Encoding human-like operational knowledge using declarative Kubernetes operator patterns

Lucas Käldström

School of Electrical Engineering

Bachelor's thesis
Espoo 10.12.2021

Supervisor

Assistant Professor Martin Andraud

Advisors

Prof. Mario Di Francesco

Dr. Stefan Schimanski

Aalto University
School of Electrical
Engineering

Copyright © 2022 Lucas Källström.
This work by Lucas Källström is licensed under the CC-BY-SA 4.0 license, unless otherwise stated.

Aalto University
School of Electrical Engineering

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the bachelor's thesis

Author Lucas Käldström

Title Encoding human-like operational knowledge using declarative Kubernetes operator patterns

Degree programme Aalto Bachelor's Programme in Science and Technology

Major Digital Systems and Design **Code of major** ELEC3056

Teacher in charge Assistant Professor Martin Andraud

Advisors Prof. Mario Di Francesco, Dr. Stefan Schimanski

Date 10.12.2021 **Number of pages** 52 **Language** English

Abstract

In recent years, the *cloud native* paradigm has emerged as an effective way of managing server infrastructure, with over 100,000 open source contributors and users such as Apple, CERN, Deutsche Telekom, JD.COM, Nokia and Spotify. Cloud native is a mindset; a set of patterns and practices that provide for efficient, scalable, and resilient server infrastructure control.

Although cloud native has a definition and has been implemented widely, accessible context on why cloud native converged to its present state is scarce. Consequently, if the reasons for a pattern is not known, the pattern might not be implemented correctly, or, worse, at all. The aim of this thesis is to gather this context and showcase the benefits of cloud native implementation in *Kubernetes operators*. In other words, the goal is to identify methodologies for being able to scale systems, even beyond human cognitive limits, while still keeping the system in a reliable state.

Through review of articles relating to server infrastructure control and open source cloud native resources, this thesis finds that some control problems are noticed mostly at large scale, which can lead small-scale users into a false sense of control. The problems include the impact of inevitable entropy increase, randomness, failures and limited information transfer rate. Through codifying human-like operational knowledge, Kubernetes operators can effectively combat these problems and allow for greater resource, knowledge, and monitoring scalability.

Kubernetes operators form a *novel programming model*, allowing a shift from humans managing servers to servers managing servers, analogously to the Industrial Revolution. This empowers users to once codify the control mechanism, deploy it using a declarative abstraction layer, then finally, benefit from both system scalability and reliability.

Keywords cloud native, Kubernetes, servers, cloud services, control engineering, entropy, automation, declaration of intent, interfaces (computer programmes), scalability

Aalto University
School of Electrical Engineering

Aalto-universitetet, PB 11000, 00076 AALTO
www.aalto.fi
Sammandrag av kandidatarbetet

Författare Lucas Käldström

Titel Automatisering av operationell serverhanteringkunskap via deklarativa Kubernetes-gränssnitt

Utbildningsprogram Aalto Bachelor's Programme in Science and Technology

Huvudämne Digital Systems and Design **Huvudämnets kod** ELEC3056

Ansvarslärare Assisterande Professor Martin Andraud

Handledare Prof. Mario Di Francesco, Dr. Stefan Schimanski

Datum 10.12.2021 **Sidantal** 52 **Språk** Engelska

Sammandrag

Under de senaste åren har "cloud native"-paradigmet (ung. "molnäkta") utvecklats till ett effektivt sätt att hantera serverinfrastruktur. Idag har mer än 100 000 personer bidragit till olika cloud native öppen källkods-projekt, varav ett är Kubernetes. Kubernetes används av exempelvis Apple, CERN, Deutsche Telekom, JD.COM, Nokia och Spotify. Cloud native-paradigmet är ett tankesätt med god praxis som ger möjlighet till effektiv, skalbar och motståndskraftig kontroll över serverinfrastrukturen.

Trots att cloud native har en definition och många implementationer, finns det inte mycket dokumenterad och lättillgänglig kontext varför cloud native designats såsom det gjorts. Om orsaken till att en rekommendation existerar är okänd, finns det risk för att rekommendationen inte blir implementerad rätt eller över huvud taget. Därför är målet med detta examensarbete att samla denna kontext samt göra den lättillgänglig och begriplig. Vidare är avsikten att visa på konkreta fördelar som uppnås i praktiken genom implementation av cloud native, exempelvis automatiserade "Kubernetes operators" (ung. "Kubernetes-operatörer"). Syftet är därmed att identifiera metoder för göra serversystem mer skalbara, till och med förbi människans kognitiva kapacitet, samtidigt som systemen hålls stabila.

Genom granskning av relevanta forskningsartiklar och öppen källkods-resurser, identifieras i detta examensarbete utmaningar med att kontrollera serversystem. Utmaningarna noteras oftast i stor skala, vilket leder till att småskaliga användare kan inledas i en falsk känsla av kontroll. Utmaningarna inkluderar effekterna av oundviklig spontan entropiökning, slumpmässighet, krascher och begränsad överföringshastighet av information. Genom att koda in operationell serverhanteringskunskap i "Kubernetes operators" kan man effektivt motverka ovannämnda utmaningar och åstadkomma högre skalbarhet av resurser, kunskap och övervakning.

"Kubernetes operators" exemplifierar denna nya typ av cloud native programmeringsmodell, vilket möjliggör en övergång från att människor hanterar servrar till att servrar hanterar servrar, i likhet med den industriella revolutionen. Detta ger användaren möjlighet att automatisera serverhanteringslogiken, sedan distribuera och konfigurera automationen via ett deklarativt och abstraherande gränssnitt, samt slutligen dra nytta av skalbarheten och driftssäkerheten.

Nyckelord cloud native, Kubernetes, servrar, molntjänster, reglerteknik, entropi, automation, viljeförklaring, gränssnitt, skalbarhet

Preface

I would like to thank my excellent advisors Mario Di Francesco and Stefan Schimanski for their advice, ideas, and feedback during the writing process. Your input has been very helpful.

I also would like to thank my partner for the support throughout the research process

Finally, I appreciate all input I have got from friends reviewing the thesis, namely: Verneri Hirvonen, Michael Gasch, Frank Sandqvist, Joe Beda and Johan Tordsson.

