution of their work until the specified number of Pods terminates successfully. We typically use it to execute some work just once[18], e.g., database backup.

[18]There is also another Controller called CronJob, which works the same way as Job, but it runs periodically.

2.1.6 Extensions

Although StatefulSet and Deployment provide enough options for running all sorts of applications, they alone would not be beneficial for more extensive applications with some extra logic. For example, if we had some Postgres applications with three replicas, each having its Volume, we would still need to synchronize these databases independently. In addition, each application may have its unique requirements regarding *starting, scaling, and self-healing*. These operations would typically require the user's (Operator's) intervention. Because the main ideology of Kubernetes is automatization, Kubernetes solves this problem by *Operators* and *custom resources*, which are described in the following subsections.

Custom resource Definitions and custom resources

Custom resources are *API extension mechanism in Kubernetes* [15]. An additional type of object called *CustomResourceDefinition* defines the *CustomResource*. After applying CustomResourceDefinition, we are capable of deploying these new types of objects defined by CustomResource. Once we create a *CustomResourceDefinition* for our new resource and then create this resource with the help of *CustomResource*, this resource will have no other use than being some structure of data. The whole concept of CR and CRDs[19] was invented to provide more domain-related objects, which can be manipulated by *Operators*. All of these can be seen in Figure 2.10, where we provide CRD for the creation of websites and accept specifications describing replicas and URL properties.

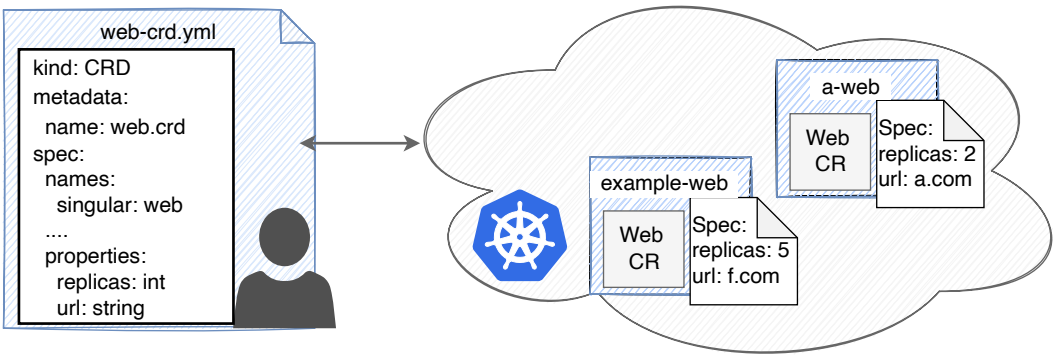


Figure 2.10: User firstly provide CRD describing new resource (i.e., website), than create two new resources of this kind.

Operators

A Kubernetes Operator is an *application-specific Controller* that extends the functionality of the Kubernetes API to create, configure, and manage instances of complex applications on behalf of a Kubernetes user [2]. Unlike other Controllers, it also has domain-specific knowledge and acts upon *CustomResources*. Concerning CRs and CRDs, we can rewrite the definition of Kubernetes Operator. An Operator is a custom Kubernetes Controller watching a CR type and taking application-specific actions to make reality match the specification in that resource, formally written as Algorithm 1. As a result of this, users

[19] CR and CRDs are commonly used abbreviations for CustomResources and CustomResourceDefinitions

can afterwards work directly with new CRs, like *Kafka, ChaosEngine, Website, or any other resource, which is much more domain-specific in comparison with trying to do so with pods and other Kubernetes native primitives.*

Algorithm 1 Operator. The Desired state is given by custom resources

```
1: procedure MAIN                                   ▷ Every operator in a nutshell.
2:     while true do                                ▷ Operator continues in endless loop
3:         currentState ← getCurrentState()
4:         desiredState ← getDesiredState()
5:         if currentState ≠ desiredState then
6:             makeChanges(currentState, desiredState)
```

These actions can be visible in the fictive example operator in Figure 2.11. The Operator's manipulation can be done using HTTP communication with the Kubernetes API server, from within or outside the Kubernetes cluster. However, we will mostly see Operator running inside a Pod which is within Deployment [27].

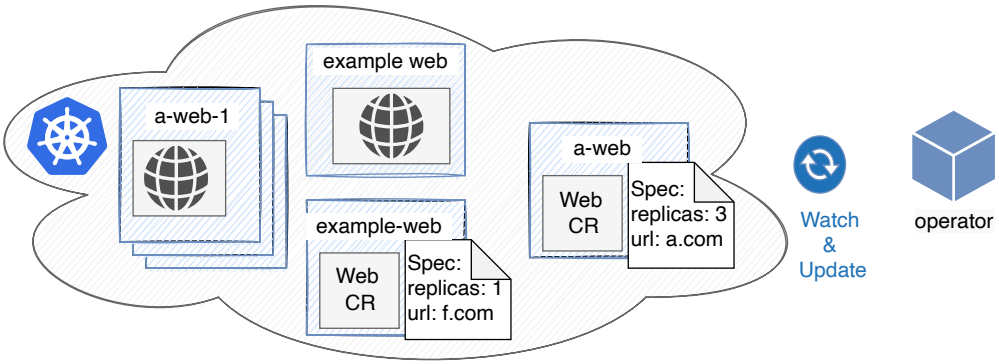


Figure 2.11: An Operator works on our newly created CRs. It uses values from specification, creates new pods with specified number of replicas which then start serving requests for specified url or any other reasonable activity.

Over the last years, Kubernetes has become a *standard for orchestration*. Numerous benefits for all members of the team, e.g., developers and administrators. These benefits are self-healing, easy deployment, declarative approach, and advanced automatization gained widespread popularity. This shift can be seen in platforms as well. Example of it is Red Hat Openshift[20] platform, which is completely managed by OpenShift operators [13]. The extendibility that Kubernetes provides, which is very benevolent regarding development (choice of language or need to be present in the cloud), allowed the start of many projects[21] through the use of operators; Both Strimzi 2.3 and Litmus 3.2 are examples of them.

Deployment of an operator on a cluster requires the provision of several other resources (besides operator and custom resource definitions); these are described in Appendix B.

[20] Openshift – https://www.redhat.com/en/openshift-4

[21] Operatorhub – https://operatorhub.io/ is the hub with numerous currently available Kubernetes native technologies, all managed by operators.

2.2 Apache Kafka

Kafka (Apache Kafka) can, in its simplest form, be defined as a *event-streaming platform* [19]. In other words, it captures data in real-time from event sources, e.g., databases, sensors, cloud services, or mobile devices, in the form of a stream of events (which is a durable, ordered, and unbounded sequence of records). Kafka may therefore be used whenever we work with lots of logs, e.g., website activity tracking, metrics monitoring, and message brokering.

Section 2.2.1 starts with basis, which are further explored in Section 2.2.2 with focus on clients. Afterwards Section 2.2.3 describes architecture of Kafka cluster and whole description of Kafka ends with additional description of its extensions in Section 2.2.4.

2.2.1 Basis

One of the authors, Jay Kreps, explained that name Kafka was given after somebody who wrote a lot, as it is a system optimized for writing. As for language, the authors decided to write Kafka in Java because of the possibility of running it almost everywhere [33]. Back before 2011, most companies did not need to solve the problem with communication of many services; at LinkedIn[22] it became a problem of much higher magnitude. As we can see in Figure 2.12, there is a tight coupling even between very few communicating components, not to mention the need to know all of the others' services protocols for communication.

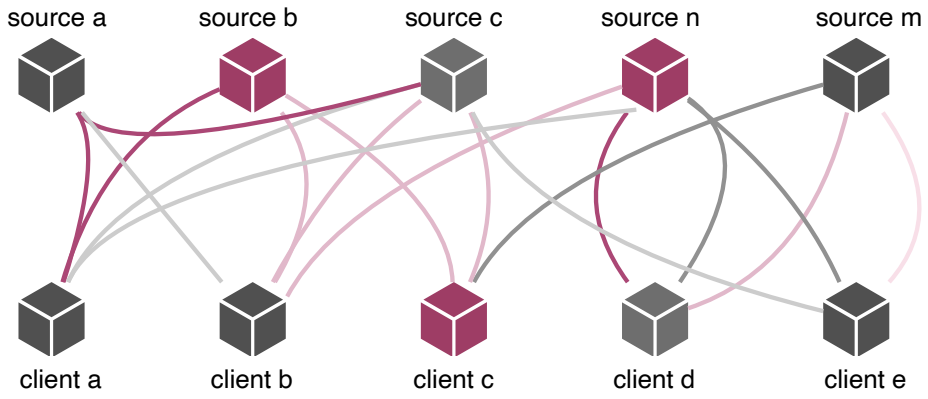


Figure 2.12: Point-to-point communication between increasing number of services

Mess in communication led to the usage of *Event-driven architecture*[23]. Instead of direct communication between applications, they (i.e., entities that either emit or consume data) communicate only using *channels*. This channel is Kafka, and it consists of entities called Brokers[24]. Because both the Producer and Consumer can communicate only with Kafka brokers (which serve as a hub), there is no longer *coupling* between them (Producer does not even know if there is some consumer [24]). The Following figure 2.13 shows communication (using Kafka) between applications.

[22] Linkedin – http://www.linkedin.com is an American business and employment-oriented online service.

[23] Event-driven paradigm – paradigm promoting the production, detection, and reaction to events

[24] A person who buys and sells goods or assets for others.

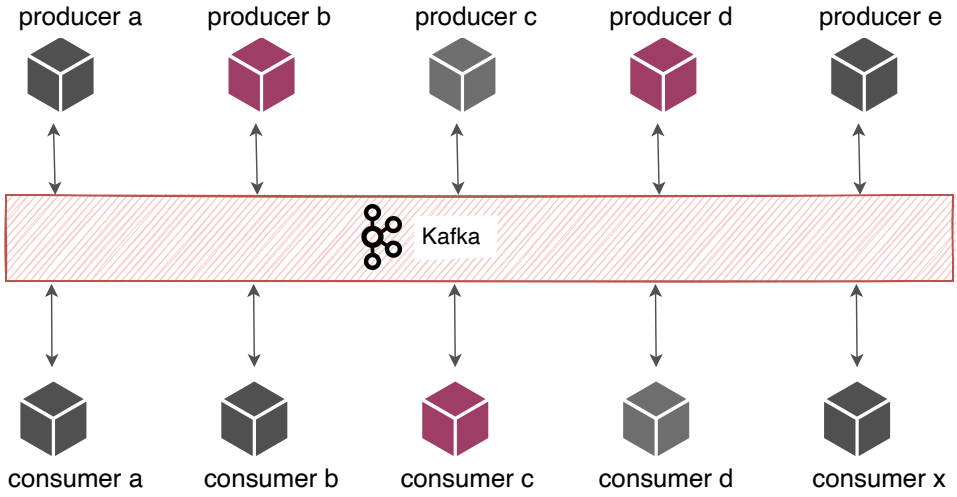


Figure 2.13: Communication of several application through Kafka.

Data

Kafka splits data into *topics*. Database analogy for the *topic* would be a *table*, but the catch here is that data in the topic are not necessarily *homogeneous* [28]. The important thing in Kafka is that data inside the topic are further split into *partitions*. The whole reason behind the split is that it will allow us to achieve *higher level of parallelism*. Each partition can be seen from a user's perspective as an *append-only log*. Because Kafka is an event streaming platform, it has to store streams in total order. This constraint does not allow for big scalability, and therefore total order is guaranteed only within partitions. Each record has its offset (a unique incremental number that represents the order of record within the partition), which identifies each record inside the concrete partition, as can be seen in Figure 2.14. Communication itself takes place in the form of messages. Each message may consist of multiple records, and each record consists of a header, data, and key. Data itself are optional, as well as key and header[25]. The key's purpose is that records with the same key end up in the same partition.

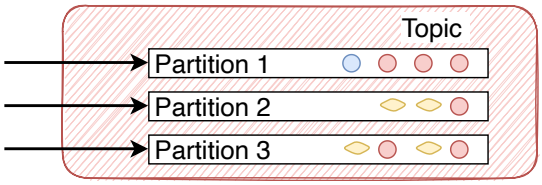


Figure 2.14: A Kafka topic with its partitions.

There are four main traits when it comes to data in Kafka:

- **Scalability** – Splitting topics into partitions makes it is easy to split the load inside the cluster. We can talk (read/write) to different brokers while still working on the same topic. Whole communication in Kafka Ecosystem is quite more complicated and will be further explained in Section 2.2.3

[25] Unless we are debugging or using some canary service, the absence of data is rarely the case

- **Durability** – Kafka is *highly configurable*, which in terms of durability means that we can configure Kafka to keep data on disk instead of memory *based on time, size, or even both of these parameters*. We can configure Kafka to keep data forever.
- **Safety** – Kafka allows *replication of partition* with custom factor n, which means that each partition will exist on n brokers, and thanks to that, data can withstand up to $n - 1$ broker failures at the same time without losing data.
- **High throughput** – Kafka gains this trait thanks to the way it stores the data. It is an append-only log (file) with a unique offset.

2.2.2 Clients

Kafka allows three main ways to produce or consume data in the Kafka ecosystem, i.e., Consumer, Producer, and Streams libraries. Both *Producer* and later described *Consumer* are implemented in various programming languages (e.g., Go, Python, Java); since Kafka is written in Java, it is also the most up-to-date version of these clients.

Producer

The Producer produces messages into a known set of *topics*. It can choose whether it wants to produce *records* into a specific *partition* or leave the decision up to Kafka. The algorithm used to split unspecified records into partitions is Round Robin. Besides expected configuration, such as addresses of servers or serialization, we can configure much more properties. All relevant configuration is explained in Appendix C.

Consumer

It is a set of libraries that allow easy implementation of an application that reads data (records) from Kafka. *The Consumer* can read topics from the broker on its own, but most often, Consumers form *Consumer Group*; it is a set of Consumers, which are each consuming their disjoint part of partitions within the same topic. We distinguish Consumer groups by their unique id. There can be multiple consumer groups per single topic (as it only means that there are multiple readers). Consumer alone or consumer group with at least one Consumer has to read all the partitions within the consumed topic. The reason for the formation of consumer groups is that it allows concurrent consumption of data within the same topic (mainly because a single consumer cannot handle the volume of produced data on his own). As depicted in Figure 2.15, every consumer group must consume the whole topic. Furthermore, a Consumer can join or leave a consumer group dynamically. Both of these events cause another event called *partition rebalance*, i.e., reassignment of partitions. If the Consumer leaves the group, the group will split its partitions amongst the rest of the active consumers. If a new Consumer arrives within one group, partitions will be reassigned to obtain an approximately fair number of consumed loads for each Consumer.

To keep data consumption within a consumer group in a relatively stable manner, consumer besides consuming data covers two extra functionalities:

- **Commit offset** – Based on the configuration (Appendix C), the Consumer saves information about its offset within the partition. Although the Consumer knows which data he should ask for next, Kafka's broker needs to know this information independently of the Consumer. For example, in case of failure of this informed Consumer, a new consumer would know how much data from the topic had already

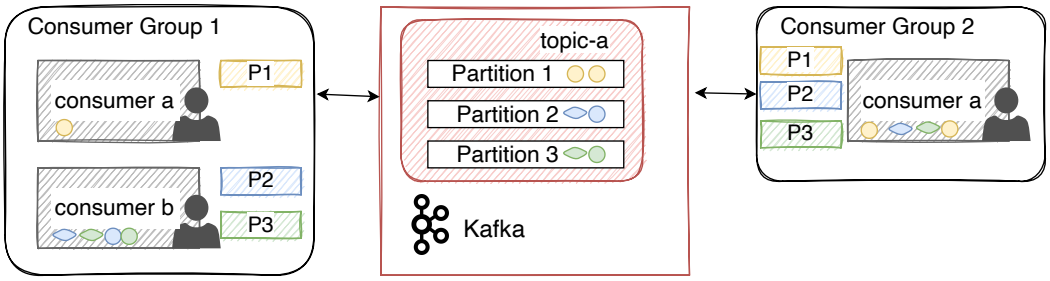


Figure 2.15: Communication using Kafka cluster

been read. Internally this is done by saving positions for pairs of consumers and partitions into the special topic called ___consumer_offsets.

- **Send heartbeat** – Consumer informs Kafka about his ability to consume data. It is done by reading data or periodically sending a message, i.e., heartbeat.

2.2.3 Kafka Cluster

Kafka cluster is a group of *Kafka broker instances*. Each of the brokers within the cluster is identified with a unique, zero-based index. There are several roles that a broker may take in the cluster (as can be seen in Figure 2.16), and depending on these roles, the broker is also assigned different *responsibilities*:

- **Controller** – One broker is always elected to be the *cluster controller*. Its main responsibility is to make sure other nodes work correctly. Besides that, also *assign partition leaders* and their followers for concrete topics.
- **Partition leader** – Broker that *handles all writes and reads* for concrete partition. Besides partition leader, there is also a role called the preferred leader. This broker is the one that would be the optimal leader from the perspective of effectiveness.
- **Partition Follower** – Brokers (in the context of a concrete topic) that only replicate messages from a specific topic's partition leader is called a follower. These serve as backup.
- **Consumer group coordinator** – The Coordinator is the broker responsible for getting heartbeats from a concrete consumer group. Heartbeats should signalize that these consumers are still active. If a consumer does not respond, other consumers from the same group will take care of what was formerly its partitions.

Zookeeper

Till Kafka version 2.8.0, it was heavily dependant on Zookeeper[26]. Its responsibility was cluster management (e.g., the election of the cluster controller and keeping track of active brokers). From version 2.8.0 on, the Kafka community is working on getting rid of Zookeeper[27] and tries to *manage all metadata internally* in a single dedicated topic.

[26] Zoeekeper – https://zookeeper.apache.org/

[27] KIP-500 – https://cwiki.apache.org/confluence/display/KAFKA/KIP-500 (Kafka improvement proposal 500) proposes the removal of Zookeeper and replacing it with self-managed quorum

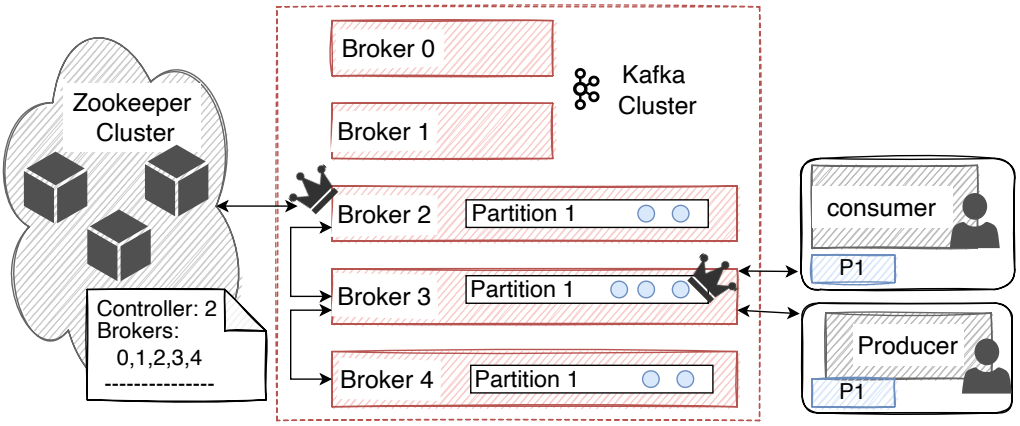


Figure 2.16: Communication and roles within Kafka cluster. Broker 2 is controller, and together with broker 4 it is also partition follower. Broker 3 is partition leader.

2.2.4 Kafka Connect

When Kafka came out, the already mentioned API was fundamentally all it provided. Using Producer and Consumer alone, we can accomplish things such as storing/loading data to/from applications that communicate via Kafka protocol, copying all data from one Kafka cluster to another one, or additionally processing these data within seconds since their creation. All these tasks would be tedious and require a lot of repeated code. Over the years, this led to the implementation of several extensions. For the sake of this thesis, we will focus on one particular extension, which is Kafka Connect. The rest of them are briefly described in Appendix D.

Kafka Connect is all about *data integration*. Almost every system needs data from other systems or data sources. Therefore, we will need to write much code that will help with transferring and transforming data to a desirable state, which will eventually lead to high coupling and divergence from focus on business logic [10]. *Kafka Connect* is a data integration framework that provides a scalable and reliable[28] way to move data between Kafka and other data stores, possibly even different combinations of stores, while serving as a middleman. This makes it the perfect choice for transferring data from one database to another, aggregating data from different sources, and all of that with minimal coupling between individual sources.

We need to address the problem here: each data store or application that can provide data may have a different way of storing and accessing them. Therefore Kafka Connect uses *plugin architecture*. The only thing which a developer, or generally any organization, which is interested in the creation of these plugins, has to do, is to extend *Connect* and *Task* Java classes. Connect plugins are currently created and managed by the community or authors of different databases, data stores, and services. On one side, it provides enough *flexibility* to handle the diversity of end objects. On the other side, the amount of *additional configuration* may become overwhelming [8]. The following list describes all components which we either need to take care of or are part of the Kafka Connect framework (see Figure 2.17):

[28] Currently, at least once delivery, which means that data may end up delivered twice.

- **Worker** – The cluster itself comprises workers. When we are talking about *Kafka Connect cluster*, we mean a set of workers which identify themselves with the same group id parameter. These workers are *fault-tolerant and self-managed*. If one worker crashes, others split the work amongst them.
- **Connector plugin** – A *Kafka Connect plugin* is a set of JAR files containing the implementation of one or more connectors, transforms, or converters [4]. Depending on whether we want to store data from source to cluster or the other way around, we implement either *SourceConnector* or *SinkConnector* classes. The main responsibility of this class is to *manage tasks (i.e., start, configure, split the load, and overall manage)*. For example, the JDBC source connector will connect to the database, discover the existing tables to copy, and, based on that, decide how many tasks are needed [28].
- **Connector** – Once a plugin is installed in the cluster, we may configure details specific to our needs and thus create our Connector (using REST API). Each connector instance *coordinates a set of tasks* that copies the data from source to Kafka or vice versa [5]. The Connector passes configuration to each *task*, so it can work individually as a separate process.
- **Task** – Notion of Task is the same as in the case of Connectors, there are two types. *SinkTask* for SinkConnector and *SourceTask* for SourceConnector. Its responsibility is to communicate (transfer data) with the data store according to the configuration given by the Connector. Splitting the workload of transferring data into multiple tasks is how Kafka Connect provides built-in support for parallelism. It is mostly up to *Connector plugin* (implementation) to decide how many tasks we can start for a specific Connector. There is also configuration parameter tasksMax for the upper bound of task count.
- **Convertor** – The worker uses *Convertor* to convert data into appropriate format (e.g., JSON) when writing to or reading from Kafka.

All that is left to the user (at least in most common use cases) is to provide correct configurations. This means that data integration can now be solved in a simple, declarative manner thanks to Kafka Connect and already implemented plugins.

2.3 Strimzi

This section describes project Strimzi, its components, architecture, and place in Cloud Native world. Information is primarily based on Strimzi documentation [8] and several blog posts. Text will build on knowledge about Apache Kafka and Kubernetes, described respectively in Sections 2.2 and 2.1.

2.3.1 Origin and Motivation

Project Strimzi, whose aim is to *simplify the process of running Apache Kafka in a Kubernetes cluster*, started in 2017 as the work of three developers. From that time onward project gained considerable traction, and since the year 2019, Strimzi has been a part of the Cloud Native Computing Foundation projects. By the time of writing these lines, several big tech companies use Strimzi in the production environment [14].

Strimzi brings Kubernetes native experience for running Kafka by providing a *collection of Operators* which simplifies work with Kafka [8]. The term "Kubernetes native" means

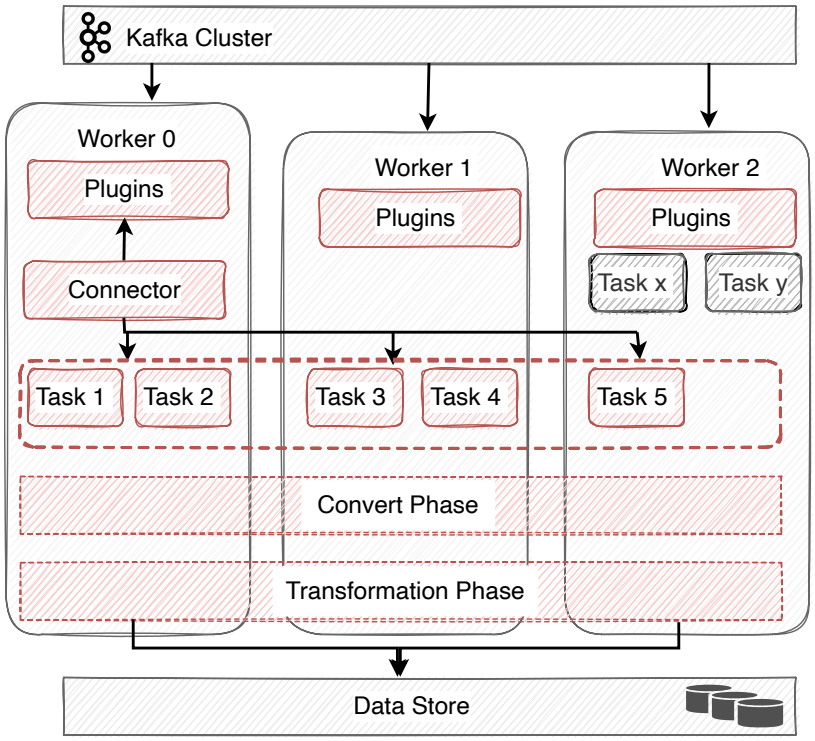


Figure 2.17: Kafka Connect architecture. Inspired by Strimzi documentation [8].

.

that we are referring to tools and applications that have been built *specifically to be run on Kubernetes platforms*, and thus making full use of Kubernetes API and components. Strimzi additionally supports Kafka by deploying and running a Kafka cluster, managing access, brokers, topics and users. Some reasons why to use Strimzi (as well as challenges Strimzi has to overcome) are summarized in the following points:

- **Straightforward routing and broker failure handling** – Once the Kafka broker dies, its replacement needs to take its exact place (its configuration) to be recognized as the same instance being recreated instead of a new broker added.
- **Updates** – If we run Kafka Pods as part of the Deployment, we can change the image and run *rolling update* (Section 2.1.5)
- **Readiness and liveness** – Alive Kafka broker and the ready one can be two different things. A container that does not match the liveness condition is restarted, whereas a ready container (i.e., Pod) can be added as an endpoint to the appropriate Service.
- **Abstraction and automatization** – Are accomplished with usage of the Operator Pattern (Section 2.1.6). This may go as far as to simplify the whole management of Kafka and its components by updating a few custom resources.

2.3.2 Architecture

Due to Strimzi's *Kubernetes native nature*, we can think of its architecture in terms of Operators, custom resources it operates upon, and of course, Kafka's parts.

Operators

Project Strimzi consists of four *operators*. As already mentioned(Section 2.1.6), when we simplify it, all Operator does is manage custom resources due to changes in the cluster and vice versa. Specific operators are described in the following lines:

- **Cluster Operator** – This is the primary Operator that *operates upon the whole Strimzi cluster, deploys Entity Operator, Apache Kafka cluster*, and all other components already described in Section 2.2, e.g., MirrorMaker, Kafka Connect. When we omit all extra resources, once this Operator is deployed, our cluster would look like the one on Figure 2.18

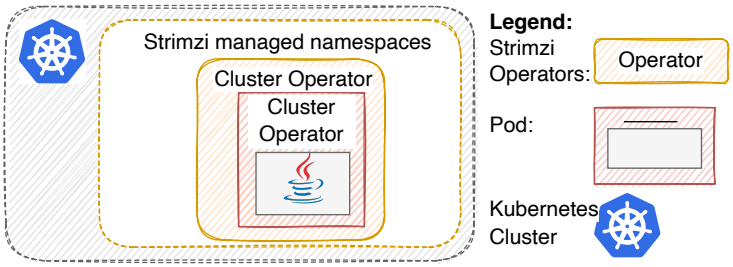


Figure 2.18: Deployed Strimzi Operator runs as a pod which does nothing more than monitor its namespaces in order to operate upon newly created custom resources.

- **Entity Operator** – This Operator is deployed by Cluster Operator. It Comprises User and Topic Operators.
  - **Topic Operator** – Topic Operator manages Kafka topics. The custom resource representing topics from Kafka is *KafkaTopic*. Manipulations such as deletion, creation, or change of specification of a topic in Kafka brokers are reflected to concrete resources and vice versa[29]. This means that Topic Operator watches for any changes to the topics inside the Kafka cluster and reflects these changes in KafkaTopic custom resources. The same kind of reflection applies to the other way. The Topic Operator reflects these changes to the Kafka cluster if a KafkaTopic is created, deleted, or updated.
  - **User Operator** – User Operator manages Kafka users. Acts in the same way as Topic operator, but regarding *KafkaUser* resources and only in one direction. Unlike topics, users in Kafka are not expected to be manipulated by both Kafka and custom resources. Parameters such as quotas, authorization (access control lists and support for Keyclock or Hydra[30] servers as well) and authentication (TLS, SCRAM-SH) mechanism can be specified here. The custom resource representing users from the Kafka cluster is *KafkaUser*.

Custom resources and components

Custom resource definitions (and consequently custom resources) provided by Strimzi, reflect to rather high degree components of Kafka Ecosystem which are described in Section 2.2.4:

[29] This configuration can be forbidden, in case of need
[30] Hydra – https://www.ory.sh/hydra/docs

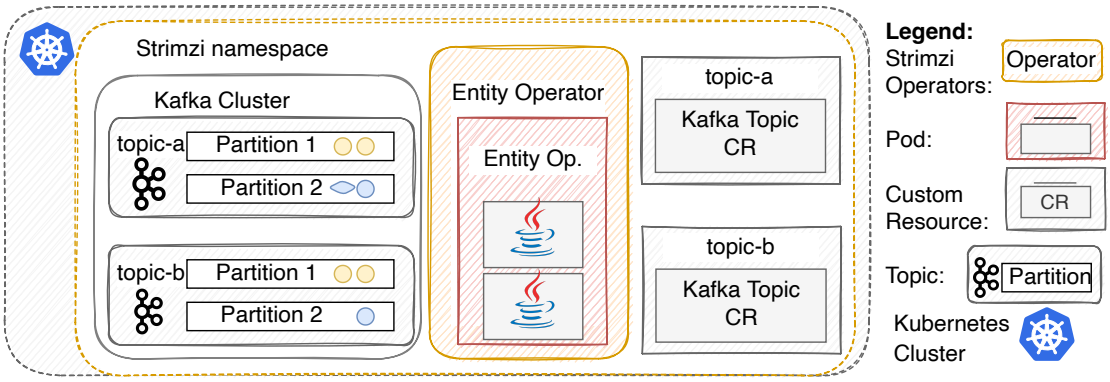


Figure 2.19: Topic Operator (running as container inside Entity Operator pod), watches over KafkaTopics custom resources.

- **Kafka** – Kafka represents cluster of broker nodes. Part of the specification is the number of brokers and the number of required Zookeeper brokers. Concrete Brokers need to store data, which means that *StatefulSet* is used. An example of this custom resource is in Listing 3.7 and also in the Figure 2.20

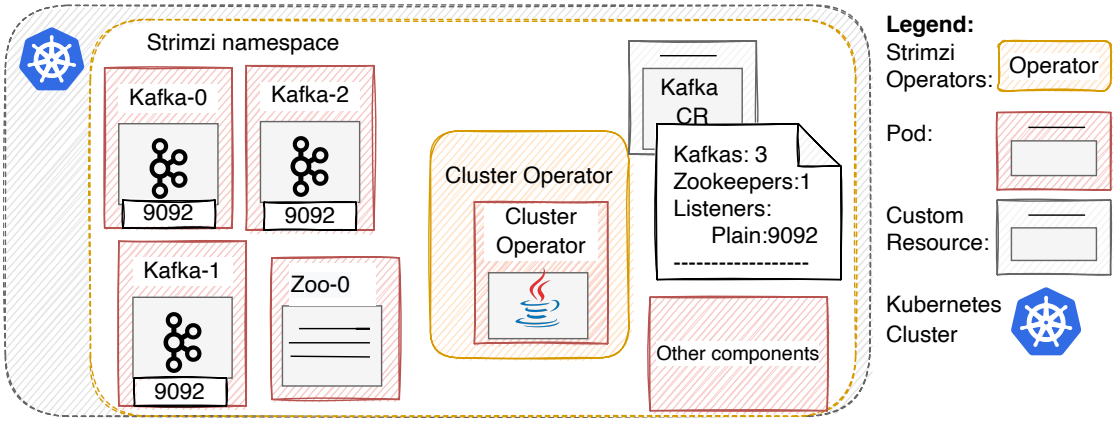


Figure 2.20: Cluster Operator act upon creation of new Kafka custom resource.

- **KafkaBridge** – This component technically serves as a proxy for integrating HTTP-based clients for Kafka Cluster in the form of Rest API. Two main exposed resources are *consumers* and *topics*. Besides typical operations, such as creating and reading, there are plenty of options regarding the assignment of partitions, committing offsets, or subscribing consumers to a topic. Clients can produce and consume messages without the requirement to use the native Kafka protocol.
- **KafkaRebalance** – KafkaRebalance works together with *Cruise control* component, which balances Kafka cluster concerning disk, network, and CPU utilization. Cruise Control is deployed alongside the Kafka cluster to monitor its traffic, propose more balanced partition assignments, and trigger partition reassignments based on those proposals [8].
- **KafkaConnector** – This custom resources represents specification about same work as is described in Section 2.2.4. To work properly, KafkaConnector binds to Kafka-Connect CustomResource, and by transitive law, also to Kafka CustomResource.

Workers from KafkaConnect are now created as Pods, and these are assigned tasks by a specific KafkaConnector.

2.3.3 Configuration

One of the many benefits of Strimzi is that it simplifies the configuration (Appendix C) of Kafka. For example, listening 3.7 shows the specification of the whole Kafka cluster using Strimzi and Kafka custom resources.

```yaml
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-kafka-cluster
spec:
  kafka:
    version: 3.0.0
    replicas: 3
    listeners:
      - name: plain
        port: 9092
        type: internal
    config:
      offsets.topic.replication.factor: 3
    storage:
      type: ephemeral
  zookeeper:
    replicas: 1
    storage:
      type: ephemeral
  entityOperator:
    topicOperator: {}
    userOperator: {}
```

Listing 2.2: Specification of Kafka Cluster with 3 Kafka brokers and 1 zookeeper node.

Chapter 3

Chaos Engineering

This chapter describes Chaos Engineering[1] as a discipline, its origin, principles frameworks, and projects that are built in this area. Section 3.2 describes details about project Litmus[2], its architecture and experiments. The description provided in the following pages is based mostly on books [12, 23, 29] and the Litmus documentation [3].

3.1 Discipline

This section describes Chaos Engineering, which is formally defined as the *discipline of experimenting on a system in order to build confidence* in the system's capability to withstand turbulent conditions in production [1]. Later parts describe its origin (Section 3.1.1), motivation for it (Section 3.1.2), principles (Section 3.1.3), experiments (Section 3.1.4), and lastly its overall adoption (Section 3.1.5).

3.1.1 Origin

Chaos Engineering has its roots in Netflix[3], specifically at the moment when the company decided to move from the data centre to the cloud. The motivation for this migration was *elimination of the problem with the single point of failure* (i.e., datacentre).

The most common trait of cluster components is that they often fail. The reason for that may be anything from cheaper hardware to network problems. There were several ways to increase the uptime of services in the cluster, e.g., redundant nodes, automatization of service discovery, and self-healing of services. Many of these were readily available by using Kubernetes (described in Section 2.1) and thus making them robust enough to *withstand occasional vanishing of instances*. Nevertheless, all of these failures and consequent fixes resulted from unintentional problems.

The first broadly accepted approach that had prevailed was using Chaos Monkey[4], a simple application that periodically picked one instance from each cluster and turned it off, which had simulated the *vanishing of instances in production*. This, as a result, forced developers to act proactively towards possible future failures and make the system resilient. The primary motivation for all of this was to build software resilient enough to withstand

[1] Choas Engineering – also known as Chaos testing or chaos experimenting, more info can be found there https://principlesofchaos.org/

[2] Litmus – Cloud-Native Chaos Engineering Framework, more at https://litmuschaos.io/

[3] Netflix – https://ir.netflix.net/

[4] Chaos monkey – https://github.com/Netflix/chaosmonkey

turbulent events in production with as little downtime for the end-user as possible. Netflix took this concept even one step further with Chaos Kong[5] This simulates the falling of the whole region of clusters.

Netflix's original approach, explicitly using Chaos monkey and Chaos Kong, served to build the resilience of their project by *injecting Chaos directly into the production*. From the beginning, every day, the most important task that needed to be solved was a problem with fallen instances in production. Only afterwards could they continue in regular work. After two years of this approach, Netflix finally decided to build a team to formalize The new Discipline called Chaos Engineering.

3.1.2 Motivation

Building confidence in monolithic systems did not need more than conventional methods of *quality assurance*, e.g., acceptance, integration, end to end testing. No matter what kind of application we have, the task remains the same; make our application resilient. The definition of a system's resilience is, in this context, extended as follows; that being system's ability to withstand different sorts of failures or turbulent conditions. In the case of microservices and the cloud environment, we need to address the system's complexity and prepare for failures beyond the scope of the application.

Complex distributed system

A generalized problem of distributed application written in the form of microservices may look like the one in Figure 3.1

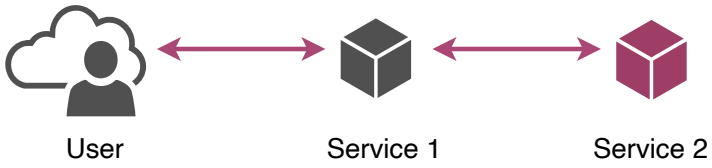


Figure 3.1: A schema of a simplified microservice.

Although the behaviour (i.e., interface) of Service 1 and Service 2 from Figure 3.1 can be clearly defined, to have confidence in this type of distributed system, we need to have answers for what should happen if Service 2 dies or start to respond slowly. What happens if Service 2 comes back after going down for a while or a connection problem occurs. Furthermore, most important, *what does all of this mean to the user* [23]? The example from Figure 3.1 is simplistic, but the services described here may include numerous other components. Most of the distributed systems have traits of the *complex system*, with *nonlinear*[6] behavior. This kind of system may behave unexpectedly due to interaction between its components. *Validation and verification of bigger projects (e.g., Strimzi) is harder than the sum of verification and validation of its components.* Concrete Operators often need to spawn a series of other Kubernetes objects to ensure the proper functioning of the system. When the situation with microservices gains a bigger scale (e.g., Netflix's

[5]Chaos Kong built on the success of Chaos monkey, and same as Chaos Monkey was inspired by actual events (Whole Amazon Region went down) – https://netflixtechblog.com/chaos-engineering-upgraded-878d341f15fa.

[6]Nonlinener system – A system in which the change of the output is not proportional to the change of the input. Taken from https://news.mit.edu/2010/explained-linear-0226

television subscription service with more than one hundred million users worldwide), we would get architecture like the one in Figure 3.2.

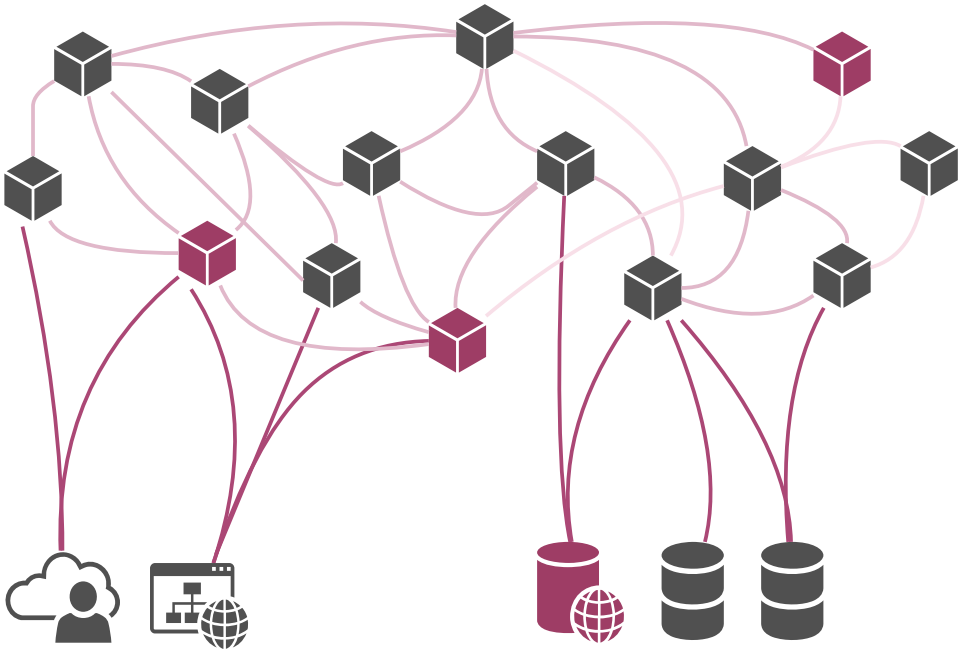


Figure 3.2: More realistic example of microservices and traffic between components. These services can be databases, cache, load balancers, API. Inspired by [22].

Example of this may be cluster with internal DNS[7] service. The cluster works correctly until a rising number of components that need to communicate with the server bring the service down. Afterwards, this behaviour periodically repeats as services increase the rate of requests to the DNS server. Nevertheless, there may not be any sign of a problem until the first fall of the DNS server. These faults are often called *Dark debts*, and the anomalies they generate are complex *system failures* [23]. Generally

Resilliency dependency

Any application intended to run on the *cloud and written as microservices depends on whole stack of other components*:

- **Infrastructure** – Physical form of hardware bears with itself countless options of failure (e.g., power loss, heat, environmental disasters). This includes as well virtual machines, or cloud providers' Infrastructure-as-a-Service (IaaS).
- **Kubernetes** – Kubernetes is still relatively young, and there are many upgrades almost every few months; each of these upgrades can be considered a *threat to resilience*.
- **Network** – Application on cloud is possibly open to the whole outside world, problems with network present plenty of scenarios (e.g., unexpected traffic) which can create a turbulent environment.
- **Other services** – Databases, monitoring tools, load balancers, CoreDns, and any other middleware or service which is either necessary for the functioning of a cloud-native environment or provides some essential service.

[7]Domain Name System – https://www.cloudflare.com/en-gb/learning/dns/what-is-dns/

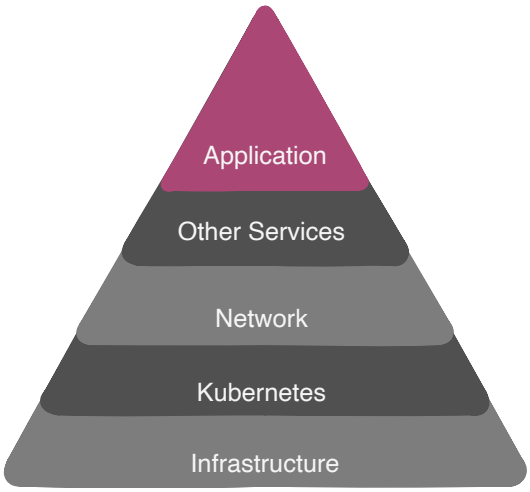


Figure 3.3: The resilience of the application

The resilience of all of these components must match expectations before we even start considering the resilience of the application itself, as depicted in Figure 3.3. It is safe to assume that *most of the resilience of service that runs on Kubernetes depends on other components* and infrastructure most of the time [26].

3.1.3 Definition and Principles

The current definition of chaos engineering is the following: It is the discipline of experimenting on a distributed system to build confidence in the system's capability to withstand turbulent conditions in production. The base unit of Chaos experimenting is *chaos experiment*, described in Section 3.1.4. In the field of quality assurance, we can classify Chaos engineering as follow:

- **Experimenting** – Chaos Engineering is a form of *experimenting*, not testing per se. Tests make an assertion based on existing knowledge, and then running the test collapses the valence of that assertion, usually true or false. Experimentation, on the other hand, creates new knowledge [12].
- **Verification** – Chaos Engineering is mostly interested in the following question. „Does the system work", rather than building some system model. Due to this fact, chaos engineering can be used for setting SLOs[8] which engineers can easily focus on, such as the success ratio of requests even during chaos injection. This is also the most important trait that distinguishes chaos engineering from fault testing; the latter is also interested in how the system works exactly.

Chaos Engineering follows several *principles*, all of which are based *empirically*. The degree to which these principles are pursued vigorously correlates to the confidence we can have in a distributed system at scale [1]. It is important to note that these principles are not all or nothing and the possibility of abiding by them is not always absolute.

- **Hypothesis build around steady-state** – Focus is on the expected behavior of the system, often by looking at key *performance indicators* (KPIs) or other metrics, the focus is on overall output instead of searching for cause [12].

[8] SLO – Service level objective.

- **Vary Real-World Events** – Instead of trying to terminate an instance, fill up the disk and turn off the network on instances, which are all relatively easy tasks that, from orchestration's viewpoint, accomplish the same thing (i.e., instance stop responding). We should focus on events that are more likely to be caused by users, such as varying latency. Prioritization of events should reflect both potential impact and estimated frequency.
- **Run Experiments in Production** – The environment in which we run these experiments is the one in which we *build confidence*.
- **Minimal blast radius** – Chaos should not involve any unnecessary parts (i.e., for that specific experiment) of the system. If we run experiments in the production environment, we should make an effort to minimize the way customer is impacted.
- **Continuous run** – Chaos experiments should be invoked at least each time any change is introduced to any of the components of the underlying technology stack that the application is running on.

3.1.4 Chaos Experiment

The *base unit* of Chaos Engineering is *Chaos Experiment*. Once the experiment is over, it either passes or fails. In the former case, we have more *confidence in our system*, as it can withstand turbulent conditions simulated by the experiment. In the latter one, we have *new knowledge*, and we can either update our experiments or system according to weakness, which was discovered before it could cause severe damage. All of these steps are depicted in the following Figure 3.4.

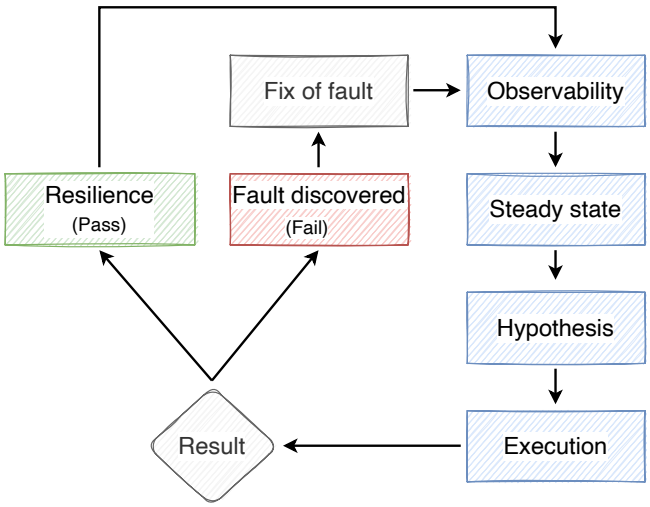


Figure 3.4: Chaos experiment workflow

All of the tools described in Section 3.1.5 implement chaos engineering in almost an entirely different manner, but every chaos experiment itself always matches the following structure:

- **Observability** – Each Chaos experiment starts with observability. It is the key factor in chaos engineering. We need to tell somehow how the experiment ended and what components or variables (e.g., the load of CPU, a crash of website, reachable endpoint

present pod) were affected by the experiment. The key factor here is to reliable see any metric we are interested in. While doing so, *we have to take into account that observing an experiment alone may itself alter its result*[9]. For example, the process for measuring CPU load may end up using more CPU than the application itself [29].

- **Defined steady state** – We define the state of system tan can be considered to be healthy or desired, both before and after the experiment. This is done by focusing on variables, which should, to a high degree, reflect the healthy state of the system.
- **Formulate hypothesis** – We run these experiments with some expected behaviour. This includes checking the desired state after the experiment in terms of implementation. A hypothesis may often represent only a hint or expectation about system properties.
- **Run experiment** – Injection of chaos.
- **Application of changes** – Upon conclusion of the experiment, we should be able to evaluate the result and, with the help of gained metrics and further investigation, suggest and implement fixes in an attempt to fix discovered issues.

Actual examples of applied experiments may be a situation that includes turbulent conditions, e.g., the responsiveness of the website during CPU spike, recovery of the Pod after CPU spike, a recreation of lost resources, and survival of service despite several lost replicas.

3.1.5 Adoption

Resilience is context-sensitive and improving it needs to be looked at as a practice rather than a specific set of tasks [26]. Each system has multiple features that can fail in different manners. However, the same problem most technical giants and systems face is an outage. With the rising adoption of Kubernetes, so does the need to ensure the stability of applications and systems running on it. This means *making our application resilient towards typical problems within the cloud* (and Kubernetes) environment. Shift to cloud and microservices made this problem only more urgent, and a proactive reaction to it is Chaos engineering.

Despite this discipline being still young, it has started to be widely incorporated by technical organizations such as Amazon, Linkedin, Google, and Microsoft. It was primarily the shift to the cloud and Kubernetes' widespread that brought it to the spotlight. Each organization approached chaos engineering in its way and had its reasons to do so (e.g., split Monolith into microservices, shift to the cloud). Some of the currently most widespread chaos testing tools are Chaos Mesh[10], Gremlin[11] which is often considered to be an improved version of the already mentioned Chaos Monkey, as it allows safely and efficiently simulate system outages (and, of course, countless other features), and many others, such as Litmus, which is described in the following section.

It is essential to know where to draw a line where Chaos Engineering becomes for system necessity. Several factors may contribute to this decision, e.g., the need to ensure availability, mission-critical software, and the vast scale of the system. The higher level of abstraction, new functionality, and making service more available (e.g., use of redundancies,

[9] Most famous example of this phenomenon is double-slit experiment - https://en.wikipedia.org/wiki/Double-slit_experiment

[10] Chaos Mesh – https://chaos-mesh.org/.

[11] Gremlin – https://www.gremlin.com/

horizontal scaling) will inevitably cause that system to gain on its *complexity*. In that case, once chaos Engineering is applied, it brings numerous benefits (i.e., proven resilience, found weaknesses, prevention of losses in revenue, reputation, reduction of downtime).

3.2 Litmus

This section covers project Litmus; *open-source Chaos Engineering platform designed for cloud-native infrastructures and applications* [31]. The section starts with its overview and continues with its architecture, terminology, and specific experiments. Information provided here are based mainly on documentation [3] and several blogposts [31, 16, 26].

3.2.1 Framework

Litmus is a complete framework for finding weaknesses in Kubernetes platforms and their applications. It adheres to four principles of cloud-native framework: Open source, Kubernetes native, extensible, and with broad community adoption [21]. All of these traits make it the perfect choice for Chaos Engineering on project Strimzi 2.3.

Litmus uses the *Operator pattern* and relies on Custom Resource Definitions, also described in Section 2.1.6 [16]. The main difference between Litmus and most other tools for chaos testing is that Litmus does it in Kubernetes Native way. While doing so, it also supports several container runtimes (including cri-o). It thus makes it possible to inject Chaos in the Kubernetes cluster on several platforms[12].

3.2.2 Architecture

Installing Litmus means that we deploy the Operator and apply several other resources (almost the same situation as in Strimzi's case 2.3). Kubernetes native nature of Litmus allows us to think about its architecture from the viewpoint of the Operator and the resources upon which it operates. These components form the backbone of Litmus which users can interact with.

Chaos API

Following components are *Operator and custom resources* that users can interact with. Communication between these components and application is described in following lines:

- **Operator** – Manages the lifecycle of the chaos *Custom resources* (that includes *Chaos experiments*). The Operator itself (and most of the experiments) are written in Golang. An example of deployed chaos operator is visible in Figure 3.5
- **ChaosExperiment** – This resource *defines the type of experiment* and its key parameters [25]. These resources are *pre-build* and YAML specifications for these custom resources are hosted at the public ChaosHub[13], which serves as a store for all chaos experiments which are currently available. The configuration provided in this resource can be considered rather static and should require changes rarely.

[12] Supportig of cri-o container runtime makes Litmus compatible also with OpenShift platform

[13] ChaosHub – hub with existing chaos experiments. https://v1-docs.litmuschaos.io/docs/chaoshub/

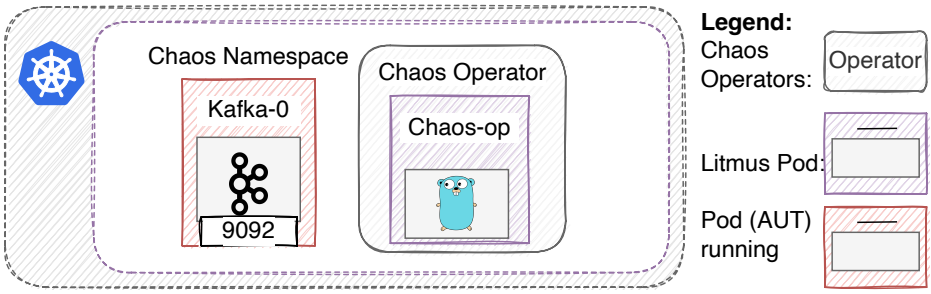


Figure 3.5: Deployment of chaos operator.