Otaniemi, 17.12.2021

Lucas J. O. Käldström

Contents

Abbreviations

API	Application Programming Interface
BCL	Borg Configuration Language
CLI	Command Line Interface
CNCF	Cloud Native Computing Foundation
CNI	Container Network Interface
CoC	Convention over Configuration
CPU	Central Processing Unit
CRI	Container Runtime Interface
CSI	Container Storage Interface
DSL	Domain-specific Language
GCL	General Configuration Language
GVK	GroupVersionKind
HCL	HashiCorp Configuration Language
HTTP	Hypertext Transfer Protocol
ID	Identifier
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
MTBF	Mean Time Before Failure
MTTR	Mean Time To Repair
POST	A HTTP verb (conventionally requesting creation of a resource)
PSU	Power Supply
REST	Representational State Transfer
SNR	Signal-to-noise Ratio
SQL	Structured Query Language
SRE	Site Reliability Engineer
TOML	Tom's Obvious Minimal Language
UI	User Interface
URL	Uniform Resource Locator
VM	Virtual Machine
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

app	Application
config	Configuration
diff	Difference
`spec`	User-provided, Desired State
`status`	Observed, Actual State

1 Introduction

In recent years, *cloud native* [1] has become a hot topic in the server infrastructure ecosystem. Cloud native is both a *mindset* and set of patterns and practices for managing server infrastructure. Since its introduction in 2015 [2], it has evolved into a community of more than 110,000 open source contributors [3]. Cloud native is defined[1] as follows [1]:

> Cloud native technologies empower organizations to build and run **scalable** applications in modern, **dynamic environments** such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and **declarative APIs** exemplify this approach.
>
> These techniques enable **loosely coupled** systems that are **resilient, manageable**, and observable. Combined with **robust automation**, they allow engineers to make high-impact changes frequently and **predictably with minimal toil**.

Kubernetes [4], the flagship cloud native project with almost 50,000 new commits in 2020 alone [5], is an open source production-grade container orchestrator. Kubernetes is designed by engineers with decades of experience building and operating server systems at massive scale, including, Borg [6] and Omega [7], the two orchestration systems deployed at Google LLC.

Kubernetes' declarative, extensible, scalable and resilient design has many benefits [8]. This has led to vast end user adoption, including, but not limited to: Apple [9], CERN [10], JD.COM [11], Spotify [12], and even Deutsche Telekom [13] and Nokia [14] as it enables 5G development [15], [16].

1.1 Problem Statement and Aims

This thesis aims to shed some light on **why** Kubernetes and cloud native were designed the way they are, rather than just *how*. What made the designers choose radically different design decisions compared to state of the art at the time? Furthermore, the aim is to compare some of its design decisions with similar server infrastructure platforms such as Docker Swarm [17] and HashiCorp Terraform [18].

Kubernetes is designed in a self-healing, robust and declarative way. In addition, it is built from the ground up to be extensible and transparent, allowing for and encouraging users to build their own self-healing, robust and declarative application-specific automation on top. Coined in 2017 [19], *Kubernetes operators* bring **a novel programming model**, following a large set of best practices, to the server infrastructure ecosystem.

This thesis aims to illustrate the (quite complex) concepts covered in a relatable and concrete way through analogies and examples. The goal is that readers get a new perspective on server management, and intuitively learn from those with decades of experience building and operating large-scale systems.

[1] Emphasis added to the quote; denotes aspects covered in this thesis

The research questions of this thesis are the following:

1. How can server-related operational intelligence and knowledge be codified such that humans can delegate management of server infrastructure to automated processes (instead of "manual" management) and thus scale server systems beyond human cognitive limits?

2. What underlying (physical, statistical and managerial) dynamics of server infrastructure become visible at large, but not necessarily small, scale and thus influence management software purposefully built for massive scale?

3. What are Kubernetes operators and their design principles, and how do they compare to other alternatives? For example, why are operators designed as declarative, level-triggered, periodic, pull-based, explicitely versioned, bidirectional, idempotent and optimistic reconciler loops, and not some other way?

1.2 Methods

The foundational theory covered in this thesis are the core design principles behind Kubernetes ([6]–[8], [20]), Kubernetes operators ([19], [21]–[23]), CFEngine ([24], [25]), other server management systems ([26]–[29]) and other control systems ([30], [31]).

Talks and slides from industry conferences, such as, KubeCon [32] and blog posts, books, or whitepapers from leaders within the CNCF community were considered to answer the research questions. Additionally, keyword searches are performed on relevant concept terms that need examination, like "declarative", "UNIX", "Raft", "Kubernetes Cluster API". The databases searched include the ACM Digital Library, arXiv, IEEE Xplore, and Google Scholar. Backward snowballing was also used for resources of particular importance.

As open source projects Kubernetes, Docker Swarm and HashiCorp Terraform are examined, their websites and documentation are thoroughly searched, and the open source code is reviewed if needed. This evaluation took place between September 2021 and December 2021.

1.3 Structure of the Thesis

This thesis is structured as follows. Chapter 2 explores the chaotic and random nature underlying server systems, and proposes a solution: declarative reconciler loops. Chapter 3 explores how to instruct (configure) these reconciler loops correctly and with minimal toil. Chapter 4 binds the theory introduced in Chapters 2 and 3 together into a concrete form that produces business value for its users. Chapter 5 discusses benefits and the shortcomings of the proposed solutions. Finally, Chapter 6 details improvement opportunities and concludes the thesis.

2 Bringing Order to Chaos

In physics, the second law of thermodynamics loosely states that the entropy of a system left to spontaneous evolution can not decrease over time. In fact, such a system strives for *thermodynamic equilibrium*, namely, the state with the highest entropy, i.e., the least ordered state. In other words, any ordered system becomes less ordered over time when left to spontaneous evolution.

As observed by Burgess [24], a similar pattern also applies to infrastructure systems. The entropy increase can be caused, for instance, by inconsistent ad hoc configuration changes, software upgrades, power outages, hardware/software failures, security vulnerabilities, or other changes of varying predictability. This claim is strengthened by more than a decade of experience at Google, crisply summarized as: "Failures are the norm in large scale systems" [6]. These changes or failures happen at random intervals and cause the system to become less ordered, therefore, more chaotic.

To solve this issue, Burgess argues that the changes going into a system should be split into a long-term trend part (a "slow" component) as well as a noisy and quickly changing part (a "fast" component), like ripples (fast) within waves (slow). The former, fast, component represents the unwanted fluctuations from the long-term trend. The latter, slow, corresponds to the policy intended to govern the evolution of the system over time. Without that, fluctuations could grow unboundedly, thereby putting the system at risk of reaching an unstable state. A corrective process countering the fluctuations needs to be applied at least as often as the fluctuations occur [24], as shown in Figure 1.