- **ChaosEngine** – This *Custom Resource* links an application instance with one or more chaos experiments. This is done by the specification of the target application using labels, kind, and optionally annotation. It serves as the main user-facing resource as well as providing options to override default parameters passed to Chaos by *Chaos Experiments* specification. Most importantly, it serves as the single source of truth for evaluating the execution of the concrete experiment. Once ChaosEngine and respective ChaosEngine are supplied, an operator can start to take action for actual chaos injection. This behaviour is depicted in Figure 3.6.
- **ChaosResult** – There is one ChaosResult Custom Resource per *ChaosEngine*. Its role is to provide information about the results of running it, and contrary to ChaosEngine or ChaosExperiment, it is runtime built.

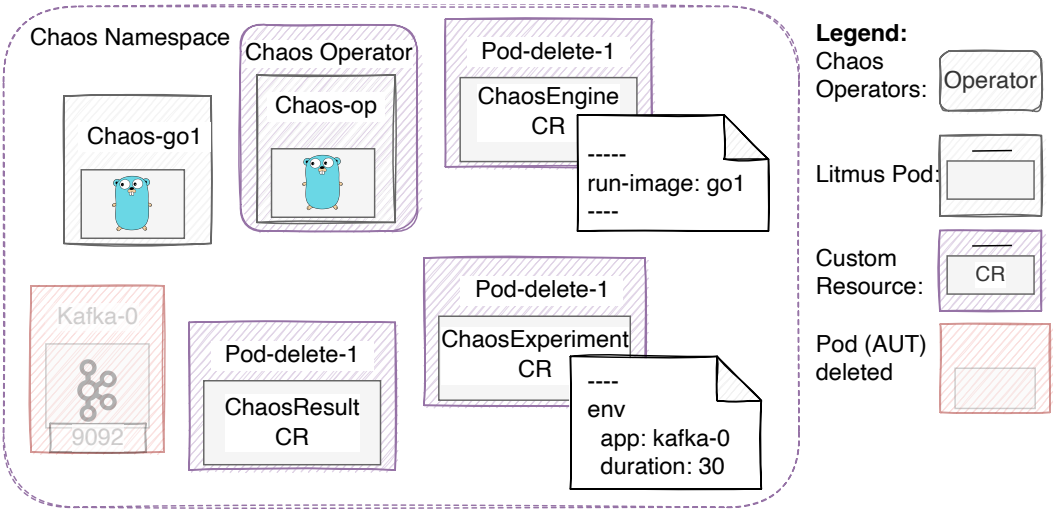


Figure 3.6: Roles of litmus custom resources demonstrated within single example of Pod deletion experiment.

Execution

In reality, actual execution includes several other steps. Previously mentioned parts of architecture describe part of Litmus architecture that holds information. This part covers execution. The Operator creates an object called *ChaosRunner* which afterwards spawns *ChaosJob*. The architecture of execution can be visible in Figure 3.7.

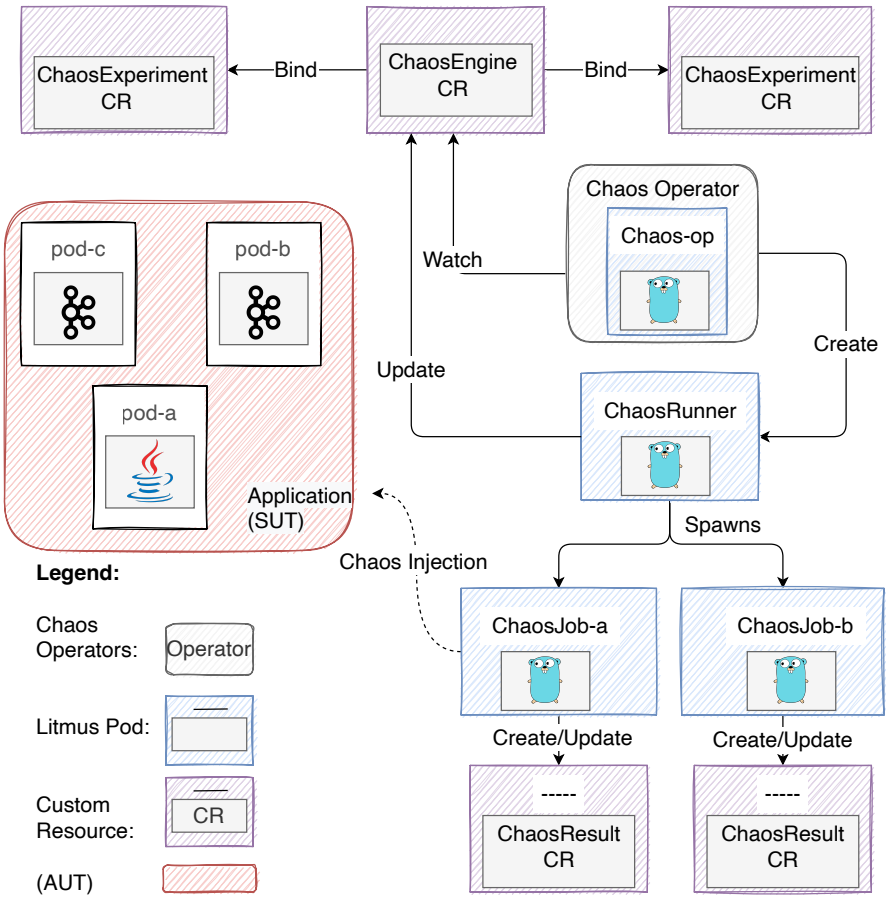


Figure 3.7: Execution of chaos experiments

- **ChaosRunner** – This component maintains life cycle of experiment. Execution of Chaos itself is done using Jobs 2.1.5. It acts as a bridge between the Chaos-Operator and the LitmusChaos experiment jobs. Before creating Job, ChaosRunner checks all dependencies that the experiment requires (e.g., presence of ConfigMaps Volumes, permissions). It takes arguments, commands, environment, variables from both *ChaosExperiment* and *ChaosEngine*, and after all necessary checks, spawn and monitor *ChaosJobs*. ChaosRunner then watches ChaosResult and updates ChaosEngine.
- **ChaosJob** – Technically it is Kubernetes Object, (Job Controller from Section 2.1.5). This component injects Chaos (and several other steps as well) and updates *ChaosResult* with each of its steps. Subsection Chaos Experiments 3.2.3 will further explain all steps and technical details. From a communication point of view.

3.2.3 Chaos Experiment On Litmus

Chaos experiments are the core part of Litmus. The whole project is written in an extensible manner. This is accomplished by publishing experiments on ChaosHub, and providing support for different libraries, e.g., Native-Litmus, PowerfulSeal[14], and Pumba[15] [26].

[14]PowerfulSeal – https://github.com/powerfulseal/powerfulseal
[15]Pumba – https://github.com/alexei-led/pumba

Existing experiments

Litmus currently provides more than 50 experiments (including appropriate RBAC resources) on ChaosHub. To avoid confusion, we will further address these litmus ChaosExperiments as templates. The template's primary purpose is to reference the actual implementation of desired chaos experiment. Which is then configured for a specific use case with the help of environment variables in the specification of another custom resource (i.e., ChaosEngine). Following templates cover the most common problematic situations we can encounter:

- **Pod chaos** – There are templates for deletion of the selected number of Pods, killing of containers, or manipulation with the Pod's resources (e.g., *Disk Fill, Pod CPU hog, Pod IO stress, Pod Network Latency*).
- **Node chaos** – Failure of a node is not a rare condition. There are also occasions when we need to restart nodes (e.g., upgrade of the Kubernetes version).
- **Service chaos** – These experiments kill services on worker modes, e.g., kubelet.
- **Application specific chaos** – Litmus provides also *template* for injecting chaos into Kafka applications. This experiment is built especially to ensure the proper functioning of Kafka while some of its instances are killed.

If provided templates do not match our use cases, we can create our own. All that needs to be done is provide an implementation that matches the expected workflow of the experiment. Then, new experiments can be created the same way as in the previously mentioned templates.

Experiment's workflow

Experiment workflow describes steps executed once the experiment of a given type begins. All these steps are depicted in Figure 3.8 as well.

Once ChaosEngine and ChaosExperiments are applied inside the cluster, the Operator spawns Chaos Runner, which will spawn Job. The following lines describe the situation from figure 3.8. ChaosJob starts its work by reading input (all parameters such as image, arguments, commands, and environment variables) provided in *ChaosEngine* and *ChaosExperiment*.

After initialization of ChaosResult experiment checks Steady-state conditions. This check differs for different experiments, but we have to check if the system under test works as expected before injecting Chaos. For example, in the case of a simple experiment such as deletion of the Pod, we would have to make sure this Pod exists and is in a running state. If the actual state does not match the expected one experiment fails. Otherwise, execution continues with launching additional resources (such as a helper Pod with privileges to mount volumes or kill containers for specific experiments). Chaos is injected once these steps are done (network goes down, a node is drained, Pod deleted). This step can be repeated if we allocate a long enough duration for a single experiment. Once the time for injection of Chaos ends, we verify that cluster is in the expected state and if it is so, we consider the experiment as successful; otherwise, it fails.

During all these steps, we generate events that will be present in *ChaosResult*, *ChaosEngine*, and also *Chaos Operator*. The reason for that is to have all events aggregated but also separated for further usage. The single source of truth for each experiment is *ChaosEngine*.

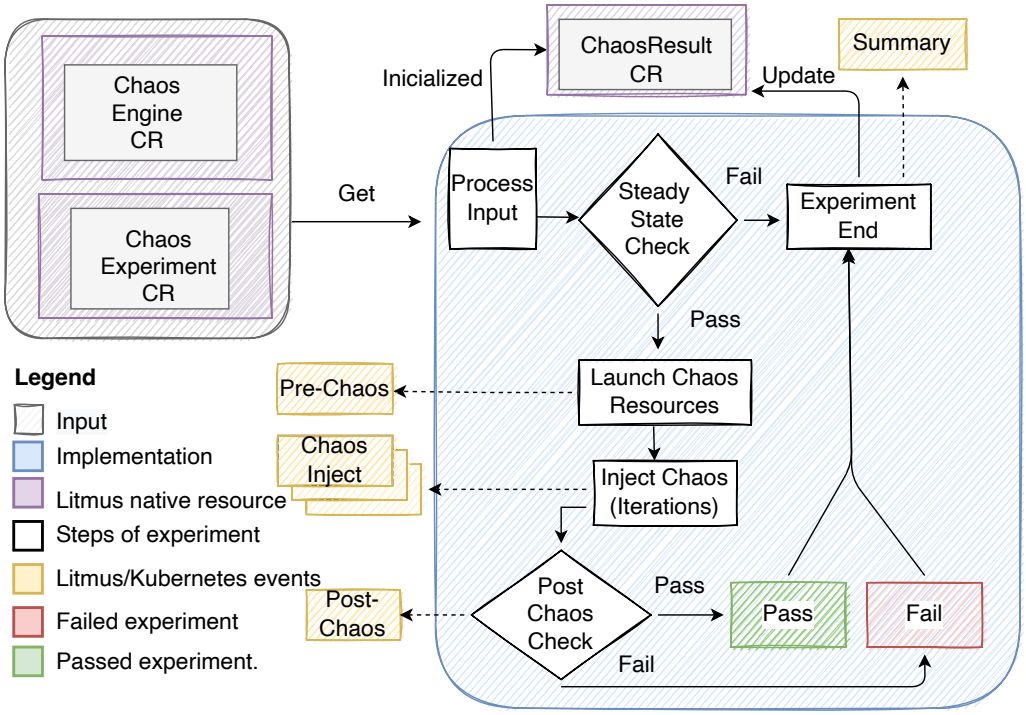


Figure 3.8: Workflow inside Chaos Job

Probes

The existence of pre-created chaos experiments on the chaos hub allows fast adoption of chaos experimenting using Litmus. These experiments are by default *configurable* enough to fit most of the required use cases. For example, we can test Pod deletion on Pods managed by different Controllers in different namespaces. However, sometimes, checks executed by these experiments are insufficient, and we need some additional checks. To avoid the need for writing new experiments, we can add additional *probes* to existing generic *ChaosExperiment*. These probes are pluggable checks that can be defined within the ChaosEngine for any chaos experiment [3]. For the experiment to pass, all these checks have to pass. Probes are specified as part of *chaosEngine* specification and can be executed before, during, or after chaos injection. Currently, supported probes are:

- **cmd** – Function that is implemented as a shell command. A typical example of this kind of check can be the presence of some records in the database.
- **http** – Http request inside cluster, with support for *Get* and *Post* methods. Check itself is done by comparing obtained and expected status codes.
- **k8s** – CRUD operations against Kubernetes resources.
- **prom** – Match prometheus metrics for specific criteria.

Chapter 4

Design

Section 4.1 describes the main weaknesses of project Strimzi, which we can (not necessarily) address. This part serves as a base for reasoning about the best possible design and proposed solution (and alternatives) for implementation. Section 4.2 then extends information about Litmus architecture (Section 3.2) and describes how our component can interact with Litmus' custom resources, e.g., *chaosExperiment*. The following Section 4.3 describes specific chaos templates. It starts with existing templates and discusses what functionality is needed to inject different kinds of Chaos. Finally, Section 4.4 describes the application of chaos engineering concerning the needs of the Strimzi project and the principles of chaos engineering.

4.1 Considerations

The design and implementation of chaos experiments are tightly coupled. The first step is identifying the properties of the system with which we want to experiment. Once we do this, we can decide what level of abstraction and what type of framework best suits our needs.

4.1.1 Strimzi's Weaknesses

The nature of the Strimzi project (an operator that incorporates Kafka running on Kubernetes), and the fact that it is usually part of other systems, means there are countless viewpoints regarding possible weaknesses and demands regarding its resilience:

- **Containers and pods** – An application always runs inside a container (Section 2.1.1), which runs inside a Pod (Section 2.1.4). That means container runtime processes, containers, and Pods can crash or become unresponsive. A typical problem that needs to be addressed is Pods' readiness and health check. Each Pod and its containers may start to consume too much memory, cause too many writing operations on disk, and take too many CPUs or RAM. This affects the Pod itself and the whole node the container is running on. On top of that, as was already mentioned, pods are known for their ephemerality.
- **Kubernetes** – By running our application inside the Kubernetes cluster, we are counting on the correct functioning of countless services. Both types of nodes (i.e., master and worker, described in Section 2.1.3) may stop working correctly, and in the case of workers, it means that a significant part of the system goes down as well:

  - **Master nodes** – Control plane of Kubernetes consist of several components that need to function properly (i.e., etcd, API server, scheduler, controllers). These components may fail under specific conditions and cause undesired behaviour in other components. For example, etcd uses distributed consensus algorithm called Raft[1], if we bring down at least half of the master nodes, it will make etcd unavailable.
  - **Worker nodes** – Worker node needs three processes to work correctly, specifically container runtime, kubelet (i.e., Kubernetes process), and Kube proxy. All of them are already described in Section 2.1.1). A crash of any of them will make this node either unresponsive or unable to work correctly. There are also options for disrupting routing (by corrupting *iptables* or killing the proxy pod responsible for their maintenance).

- **Cloud and network** – There are a series of services communicating with each other using a network. Besides communication within the cluster, services that incorporate Strimzi are often public (e.g., it communicates with services or clients outside of the cluster). This means that besides common networking problems (i.e., packet corruption, packet loss, latency), we may encounter other turbulencies (e.g., spikes of traffic, DOS attacks[2]).
- **Implementation** – Strimzi is implemented using Java programming language and operator pattern (described in Section 2.1.6) to simplify work with Kafka inside the Kubernetes cluster.
  - **Operator** – There are plenty of objects that are responsible for the correct functioning of our application (e.g., Services, configuration maps, Pods). They can succumb to failure, be removed, or be replaced. Strimzi operator runs inside the Pod as well and therefore can also easily succumb to failure.
  - **Java** – most of Strimzi components (e.g., Strimzi Kafka Bridge, Apache Kafka, Apache ZooKeeper) run inside JVM[3]. We can easily inject failure directly into JVM and cause unexpected exceptions, but most of these scenarios are already tested within the scope of conventional quality assurance. Although project Strimzi, like most of the other projects of that scale, involves a series of tests (i.e. unit, integration, end-to-end tests), thus most of the troubles connected to concrete implementation are already covered.
- **Kafka** – Strimzi is Kubernetes native project which may be involved in mission-critical software; it is necessary to enforce this quality in the Strimzi (if desired) as well. Depending on the concrete configuration of Kafka, a user may require high availability or other traits covered in Section 2.2. Most of the experimenting will need to verify that proper functioning (i.e., availability or guarantee of delivery) is ensured even during chaotic conditions. Strimzi is a highly configurable project with many additional features (e.g., Bridge, metrics) that will require experimenting with many configurations. Concerning the used Kafka version, we will also consider Zookeeper and its quorum dependent algorithm.

[1] **Raft** - More at https://raft.github.io/
[2] **DOS attack** - Denial-of-Service attack, more at https://en.wikipedia.org/wiki/Denial-of-service_attack
[3] **JVM** - Java Virtual Machine

4.1.2 Choice of Approach

The choice of design (and coupled implementation) may make some experiments impractical to maintain or implement. Despite the relatively straightforward structure of chaos experiments from Section 3.1.4, even simple experiments such as killing a Pod may be approached from many perspectives.

There are two main factors regarding the choice of the chaos design. The first one is to cover most of Strimzi's weaknesses. The second one is fulfilling the criteria regarding the correct approach to chaos engineering (Section 3.1.3) and allowing us to create maintainable chaos experiments (Section 3.1.4).

Imperative design and implementation

Strimzi is a Kubernetes native project. Kubernetes always depends on some containerized solution, e.g., docker. This means that we have quite a big choice in the desired level of chaos experiment's abstraction :

- **Linux utilities** – The Most common choice for simple experiments is writing scripts. The biggest advantage of this approach is that we have access to all low-level parameters. We could use Linux utilities (e.g., df[4], ab[5], tc[6]) for manipulation of the system system resources, i.e., CPU, RAM, storage, network.
- **Containerized solutions** – Represents approach with higher level of abstraction. Instead of working with processes, we manipulate containers directly (working with containers is briefly explained in Section 2.1.1). This can be done using third party libraries, e.g., Pumba[7].

The higher level of abstraction significantly decreases problems with maintainability and depends less on concrete solutions (i.e., scripts, systems, runtimes). The need for inconsistent solutions often makes these types of designs rather impractical for bigger projects. Continuous run can be accomplished with use of utility called cron[8], or using some continuous integration and delivery software, e.g., Jenkins[9]. However, problems with observability remain, as well as blast radius (especially in the case of lower abstraction). An example can be killing processes or causing higher traffic between database and service. This scenario can be accomplished using Linux utilities or additional libraries. Still, unless being specific with names of processes or the node's location (which may be even virtual), we may induce Chaos elsewhere.

Declarative design and implementation

When we start using the *Kubernetes API*, the simple addition of a few YAML files with a description of resources allows for the creation of rather advanced chaos experiments.

[4] **df** - abbreviation for disk free, more at https://en.wikipedia.org/wiki/Df_(Unix)

[5] **ab** - Apache Benchmarking tool, more at https://httpd.apache.org/docs/2.4/programs/ab.html

[6] **tc** - (Traffic control) manipulate traffic control settings, more at https://man7.org/linux/man-pages/man8/tc.8.html

[7] **Pumba** - chaos testing command-line tool for Docker containers, more at https://github.com/alexei-led/pumba

[8] **cron** - https://en.wikipedia.org/wiki/Cron

[9] **Jenkins** - Continuous integration and delivery automation software, more at https://www.jenkins.io/

An example of this can be using *toxiproxi*[10], a simple framework for simulating network conditions on a Kubernetes cluster. Users can specify almost all of Chaos's parameters in a declarative manner. With the help of similar tools, we could accomplish an almost fully declarative chaos design. The following lines describe vital options for fully declarative chaos engineering on the Kubernetes cluster:

- **PowerfulSeal** – Is a chaos engineering tool for Kubernetes. Implementation of chaos experiments is afterwards accomplished by writing Yaml files. We can write any number of scenarios, each listing the steps necessary to implement, validate, and clean up after a given experiment [29].
- **Litmus** – Besides the numerous benefits that Litmus has, there are two points which we later consider regarding our needs. The first one is that Litmus is the youngest of mentioned projects. Therefore, it may still undergo some significant changes, and the second is that some experiments, although provided, may not be possible on a specific infrastructure, e.g., experiments with nodes.
- **Others** – Besides already meantioned approaches, there are also numerous other options (i.e., tools and frameworks) for application of chaos engineering on Kubernetes cluster, e.g., ChaosToolkit[11], kraken[12].

Considering all previously mentioned attributes, regarding project Strimzi, a declarative approach to Chaos seems *more durable* and with lower *blast radius*. Moreover, with the help of frameworks and libraries, this approach also provides sufficient options for *observability* and options to accomplish *continuous* run.

4.2 Communication

If we decide to implement templates and inject Chaos into some production project or artificial system in a Kubernetes native way, we will work primarily with CustomResources. The whole communication can be visible in Figure 4.1. Assuming that *Chaos Operator* and respective *ChaosExperiment* CustomResource already exist, each experiments starts with creation of *ChaosEngine* CustomResource. This is all interaction regarding the initialization of Chaos we will need from the user. After a series of steps explained in Section 3.2.2, eventually, the Pod containing the specified image is created. It contains all the specifications it needs in the form of environment variables and arguments it was executed with.

Besides execution of Chaos itself, pod is also responsible for continual updating of chaos custom resources (i.e., *chaosResult*, *chaosEngine*). Once Job (Pod) is finished with the experiment's execution, it updates the *ChaosResult*, and afterwards, the experiment is done. Users can then verify messages and results present in *chaosResult* and examine *chaosEngine* (this resource is supposed to be the single source of truth for a given experiment). Based on the specified cleaning policy, the user can evaluate logs from Pod responsible for chaos injection and other additional resources involved in the chaos experiment.

[10] **toxiproxi** - more at https://github.com/Shopify/toxiproxy
[11] **ChaosToolkit** - https://chaostoolkit.org
[12] **Kraken** - https://github.com/cloud-bulldozer/kraken

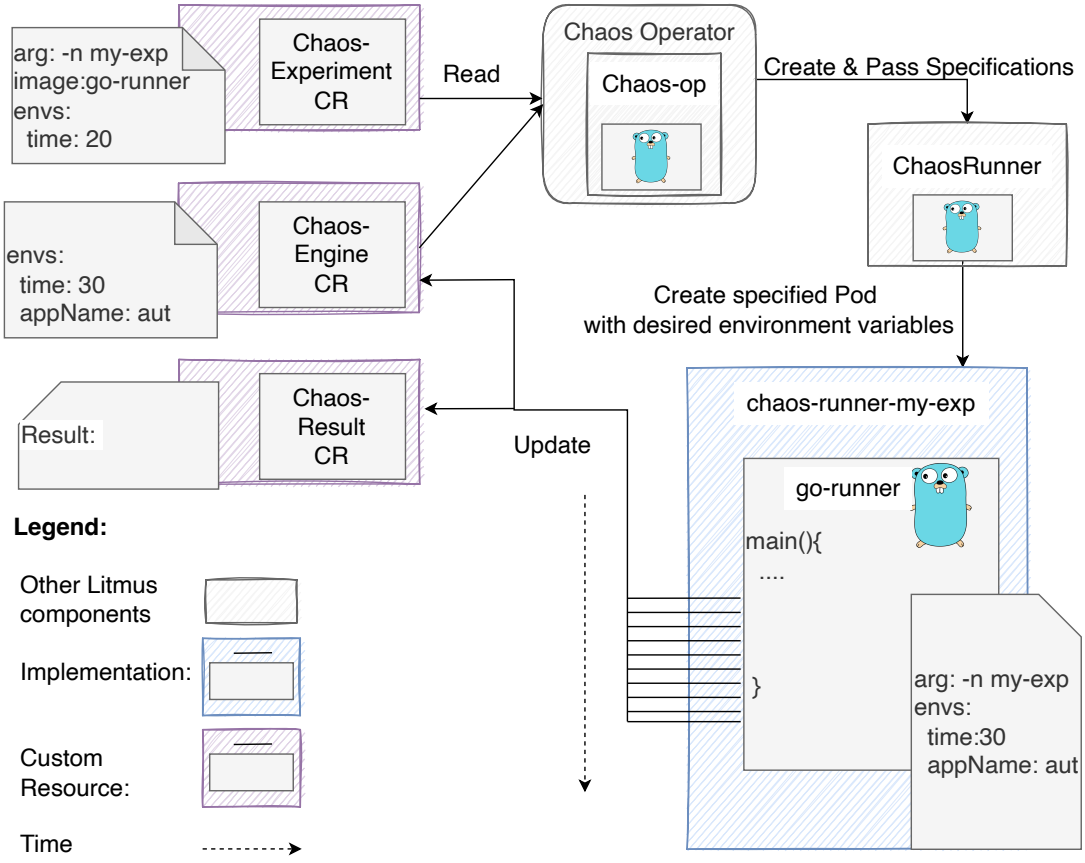


Figure 4.1: Communication between our template and other components of Litmus.

4.3 Chaos Experiments on Project Strimzi

The focus is on replicating the events that are likely to happen in our system. For example, the application will go down, networking will be disrupted, nodes will not be fully available all the time, and human errors will occur [36]. Section 4.1.1 describes common concerns we should address in Strimzi and similar projects. These factors represent categories of potential problems that may occur within the scope of Litmus chaos experiments. Litmus chaos experiments serve as templates for creating chaos experiments. For simplicity further called *templates*. Each template follows the structure of chaos experiments, as was already described in Section 3.1.4. Implementation of desired chaos experiments can be accomplished using provided generic templates (i.e., creating Litmus Chaos Engine components bound to these templates). These generic experiments are described first. Afterwards, we will suggest and describe the implementation of brand new templates (these will also be available on Chaos Hub) for situations where we cannot accomplish desired behaviour with provided generic templates. These templates can be later used in the same manner as other already existing templates, just like all other existing templates.

4.3.1 Generic Chaos Experiments

When we think about chaos experiments (in the scope of using Litmus on our project), we must consider that each chaos experiment is bound to options provided by a template (i.e.,

litmus chaos experiment). This section is called generic because to create these experiments, it is possible to reuse existing templates (i.e., litmus chaos experiments described in Section 3.2.3).

For example, if we want to test that our Operator will work correctly even if that Operator Pod dies or that communication with the Kafka Bridge is restored once the problem with the Kafka Bridge pod is resolved. Because Strimzi and its components are running inside Pods, the correct functioning of these components depends on the proper functioning of these Pods. We could easily accomplish this with templates for pod deletion or container kill. These templates refer to specific chaos executions. With the help of environment variables and other components (i.e., ChaosEngine), we can set up this chaos experiment to the desired form. In our case, it can be done simply by selecting the Pod and the appropriate Controller. In the case of controlling correct behaviour, from the Kafka Bridge example, we may also use probes (described in Section 3.2.3). We are explicitly checking the availability of the HTTP endpoint once chaos duration (i.e., the time after which we expect the system to regain its former state) ends. Provision of the system under test for this simple scenario may look something like the one in Figure 4.2. Some other scenarios that may use generic templates are the following:

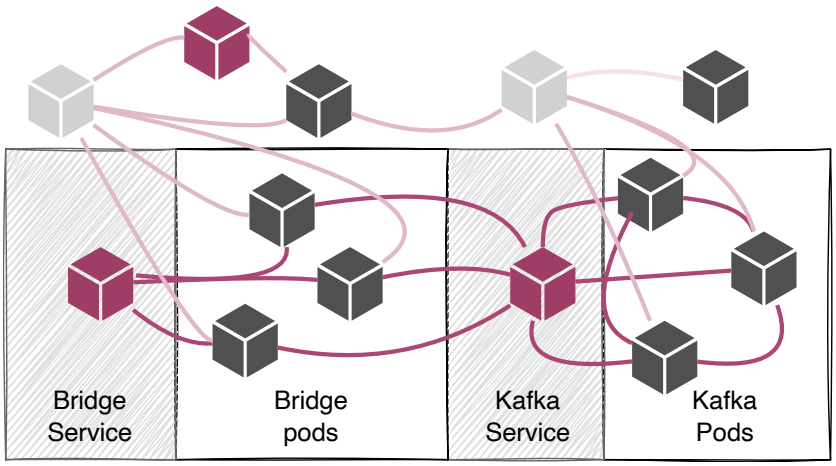


Figure 4.2: System under test in case of chaos experiment with simple Kafka Bridge.

- **KafkaConnect process kills** – The way how Kafka connect works is already described in Section 2.2.4. This experiment can use templates for container kill and, by doing so, disrupt the correct functioning of this component. The expected behaviour is that work will continue, and new tasks are created in place of those killed.
- **Broker killed** – Kafka broker pod deletion is already provided as a template by Litmus. Different types of brokers and their roles are described in Section 2.2.3. This experiment allows us to verify the undisrupted flow of messages during the event of killing some brokers. This template may need slight adjustments to observe correct functioning under the circumstances of killing different brokers with different roles (e.g., partition leader, cluster controller) and with varying configuration goals (e.g., availability). The problem with Kafka itself may become even more complex, as Litmus does not provide a way to identify the different roles of brokers. The bigger problem is the fact that due to the already mentioned KIP 500 (Section 2.2.3), we will

need to execute these experiments even in the absence of the Zookeeper. The current implementation of Litmus does not yet provide support for this Kafka's improvement.

- **Kubernetes chaos** – Lot of components that help with the provision of functionality are pods and services as well. Besides these, we will also inject Chaos into the essential services and processes (e.h., container runtimes) and indulge network and CPU hogs.

Templates (and experiments that can be built on top of them) described in previous points cover most of the general requirements. However, because Strimzi is an Operator, our needs are slightly different. As a result, several types of these provided templates will be redundant, while some additional cases need even more specific approaches.

4.3.2 Strimzi Specific Chaos experiments

As was already mentioned, Litmus provides more than 50 already implemented templates. Unfortunately, most of them are not suitable for our use case (e.g., templates for injection of Chaos to other operators, injection of Chaos into Kubernetes internal components[13]). To be able to inject a specific type of Chaos (i.e., aimed to disrupt precisely our resources), we have to design and implement our own additional chaos templates first.

Templates should implement some basic general behaviour. Additionally, they should allow users (these templates will be available from the chaos hub to the whole community) to specify scenarios based on their needs, i.e., specify application under test, mark resources, and provide all necessary configurations for their specific application. This means implementing the logic of the experiment that will be part of the runner image, afterwards creating necessary resources for sufficient minimal permissions, i.e., service account, role, and role binding (all of them described are described in Appendix B). The following parts describe two main templates that will need to be implemented to create desired chaos experiments afterwards. Once these templates are implemented, we can use them to create chaos experiments.

Resources Deletion

Strimzi operators and any other operators operate with numerous other components. In the case of Strimzi cluster operator, it should be able to recover lost resources (e.g., configuration maps, secrets). Deleting these resources is not something that would happen as often as failures of pods. Still, because they are managed by the Operator and are likely to be part of larger systems with numerous other similar resources, they may be removed due to other activities, such as human error or invalid scripts. Deleting or losing some of these components may also cause different events (e.g., deletion of specific secrets may cause rolling update of the whole cluster).

The workflow of a template can be visible in Figure 4.3. It matches the desired specification of Litmus chaos experiments. Implementation of this is simplified due to provided bootstrap[14] in the Go language. This provides the implementation for generating basic events and parsing input.

Besides controlling that expected resources are recreated, the experiment also verified that this interference (i.e., Chaos) did not touch the consumption of data from Kafka.

[13] Behavior in these cases in most cases is already well documented.

[14] Repository with the implementation of existing templates – https://github.com/litmuschaos/litmus-go

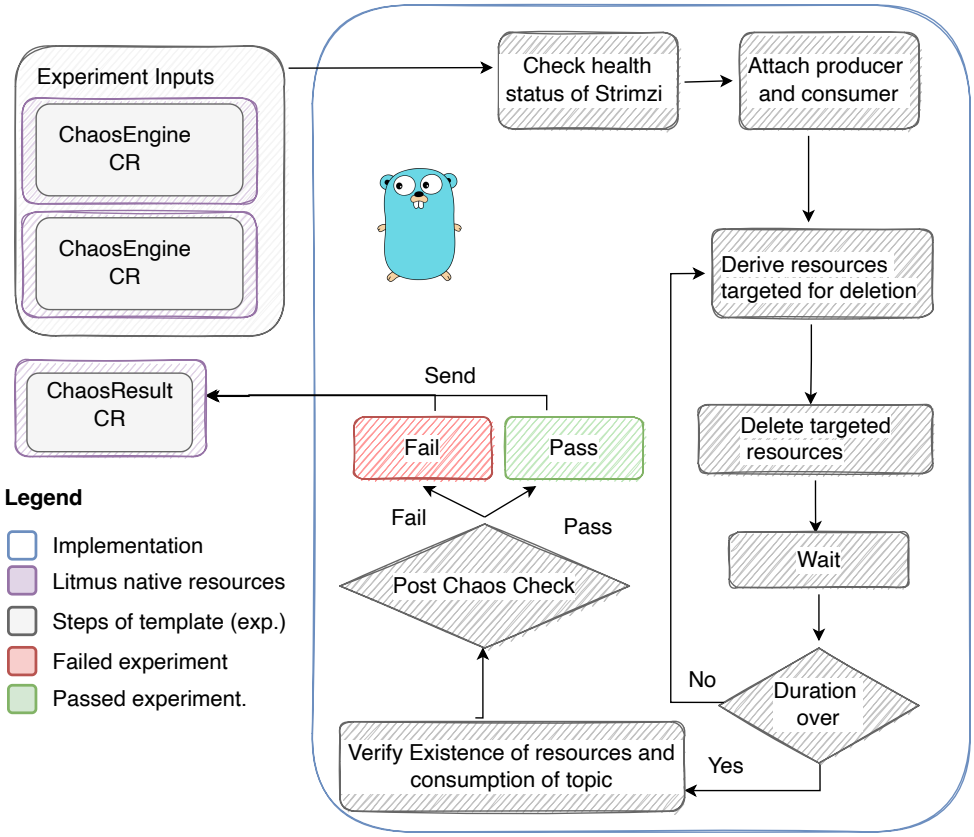


Figure 4.3: Workflow of Strimzi specific implementation (using Golang) of Litmus Chaos Experiment, focused on Strimzi's capability to restore its kubernetes resources.

Update Resilience

In Kubernetes, one of the weakest points[15] of any service's availability, is during its update (it is usually rolling update, also described in Section 2.1.5). Kafka's update means that every broker goes down and is replaced with a new one. This can mean a problem with availability if we configure Kafka to accept data only once all followers are synchronized, and also a significant vulnerability if we have only a few replicas for any of the topics. The whole process represents a weak spot not only because there is one missing component for the entire duration of this event but also because several things may go wrong with the update itself. Implementation of this template use lot of the same steps as the previous template. The main difference here is that we need to force the recreation of Kafka's Pods. This can be accomplished with the renewal of certificates, already mentioned upgrade of Kafka brokers, or updating Kafka resources to such degree[16] that update will be required, e.g., the addition of internal listener.

[15]Situation like this is when we need to drain all Kubernetes nodes, but this situation is already handled with the help of another Strimzi component called drain cleaner (https://github.com/strimzi/drain-cleaner), and it uses even more advanced Kubernetes principles.

[16]Strimzi support dynamic changes, i.e., it does not need to recreate Kafka pods in case of minor changes.

4.4 Application Of Chaos Experiments

With all templates implemented and chaos experiments designed, all we need to do is to apply them to the system under test. This step is trickier than it seems, as we want to follow several practices. We want observability, continuous run, easy deployment, and preferably some production system to apply this Chaos on. Setting up the system under test itself is no easy task. Even a simple experiment of bringing down a few pods from Kafka Bridge requires the presence of operators, services, and multiple pods (all visible in Figure 4.2).

Two main possible options to accomplish this are an extension of existing Strimzi system tests or an application on the system (either real or one created exclusively for the purpose of Chaos engineering). These variants have trade-offs and are further examined in the following lines.

4.4.1 Extension of Existing System Tests

Chaos engineering should match the needs of the system under test. In our case, it is Strimzi, which is technically an Operator. This means we have to cover many possible scenarios, different configurations, and deployed components. Setting up all these configurations of Strimzi (e.g., Kafka, Zookeeper, KafkaConnect) only using YAML files may be tedious. Nevertheless, Strimzi system tests provide implementation with available builder[17].

The whole process of running chaos experiments can be accomplished with the use of Azure[18] pipelines or Jenkins jobs; in simple terms, Azure pipelines (and Jenkins jobs as well) have several features (i.e., step, stage, task). Within a step, we can specify the concrete command. A set of these steps builds our testing environment. These steps prepare tools, such as Java and Kubernetes clusters. Depending on our needs from the underlying infrastructure, we can use one-node (i.e., Minikube, which is used Azure pipeline) or a multi-node Kubernetes cluster. In Figure 4.4, we can see two main parts of these pipelines. The first part is the preparation of the execution environment and the second one is the arrangement and logic of execution. The preparation phase consist of steps required to provide concrete machine with environment suitable for chaos experiment execution (e.g., download Strimzi repository[19] , Java, docker). The verification phase uses the predefined experiments (written in YAML), test cases annotated with „chaos“ (This will allow maven[20] to identify chaos experiments that should be executed), and finally the chaos test suite. These three components are the backbone of the proposed design of chaos engineering on the Strimzi project. Specific Results, logs, and metrics can be later visible in reports.

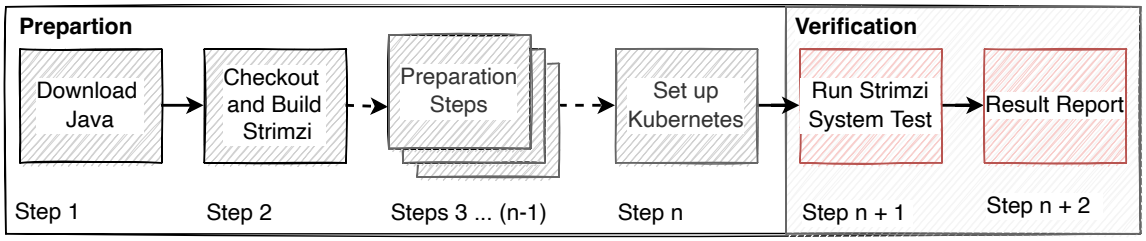


Figure 4.4: Workflow within azure pipeline/jenkins job, which is executing chaos tests.

[17]Builder is a design pattern used for the construction of complex objects step by step.
[18]more at https://azure.microsoft.com/en-us/
[19]**Strimzi repository** – https://github.com/strimzi/strimzi-kafka-operator
[20]**Maven** - Maven is a build automation tool used primarily for Java projects.

Two main drawbacks of this solution would be that we are not testing any particular running system, which is a missed opportunity as we would also provide confidence in that system. The second one is over-complication of the result pipeline, i.e., significant communication overhead in the code.

4.4.2 Experimenting in Production

In contrast to applying Chaos with CI-driven execution, this approach monitors and manages Chaos for a specific configuration of Strimzi in the long run. The first variant is a dedicated cluster with artificial Strimzi and some workload running on top of it. We can deploy only one or very few Strimzi Kafka clusters. Continuous scheduling and application of Chaos can be accomplished with the Litmus component called *ChaosScheduler* (described in part 3.2.2). Both Litmus and Strimzi provide support for the export of metrics. So information such as the number of online brokers, incoming and outcoming traffic, CPU or memory usage, and countless other metrics that accompany the functioning of Strimzi under conditions of injecting Chaos can be monitored with Prometheus and additionally displayed with the help of Grafana dashboards.

A slight variant of this is the application of Chaos on an already existing system. The only disadvantage of this would be that this kind of system may not use all features of Strimzi, so there would not be a need (unless forced) to create chaos experiments for them.

Chapter 5

Implementation

This chapter covers the actual implementation of the solution proposed in Chapter 4. Firstly, Section 5.1 briefly describes how we accomplish communication with Litmus custom resources. After that, the main focus is on the implementation of new templates (Section 5.2). Finally, the chapter ends with implementation of the actual application of chaos, i.e., the execution of specific chaos experiments against the running system (Section 5.3).

5.1 Components' Communication

To inject chaos into our system, we need to transfer much information between different components managed by autonomous entities, e.g., Kafka Pods managed by the Strimzi cluster operator. All communication springs from chaos runner, so naturally, Litmus provides implemented ways to simplify repetitive needs to obtain and propagate information from the Litmus native components.

Invocation of the desired template is accomplished by passing the proper flag as part of the ChaosExperiment resource (Section 3.2.2), also shown in Listing 5.1. This property is then propagated and used inside the Pod executing actual chaos, which is shown in Listing 5.2. Main function simply parse provided flag and invokes the according branch.

```
kind: ChaosExperiment
metadata:
  name: strimzi-pod-delete
spec:
  image: "litmus/go-runner:ci"
  args:
  - ["-c", "./experiments -n
      strimzi-pod-delete"]
  command: /bin/bash
  env:
  - name: CHAOS_DURATION
    value: '30'
```

Listing 5.1: Definition of chaos experiment.

```
package main

import (
    nr "experiments/litmus/..."
    spd "experiments/litmus/..."
)
func main() {
    name := flag.String("n")
    switch *name {
        case "node-restart":
            nr.NodeRestart()
        case "strimzi-pod-delete":
            spd.PodDelete()
            // other experiments
```

Listing 5.2: Determining which experiments to be invoked.

The rest of the input (by input we mean specification provided in custom resources responsible for chaos invocation) resides in a resource called *chaosEngine*. Its attributes are passed to running containers as environment variables, so reading them is simply a question of extracting them directly from the environment and storing them inside desired structures.

The second part of communication represents a process of sending information about the progress of the experiment to the Kubernetes. This means updating resources (i.e., *chaosResult* and *chaosEngine*) and generating Kubernetes' events. Litmus implements necessary interfaces to allow usage of Kubernetes client[1] upon its resources. This client is a library for communicating with Kubernetes' resources, generating events (as is required by Litmus) depicted in Figure 3.8 is then accomplished easily with the help of the client, as shown in Listing 5.3. Here we can see function which takes as input all necessary specification of event and afterwards use with usage of client invokes creation of this event in the cluster. Manipulation of all other Kubernetes resources[2] can be accomplished in a similar manner.

```
import (
   apiv1 "k8s.io/api/core/v1"
   metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
)

func CreateEvent(eventName, ns string, details *EventDetail, ...) error{
   event := &apiv1.Event{
      ObjectMeta: metav1.ObjectMeta{
         Name: eventName,
         Namespace: ns,
      },
      Message: eventsDetails.Message,
      Reason: eventsDetails.Reason,
      Type: eventsDetails.Type,
      Count: 1,
      // other properties ...

   _, err := client.CoreV1().Events(ns).Create(event)
   return err
}
```

Listing 5.3: Communication with rest of Kubernetes from inside of Pod.

The last part is about using data regarding the results of an experiment. This information is stored inside *ChaosResult* custom resource, and it will be up to another system to parse this resource. We will implement two different approaches to accomplish this. The first will be for the evaluation of results in Section 5.3.2, and the second one takes place is in Chapter 6.2.4 where we will visualize some of the results.

[1] Kubernetes client library – https://github.com/kubernetes/client-go

[2] Another essential library for manipulating Kubernetes objects can be found here https://github.com/kubernetes/apimachinery

5.2 Templates

We implement four own templates in this thesis. Our choice of language is, in this case, Golang, which allows us to extend repository[3] containing most of the currently existing experiments. Implementation of each of the templates follows the structure described in Section 3.2.3. Complete implementation of template requires also relevant access resources, such as *role*, *roleBinding*, *serviceAccount*. These resources provide all means necessary for correct authorization and authentication in a Kubernetes cluster and are further explained in Appendix B.1. The scope of privileges we have to give to the entity executing the experiment depends on the resources it will need to manipulate.

Besides two originally proposed templates (Section 4.3.2), e.i., *Kafka Rolling Update* and *resource deletion*, two additional are implemented. This decission is the direct result of updates of underlying technologies (e.g., new version of Kafka) and the fact that Litmus did not reflect these changes in templates which it provides. The content and reasons for addition of these templates will be further explained in following sections. In order to better understand quite tangled concepts in implementation, this section is further split, beginning with Section 5.2.1, which describes all additional implementations needed for the smooth functioning of every single templates template from the perspective of implementation of the first template, called Pod delete. Afterwards, the rest of this section describes implementation of the three remaining templates.

5.2.1 Preliminaries

The templates we have implemented within this thesis require additional logic and working with specific custom resources Therefore, we will start with a description of a simple template and additional components we will implement and use in all other implemented templates.

Broker pod delete

Litmus already implements template for *pod* deletion, as well as another template called *kafka-broker-pod-delete*. Unfortunately, none of these templates can cover all of our needs in regard to actually injecting chaos to all our environments. The first template does nothing more than delete Pod and verifies that it is recreated after the chaos is over. If we assume that Kubernetes works correctly, this behaviour is expected, and experimenting with it alone would be much more about testing Kubernetes itself than the ability of our application to withstand the actual condition of brokers going down. The problem with the second provided template is that Litmus did not make the template flexible enough to overcome the update of Kafka to version 3.0.0. As was mentioned in Section 2.2.3, Kafka is working on getting rid of its dependency on Zookeeper. Despite it not being fully reached, Zookeeper usage as an option when connecting to Kafka is no longer supported. So instead, an additional template is implemented. This can overcome the mentioned issue by working with Kafka instead of Zookeeper. It also provides flexibility by allowing users to choose possible other ways to connect to Kafka. Another imporant point here is also addition of options to communicate with Kafka despite it being secured, but this is the topic which will be described further in following Section.

[3] More at – https://github.com/litmuschaos/litmus-go

This template aims to provide users with enough functionality to experiment with different scenarios of Kafka broker Pods unavailable instances (potentially even different Pods, such as Zookeeper). Identification of desired Pods is possible in three ways, i.e., by names, labels, or the function of the pod[4]. The purpose of this experiment, in a nutshell, comprises the identification of Pods (depicted in Figures 5.1 and 5.2), their deletion, and finally, checking that they are recreated. Of course, all of this must be done within the expected time.

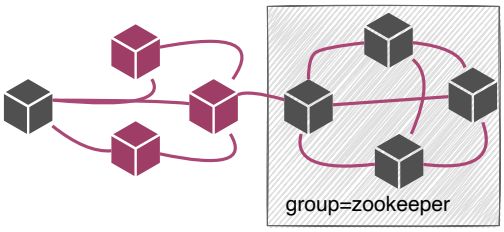


Figure 5.1: Identification of Pods by their labels.

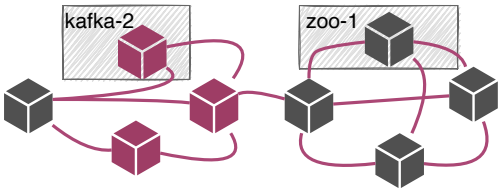


Figure 5.2: Identification of Pods by their names.

Clients

To read and manipulate custom resources (Section 2.1.6), we have to implement an appropriate Kubernetes client. To do so, we start with the provision of models, i.e., data structures which will represent given custom resources in our running instances of code. We will do this for all Strimzi custom resources we will work with, e.g., KafkaTopic, Kafka, etc.

There are no advanced tools for the automatic creation of builders or already implemented reliable clients in Golang, so the minimal client is implemented by hand. Libraries used for this extension are already mentioned at the beginning of Section 5.2. The minimal client refers here to the fact that we will provide only necessary methods and allow access only to the necessary properties of given resources. Doing more without an automated solution would be counterproductive in cases such as this[5].

Specific methods and reasons for their support are described in Table 5.1.

Liveness stream

Liveness stream is not a template on its own. Instead, it extends existing templates, e.g., *resource-delete* or *broker-pod-delete*. This extension is the actual solution to the problem with Zookeeper, described in the previous section. This part aims to provide a way to ensure that the functioning of Strimzi (from the client's viewpoint) was not damaged and it managed to recover, depending on its specification. We will ensure this with the creation of Consumer and Producer from Section 2.2.2 and check that they work as we expect them, i.e., make sure no messages are lost. There were two ways how to accomplish the creation of Producer and Consumer. First, implement them inside a running container and let it handle all the production and consumption. This solution is straightforward but not a good one, as all responsibilities lie in one Pod that should only take care of the execution of the experiment itself. Instead, we will use a separate container for both

[4]We allow a user to kill the partition leader, this will be explained in Section 5.2.1
[5]Composition of custom resources may change in subsequent versions

Table 5.1: Method supported by client for given custom resources.

Resource	Operations	Usage
KafkaTopic	Get, Patch, Delete, Create	Experiments that support liveness need to support manipulation with Kafka topics. This includes creation at the beginning of the chaos experiment and deleting the given Kafka topic at the need of it.
Kafka	Get, Patch	Kafka custom resource represents the Kafka cluster. Custom resource holds Kafka's desired and actual state by keeping its specification. We can patch this specification and thus (under specific circumstances) cast update.
KafkaConnector	Get	KafkaConnector contains information about workers (from the KafkaConnect cluster) performing tasks. We then can identify these Pods quickly.

Producer and Consumer. Figure 5.3 depicts this situation. A part of the specification given inside *chaosEngine* and *chaosExperiment* resources specifies desired properties of the Kafka topic, Consumer, and Producer. This configuration is retrieved and passed again as environment variables to create containers for the Producer and Consumer. To minimize the number of required attributes provided by the user, we have default values for Consumer configuration. These follow default values for a specific configuration that Kafka also assigns (Appendix C.2), so we also prevent the addition of confusion in parameters. The only responsibility of the Producer image is to implement Kafka Producer with the desired configuration and send the requested number of messages. Respective work is required from consumer image.

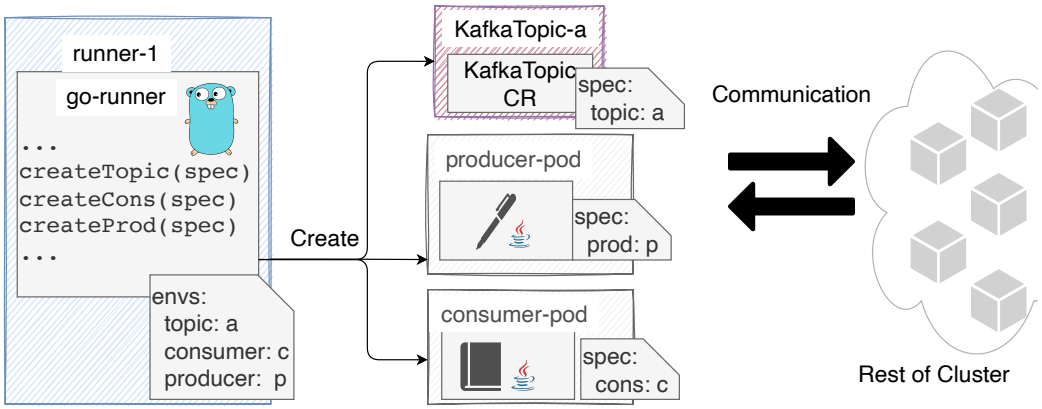


Figure 5.3: Chaos runner creates during its execution producer and consumer containers.

The choice of image for the Producer and Consumer is up to the user. There are no hardcoded scripts that would make other changes impractical; instead, a few commonly known[6] parameters are passed. The extension also provides additional default images,

[6]These parameters are described in Appendix C.

so the user does not need to solve this problem from scratch. Strimzi comes with several examples in its repository[7]. Implementation of the used images is in Java, which is preferred due to the fact already mentioned in Section 2.2.2, which is that Java clients are always the most up to date. Listing 5.4 shows a part of the Consumer's work. Its core is a loop in which the consumer accepts data consumed from the given topic and afterwards commits offset about the position of the last read message.

```
public class ExampleKafkaConsumer{
   public static void main(String[] args){
      // setup ...
      KafkaConsumer consumer = new KafkaConsumer(props);
      consumer.subscribe(Collections.singletonList(config.getTopic()));

      while (receivedMsgs < config.getMessageCount()) {
         ConsumerRecords records = consumer.poll(Duration.ofMillis(Long.
            MAX_VALUE));
         for (ConsumerRecord<String, String> record : records) {
            log.info(record.value());
            receivedMsgs++;
         }
         consumer.commitSync();
      }
```

Listing 5.4: Consumer reading data and committing offset.

A liveness stream can be afterwards used in a chaos experiment by enabling it in the provided environment variable, shown also in Listing 5.5. Here, we enable creation of liveness stream and also specify the name of Kafka cluster which will be targeted by our chaos experiment.

```
kind: ChaosEngine
metadata:
  name: example
spec:
  appinfo:
    applabel: 'app=kafka' # identify pods
  experiments:
    - name: strimzi-pod-delete
      spec:
        components:
          env:
            - name: STRIMZI_KAFKA_CLUSTER_NAME
              value: 'my-cluster'
            - name: LIVENESS_STREAM
              value: 'enable'
```

Listing 5.5: ChaosEngine representing chaos experiment with intention of killing partition leader while keeping Kafka fully available.