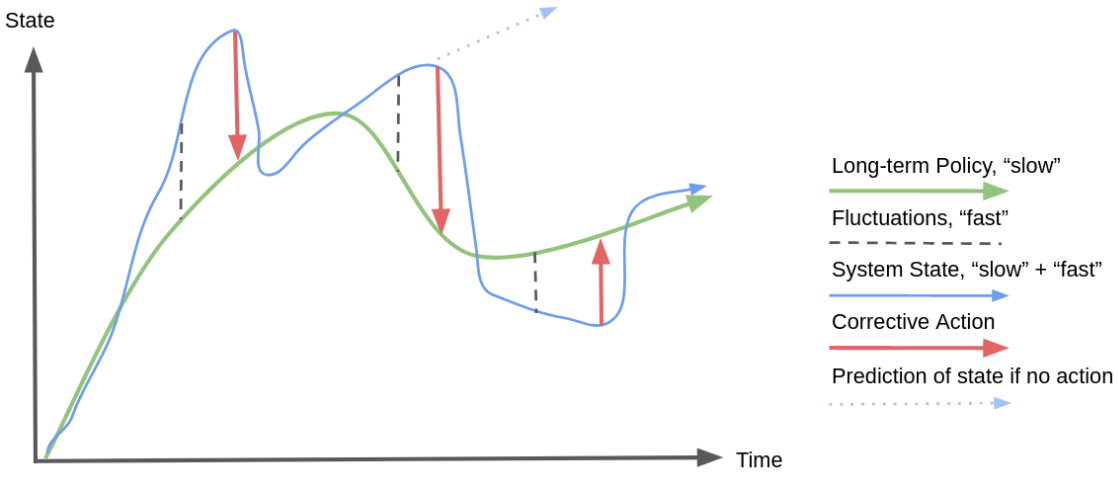

Figure 1: Sample illustration of system state evolution over time, highlighting the long-term policy, random fluctuations from such a policy, and corrective actions being applied to prevent indefinite growth. The illustration is based on the idea described in [24].

The key takeaway from this analogy is that a spontaneously evolving system

becomes *less ordered*, hence, it needs periodically executed *corrective actions* to steer the system towards *a policy* governing the intended evolution of the system over time. Changes to the system state over time is using this model either *emergent*, e.g., a power outage, or *desired*, e.g., a new application deployment.

In this thesis, the corrective mechanism is referred to as a *reconciler loop* (or reconciler) which drives the *actual state* of the system (the sum of the fast and slow components) towards the desired long-term evolution policy – the slow component, referred to as *desired state*. This mechanism is called *state reconciliation* [33]. The desired state is defined such that the reconciler loop can obtain the difference (diff) between the desired and actual state, the fluctuation, referred to here as *state drift*.

For those familiar with control theory, the desired state is often called the *desired output*, the actual state the *actual output*, and state drift the *error signal* [30]. The reconciler loop represents a *closed-loop* control mechanism, because it takes the observed, actual state into account when deciding what actions to perform.

2.1 Entropy

Let us explore the concept of entropy in infrastructure systems a bit more precisely. Claude Shannon introduced the concept of *information entropy* in his paper "A Mathematical Theory of Communication" [25] already in 1948. Information entropy is a measure of how much "uncertainty", "surprise" or "choice" there is in a random variable, denoted here by X. Given a probability density function $P(X)$, the information entropy $H(X)$ expressed in bits is defined as $H(X) = E[-\log_2(P(X))]$, where E denotes the *expected value operator*. For a discrete random variable with outcomes x_i, it is $H(X) = -\sum_i P(x_i)\log_2(P(x_i))$.

For illustration purposes, let us consider the following: $H(X) = 0$ when the outcome is known, i.e., there is one outcome x_k where $P(x_k) = 1$. $H(X)$ simplifies to $\log_2 n$ when X is uniformly distributed, i.e., all outcomes have the same probability $\frac{1}{n}$, where n is the number of outcomes. A coin's entropy is as largest when the probability is 50-50, which makes sense as entropy is a measure of uncertainty. To connect this to server systems, one can identify a set of distinct categories resources can be grouped by, and model the random variable X as "how likely is it that a randomly selected resource falls into a given category". For example, if the set of resources are servers, the categories can be set of OSes available, assuming one server can only run one OS at a time (Figure 2).

Burgess proposes three dimensions of modelling infrastructure consistency [24]. In the "consistency" dimension (Figure 2) the resources are categorized and minimum entropy (order) is desired. The "compliance" dimension (Figure 3) models how consistently a *given* category is distributed across the resources. Finally, the "time" dimension models the evolution of a setting over time, where commonly maximum entropy is desired. Interestingly, Burgess points out that Shannon's definition of information entropy agrees with the previous definitions of entropy in the field of thermodynamics, highlighting a profound connection between information theory and physics.

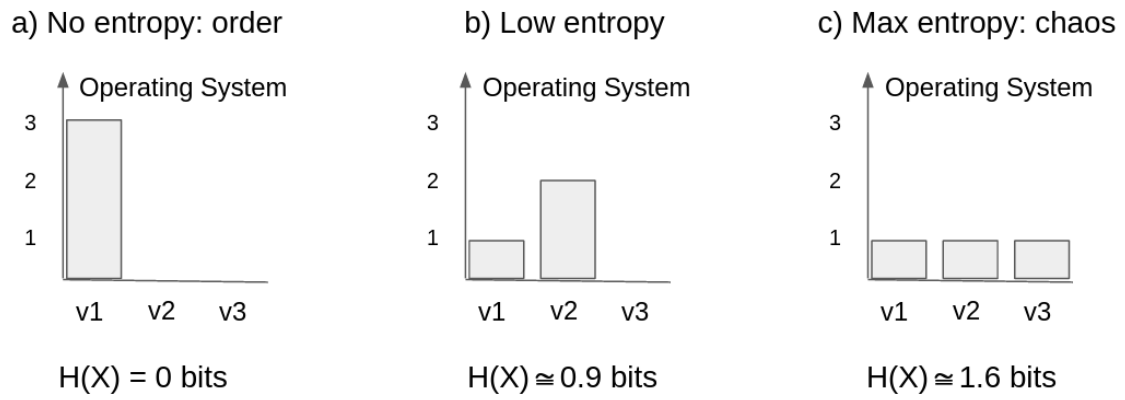

Figure 2: Histograms of how many servers are running a given OS. In case a) the probability of a random server having OS v1 is 100%, and thus $\log_2 1 = 0$ – in other words, the system is ordered. In b) and c) the entropy gradually increases, and in c) the maximum entropy state is reached, where the system administrator experiences the maximum amount of "surprise" when managing the servers.

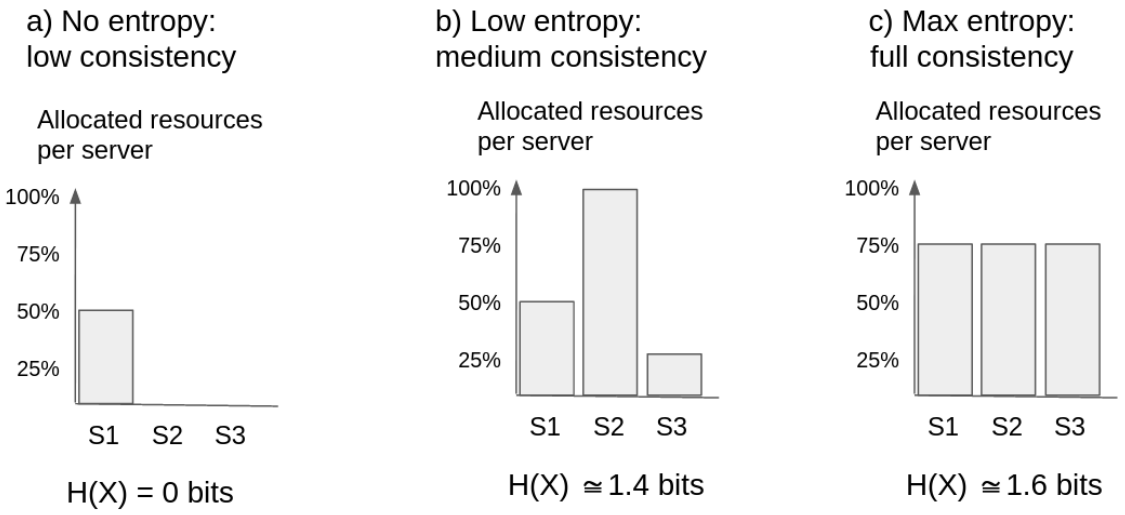

Figure 3: A company targets achieving maximum spread (or utilization consistency) of its workloads across servers 1-3. Case a) contains minimum spread or consistency, and thus no entropy. In b) the spread is a bit more even, but not as even as in c), where the maximum entropy state is achieved, and thus all utilization consistently distributed across the resources.

Example 1: A company buys three servers with exactly the same configuration A and OS v1. The servers are powered on. The system is ordered as the state is completely uniform. Later, a critical security OS upgrade v2 is released, thus, sysadmin S upgrade all but server 1 that is

running some critical workload and must not be disturbed. A day later, server 1 experiences slow disk access time due to misconfiguration. S pokes around imperatively (by using a web UI or CLI tools) on server 1 until the error is resolved. S notices that the user load has declined due to a seasonal variation, hence turns off server 2 to save power. A week later, while sysadmin S is on vacation, sysadmin T notices that server 3 reports an error, unknowingly the same error S fixed on server 1 earlier. T imperatively "fixes" the issue for server 3, but in a different way than S for server 1. A brand-new OS v3 with desired but incompatible features is released, hence, only server 3 is upgraded as an experiment. Unexpectedly, a lightning strike makes the PSU of server 3 unusable, shutting it down. This evolution is illustrated in Figure 4.

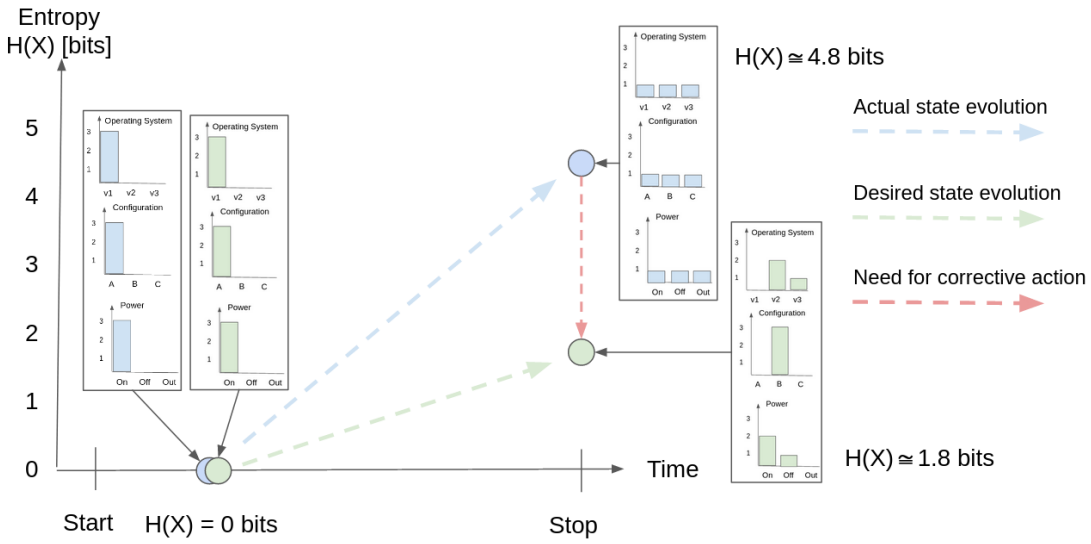

Figure 4: In Example 1, the entropy can be modelled with "consistency" histograms, showing how many servers belong to the different OS, config and power categories respectively. It is worth noting that the desired entropy increases slightly due to the experiment with OS v3 and reduced seasonal load, but the actual state is still vastly more unorganized. There is a clear need for corrective action to lower the entropy to the desired level.

2.2 Expect the Unexpected

Nassim Nicholas Taleb, in his masterpiece book *The Black Swan* [34], describes how the world is significantly more random than we as humanity believe, but also how blind we are to it. This in part because of our *retrospective* explanations of inconceivable events which fools us into thinking the world is more predictable than it is. The dynamics become even more interesting when the perceived low-probability event carries extreme impact: this is called a *black swan* event.

The first step on the journey of understanding the black swan phenomena, is to realize that most of our predictions carry errors magnitudes larger than we think and understand that it can be more efficient to focus on the extraordinary events, not as much on the ordinary. Note that there is no "silver bullet" solution to such a dynamic in this thesis; however, it is imperative to be aware of the need to design for failure and also to acknowledge that nothing is certain.

At Google, engineers acknowledge that unexpected events will happen, at unknown times, and thus allocate approximately 17% more resources than needed in order to "deliberately leave significant headroom for workload growth, occasional 'black swan' events, load spikes, machine failures, hardware upgrades, and large-scale partial failures (e.g., a power supply bus duct)" [6].

Other examples of randomness-dominated activities are large application deployments and service latency. This phenomenon is referred to as *The Long Tail* [27], [28] which highlights, similarly to the black swan ethos, that a small percentage of tasks to be executed (i.e., "stragglers") account for a significant, disproportionate percentage of instances of unwanted behavior, such as failures or high latency. For example, Garraghan et. al. [28] found that "5% of task stragglers caused by data skew impact 50% of the total jobs", and Dean and Barroso [27] noticed that 5% of fan-out requests [of a Google service] account for 50% of the 99^{th} percentile latency .

Eric Sorenson makes a related remark about that large-scale systems – particularly, Twitter's server infrastructure always experience a constant rate of errors [26]. Sorenson points out that the most interesting metric is not what specific system is experiencing errors; it is rather the derivative, i.e., the change in the rate of errors. In control-theoretic terms, it means that although a *steady state* [30] never is reached, the system can function correctly by *compensating for the error rate*. This practice is applied in many other domains as well; for instance, ships both offset the navigation error induced by the magnetic North not being equal to the geographic North and compensate for other factors such as wind.