[7]Strimzi example clients – https://github.com/strimzi/client-examples

By applying ChaosEngine from Listing 5.5 we can create numerous different situation. The ChaosEngine is configured to kill the partition leader of the test topic without encountering significant disruption in the production and consumption of a given topic. Once the partition leader is killed, a new one is selected from brokers with all the messages. Afterwards, when the broker (i.e., Pod with running Kafka broker) is recreated, it reads all messages produced in its absence and becomes synchronized. Consumer and Producer will notice the change of partition leader and start to communicate instead with the newly elected partition leader. Otherwise, their proper functioning is not disrupted. All of this also depicted in Figure 5.4.

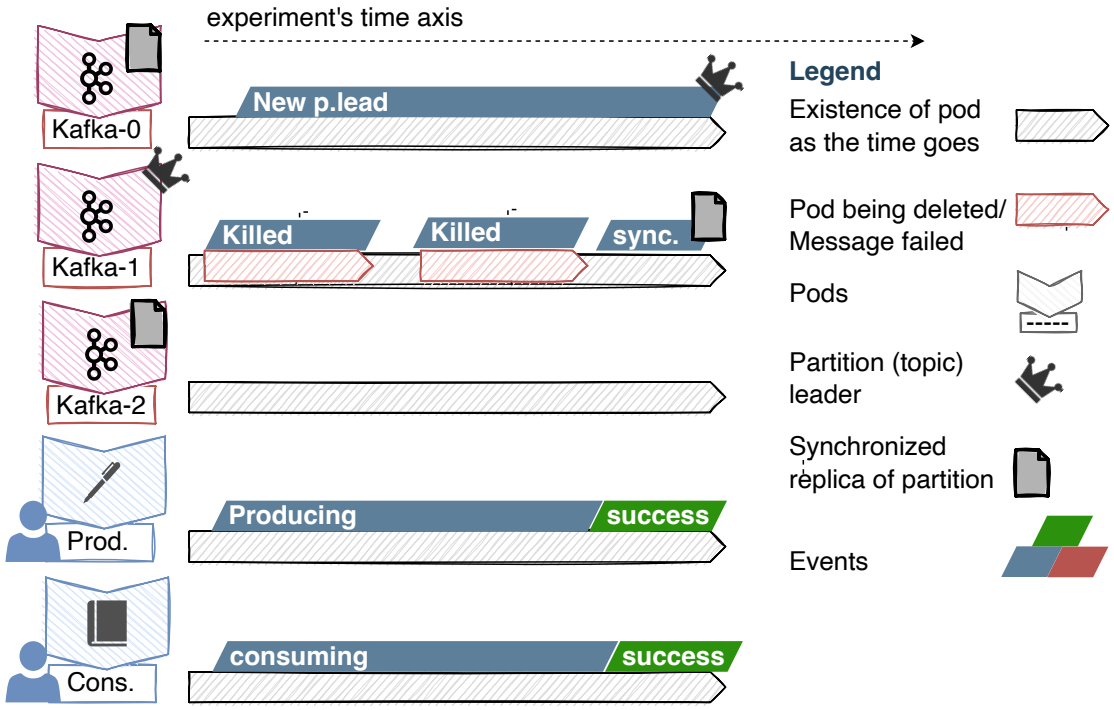


Figure 5.4: Situation in cloud from start to end of *chaos experiment.*

The only trade-off for providing a liveness stream is that it includes many configurable parameters. Even after reducing all unnecessary configuration and rather a simplistic choice of images (the default Producer image is preconfigured regarding the size of messages, which would otherwise need additional configuration), we still introduce fifteen new variables to consider. The solution for this is the implementation of logical defaults, countless examples, and possibly providing additional documentation. The following two templates use precisely this type of liveness stream, and we will continue discussing them with having this part of implementation at our disposal.

5.2.2 Resource Delete

Strimzi creates and updates a lot of other resources in order to work correctly. Essential resources for this purpose are Secrets, Configmaps, and Services. Litmus provides a template for Pod deletion, but not for all other resources we mentioned in the previous sentence. According to one of the developers, this is because there is only rarely some controller (Section 2.1.5) that would recreate them. Therefore a failure of these resources would be

permanent. This behaviour is a shortcoming in the case of Operators. Nevertheless, because Kubernetes already provides clients for communication with native resources, we can use it. Therefore, there was no need to implement a client of our own as in the case of Strimzi custom resources.

The user is allowed to specify resources he wants to delete while also (same as in the case of all other experiments) specifying details about the liveness stream. Deletion of various resources may cause a different set of events. For example, deletion of the specific certificate (i.e., Secret) causes the whole Kafka cluster to restart.

5.2.3 Kafka Rolling Update

There are several ways to initiate updates on the Strimzi Kafka cluster. The easiest is annotation i.e., `strimzi.io/manual-rolling-update=true`. However, to simulate Kafka Rolling Update in a real environment, we will update the Kafka configuration, specifically creating or deleting one internal listener. This configuration of Kafka is held within the specification of Kafka custom resource, which will be patched with the use of an implemented client (Section 5.2.1). The described change of configuration (i.e., a configuration of Kafka custom resource) forces Strimzi to update all Kafka brokers, as their configuration cannot be changed dynamically [8] The experiment expects the update to finish within the expected time while still keeping Kafka fully available.

This behaviour can be seen in Figure 5.5 with some of the actual events accompanying this occasion. Based on the correct configuration of Producer and Consumer, we may assume that cluster remained perfectly available, as all messages were delivered, and what is more, this delivery took place without additional delay.

5.2.4 Worker Delete

Worker is the name of a Pod that constitutes the Kafka Connect cluster. Principles of how Kafka Connect works are described in Section 2.2.4. Regarding Strimzi, Kafka Connect is specified in the KafkaConnect custom resource. The main focus of this experiment is on tasks. Workers which are assigned tasks are identified from a specification of KafkaConnector. Chaos itself includes either deletion of tasked workers or any random workers (this may or may not include tasked workers). Regain of a healthy state is observed by reassigning tasks to new workers.

A similar experiment could also be casting Rolling Update inside Connect cluster. Implementation of this would be simple because most of the needed methods are already implemented. We would again patch the KafkaConnect resource and wait for the recreation of all resources. While Kafka Connect cluster resembles the Kafka cluster, workers (in contrast to brokers from the Kafka cluster) are pending in their normal state. Real work takes place only once some Connector is applied, and therefore KafkaConnect does not represent that crucial component, and this experiment (i.e., another template) is omitted.

5.3 Application of Chaos

This section describes an application of both newly implemented templates and those provided by Litmus. In Section 5.3.1, we propose an extension of the Strimzi system's tests

[8] Strimzi allows changing some of the configurations in such manner, but this must not involve configuration written directly in broker's specification.

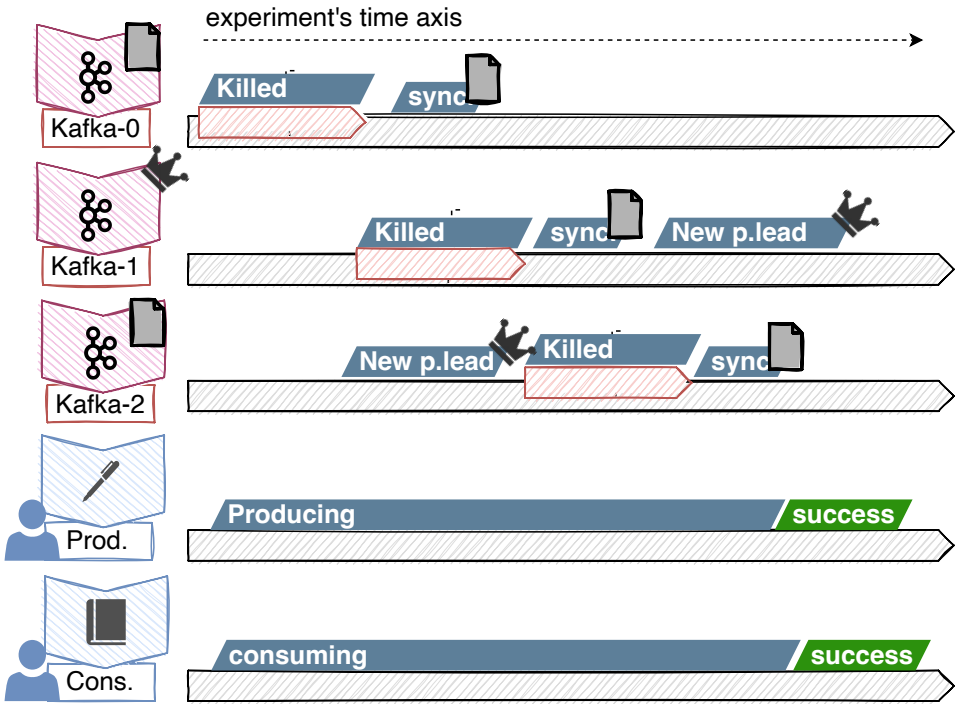


Figure 5.5: All Kafka broker pods are being updated (i.e., deleted and being replaced by never versions). Consumer and Producer works as nothing would happen.

as the primary approach to the implementation of chaos application. In order to follow the Principles of Chaos Engineering and the nature of the tested system as much as possible, This solution is eventually split into two different parts. The first part (Section 5.3.2) is the creation of new tests that run chaos experiments against a simple cluster running Strimzi and its components. The second part (Section 5.3.3) is an application of all suitable experiments against the system running in production.

5.3.1 System Tests Extension

As mentioned in Section 4.4.1, this approach is based on extending the already existing system's tests. The proposed implementation relied on the provisioning of desired Strimzi cluster, which would be accomplished with implemented builders. Once the system under test is up and running, the prepared configuration of ChaosEngines will be applied. Afterwards, there is a period of waiting for the end of chaos by observing either ChaosResult or the state of the system under test. This architecture is shown in Figure 5.6. As shown in the the Figure, the main part of this is the creation of the new *Chaos* package, which will use provided utilities for all provisioning. Firstly, to provide the desired system under test, and secondly, to apply provided ChaosEngines' specifications to produce desired chaos. Concrete test cases then sequentially make these calls and evaluate the results of chaos injections.

This solution bears its pros and cons. As is shown from Figure 5.6, the main benefit of using this approach is that there are many implemented utilities at disposal, e.g., an implemented builder for provision of the desired system under test.

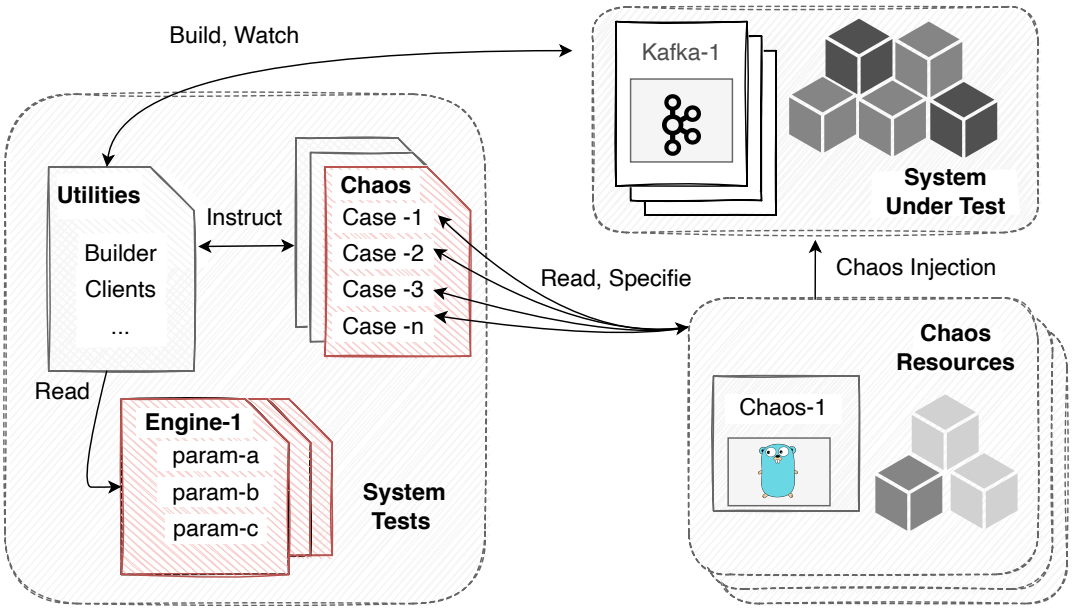


Figure 5.6: Architecture of system test extension

Each chaos experiment can be implemented as a single test case. This test is located inside a chaos test suit and consists of several main parts. After the provision of the system, the implementation is focused solely on chaos, i.e., chaos *injection* and chaos *observation*. Unfortunately, the implementation of these steps proved more complicated than expected. Moreover, the possible solutions brought other divergences from principles of chaos engineering or involved lousy software engineering practices. The following points summarize some of these factors:

- **Environment** – System tests run against environment created only for the reason of testing. One of chaos principles (Section 3.1.3) is to inject chaos as close to production as possible.
- **Continuos run** – It is essential to execute chaos experiments often and continuously. We can then see if there are some long term damages or unexpected conditions. System tests allow automation and periodic run, yet this can cover only the first half of demands regarding the running of chaos.
- **Communication** – Parts of presented utilities are also clients for communication with Strimzi and Kubernetes resources. However, most of the communication this implementation requires is amongst test and Litmus components. This will require the addition of the client, which will overload the existing one.
- **Performance** – Even running around thirty chaos experiments may take around three hours to complete. This number is due to the fact that some events in the cloud take a longer time, e.g., updates. The additional reason is that when we start to execute a set of experiments one after another, instead of being completely isolated, several new events occur. For example, one experiment may bring down several components whose absence lengthens the next experiment dramatically, maybe even causing it to fail. The second additional factor comes from the fact that reflection of desired changes goes throw a lot of components (Section 2.1.6), and propagation takes additional time per each experiment.

- **Observabillity** – Aside from test results and possible control of Strimzi itself, there is very little to no observability of events that takes place in the long run. This is a direct result of the fact that a completely new system is introduced each running time or that most of its components are replaced.
- **Additional chaos experiments** – Some experiments are not possible due to the choice of Litmus. We are still able to compensate for (Section 6.2.4), but not inside the system test.

This solution bears countless benefits (see Section 4.4.1), but to harvest the most from chaos engineering and good software principles, the two additional approaches were implemented. Firstly, there is the implementation of a new testing repository, which will solve problems of performance and communication. Secondly, chaos will be applied to the long-running system, which will solve the rest. These solutions are described in the following Sections.

5.3.2 Chaos Test Suite

As mentioned in the previous section, the approach taken concerning applying chaos is split into two parts. The first one (i.e., this part) is implementing a new repository to automate the testing of a simple Strimzi cluster. Thus the name *Chaos testing* is actually on purpose. The implementation itself is in Golang. This choice is made because most communication will occur with Kubernetes and Litmus' resources, all written in the same language. The main goal is to provide a maintainable, automated way of running chaos experiments. Figure 5.7 depicts all essential steps of the implementation.

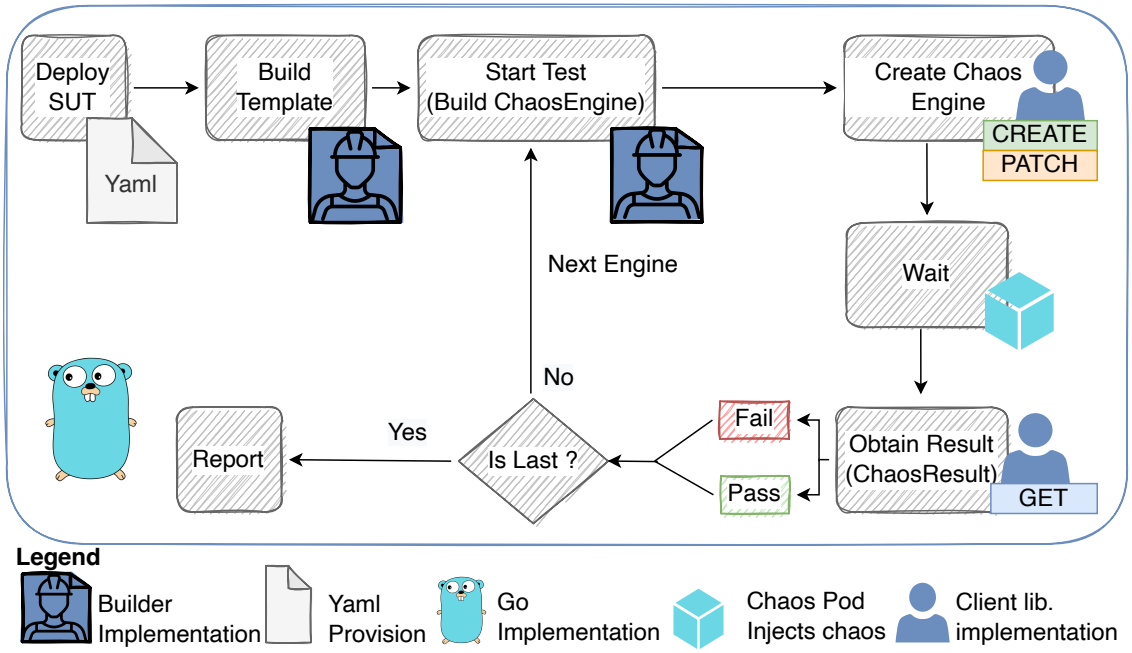


Figure 5.7: High level workflow of testing.

The implementation starts with an assumption (and also requirement) that the user has admin access to the cloud[9], which has Strimzi and Litmus operators installed. The

[9]This cloud can be Minikube, Openshift, etc.

first step is the provision of the desired cluster and all additional resources. This step (i.e., Apply System Under Test) is the only one where YAML files are applied instead of working directly with the code. Some of this configuration is necessary[10] as there are multiple different resources (e.g., RBAC[11]) that need to be configured.

With all prerequisites done, the provision of the system begins. This system is simplified in Figure 5.8. It is a small system with Kafka, Zookeeper, Kafka Connect cluster, Kafka Bridge, and some additional components (e.g., Services, Pods) managed by Strimzi. Test cases are implemented so that the user can specify desired replicas of each component, and they will still work correctly. The scale of system is chosen for simplicity and the possibility of running locally. All important setup is already preconfigured. The only important thing here is having a cluster with known names, labels, namespaces, desired numbers of replicas, and other necessary configurations.

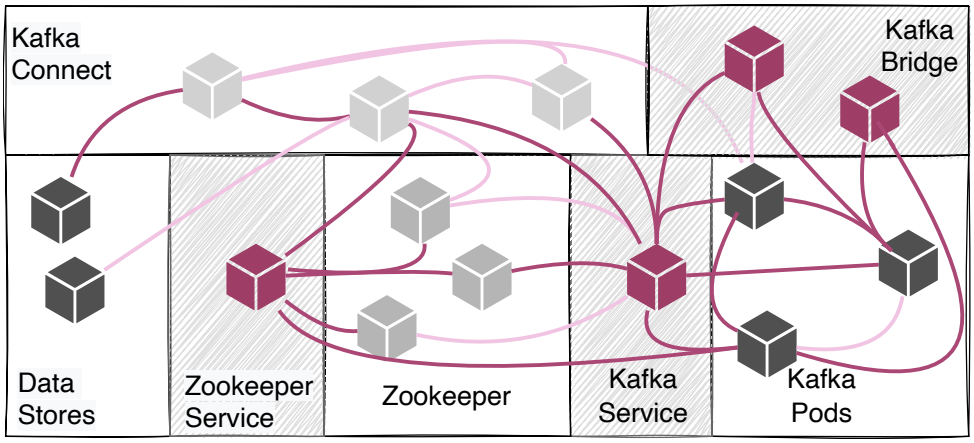


Figure 5.8: Ilustrational schema of SUT.

Builders

The following two steps in the workflow (Figure 5.7) are the installation of Templates[12] and the building of ChaosEngines. The decision to build all chaos-related specifications (i.e., ChaosEngines and Templates) inside the code has two main reasons.

The first one is maintainability, as keeping tens of declaration files may become hard to keep up with. Furthermore, every important change would need to be copied to each of these files. This is tedious, even on a scale of a few experiments, not to mention hundreds of them.

The second reason is the clarity and focus of implementation. For example, Listing 5.6 shows the creation of the ChaosEngine, which will spawn the chaos experiment responsible for the Kafka Rolling Update and all related events. Whereas specification of this experiment using builder remains focused and takes just a few lines of code, specification of the precisely same ChaosEngine would otherwise take forty lines in YAML manifest[13].

[10] It was also possible to use the client for the creation of these resources inside the code, but it would be hard to follow why there is such vast implementation for one-time usage.

[11] Role-based access control (see Appendix B).

[12] This is a reference to the actual object ChaosExperiment.

[13] Part of this repository is also a folder with examples, which holds more than fifty YAML manifests which were initially supposed to be used in formerly proposed implementation (Section 5.3.1) .

Moreover, we would still need to reference these pre-created ChaosEngines from the code by the file's name.

```
func Test_kafka_update_1(t *testing.T) {
    engineName := "ku-pass"
    chaosEngineBuilder := engines.ChaosEngineBuilder()
    chaosEngine := chaosEngineBuilder.
        SetName(engineName).
        // term Template used instead of ChaosExperiments in thesis
        SetExperimentName(experimentNames.KAFKA_UPDATE).
        SetAppLabel("app.kubernetes.io/name=kafka").
        AddEnv(envNames.END_CHAOS_INJECTION_ASAP, "enable").
        AddEnv(envNames.CHAOS_INTERVAL, "400").
        AddEnv(envNames.STRIMZI_KAFKA_CLUSTER_NAME, "my-cluster").
        GetEngine()
    engineEvaluationPass(t, chaosEngine)
```

Listing 5.6: ChaosEngine builder

Creation of Templates follows the same pattern as the creation process of ChaosEngines, but with its own builder.

Clients

With ChaosEngine resources created locally, all that needs to be done to start chaos is to Post this object to the Kubernetes API server; the chaos operator handles the rest. The Kubernetes API client implementation is mandatory for this step. To implement it, we provide the implementation of a similar method as in the case of the Strimzi client (Section 5.2.1). From an implementation viewpoint, we have to provide all serialization, models, and additional methods to provide the necessary attributes to implement the Kubernetes API interface. The most crucial resource, in this case, is *ChaosEngine*, and the implementation of its API interface is shown in Listing 5.7. By implementing all necessary methods for manipulation (e.g., read, create, delete) for ChaosResults, ChaosEngines, and Templates, we have all the required means regarding communication for the purpose of chaos testing.

```
type EngineInterface interface {
    Get(name string, options GetOptions) (*ChaosEngine, error)
    Create(engine *ChaosEngine, opts CreateOptions) (*ChaosEngine, error)
    Delete(name string, opts DeleteOptions) error
}
```

Listing 5.7: Create ChaosEngine API method implementation.

Communication

The simplified model assumes communication of four main entities, i.e., Litmus Operator, ChaosEngine, Test client, and chaos Job. This flow of communication shown in Figure 5.9. ChaosEngine serves as a data structure for storing/providing information. The rest of the entities must implement a client for communicating with this c*ustom resource*. The chaos operator triggers the creation of chaos Job each time it sees some ChaosEngine

resource in the active state (i.e., with a given configuration). All the steps between posting ChaosEngine resources to the Kubernetes API and the actual creation of chaos Job are described in section 3.2.2.

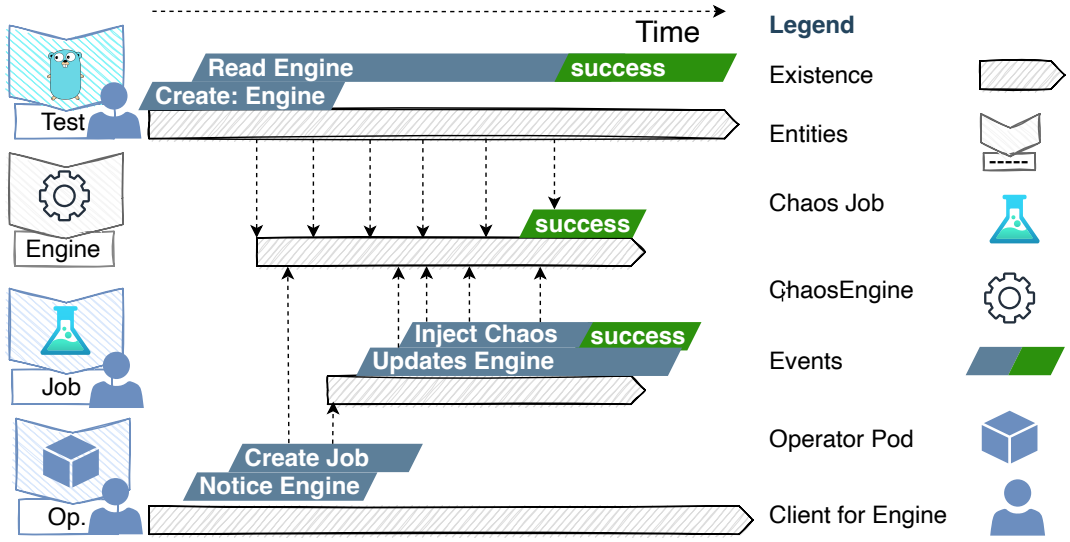


Figure 5.9: Communication between test client and ChaosEngine.

When chaos Job starts, it continuously updates ChaosEngine and ChaosResult resources. From the tests' viewpoint, the chaos remains *completely transparent*, i.e., tests only wait for the result and possibly other information. Of course, in reality, the client also communicates with ChaosResult, but in practical terms, ChaosEngine can serve as a single source of truth for a given experiment.

Evaluation

Each test from this section fails or passes according to obtained result from the ChaosResult custom resource. However, when a chaos experiment is done, several components involved in chaos may still be accessible based on a specified policy, e.g., chaos Job, Producer, Consumer.

Although the implementation of Templates follows the idea to keep all manipulation on the components level (templates evaluate results of Jobs instead of directly entering Pods), in comparison with other templates, it also contains additional tunable logging, as these logs could provide some additional info while debugging unexpected conditions in the cloud. Furthermore, this represents the possibility of obtaining additional trends and metrics, which will be further discussed in Section 6.1.

5.3.3 Production Environment

When it comes to chaos Engineering, it is essential to include experiments also in production. We build confidence in the environment we apply chaos in, and despite Strimzi's specific demands, resilience of generic clusters represents only one side of the coin. The system which will be used is part of ExcelentProject[14]. The purpose of this project is to provide long term observation of Strimzi in production. The system itself resembles the

[14]More at https://github.com/ExcelentProject

one from Figure 5.8, but this one is on a much bigger scale, with several other components (e.g., Cruise Control) and running a monitoring mechanism (More on that in Section 6.1). There are two most important factors to consider once running chaos inside the production environment. Firstly, we need to take into account aspect of the access control. This is briefly described in Section 5.3.3. Secondly, we also need to reconsider what types of chaos experiments we will apply, precisely, the exact configuration of these chaos experiments.