> **Example 2:** If sysadmin S wants to run 100,000 instances of a web app, S can account for an error rate of, say 9%, by declaring the desired number of instances is to be 110,000, which to most likely yield 100,100 running instances on average in practice.
>
> One can think of a system never reaching the steady state but offsetting the errors as "moving", in contrast to a system not designed for the inevitable error rate as "static". Consider certification rotation or upgrades: if they are continuous, you do not "feel the acceleration", just as when an object already is in motion; whereas if the tasks are done manually every quarter, you feel the acceleration when temporarily moving a bit forward. That said, if you always are moving, the risk of falling is higher, but at a large scale you do not have the choice to stay static.

2.3 Declarative Programming

As the system might spontaneously drift into unwanted states due to entropy or randomness, it is instrumental to declare the desired state. The word *declarative* means "making a declaration", while *declare* refers to "to make known as a determination" [35], [36]. An example of a declarative sentence is "The door is shut", whereas an *imperative* example is "Shut the door!". A *declarative language* is one where the writer describes the links between the entities involved and the desired output or transformation of a process ("the what"), in contrast to an imperative language, which focuses more on exact sequences of operations to be executed ("the how") [37].

In *declarative programming*, this means expressing only the theory or logic of a computation, delegating the production of an appropriate control flow to a compiler or actuator. Using Kowalski's equation $algorithm = logic + control$ [38], the programmer supplies the logic part while the compiler deduces the control part, according to given rules and underlying constraints. Notably, it takes two to tango, thus logic without control is not sufficient in this imperative world we are living in. Therefore, a declarative programmer spends most of their attention describing the end state ("the what") to the computer while not focusing much on the imperative steps to get there ("the how"). That said, "declarative vs imperative" is not a binary choice, but a spectrum. Very few, if any, (programming) languages are purely imperative or declarative.

One can think of the declarative logic as a "claim", and the compiler or actuator fulfilling that claim. However, errors might result in the claim not being fulfilled at all. Generally the execution order or state evolution of declarative logic is not defined, which applies to at least Haskell [39], SQL [40] and Kubernetes. In fact, the *lack of* a defined execution order signals that the logic likely is declarative.

> **Example 3:** Code, written in a high-level programming language, in text files ("the what") is essentially a declarative claim of a computer-executable binary file with imperative CPU instructions ("the how"). Syntax errors in the code of an invalid claim result in a failure, but also the lack of enough computer memory can hinder the compilation process. The result varies depending on the compiler version, the machine architecture and applied optimizations (among others).

Examples of (at least mainly) declarative programming languages are SVG, GraphQL, HTML, CSS, React and SQL. *Functional programming* is a subset of declarative programming. Haskell [41] is functional, thus also declarative. The *pure* subset of functional programming is characterized by the fact that functions are *side-effect free*: they deterministically return the same output for a given input, and do not mutate any global or shared state, nor otherwise produce effects not directly related to the computation.

2.4 Abstraction Layers

By definition, a declarative claim forms an *abstraction layer*, namely, a generalized model hiding specific implementation details. Abstraction layers are a very common approach in Computer Science to manage complexity, such that whoever is using the abstraction employs some desired functionality without the need for knowing all the details under the hood.

Declarative claims do not force the programmer to manually hard-code specific control logic, but quite the contrary: they allow the compiler or actuator to optimize the returned set of steps to be executed, depending on the environment. For example, if the declarative claim is to compute the sum of a set of numbers given as input, the control logic can parallelize the summing by utilizing multiple CPU cores at once. This is in contrast with a hardcoded, iterative for-loop that just adds the items one by one into an aggregated variable; such a control flow misses out on the parallelizability.

Coupling is defined as the "manner and degree of interdependence between software modules" [42]. It is characteristic for cloud native [1] software to be interface-driven, thus *loosely coupled*. For instance, Kubernetes itself is designed as a set of five loosely coupled *microservices* interacting through well-defined APIs [43]; for many cruicial functions it does not ship with any implementation at all. For illustration, Kubernetes leverages workload isolation and runtime support from CRI implementations, networking from CNI implementations, and storage from CSI implementations.

Hyrum's Law states that "With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody" [44]. In other words, the abstraction layer (here equivalently referred to as API) effectively has zero "degrees of freedom" once its user base reaches critical mass. This indeed is a challenge; however, well-defined and -specified API contracts are still of utmost importance.

In some cases, like Haskell and Kubernetes, however, part of the API contract might explicitly state that specific implementation details, such as order of execution, are undefined. Kubernetes and SRE best-practices [45, ch. 22] even *inject randomness* into reconcilers, e.g., their reconcile period, to reduce the probability of too many clients issuing requests to the API server at a given time [46]. More "degrees of freedom" are obtained as a side effect as this approach prevents users to rely on resource reconciliation at pre-defined times.

2.5 Desired and Actual State Separation

Using a linear algebra mindset, one can think of the desired and actual state as *coordinates* in some abstract state space, and the distance between them the state drift. Because the coordinates represent specific *states* of a system, the desired and actual states are hence *declarative*. In contrast, a vector in this space could be thought of as an *imperative*, state-transforming action.

> **Example 4:** When you call a taxi, it is obvious to declaratively tell the actor (taxi driver) the coordinates of your actual and desired state. Imagine how awkward it would be to imperatively describe your movement as a vector, that has no fixed starting point, just an angle and length: "Drive me 50 kms southeast, but pick me up anywhere". It is just as awkward to execute an imperative action against a target whose actual state is unknown.
>
> Thus, it is instrumental to describe both the actual and desired state, otherwise you get the same experience as Alice in Wonderland, where she asks the cat for directions without knowing her destination. The cat responds: "If you do not know where you are going, any road will take you there".

Given this separation, consider an *emergent* change to the actual state coordinate, e.g., a power outage that transforms the actual state from 100 servers running to 25 servers running. An emergent change moves the actual state coordinate; it can be caused, for instance, by humans, other systems or, as discussed above, randomness. Furthermore, the requirements on the desired state tends to change over time. For example, a new application maybe is deployed into production, additional available servers are added, or the resource limits of an application are adjusted. This is called a *desired* state change that only affects the desired state coordinate.

An *idempotent* operation f mathematically satisfies the equation $\forall x : f(f(x)) = f(x)$. In other words, f is such that its value does not change after the first application $f(x)$ after being applied any number of times to its own output. In Borg, one of the strategies to deal with failures is having idempotent mutating operations, such that a failed or occasionally disconnected client can submit API requests more than once harmlessly [6].

This distinction between desired and emergent state changes highlights that they affect the desired and actual state coordinates independently. The state drift can be modelled as the distance between the state space coordinates; thus, it is natural for an actuator to be idempotent and execute no action when the state drift is zero.

This abstract state space can be an useful analogy when reasoning about these concepts, although not necessarily a 1-1 mapping to the real world. *Orthogonality* is a key concept in linear algebra; orthogonality of a state space represents operational independence. For example, in Kubernetes there are reconcilers that operate on completely independent state, e.g., Deployment and CertificateSigningRequest reconcilers. In other words, they operate in *mutually orthogonal sub-spaces* within the complete state space. Thus, each reconciler just sees its own, say one- or two-dimensional, state sub-space and converges the actual state coordinate towards the desired state in that *projection*. Figure 5 illustrates two sample dimensions from Example 1 above.

2.6 Control Through Choreography

Kubernetes' design is an example of the *control through choreography* design pattern, applied at a large scale [8]. Kubernetes reconcilers are loosely coupled, independent,

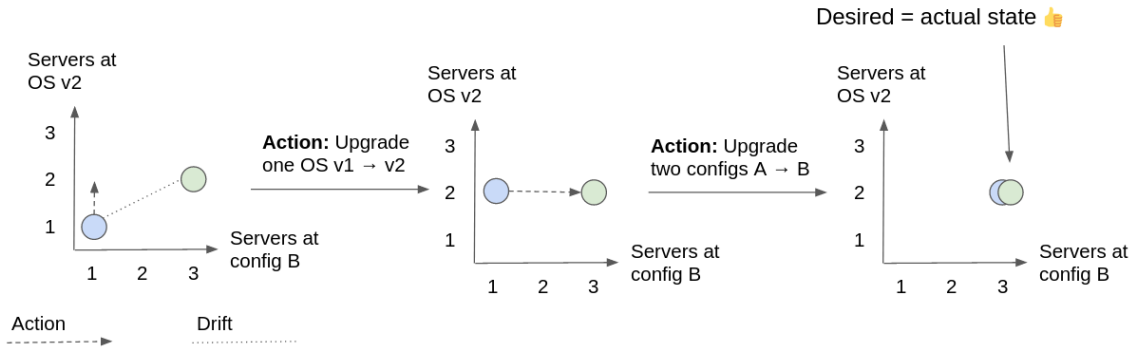

Figure 5: Desired and actual state given two state dimensions in Example 1; the number of servers running an OS at version v2 and with a given configuration B. At first there is one server (of desired two) running OS v2, and one (of desired three) with config B. This is addressed by one (or two) reconcilers issuing two possibly independent actions, eventually making the desired and actual state converge.

executing in parallel and base their actions on observation instead of a fixed state diagram, which makes them more robust to perturbations and failures [8]. The system as a whole achieves the desired state through the combination of all these autonomous reconcilers cooperating together.

There are similarities in how the reconcilers are broken up into small, independent units and the UNIX mantra "make each program do one thing well" [47]. Using the linear algebra analogy from above, finding a set of cooperating reconcilers that operate independently can be thought of as *orthogonalizing* the state space.

This is quite different from the more centralized, graph-based model deployed in Terraform, where tasks can be executed in parallel only to a limited degree, if allowed by the dependency graph [48]. Reconciliation is done simultaneously for all managed resources in the Terraform Root Module Configuration; thus all root module resources share the reconciliation interval in contrast to Kubernetes, where intervals are resource-specific. Remote state synchronization is handled by *backends* [49] in Terraform (the API server in Kubernetes), thus allowing for "true" parallelization if one root module is split into many independent ones. This, however, requires conscious and careful design by the user.

Control through choreography is also quite the opposite from Borg and Docker Swarm, that have monolithic and centralized designs [6], [50]. A centralized design can be "easier to construct at first but tends to become brittle and rigid over time, especially in the presence of unanticipated errors or state changes" [8]. It also allows the user to declare the desired state for multiple dependent resources at once, and then just wait for convergence organized by multiple independently operating reconcilers. This is in contrast with applying steps in some order defined by a state diagram, thereby assigning the actuator the job of managing temporal dependencies.

Mean time to recovery (MTTR) and *mean time before failure* (MTBF) are two critical metrics for systems either in healthy or error condition. MTTR is the mean

duration between a system entering a failing state and it becoming healthy again. MTBF is the mean duration that a system continuously stays operational. Clearly, it is required that $MTTR \leq MTBF$ to even have the slightest chance of keeping the system healthy, even though $MTTR \ll MTBF$ is needed in practice [24]. Corrective actuators "race" against entropy- and randomness-related actors (see Sections 2.1 2.2). This presents factual evidence and a theoretical background on why periodic reconciliation is needed, instead of a one-off execution.

Actions repairing a system in an error condition should happen in parallel to minimize the MTTR [24]. In fact, repair actions carried out sequentially consume more time in total and/or might block each other unnecessarily, possibly even leading to further problems. Of course, parallelism can be both a friend and foe to stability and consistency. But just like *firm real-time systems*, there is often some grace period before warnings or errors appear and the system severely fails [31, ch. 1]. For example, a system receiving a constant influx of objects eventually would crash due to finite disk space if garbage collection never would be executed.

2.7 State Relativity

As pointed out by Burgess, as speed of information transfer is finite and errors inevitable, therefore, consistency is not instant nor guaranteed but *assumed with some probability* to have occurred *after a given amount of time* ("stochastically probable consistency") [24]. Consistency cannot be guaranteed because of the "MTTR-MTBF Race" between corrective actuators and inevitable randomness Accordingly, Burgess argues for "configuration management from within", a core CFEngine design principle. In other words, actuators are *pull-based* and as close to the target system as possible, empowering *topologically* sensible caching and reconciliation of the desired state.

When the desired state is cached, yet periodically pulled, "local" reconciliation within a topological failure domain guarantees this *stochastically probable consistency* within that failure domain, even in the case of disconnection from the rest of the world. This is slightly analogous to Schrödinger's Cat, whose state cannot be observed from the outside but yet can be alive. Thus, it is possible to achieve local, per-failure-domain ("many worlds") consistency even when multiple failure domains are disconnected at once, although this cannot be verified [24].

In contrast, a centralized, "push-based" actuators cannot guarantee any form of consistency for a remote system when its failure domain is disconnected, because the push-based approach *relies on remote system connectivity*. The more distant the remote system is from the actuator, the more inevitable latency is attached to information transfer, the more unreliable actuator actions are. Recall Example 4: acting on a system in an unknown state is awkward; it can even be dangerous.

As the speed of information transfer is finite, request latency is always positive, regardless of if the actuator is push- or pull-based. When reasoning about these distributed system problems, it can be helpful to attach space and time metadata, *a frame of reference*, to pieces of information handled. Figure 6 illustrates how state can have changed between observation and corrective action, how latency need to accounted for, and how a system can be consistent yet not verifiable. In this

illustration, it becomes clear that all *state is relative* to its time and location (space).