Access Control

Part of Strimzi we did not consider that much yet is its access control. The entities that Kafka's authorization uses are called Access control lists. For the sake of clarity, this access control is different from Role-based access control (Appendix B), which is used in Kubernetes.

Strimzi provides ways of specifying desired authorization and authentication for access and security purposes. Using KafkaUser custom resource, the admin can create a role inside the cluster. Afterwards, authorization is managed by specifying desired rights as part of the custom resource specification. This style of access control follows the one already used by Kafka (access control list), which specifies allowed operations per given resources. An example of KafkaUser custom resource is shown in Listing 5.8. Strimzi supports different style of authentication, (e.g., TLS[15], SCRAM-SHA[16]), and handles all work regarding creation and signing of certification. Authentication is configured independently for each listener. Authorization is always configured for the whole Kafka cluster [8]. This is shown in Listing 5.9.

```yaml
kind: KafkaUser
metadata:
  name: litmusUser
  label: strimzi.io/cluster:my-cluster
spec:
  authentication:
    type: tls
  authorization:
    type: simple
    acls:
    - resource:
        type: topic
        name: litmus-*
        patternType: literal
      operation: Read
```

Listing 5.8: A KafkaUser custom resource with access to read all topics with Litmus prefix.

```yaml
kind: Kafka
metadata:
  name: production-cluster
spec:
  kafka:
    ...
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: enable
        authentication:
          type: tls
    authorization:
      type: simple
```

Listing 5.9: A part of Kafka specification which sets authentication on given port and authorization in whole cluster.

[15]TLS mechanism – as https://datatracker.ietf.org/doc/html/rfc8446
[16]SCRAM-SHA mechanism – as https://datatracker.ietf.org/doc/html/rfc7677

Once an appropriate KafkaUser custom resource is provided, the only thing the user has to do is use data stored in the respective Secret resource and provide them as part of configuration properties when communicating with Kafka cluster. Because different types of authentication may require different types of parameters, templates need to accept, evaluate, and propagate many different variables. Therefore, the implementation of templates solves the issue of provisioning such a vast number of possible different variables by providing the choice of specifying custom environment variables and the desired authentication method.

Chaos Experiments

With access to Kafka resolved, we can prepare all adequate chaos experiments. However, in comparison with applying chaos inside the test environment (i.e., a cloud that lives only for the duration of chaos), applying chaos to production may require other adjustments, i.e., not all experiments applicable in the test environment are suitable. On the other hand, we will also include some of Litmus's original chaos experiments.

Naturally, a different environment means different names and labels, a relatively minor factor. However, the difference in provided memory, number of available nodes, CPUs, and additional constraints is something completely different. This requires addition of several parameters in several chaos experiments. The most crucial factor is that this system runs on the Openshift cluster.

In terms of chaos experiment configuration, this plays a significant role. This is because Openshift uses crio container runtime, making some chaos experiments more complicated, as they often work with runtime directly, and some of the templates support docker runtime exclusively. The second limitation is that Openshift does not provide an option to connect to the node directly, which makes most of the chaos experiments focused on nodes impossible (we will demonstrate one of these chaos experiments anyway, but more on this in Section 6.2.4).

Scheduling

The only remaining factor is the automatization of periodic chaos. There are several considered possibilities. Among them, the most interesting are usages of the ChaosScheduler or Workflow custom resource.

The first implemented approach is the usage of Workflow custom resources. The motivation for preferring this approach to ChaosScheduler is that Litmus makes steps (e.g., keeping documentation updated and offering semi-automated creation of resources from the user interface) to make Workflow a default way of chaos scheduling. Workflow custom resource has nothing to do with Litmus. It is part of Argo, and all it does is execute defined steps.

The steps defined in this Workflow represent the actual application of custom resources, which will consequently trigger chaos experiments, also shown in Listing 5.10. The mentioned workflow consist of goals, and each of goals consequently consist of steps. Afterwards, each step is nothing more specification of specific commands which are supposed to be executed. The silent rule of Litmus in regard to workflow is that first goal is always called chaos. Because most of configuration is supposed to be done automatically, user may stay unaware of this fact.

```yaml
kind: Workflow
metadata:
  name: example-wf
spec:
  goals:
    - name: chaos
      steps:
        - name: install-chaos-experiments
        - name: pod-delete
    - name: install-chaos-experiment
      inputs:
```

Listing 5.10: Simplified specification of steps in Workflow custom resource. The goals is chaos which consist of steps install-chaos-experiment and pod-delete.

The primary motivation from Litmus's viewpoint is that they want to hide internal dependencies of ChaosEngine, EchaosExperiment, and ChaosSchedule custom resources. Nevertheless, this resource bears more cons than pros for our use case (and many others). Firstly, the specification of Workflow custom resources is very wordy (two hundreds line of configuration on average), as they need to cover many steps that are to be done and incorporate other resources' specifications. Moreover, because Workflow keeps the specification of other resources as a string value, it does not offer any validation before the actual attempt of creation.

The second problem is that this approach is still relatively young and not crated by Litmus natively, which bears additional negatives from a practical viewpoint, e.g., the need for additional extensive Controler. An additional level of abstraction and hidden manipulation with custom resources may also cause unexpected behaviour, e.g. if Template already existing in the cluster specification provided in some part of Workflow will not be applied. Instead, this Template is bound to the newly created ChaosEngine. The most significant problem is that Litmus still does not provide required manifests with a specification of Kubernetes objects needed for the correct working of the Workflow component. Whereas we still can investigate this problem and configure our resources (this configuration can be found in provided implementation), it would bear additional negatives in the scope of security and access, which we cannot allow in production, i.e., we would provide too extensive access to entities responsible for chaos Scheduling. To conclude, the usage of Workflow is an excellent idea for development. Once Litmus provides all necessary means to specify these resources and provide all necessary Kubernetes resources responsible for granting desired minimal permissions, it will also be suitable for the production environment.

ChaosScheduler is a simple abstraction over ChaosEngine that creates or activates given ChaosEngine every given time. It is another custom resource with a straightforward Controller (i.e., Operator). ChaosScheduler custom resource itself contains all necessary details for creating exactly one ChaosEngine and several additional fields that specify desired scheduling. For example, listing 5.11 depicts a chaos experiment which will be injected every fifth minute. The `schedule` part of the specification is later translated into individual cron expression and afterwards invoked adequately. Invocation later results either in creation of brand new ChaosSchedule custom resource, or configuration of the existing one.

```yaml
kind: ChaosSchedule
metadata:
  name: scheduler-example
spec:
  schedule:
      properties:
        minChaosInterval:
            everyNthMinute: 5
  engineTemplateSpec:
    ...
```

Listing 5.11: Simplified specification of ChaosSchedule custom resource specification. The rest of the specification is afterwards very simmilar to the one from Listing 5.5.

The only job remaining is that either an admin or a script initiates these resources (Figure 5.10). Communication after the creation/activation of ChaosEgine is already described in previous parts. The most significant benefit of this approach is its simplicity. A correct application of this approach in the Opensshift cluster required significantly smaller overhead regarding additional resources and very few new resources for necessary access requirements.

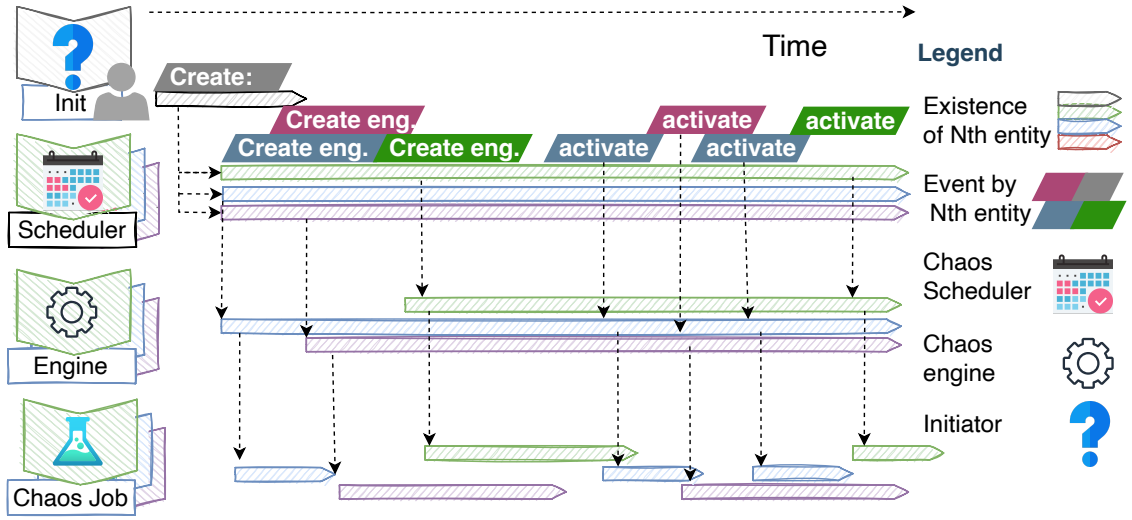


Figure 5.10: Role of ChaosScheduler in automatic scheduling and applying of chaos.

Chapter 6

Monitoring, Evaluation, and Experiments

This chapter completes the practical part of this thesis. Firstly, Section 6.1 covers monitoring of chaos, *why we want it*, *how it is accomplished*, and what benefits it brings to our solution. Afterwards, Section 6.2 discusses all details about all different types of injection of chaos, the impact it has, results we obtained, and the importance it has for the system as a whole. This part concludes all chaos experiments (created by templates), which we apply in both production and generic environment.

Once we evaluate the resilience of these systems, we will discuss and demonstrate additional alternative experimenting (e.g., proposal of changes in system configuration, new ways of chaos injection, etc.), which may either increase system resilience or be otherwise valuable to consider regarding applied chaos engineering.

6.1 SUT Monitoring

Although monitoring can mean many things in many spheres, we will focus solely on the purpose of chaos monitoring (and its overall impact on the system). Therefore, in Section 6.1.1, the discussion starts with the establishment of the situation and creating the case for monitoring. Afterwards, Section 6.1.2 describes tools used for this purpose, and lastly, Section 6.1.3 shows what and how needs to be set up in order to monitor chaos correctly.

6.1.1 Motivation

Consider a Kafka Rolling Update chaos experiment with three Kafka brokers and a liveness stream with a minimum of two synchronized replicas[1]. The rest of the configuration is set to make the Producer resistant enough. We expected the experiment to pass, but it failed, and our most accurate source of data are the statuses of Pods and logs (including error messages) from chaos Pod (Listing 6.1) and from clients (we consider only Producer), which is visible in Listing 6.2.

[1] The configuration means that unless at least two replicas (including the replica contained in the partition leader) write a message, the Producer is not acknowledged about successful write. For further explanation see Appendix C.1

```
time="22:56" msg="[Info]: Chaos injection begins
time="22:57" msg="[Wait]: Chaos Interval Duration=400
time="23:17" msg="[Wait]: Time 20/400. Updated kafka pods:1/3"
time="24:17" msg="[Wait]: Time 80/400. Updated kafka pods:2/3"
time="25:46" msg="[Wait]: Time 179/400. Updated kafka pods:3/3"
time="25:42" msg="[Info]: ending successful chaos injection ASAP"
time="25:44" msg="[Liveness]: Producer failed"
```

Listing 6.1: Logs from chaos Job

```
2022-04-15 22:23:17 INFO - Sending messages "Hello world - 1"
2022-04-15 22:23:18 INFO - Sending messages "Hello world - 2"
2022-04-15 22:23:19 INFO - ISR Set(2) is insufficient, have 1"
```

Listing 6.2: Logs from liveness Producer

Even if we use all logs, events, and results, we may still be only hardly able to reconstruct what led to this situation. Meanwhile, the actual root cause is depicted in Figure 6.1, showing us that while one of the three brokers was updated, another one was eventually killed by a different chaos experiment. Then, suddenly the logs make complete sense, and the situation becomes understandable.

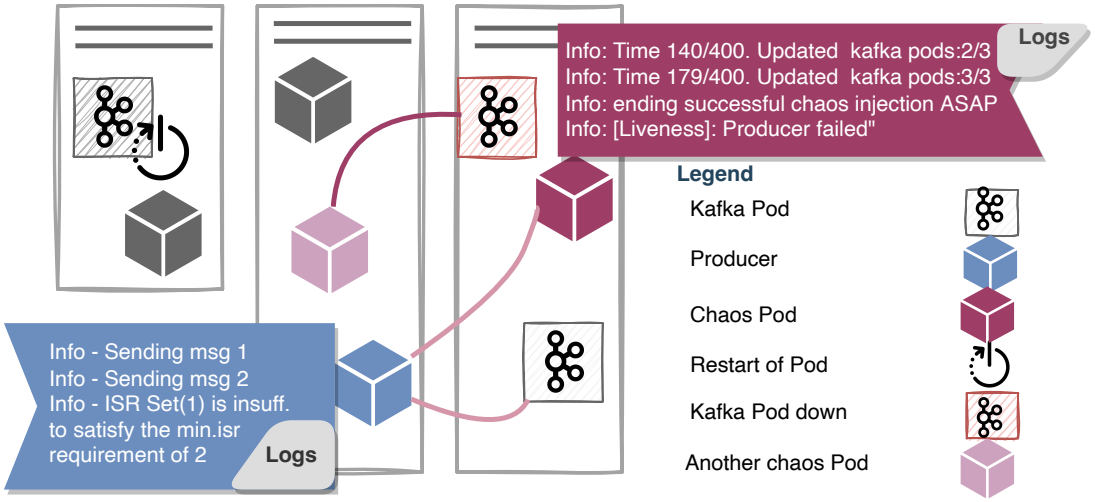


Figure 6.1: The situation inside cluster during a failure of the chaos experiment.

With the proper tools, we can set up our cluster to collect desired data and help us understand the situation in the cluster at a given moment.

6.1.2 Tools

The primary tool we will be using to obtain and evaluate data is Prometheus. It is an open-source, metric-based monitoring system with its own data model and query language [9]. We will simplify it to three main components (Figure 6.2), i.e., Retrieval, Storage, and HTTP server.

Data (i.e., metrics) are obtained using the pulling method. In other words, monitored components expose data (e.g., CPU status, request count, space usage), and Prometheus

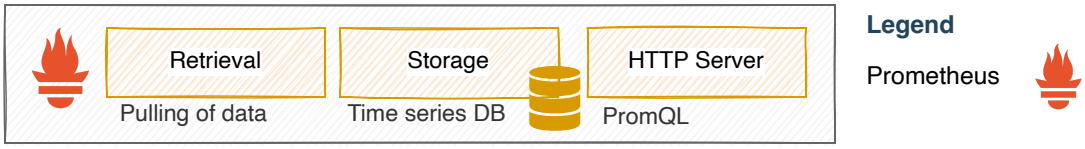


Figure 6.2: Prometheus simplified architecture

scrape them. Data are stored in a time-series database. We can store three types of data: counters, the current value of a given variable, and histograms. Furthermore, these data are finally exposed by the Prometheus HTTP server for further use. We can request data from this server with the help of the PromQL[2] language.

6.1.3 Configuration

With the Prometheus server up and running, all that needs to be done inside the cluster is configuring data sources (i.e., components that expose their data) and later query the Prometheus server. The three main parts of the cloud that are of interest for obtaining data relevant to chaos are Kubernetes, Strimzi, and Litmus. The following description is also illustrated in Figure 6.3

The exposure of data can be accomplished with the use of libraries. Because Prometheus is practically a standard in monitoring technologies, Kubernetes already implements automatic exposure of all metrics it has (e.g., pods, nodes, CPU usage) by default. Litmus and Strimzi simplify the process of configuring monitoring by providing exporters. An exporter is usually a Pod or container which monitors all crucial data from a given system and exposes its own HTTP server with a metric endpoint available. The provision of a Strimzi exporter and a Litmus Exporter is accomplished using additional Deployments.

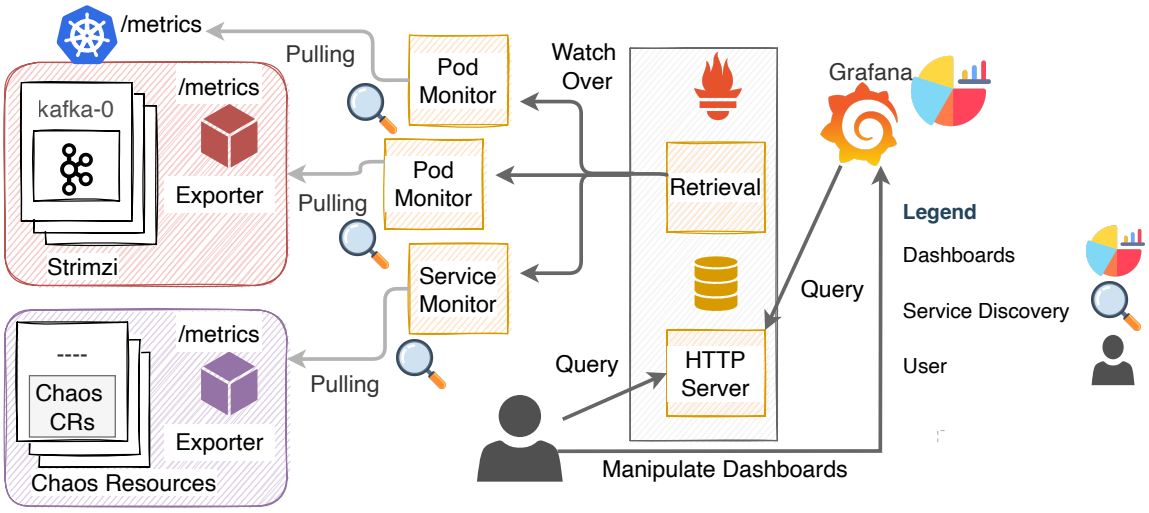


Figure 6.3: Prometheus simplified architecture.

Retrieval of data is done by querying the HTTP server. We will also use additional tool that will help visualise these data. This tool is called Grafana[3]. It is a simple, yet powerful

[2]More at – https://prometheus.io/docs/prometheus/latest/querying/basics/
[3]Grafana official website – https://grafana.com/

visualization application. It queries the Prometheus server, so some provided visualization will be created without the need to even communicate with Prometheus directly.

The last part that needs to be done is creating a new Dashboard. The dashboard structure can be relatively simply specified using Grafana UI. However, the correct provision of data is more complicated, as we use Promql, which has several unique concepts, and a correct application is rather a slow process.

6.2 Running and Evaluation

The evaluation covers all platforms, provided systems under tests, specific templates (and chaos experiments based on them), and different types of sequences used to test the resilience of Strimzi and systems that use it. The section starts with a brief description of what needs to be set. Afterwards, Section 6.2.2 describes sequences of application chaos experiments and their consequences. Section 6.2.3 covers the results of chaos experiments, system resilience as a whole, and also its components. Finally, Section 6.2.4 describes the way to accomplish other beneficial traits and covers some details about additional experiments.

6.2.1 Setup

Chaos Engineering on Project Strimzi was a unique task right from the beginning. The fact that Strimzi is an Operator (Section 2.1.6), and one specific Application, we need to cover several additional factors. The solution should cover both main orchestration platforms, i.e., Kubernetes and Openshift. Another concern is that we want to induce chaos into systems with a different configuration of Strimzi. We simply keep the same naming and layout of provided systems under test to accomplish this. By doing so, we can reuse most of the manifest in several configurations[4]. The only exception to this is Tealc (Section 5.3.3) which has its own layout, naming convention, components, etc. Therefore we will need to use different sets of chaos experiments. Moreover, different combinations of these chaos experiments, their durations, and order will also be used. In any of these cases, monitoring is set up. Figure 6.4 summarizes this reasoning.

6.2.2 Sequences

Injecting chaos from the test suit is straightforward. We simply run all of the tests (with parallelism disabled). This is done because most chaos does not make sense if run in parallel mode, and specific chaos experiments would mostly fail on waiting for the steady state. So instead, a new ChaosEngine is posted to the API server (and afterwards converted into an appropriate chaos experiment) only once the previous one is finished. The gain from this approach is that we inject chaos in an unknown order and ensure that the system withstands the chaos; it also regains a state capable of handling other unknown chaotic conditions. Figure 6.5 depicts this kind of chaos execution, where we can see a range of time in which the given chaos experiment (these experiments are focused on the deletion of different resources) was active.

Applying Chaos with ChaosScheduler means that we can simply specify cron-like expressions with desired scheduling of the given chaos. Scheduling was done in such a manner

[4]This is also accomplished thanks to the fact that Templates implemented for using Strimzi are not dependant on specific kinds of Orchestration Platform.

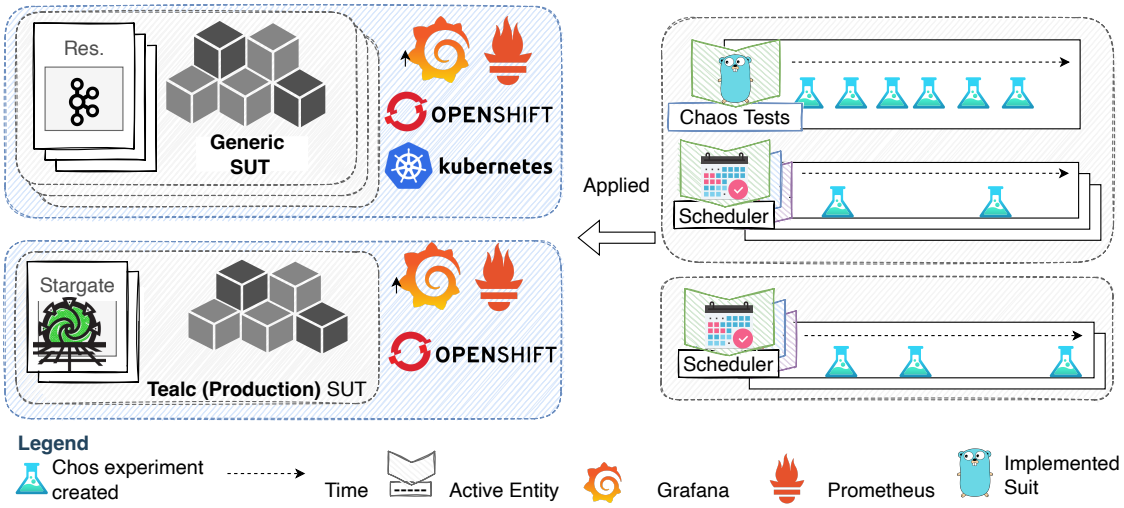


Figure 6.4: Architecture of running chaos.

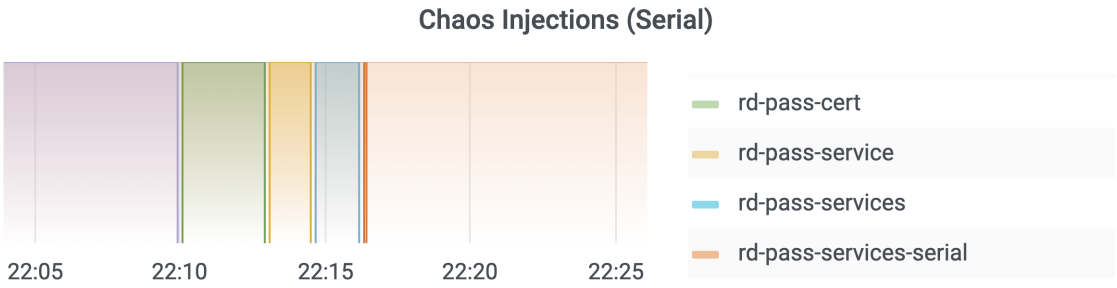


Figure 6.5: Serial chaos experiments execution sequence.

that as long as it could make any sense, different chaos experiments sometimes overlap. This brings systems under even greater stress. An example of this execution can be seen in Figure 6.6 where we can see that while the chaos experiment with the aim of casting Kafka Rolling Update was taking place, another chaos experiment started deleting ConfigMap and Services (i.e., Resources) belonging to Zookeeper of given Kafka.

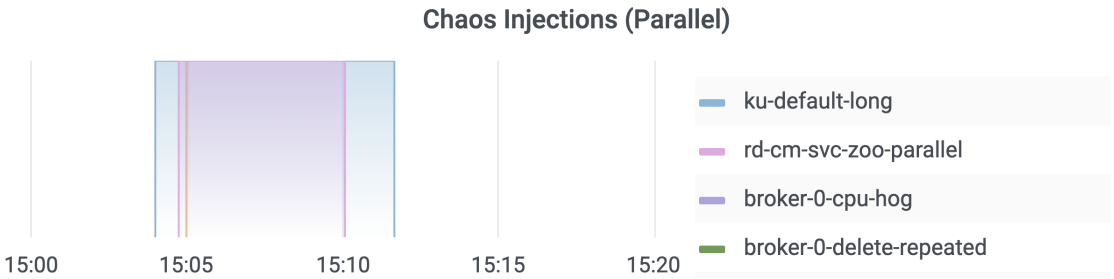


Figure 6.6: Parallel chaos experiments execution sequence.

6.2.3 Results

When injecting Chaos into systems under test (including Tealc), we used thirteen Templates. Nine of these templates are implemented by Litmus, i.e., memory hog, CPU and

traffic spikes, killed Pods and containers, a restart of node, corrupted network, etc. The remaining four templates are part of this thesis implementation and are already described in previous Sections. The results we gained are obtained by monitoring the execution and impact of each chaos experiment and by long-term monitoring of key attributes of systems. Therefore, most trends, results, and behaviours explained in the following lines will focus on critical components and factors that play an important role in the system's functioning.

The most notable of factors here is that the system keeps on repairing itself despite numerous choas injections. System Strimzi is additionally used in many other projects (e.g., Managed Kafka[5]), the key factor which is also the most common part of SLO[6] incorporated by these project is service availability. Therefore, we will also start our evaluation by considering the exact same factor. Figure 6.7 depicts how easily Operator can recreate deleted/damaged Pods. Kafka brokers are recreated in a few seconds despite facing chaos of various scales; therefore, most of the simple choas injections are not even visible[7], so the number of broker Pods is visible only when several chaoses overlap and make some of the brokers unavailable for a more extended time.