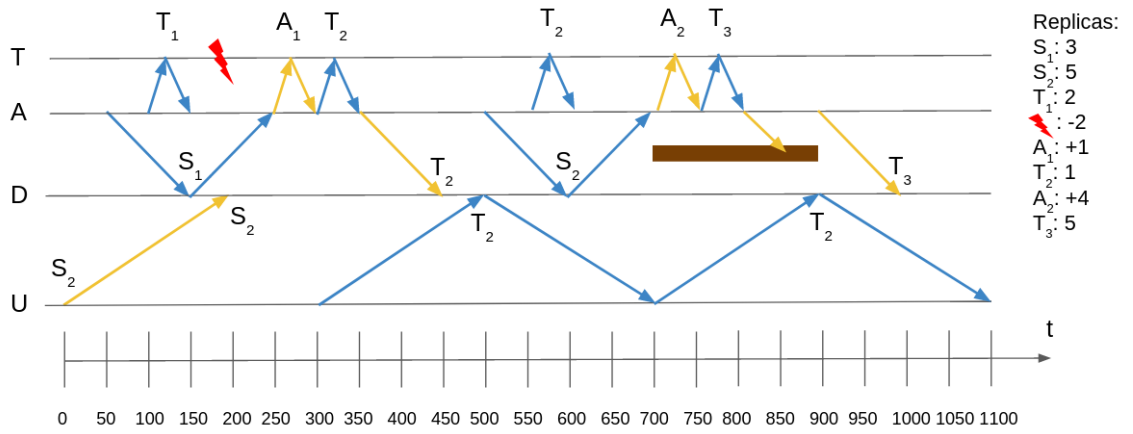

Figure 6: State relativity illustration. User U declares a new desired state $S_2 = 5$ [replicas] to desired state store D. At $t = 50$, actuator A pulls desired state from D. U is located 200ms and A 100ms from D. $S_1 = 3$ is returned to A, as S_2 does not arrive in time to D. A reconciles the local target system T, 25ms away. After A observed $T_1 = 2$, both replicas crashed (-2). A compares S_1 and T_1, and adds one replica in $A_1 = +1$. A inspects T again, and reports the actual state $T_2 = 1$ back to D, such that T's state is visible to U. A's second reconciliation receives S_2 correctly, applies $A_2 = +4$, and observes $T_3 = 5$. Although T is consistent ($T_3 = S_2$) at $t = 800$, U cannot verify this due to a temporary network failure between A and D.

In fact, Kubernetes was designed with the objective that control plane connectivity is not required for successful node operation, just like airplanes can shut off their engines during a flight [51]. Clearly, nodes cannot receive any desired state updates while disconnected, but they keep the system in sync with the latest desired state.

2.8 Concurrency Control

For a distributed system, keeping state in sync and aligned with some desired policy is a primary concern. At its extremes, two main categories of concurrency control can be identified: optimistic and pessimistic. Optimistic concurrency control allows multiple independent processes to compete in parallel for the same resources, because the frequency of conflicts are relatively rare compared to the gains in the increased throughput from the parallelism (beneficial for lowering MTTR; see Section 2.6). Optimistic concurrency can be implemented, for instance, through an ever-increasing number or timestamp field that is increased every successful write. Read requests are populated with the current value of the field; in order for a write to be successful, the field in the write request must match the "current" value, indicating no conflicts.

Example 5: Clients A and B read resource R, with current concurrency field $F_{req} = 1$. B tries to write a modified R. The server S accepts the

write as $F_{req} = F_{server} = 1$; then sets $F_{server} = 2$. Later, when A tries to update R, S notices that $1 = F_{req} \neq F_{server} = 2$, and declines the request. A must re-read the updated R and try again.

Pessimistic concurrency control uses a locking mechanism – for example, some kind of mutual exclusion lock – to only allow one actor access to a particular set of resources for some time. This can avoid conflicts at the cost of decreased throughput and potential lock contention, depending on how fine-grained the locking is.

Google's Borg scheduler is monolithic and not parallelized, thus, it suffers from saturation when the computation time for a single scheduling decision increases [7]. To solve this, Google built Omega, with a novel scheduling algorithm that is parallel, lock-free, optimistic, and uses shared state. This algorithm is more scalable, removes head-of-line blocking and thus useful for both synthetic and real-world data [7]. For this reason, Kubernetes API server uses optimistic concurrency [52], with an opt-in [53] for pessimistic concurrency when needed.

Docker Swarm implements a Borg-like monolithic scheduler, where the cluster managers form a Raft [54] cluster that elects one "leader" which provides the API and scheduling [55] [50]. The popular Git workflow is one kind of optimistic concurrency, many contributors work on changes in parallel, and once a change is merged to the *main* branch that affects others' changes, they will need to *rebase* to cleanly apply their change. Terraform employs a pessimistic concurrency strategy using the locking feature [56] for state backends [49].

2.9 State Reconciliation

This section summarizes the lessons learned from above sections and present a set of design principles for how to build reconcilers. First of all, it can be concluded that the purpose of the reconciler is to *keep the target system in control* and provide a *declarative abstraction layer* for managing the target system to its users. By encoding operational logic about the target system in the reconciler; the user of the declarative API does not have to take a state diagram or temporal dependencies into account. All in all, this is a way of *centralizing complexity*; moving it from the user to the reconciler. This is a reasonable design decision, at least in the case of Kubernetes.

Because of the discrete nature of computers, state reconciliation cannot be continuous as in physical or electrical controllers, but periodic at best. The period varies greatly by reconciler. The reconciler needs to react to both emergent and desired state changes (Section 2.5). The reconciler should do one thing and do it well (Section 2.6). The reconciler should support any state transition, including rollbacks, such that "declarative [desired] state makes rollback of a change trivially easy" [23, ch. 1]. The reconciler should be *re-entrant*, i.e., be such that the control flow can be interrupted at any time and then re-entered without issues [57].

The reconciler should be *idempotent* (Section 2.5), in other words always check for if an action *actually needs executing*, or if it already has been executed with return values readily available. For example, if the desired and actual state equal, no action should be taken. Alternatively, if an automatically-generated ID field already has been set, it should not be re-generated.

The reconciler is tasked with managing temporal dependencies (Section 2.6). Thus, the reconciler should "re-queue" a resource (postpone the resource reconciliation after some duration) whose dependencies are not ready, e.g., a non-existent Kubernetes ConfigMap on which Pod P depends postpones reconciliation of P. The retry ("backoff") period should be randomized and grow exponentially with the amount of sequential "failures" [45, ch. 22]. Finally, three reconciler properties are further investigated in the sub-sections below.