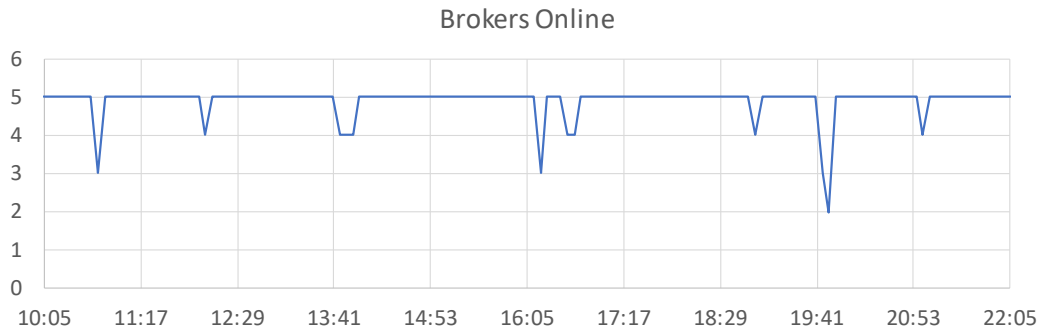


Figure 6.7: Resilence of Brokers.

Claim that chaos experiments are passing as expected almost always bears two important consequences. Firstly, the single chaos experiment pass bears assurance that the system can withstand given chaos if it is in a given situation (i.e., in a given state of components and with expected time to recover all concerned parts). As will be demonstrated in further lines and figures, Strimzi is perfectly capable of recovery and withstanding the type of chaos expected from a given configuration. Secondly, it is impossible to claim one hundred per cent success when injecting chaos into the production environment; This is only natural as spikes, traffic, or any other problems may occur even during the kind of chaos which cannot afford such a thing.

To clarify the impact of chaos, we will briefly describe chaos from one of the metrics, which are of the customers' concerns, i.e., produced messages per given time. Another reasonable metric is message lag, but it is very variable even in normal conditions (the network would be a too variable factor to cover). Figure 6.8 shows the traffic[8] in a cluster

[5]Managed Kafka – More at https://www.redhat.com/en/blog/introducing-red-hat-openshift-streams-apache-kafka

[6]Service level objective are agreed upon as a means of measuring the performance of the Service.

[7]If for example Pod is deleted and recreated within such a short time as ten seconds, it may be not even visible (as scraping in the cluster usually takes place with larger steps between two individual data pulls).

[8]This traffic is still relatively low. Strimzi in a given configuration is capable of handling a much higher load, on a scale of tens of thousands of messages per second.

without injecting chaos on purpose. Nevertheless, there is visible that sometimes traffic fails even without the interference of chaos experiments.

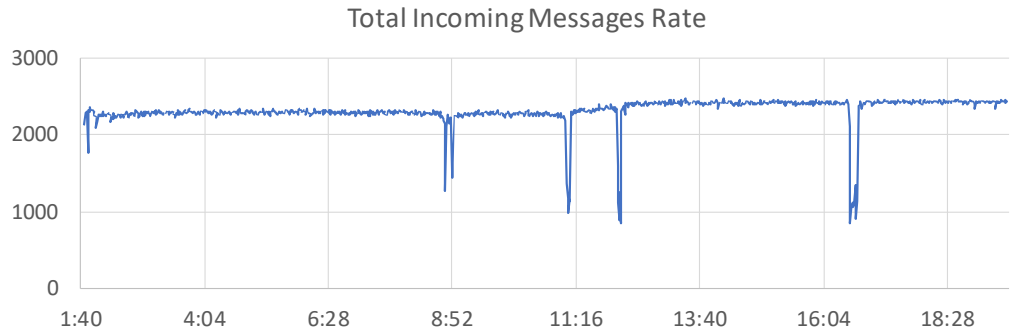


Figure 6.8: Day in a life of the system without chaos.

Nevertheless, after applying chaos for an extended period, we will end up with results similar to those depicted in Figure 6.9. There are no dramatic spikes in traffic (which are used here to indicate the overall state of the cluster within a single diagram). The scale of chaos induced is in such a scale that experiments may overlap, but if so, usually some less damaging chaos, e.g., memory or CPU spikes. The visible spike in traffic is caused due to messages being suddenly delivered at once.

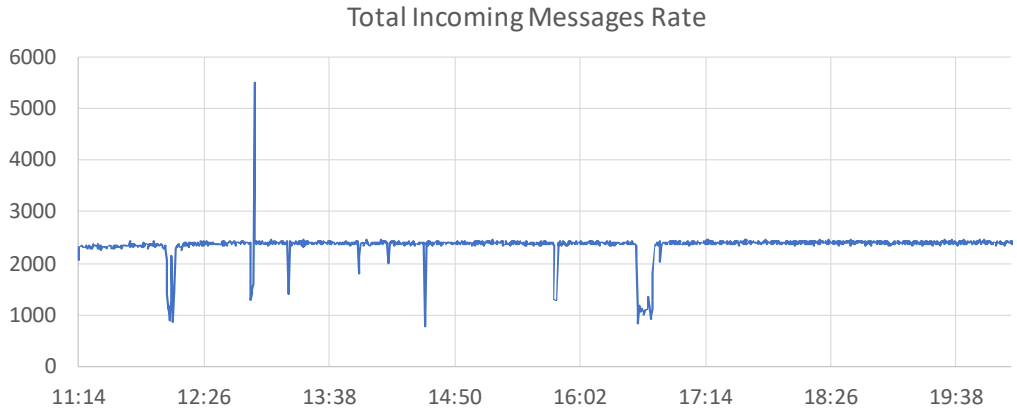


Figure 6.9: System under reasonable chaos.

Of course, if we keep on injecting chaos frequently, allow different chaos to overlap and focus chaos primarily on sources of messages, we could bring down the number of incoming messages to zero, but chaos of such extent is practically equal to bringing down the whole infrastructure.

The two other crucially essential parts of the cluster which have to work correctly are Kafka Connect and Zookeeper. Zookeeper plays a significant role in keeping all metadata and allowing most of the needed election, which takes place inside Kafka. The chaos introduced here was in the form of spikes, failed containers, and deleted Pods. Figure 6.10 depicts the capability of Zookeeper to handle Pod failures. Although Zookeeper can work correctly only while at least most than half of the nodes remain working correctly. Because Strimzi assigns Volumes (Section 2.1.4) to these Pods, they can be easily recreated even after the failure of all of them without losing any of the data.

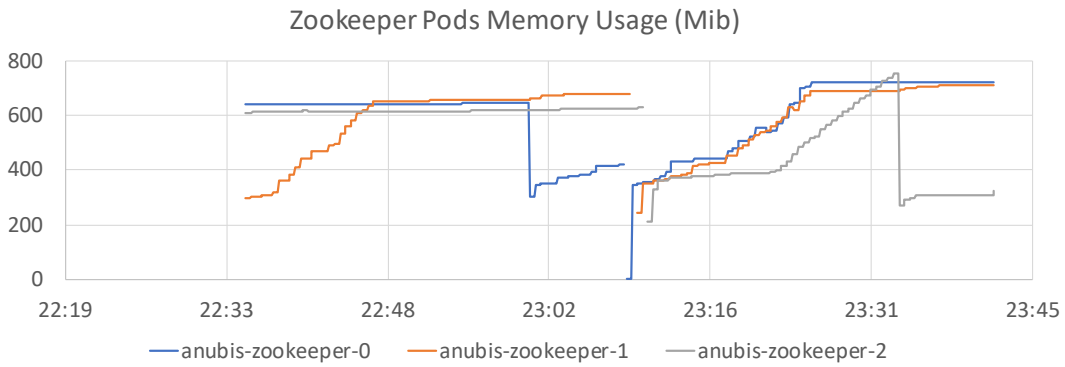


Figure 6.10: Zookeeper recovery.

When it comes to the Kafka Connect cluster (Section 2.2.4), the cluster itself handles failures just as smoothly as the Kafka cluster itself. Any time one of the tasks is killed (i.e., Worker assigned with this task), the new one is created within seconds, and the new assignment takes place in a short time after, as also visible in Figure 6.11.

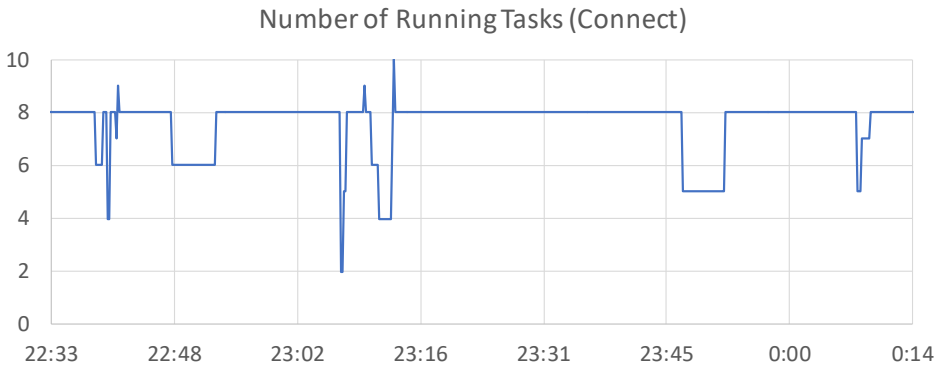


Figure 6.11: Kafka Connect tasks recreation

The only difference here is that Workers are part of Deployment instead of StatefulSet. The implication is that new Pods of created each time the previous one fails; instead of recreating the one with the given name, the new one is created. Figure 6.12 shows Workers and how they handle failure, there is also visible that the load on given Pods is eventually more or less balanced.

Besides the capacity of the critical components to handle failures, there are also several other factors. For instance, memory usage and potential drain of all resources. One of the templates wastes a big part of memory available to a given Pod. The problem may occur once the application starts to waste resources on such a scale, making the scheduling of new Pods impossible.

Tealc does not need to take care of this as there are enough resources to cover spikes needs and the fact that it is a system built especially for handling this kind of traffic and conditions. When it comes to smaller clusters, Kubernetes allow the specification of Limits on given resources (Strimzi provides options to specify these limit in Kafka custom resource). Figure 6.13 shows the cluster which does not limit memory assigned to given Pods, which leads to starvation of other brokers.

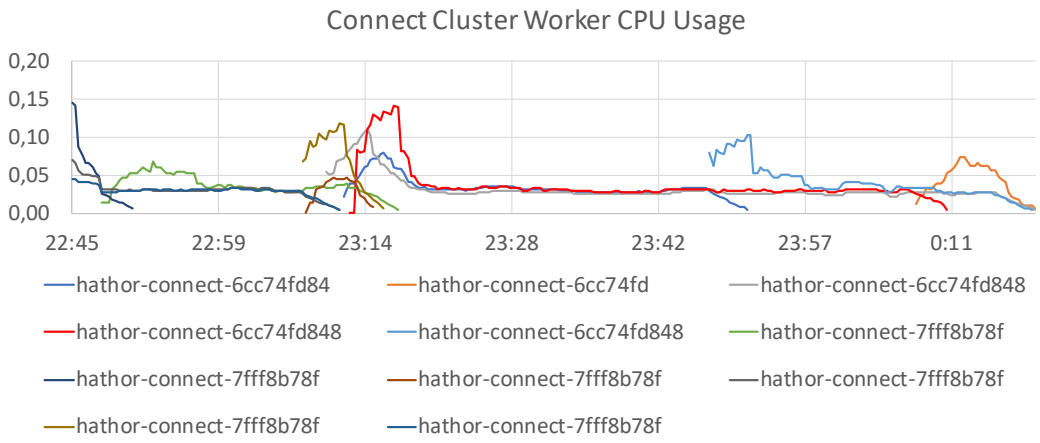


Figure 6.12: Workers' recovery and load balance

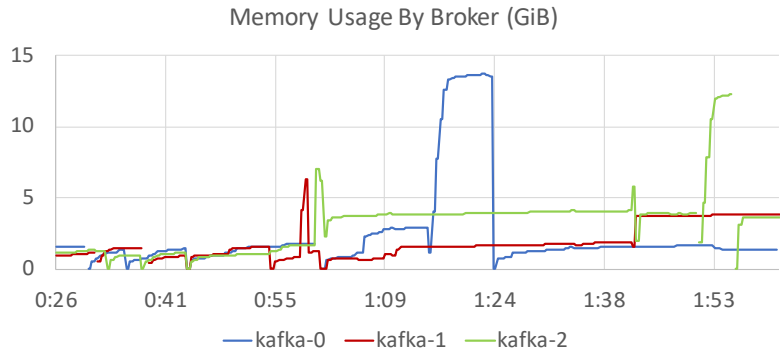


Figure 6.13: Memory wasting

After applying chaos to several systems (i.e., different configurations of Strimzi and one specific production environment) and seeing that experiments are passing almost all the time, we could conclude that Strimzi does provide desired traits regarding resilience.

6.2.4 Experiments

The end of this Chapter is dedicated to configuring, managing, and otherwise manipulating custom Kafka clusters into specific possible scenarios.

Thrughput

Strimzi is capable of recovering the cluster into a healthy state. However, frequent overlapping of chaos (e.g., deletion of resources needed for correct recreation of brokers) significantly increases the time for which the cluster remains in a weakened state, e.g., not allowing data consumption or production. The general solution for the increase of throughput lies in full usage of existing brokers, i.e., higher replication factor with reasonable settings regarding the acknowledgement of Producer about successful write of a message. Figure 6.14 depicts the root of decreased rate in production and later consumption of messages. Once the

partition leader dies, production is blocked by the Producer retrying to communicate with the fallen instance.

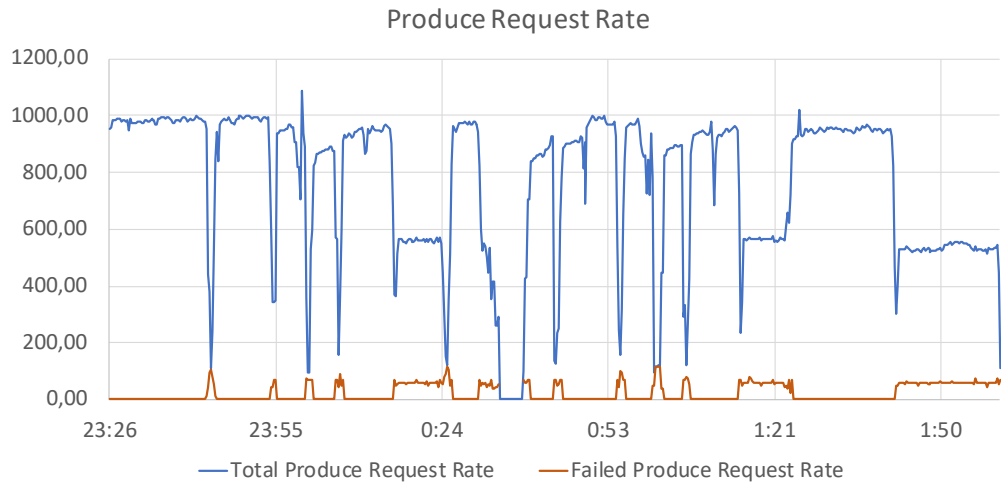


Figure 6.14: Increased rate of failed requests after deletion of some of brokers.

With proper configuration (i.e., high replication factor, small requirement regarding synchronized replicas), we can obtain relatively smooth throughput even during almost constant chaos, as is also visible in Figure 6.15. Firstly, production continues even with a higher number of broker downs. The spikes in traffic are caused by the asynchronous production of messages, with each one being resent numerous times in case of failure. As a result, once the partition leader against works, all messages are delivered at once.

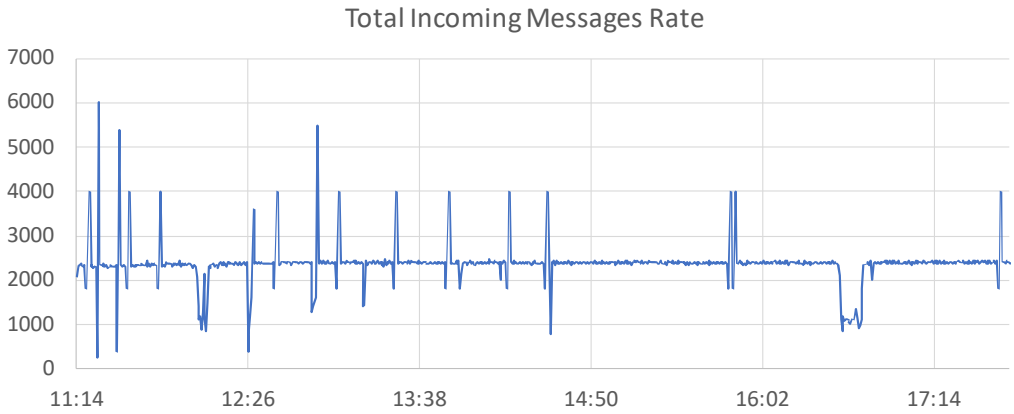


Figure 6.15: Increase of resielence by change of default configuration

Security

Most of the scenarios described in this Chapter did not cover the deletion of Security resources. This is mainly because these resources play an essential role in access, and if destroyed, repair of the cluster would afterwards require human intervention. An example of this is the deletion of signing certificates for clients, which would cause all newly created

pods to fail. The reason for this is that Kafka tries to find an old certificate, which is simply missing. This kind of chaos is even more fatal as it may also cause problems with the exporting of metrics, and before doing any hard restarts, Figure 6.16 depicts all that can be obtained once such a problem occurs. The problem only propagates itself a few minutes later when brokers die and needs to be restarted. Monitoring of cluster fails in precisely the same manner afterwards.

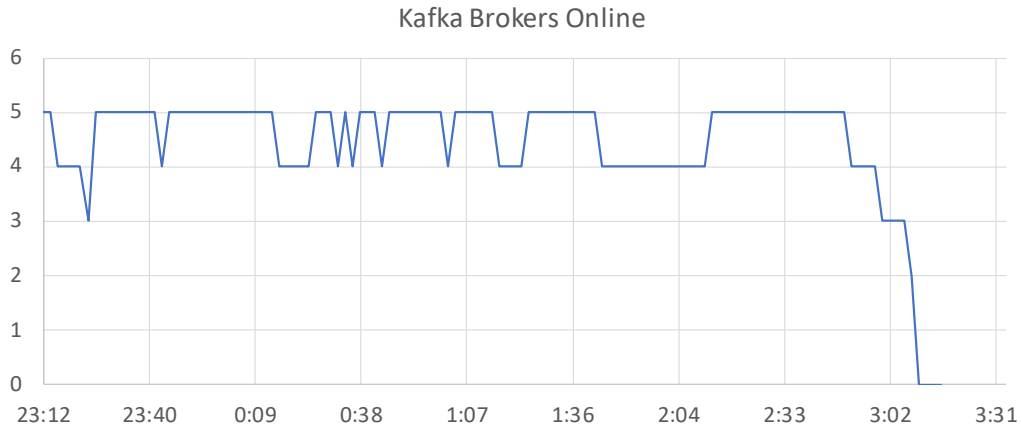


Figure 6.16: Certificate failure.

Others

Not all experiments (i.e., types of templates provided by Litmus) were applied in implementation. There are several reasons for that. Some of these templates are much more focused on Kubertnes itself than our system or at least some components they interact with. These templates are deletion of kubelet or container runtime. Other templates are impossible with the usage of crio runtime or on Openshift cluster. A Member of this category of experiments is also Node Restart. Restart of a node is an event that can happen in a cluster even on a daily base. Although we cannot use a Litmus template to inject this kind of chaos, we can use Kubernetes API to instruct it.

Similarly, as in the previous experiment, this chaos may result in actual deletion of monitoring and even other chaos injections. This is done if the Pods responsible for the given task are located in the restarted node. However, the effect this chaos has on the system itself is not very significant. One or at most two replicas of brokers may get deleted, but we already tested the system against even more severe chaos. The assurance that node failure will not cause that big harm in regard to the number of deleted brokers is obtained by using the Kubernetes feature called Node Affinity. This setting instructs Kubernetes to disperse replicas controlled by given Controllers amongst all suitable nodes, therefore making the application resilient.

Chapter 7

Future work and ideas

Templates (provided by Litmus) available on ChaosHub are expected to work correctly and be flexible enough to cover most of the desired use cases. Two originally proposed templates (i.e., Resource Deletion and Kafka Rolling Update) will be merged to the Litmus Go repository[1] and will become publicly available with the next release. This publication of templates will require additional work in the future, as underlying components will evolve, e.g., Kafka and Strimzi. However, together with the rest of implemented templates and those provided by Litmus, these templates are already applied in the production environment as part of ExcelentProject.

7.1 Advanced templates

Strimzi also has several additional components that play a part in specific scenarios in the cluster. This thesis does not cover them as they are even more advanced components used sporadically or under specific conditions. Implementation of this scale would require additional consideration and would either need different implementation for different underlying Orchestration platforms, a particular configuration of system under test, additional components, or cast special Kubernetes events in whole the cluster. Examples of these components are Drain Cleaner, Cruise Control, and MirrorMaker. More about these components can be found in [8].

The most straightforward is MirrorMaker, which transfers data from one Kafka cluster to another. However, it relies on processes (Pods) that do the actual work. The provided template would then inject chaos into the data transfer process.

Kafka Cruise Control keeps new brokers loaded equally compared to existing ones, so this kind of template requires the creation of load, increased Kafka brokers replica count, and actual chaos inside brokers.

Nevertheless, the most advanced template is the usage of Drain Cleaner, as this component takes over the management of the update of the whole Kubernetes cluster. A template of this scale is still possible but requires significantly more consideration and knowledge about advanced Kubernetes configuration.

[1] Litmus Go Github – https://github.com/litmuschaos/litmus-go

7.2 Decoupled probes

This extension refers to the Litmus component described in Section 3.2.3. The whole idea is built around separating the specification of probes from the specification of chaos. Although this is currently not possible, changes in Litmus implementation to accomplish this would not be invasive. With growing adoption and an increasing number of experiments, Litmus has to develop a way to keep existing templates reusable for much more generic purposes. For example, Litmus already provides several templates for Pod deletion. The things which they primarily differ in are checks and additional Kubernetes Jobs they create. If we find a way to make these checks separately, these templates could be highly simplified, and what is more important, there would not be a need to create a new template each time we want to create very specific liveness checks.

Litmus provides a way to check for additional conditions during the chaos with the help of formerly mentioned probes. This could be very easily used to provide options to create custom liveness streams. So theoretically, we could implement all liveness checks outside of the actual template and instead only reference (inside ChaosEngine custom resource) the kind of liveness stream we wish in the chaos.

Currently, the problem is that these probes are a part of the ChaosEngine specification and offer minimal options to configure own images (and consequently jobs), which could be spawned from them. Consider templates designed in Section 4.3.2. The liveness stream provided here must be implemented inside the template and configured inside the given ChaosEngine resource. Chaos Job later spawns Pods with Producer and Consumer images. Afterwards, chaos Job collects these Pods' results and finishes the chaos experiment. This is visible in figure 7.1. Here, we can see that the user specifies the configuration of probes and ChaosEngine separately. The probe contains all configurations related to the liveness stream, and ChaosEngine contains significantly fewer configurations, focused solely on the purpose of chaos itself.

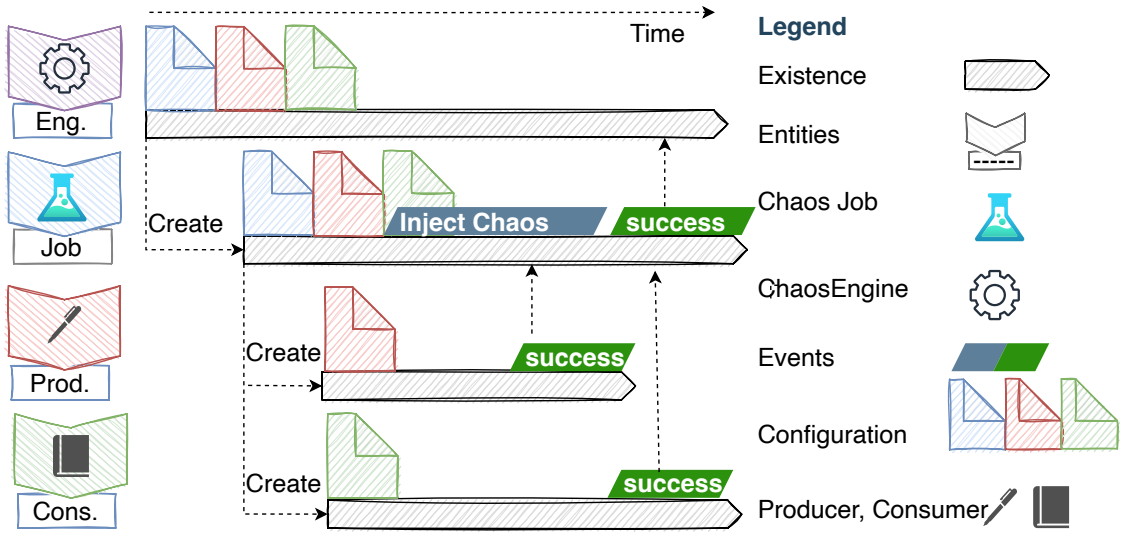


Figure 7.1: Chaos workflow in case of liveness applied.

In case of the addition of support for Probes to run *configurable* images on their own, we could easily accomplish workflow from Figure 7.2.

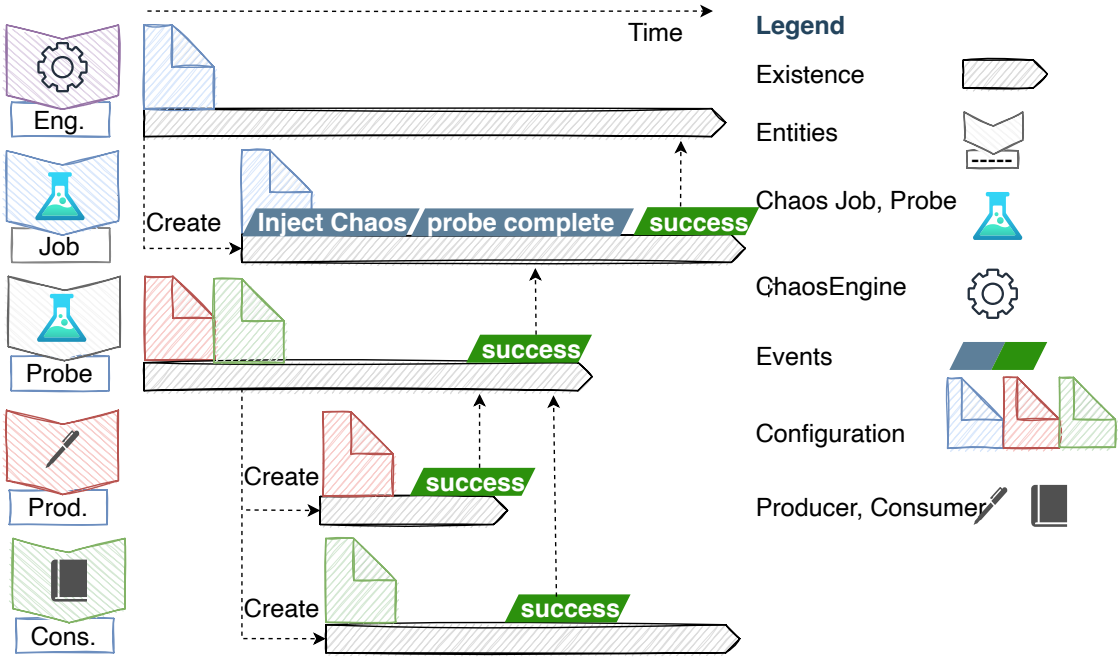


Figure 7.2: Chaos workflow in case of separated responsibilities.

The configuration of these probes can be solved by allowing each probe to specify its environment variables. There is no need to create separated custom resources (e.g. ChaosProbe), but it would be a perfect addition, and several templates could be significantly simplified, and some even removed.

Chapter 8

Conclusion

This thesis examined the problem of Chaos Engineering on project Strimzi. The whole work started with identifying key features and weaknesses the project has: dependency on Zookeeper, need to withstand Rolling Updates, and creation of numerous additional resources. Then, the central part focused on the proposal and implementation of the suitable way to cover all needs of the Strimzi project concerning Chaos Engineering, particularly: injecting chaos into generic Strimzi clusters and also into a production environment. The last part was mainly about finding and configuring a reasonable way to monitor all parameters which either influence or are influenced by the chaos itself, which also helped with the correct evaluation of the overall findings and results, which are also described in Section 6.2.3.

Because chaos engineering is not yet very standardized, and all frameworks and tools this thesis works with (i.e., Litmus, Strimzi, etc.) are changing, so did parts of the design and implementation of this thesis. For instance, the addition of two more templates needed to be implemented due to the new needs of Kafka, which were not yet reflected in the Litmus ecosystem. Another example of this is the actual decision to split the implementation into two different parts, focusing on entirely different systems. Consequently, we gain confidence in Strimzi running smoothly with all expected resistance while deployed in different configurations and environments. There are three main parts of the implementation. The first part of implementation are four templates; these will become available on ChaosHub; it will be possible to apply them in other projects that use Strimzi. The second part is the implementation of a generic test suite for new possible Strimzi configurations with all necessary utilities for building, deploying and obtaining all necessary chaos components for the addition of future templates and chaos experiments. The final part of implementation is the creation, configuration, setup, and monitoring of chaos experiments in the Production environment.

Despite this thesis's rather extensive scope, injection of chaos did not disrupt the system (deployed Strimzi) into extent in which significant changes would be needed. On the other hand, we got confidence in the project's ability to withstand turbulent conditions. Nevertheless, the overall work can be considered a beginning of chaos engineering on Strimzi on an even larger scale, as there are still many use cases and components that we did not yet cover and play a significant role in the system's behaviour. Additionally, this future implementation can be done in much more manageable way, with new way of implementing probes. Details about these changes are already described in Chapter 7.

Bibliography

[1] AUTHORS, C. P. *Chaos Principles* [online]. 2019 [cit. 2021-11-25]. Available at: https://principlesofchaos.org/.

[2] AUTHORS, R. H. *What is a Kubernetes operator?* [online]. Red Hat, Inc, may 2020 [cit. 2021-10-14]. Available at: https://www.redhat.com/en/topics/containers/what-is-a-kubernetes-operator.

[3] AUTHORS, T. litmus. *Using Litmus* [online]. 2021 [cit. 2021-11-18]. Available at: https://v1-docs.litmuschaos.io/docs/getstarted/.

[4] AUTHORS, T. C. *How to use Kafka Connect - Getting Started* [online]. 2021 [cit. 2021-10-18]. Available at: https://docs.confluent.io/home/connect/userguide.html.

[5] AUTHORS, T. C. *Kafka Connect Concepts* [online]. 2021 [cit. 2021-10-18]. Available at: https://docs.confluent.io/platform/current/connect/concepts.html#connect-tasks.

[6] AUTHORS, T. K. *Apache Kafka documentation* [online]. 2021 [cit. 2021-08-14]. Available at: https://kafka.apache.org/documentation/.

[7] AUTHORS, T. K. *Kubernetes* [online]. 2021 [cit. 2021-08-10]. Available at: https://kubernetes.io/.

[8] AUTHORS, T. S. *Using Strimzi* [online]. 2021 [cit. 2021-11-18]. Available at: https://strimzi.io/docs/operators/latest/overview.

[9] BRAZIL, B. *Prometheus: Up Running: Infrastructure and Application Performance Monitoring*. 1st ed. O'Reilly Media, Inc., 2018. ISBN 1492034142.

[10] BREBNER, P. *Apache Kafka Connect Architecture Overview* [online]. 2018 [cit. 2021-10-10]. Available at: https://www.instaclustr.com/apache-kafka-connect-architecture-overview/.

[11] BRENDAN BURNS, K. H. *Kubernetes: Up and Running: Dive into the Future of Infrastructure*. 2nd ed. O'Reilly Media, 2019. ISBN 978-1492046530.

[12] CASEY ROSENTHAL, N. J. *Chaos Engineering: System Resiliency in Practice*. 1st ed. O'Reilly Media, 2020. ISBN 1492043869.

[13] COLMAN, G. *The New Kubernetes Native* [online]. 2020 [cit. 2021-11-25]. Available at: https://graemecolman.medium.com/the-new-kubernetes-native-d19dd4ae75a0.

[14] Developers, S. *Strimzi Apache Kafka Operator joins the CNCF* [online]. 2019 [cit. 2021-11-26]. Available at: https://strimzi.io/blog/2019/09/06/cncf/.

[15] Dobies, J. *Kubernetes Operators*. 1st ed. O'Reilly Media, 2020. ISBN 9781492048053.

[16] Domnu, R. *Chaos Engineering With LitmusChaos* [online]. October 2021 [cit. 2021-11-28]. Available at: https://cloud.redhat.com/blog/chaos-engineering-with-litmuschaos.

[17] Gwen Shapira, R. S. and Petty, K. *Kafka: The Definitive Guide, Real-Time Data and Stream Processing at Scale*. 2nd ed. O'Reilly Media, 2017. ISBN 978-1-492-04301-0.

[18] kafka.apache.org. *Streams* [online]. 2017 [cit. 2021-10-10]. Available at: https://kafka.apache.org/0102/documentation/streams/.

[19] Koutanov, E. *Effective Kafka: A Hands-On Guide to Building Robust and Scalable Event-Driven Applications with Code Examples in Java*. 1st ed. Independently published, 2020. ISBN 9798628558515.

[20] Luksa, M. *Kubernetes in Action*. 1st ed. Manning, 2018. ISBN 978-1617293726.

[21] Manna, N. *A Beginner's Practical Guide to Containerisation and Chaos Engineering with LitmusChaos 2.0* [online]. June 2021. Available at: https://dev.to/neelanjan00/part-2-a-beginner-s-practical-guide-to-containerisation-and-chaos-engineering-with-litmuschaos-2-0-253i.

[22] Manna, N. *Chaos Engineering Made Simple* [online]. 2021 [cit. 2022-02-20]. Available at: https://thenewstack.io/chaos-engineering-made-simple/.

[23] Miles, R. *Learning Chaos Engineering: Discovering and Overcoming System Weaknesses Through Experimentation*. 1st ed. O'Reilly Media, 2019. ISBN 1492051004.

[24] Mitch, S. *Mastering Kafka Streams and ksqlDB Building real-time data systems*. 1st ed. O'Reilly Media, Inc., 2021. ISBN 1492062499.

[25] Mukkara, U. *Cloud native chaos engineering – Enhancing Kubernetes application resiliency* [online]. November 2019 [cit. 2021-11-28]. Available at: https://www.cncf.io/blog/2019/11/06/cloud-native-chaos-engineering-enhancing-kubernetes-application-resiliency/.

[26] Mukkara, U. *Chaos Engineering With LitmusChaos* [online]. August 2020 [cit. 2021-11-28]. Available at: https://www.cncf.io/blog/2020/08/28/introduction-to-litmuschaos/.

[27] Mwila, L. *Kubernetes Controllers | Complete Guide for 2021* [online]. ContainIq, october 2021 [cit. 2021-10-16]. Available at: https://www.containiq.com/post/kubernetes-controllers.

[28] Neha Narkhede, T. P. *Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale*. 1st ed. O'Reilly Media, 2017. ISBN 979-8703756065.

[29] Pawlikowski, M. *Chaos Engineering: Site reliability through controlled disruption.* 1st ed. Manning, 2021. ISBN 1617297755.

[30] Poulton, N. *The Kubernetes Book.* 1st ed. Independently published, 2021. ISBN 979-8703756065.

[31] Rodal, R. *5 Best Chaos Engineering Tools* [online]. October 2021 [cit. 2021-11-25]. Available at: https://harness.io/blog/chaos-engineering-tools/.

[32] Sayfan, C. *Mastering Kubernetes.* 3rd ed. Packt Publishing Ltd., 2020. ISBN 978-1-83921-125-6.

[33] Stopford, B. *Designing Event-Driven Systems.* 1st ed. O'Reilly Media, Inc., 2018. ISBN 9781491990650.

[34] Turnbull, J. *The Docker Book: Containerization is the new virtualization.* 1st ed. James Turnbull;, 2014.

[35] Velichko, I. *Exploring Kubernetes Operator Pattern* [online]. Ivan Velichko, january 2021 [cit. 2021-10-10]. Available at: https://iximiuz.com/en/posts/kubernetes-operator-pattern/.

[36] Viktor Farcic, D. P. *The DevOps Toolkit: Kubernetes Chaos Engineering.* 1st ed. Independently published, 2020. ISBN 979-8634359939.

[37] www.oracle.com. *What is OLTP* [online]. 2020 [cit. 2021-10-10]. Available at: https://www.oracle.com/database/what-is-oltp/.

Appendix A

CD content

- 01-templates – extension of Litmus (i.e., templates)
- 02-test – implementation of test suit
- 03-applied-experiments – all manifest that apply chaos.
- README – more detailed map of CD, and all necessary information to start
- extra – examples, and obtained logs
- install – manifest which are needed for installation of components
- monitoring – manifest for correct monitoring setup
- navody – extra information for applying chaos in different environments
- install-guide – step by step lead to installation and application of all the chaos

In any case, please start with README file.

Appendix B

Role based access control

This chapter briefly describes how authorization and authentication works within the Kubernetes clusters. This play crucial role in this thesis as deploying any Operator (Section 2.1.6) and in case of Litmus, also proper working of experiments, requires correct configuration of these resources. Chapter B.1 describes basics of these resources (i.e., Role, ServiceAccount, RoleBinding), and the following one (B.2) its place within working with Litmus and Strimzi

B.1 Resources

Communication with the Kubernetes API server allows us to manipulate all resources within the cluster. Authentication and authorization of applications running within-cluster (i.e., within pods) is accomplished by use of *serviceAccounts*. By providing secrets associated with given service account, application can easily authenticate itself once it needs to manipulate some data from API server (Figure B.1). Every pod is assigned exactly one service account. The assigned service account represents the identity of the application running inside the pod. Assignment of service account takes place in the time of pod's creation and cannot be modified afterwards.

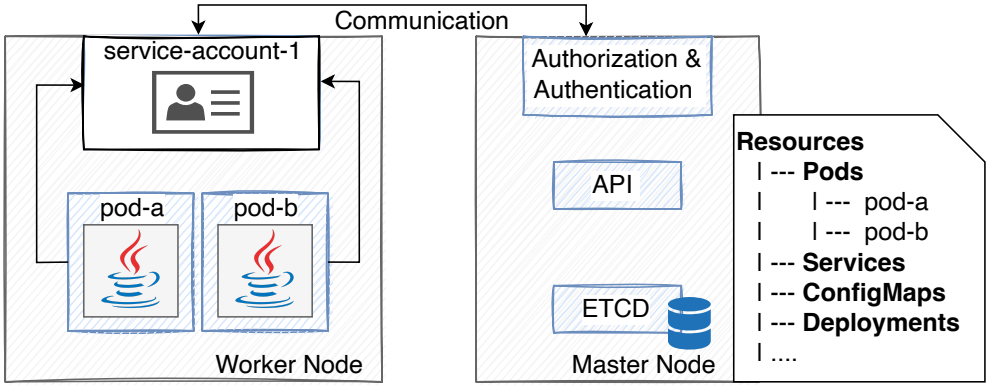


Figure B.1: Authentication within the *Kubernetes* cluster.

Kubernetes' approach to security can be than visualized and think of as model depicted in Figure B.2, where application require to execute operation on given object.

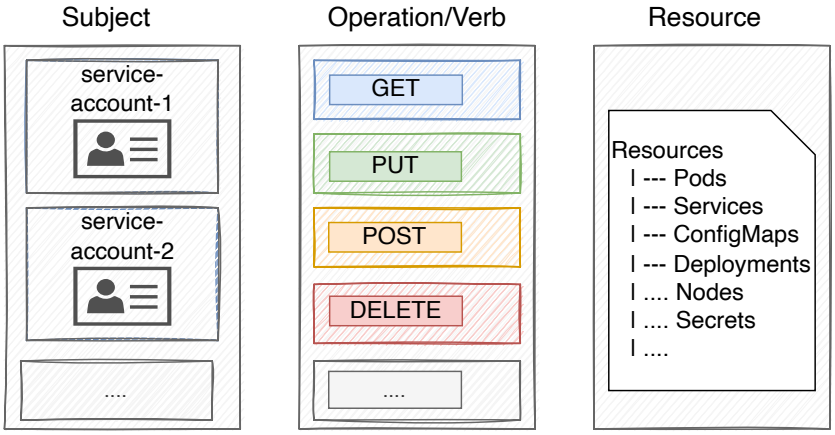


Figure B.2: Authorization and Authentication model that represents subjects, verbs and resources within Kubernetes cluster.

The subject is entity (mostly application identified by its service account), operations are methods we know from REST[1] API model, and finally resource is everything that is the content of ETCD storage, and accessible by Kubernetes API server. We can apply this model with help of two additional Kubernetes objects[2], i.e., *Role* and *RoleBinding*. The former object binds operations to specific resources, yet it does not contain any information about subject (i.e., who wants to perform operation) as is clarified in Figure B.3 and also in Listening B.1.

```
1
2 kind: Role
3 metadata:
4   name: example—role
5   label: example
6 rules:
7 — apiGroups: [""]
8   resources: ["pods","services"]
9   verbs: ["create","delete"]
```

Listing B.1: Manifest of Role object from the Figure B.3.

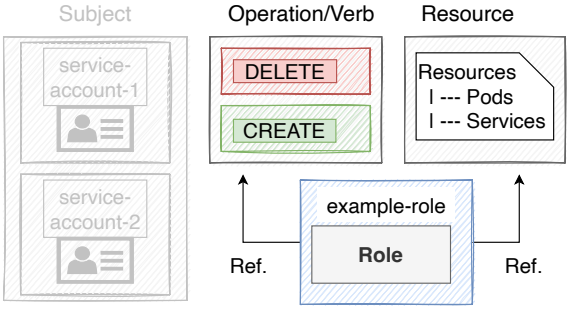


Figure B.3: Role that allows operation delete and create upon services and pods.

RoleBinding simply binds role to Subjects that may embody it (also visible in Figure B.4) and Listening B.2. When some object later wants to execute operation, API server firstly find all roles that are associated with given service account, and allow execution in case there is at least one role that permits it.

[1]REST API – more at https://www.techtarget.com/searchapparchitecture/definition/RESTful-API

[2]There are also ClusterRoleBinding and ClusterRole which have the same responsibilities as RoleBinding and Role yet they are not bound to namespace.

```yaml

kind: RoleBinding
metadata:
  name: example-rb
roleRef:
  apiGroup: ""
  kind: Role
  name: example-role
subjects:
- kind: ServiceAccount
  name: service-account-1
- kind: ServiceAccount
  name: service-account-2
```

Listing B.2: RoleBinding bject's manifest from the Figure B.4.

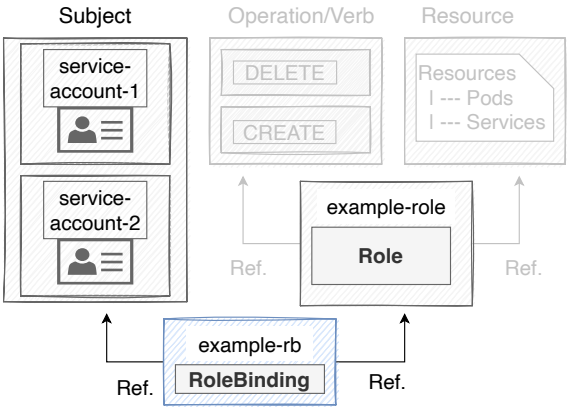


Figure B.4: RoleBinding that binds two service accounts to Role defined in Listening B.1

B.2 Operators

When we deploy operator in Kubernetes cluster, part of its work is to monitor state of cluster, as is in detail described in Section 2.1.6. From authorization and authentication point of view, we need to provide adequate resources to allow operator to watch over and manage its resources. For this reasons operators are usually shipped with predefined Roles and RoleBindings (or their Cluster-wide counterpart) for specific resources. Strimzi operators has to be able to watch over KafkaTopics Kafkas etc. but also being able to create all necessary kubernetes native resources (e.g, services, secrets).

More complicated scenario is in case of Litmus. One of the few disadvantages of Litmus is its need for a lot of previously mentioned resources. Because each experiment that we will run on our cluster may interfere with different resources, we have to provide pods responsible for that chaos experiment with different minimal permissions[3].

[3]Running these experiments with admin privileges would be unacceptable.

Appendix C

Kafka Configuration

This chapter starts with explanation of configuration problem as such in Kafka (Section C.1) and continues with individual configurations, what is content of Section C.2.

C.1 Problem of configuration

One of the most notable traits of Kafka is its *configurability*. It is mostly about which properties we need for a specific business model. Depending on our settings, we will end up with compromises between *availability and safety, latency and traffic, as well as space and durability*. Everything mentioned till this line is configurable (from how many partitions should topic have to allowance of unclean election[1]). The size of *batches of messages* before sending them to Kafka, how many times the producer should retry to send data in case of failure, and what politic should be applied when waiting for acknowledging that data were successfully stored for. All these configurations represent a *trade-off between latency on the producer side and traffic on the server side*.

We speak about one order *higher magnitude of problems with configuration* once we start to configure Connects or Mirror Makers. Truth be told, the configurability of Kafka is a double-edged weapon. On one side, it allows Kafka to match almost every need in its domain. On the other one, setting up, manipulating monitoring, and validating choices we made during configuration may be far too unnecessarily complicated[2]. New solutions that build on top of the Kafka, such as Confluent[3] or Strimzi (described in chapter 2.3) are trying to minimize the amount of necessary configuration.

We may pass our configuration to Kafka, and especially if it is managed by Strimzi, as part of specification of several entities:

- **Broker** – Beside all other configurations, it contains information about storage, its id, port it is listening on and lot of default properties for topics and clients.
- **Topic** – When we create Kafka topic we are mostly interested in its count of partitions, replicas.
- **Strimzi** – Strimzi firstly automatize most of configuration present in brokers, but also add some of its own, e.g., separate configuration for internal topics.

[1] In this case we allow partition which is not fully in sync with fallen partition leader
[2] There are more than sixty properties to customize just as part of producer's configuration.
[3] https://www.confluent.io/

- **Clients** – This includes Producer, Consumer, and indirectly also Kafka Connect cluster. We are interested in specification of connection, how to proceed in case of failure, and when to consider message written.

C.2 Entities and properties

Each of Kafka's component (e.g., topic, consumer, broker) has its own set of attributes, which help with their configuration to setup desired behavior. Configuration in split into following tables with regard to what we want to do with Kafka, i.e., creation Producer, Consumer, or Topic.

There are two types of topic, i.e., internal or created by user. The most important of internal topics is *__consumer_offsets* from Section 2.2.2. Most important configuration regarding creation of topic is visible in Table C.1.

Table C.1: Selected Topic creation related configuration

Configuration name	Set by Entity	Description	Works with	Default value
auto.create.topics.enable	Strimzi	We can disable clients from creating topics automatically. A client would otherwise create topic in case it was not present at that time		
replication.factor	Strimzi, Topic	Specifies how many replicas we want either for the given topic or for all of them		
offsets.topic.replication.factor	Strimzi	Replication factor for the given internal topic		
num.partitions	Strizi,Topic	Number of partitions (it is usually a mandatory parameter when creating a topic)		
min.insync.replicas	Strimzi, Topic	Specifies how many replicas (including partition leader) must write a message to consider it written	replication.factor, acks (producer) commit (consumer)	

Since its creation, Kafka has switched its default configuration in favour of safety of produced messages. Following Table (C.2) depicts small part of configuration responsible for smooth production of data. We will not cover all of configuration that play some role as it would still leave at least twenty fields, which would be counterproductive.

We omit heartbeats, belonging to group and other things that are already described in Section 2.2.2. Instead we will only cover configuration that will interfere with our experiments.

Table C.2: Selected Producer creation related configuration

Configuration name	Description	Works with	Default value
bootstrap.servers	List of comma-separated pairs (address:port), which are addresses of brokers to be used for obtaining metadata for communication.		
retries	Specifies how many times should the producer retry to send a message in case of a potentially transient error	delivery.timeout.ms	2^{31} - 1
delivery.timeout.ms	We can simplify it as the maximal time to wait for success or fail regarding creation of the message	unmentioned	0
request.timeout.ms	The maximum amount of time the client will wait for the response of a request	delivery.timeout.ms	30 000
Acks	How many replicas (including partition leader) must write a message to consider it written. If set to value *all,* only the number of replicas specified by property min.insync.replicas are required to register it, to consider it written.	min.insync.replicas (Topic)	all

Table C.3: Selected Consumer creation related configuration

Configuration name	Description	Works with	Default value
bootstrap.servers	List of comma-separated pairs (address:port), which are addresses of brokers to be used for obtaining metadata		
consumer.timeout.ms	This option is deprecated from Kafka 1.0 onwards.		
retry.backoff.ms	The amount of time to wait before attempting to retry a failed request		100
reconnect.backoff.ms	The amount of time to wait before attempting to reconnect using a given host	bootstrap.servers	1000
enable.auto.commit	The consumer is also a producer (in the case of the offset topic). This option allows the library to take care of commits. The consumer cannot commit position if all acknowledgements are required and not enough replicas are synchronized.	min.insync.replicas	true
auto.commit.interval.ms	Specifies How often does the consumer commit his offset regarding consumption.	enable.auto.commit	5000

Appendix D

Kafka Streams

Is the most used extension of Kafka. Its purpose is to work with endless data (i.e., data stream) and apply operations on it. The need for it emerged by natural demands of market. IoT[1], sensors, financial systems, medical records, and other sources provide a large amount of data. Use cases that are built upon these data require us to *process, enrich, transform and react to* them as soon as possible. There were two main approaches to data processing:

- **Request/Response processing** – In the database world, this approach is known as OLTP[2] when the client requests some data, wait for them to process them, and repeat to process with the next data.
- **Batch processing** – The processing system wakes up every *scheduled time* and processed all pilled up data [28]. Not suitable for processing data in real-time, as we would have to wait often and run a job or query at some interval of our choosing [24].

Kafka Streams implements *Stream processing*. Stream is an abstraction for unbounded data set (i.e., as time continues, new data arrive, either as they are created or loaded). The processing itself goes in an endless continuous manner by leveraging a programming paradigm called *Data Flow Programming*[3]. This means that programming is represented as a series of inputs, outputs, and processing blocks that form an acyclic graph. Kafka Streams itself is nothing more than a simple and lightweight client library that provides few primitives for implementation of these graphs [18]:

- **Source processor** – Internally is implemented using consumer libraries which read data from Kafka topic. It can store more records in the buffer, visible in figure D.1.
- **Stream processor:** Performs transformations, e.g., filter, map. In addition, it supports local states, enabling even advanced operations such as joins and aggregations.
- **Sink processor:** Producer part of streams. Store data to destination Kafka topic. Due to working with data from topics, Kafka Steams depends only on a cluster from which it can read and eventually write.

Some of the essential benefits that Kafka architecture brings are *scalability, reliability, and speed*. The atomic unit of scaling is the partition. Once we have a topic with multiple partitions, we can spawn a new consumer application that will take care of its part of the

[1] IOT – Internet of Things

[2] OLTP stands for online transaction processing, a type of data processing which consists of executing several transactions occurring concurrently [37].

[3] Data Flow programming – https://devopedia.org/dataflow-programming

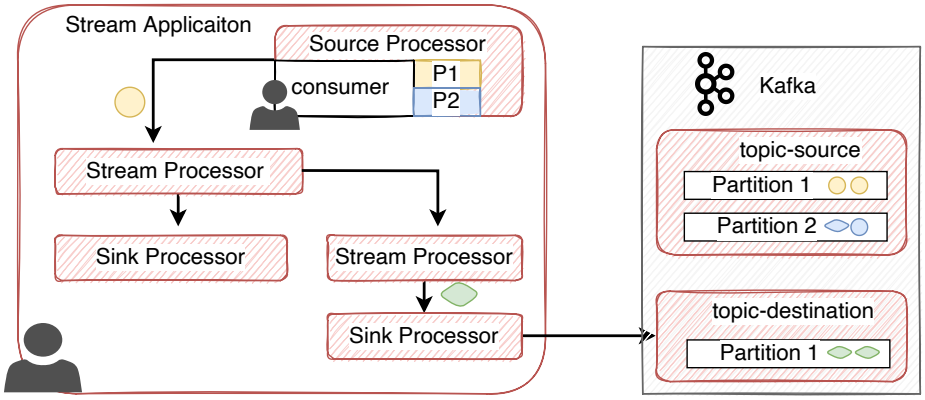


Figure D.1: Example of data flow within Kafka Streams application

partitions. *Reliability* is obtained due to the usage of Kafka as both source and destination. *High throughput* due to Kafka's way of storing and splitting data, and finally, *simplicity* that allows developers to focus more on business requirements.

D.1 Other extensions of Kafka

There are some other extensions over base Kafka, i.e., Quotas, Mirror Makers). Mirror makers are Apache Kafka's built-in cross-cluster replicators. Today there are two versions of Mirror Maker. The first version internally uses producers and consumers, but due to many flaws (mainly with configuration), MirrorMaker 2 was created, which was built on top of Kafka Connect. Because of this internal usage of Kafka Connect, there is no need to describe its component as much as in the case of Kafka Streams or Kafka Connect. Mirror makers are just another example of fast evolution in Kafka. As a whole ecosystem grows and responds according to the new needs of businesses, some parts are slowly replaced due to their many flaws. Yet, we can still see them, just like in the case of MirrorMaker 1[4] and in no time, Zookeeper as well.

[4]MirrorMaker 1 is deprecated at the time of writing this thesis