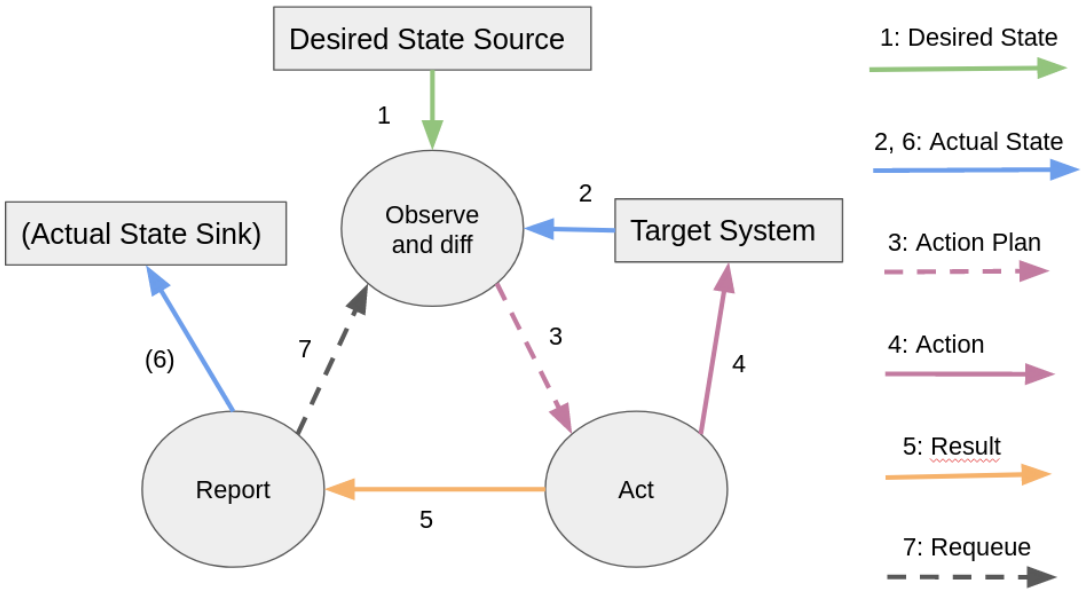

Figure 7: The control flow of a reconciler loop. 1) The desired state is fetched. 2) The actual state is computed. 3) The actual and desired state is compared, and an action plan is computed, if any. 4) The action plan is executed, if it exists. 5) The result of the action(s) is observed. 6) Optionally, the new actual state is reported. 7) The resource is scheduled for re-reconciliation after some time.

The control flow of the reconciler is described in Figure 7. At a high level, the reconciler needs to: receive the desired state from a *Desired State Source*; access the *Target System* being controlled; and optionally access a high-level *Actual State Sink* for reporting the observed the actual state, such that other actors can access the actual state without the need to observe the target system themselves.

2.9.1 Level- vs Edge-triggeredness

This section will cover two strategies for *triggering* reconciliation of a resource. *Level-triggered reconciliation* is carried out by *comparing two input signals (levels) against each other*, be it two voltages as for a CPU or desired and actual state for a reconciler. *Edge-triggered reconciliation* reacts to *events*, like a *falling or rising edge* for an electrical signal, keyboard press, or a HTTP request. The levels are *continuous*, and

can be "checked and re-checked" as often as needed, e.g., on a *periodic schedule* like most reconcilers. The events, however, are *asynchronous, sporadic and available for a finite amount of time* by their nature.

For keeping server infrastructure under control, level-triggered logic is preferred over edge-triggered logic [33], [58]. Although a level-triggered reconciler crashes or otherwise is unavailable for some time, all data needed for making decisions is still available in the form of desired and actual state, in contrast to past events that might have disappeared. The risk of missing events becomes particularly significant in the light of risk of network partitions and inevitable inter-component latency discussed in Section 2.7.

> **Example 6:** A workload is scheduled to a node in Kubernetes. The node agent periodically pulls the desired state (Pod objects) and compares it with the actual state observed from the local container engine.
>
> Conversely, if the node agent were edge-triggered (at its extreme), it would would only act on "Scheduled" events from the scheduler, thus not run any Pods at all during startup, although it is already assigned 10 "old" Pods.

Finally, sometimes edge- and level-triggered designs can be combined. If the fundamental design is level-triggered, reconciling the desired and actual state periodically, the *response time* can be minimized by *additionally* receiving resource created/updated/deleted events triggering reconciliation before the periodic schedule. Kubernetes does this to turn the response time from minutes to seconds [33], [59].

2.9.2 Uni- vs Bidirectionality

With *unidirectional reconciliation* a reconciler sources the actual state from some cache at least occasionally; in contrast, the reconciler always observes the target system in *bidirectional* mode [33]. Unidirectional mode is used when observing the target system (e.g. a public cloud API) is monetarily too expensive, rate-limited, too slow, or otherwise infeasible. Thus, it is still a valid option although consistency guarantees are weakened when inconsistent actions may be performed if the cache is stale. Deletions need to be level-triggered to empower the unidirectional reconciler to "notice" the deletion and update its cache and target system accordingly [33].

> **Example 7:** A unidirectional reconciler R is responsible for keeping a cloud loadbalancer L well-configured. L is created during the R's first reconcile round (sync); the actual state cache records the existence of L. Accidentally, L is deleted due to human error. R does not notice the deletion of L (and thus, re-create L) during the second sync, as L still exists in the cache. Thus, an admin re-creates L "by hand". Later, L is not needed anymore; thus deleted from the desired state. R notices the deletion request in a level-triggered fashion through L being in the cache but not desired state.

In bidirectional mode, however, resource *ownership* needs particular attention. By design, a bidirectional reconciler should delete any actual resources not present in the desired state. But without an ownership mechanism, two bidirectional reconcilers syncing disjoint subsets of the same resource type would delete the other's resources it does not recognize [33].

The majority of the Kubernetes reconcilers are bidirectional, however, some of them are unidirectional because of said reasons. Level-triggered deletion in Kubernetes is done through *finalizers*. Terraform has a CLI flag for mandating unidirectional mode, and supports level-triggered deletions "automatically" thanks to its architecture that separates storage of desired and actual state [60].

2.9.3 Actual State Reporting

In order for external actors and humans to know what the reconciler is doing, and in what state its target system is at the moment, the reconciler should to report what is the actual state to some actual state sink (recall Figure 6). Such "high-level" state reporting makes it possible for higher-level actors (without access to the target system) to build additional automation "on top" of the *base* resource reconciliation. In other words, actual state reporting allows a set of reconcilers to be loosely coupled, communicating if needed through well-defined APIs.

> **Example 8:** A Kubernetes Pod is a declarative claim for a running container. Before execution, the IP address of the unknown and undefined. Once running, the Kubernetes node agent populates the Pod `status` field with info about the Pod IP, container ID, etc. such that higher-level reconcilers, e.g., can route network traffic to the Pod correctly, without modifying the node agent.

However, there are further use-cases for this reporting. The reconciler should, by design, transform resource claims into resources. Thus, it needs to somehow keep track of what desired state claim maps to what target system resource. In some cases this is trivial, in some not. Consequently, by reporting to the actual state sink that "this claim maps to this resource"; the reconciler itself can become stateless, thus easier to deploy and more robust [60].

When the set of resources to reconcile grows large, the reconciler needs to become increasingly efficient to be able to process all claims in a timely manner (recall MTBF from Section 2.6). Thus, metadata and *metrics about the reconciliation process itself*. OpenMetrics [61] and OpenTelemetry [62] are two popular telemetry data specifications for this and many other purposes.

Finally, this information can be used for visualization through UIs, aid debugging, and overall increase introspectability and observability of the system. The actual state becomes yet more valuable if it conforms to some well-specified set of conventions, such that any convention implementer can process data from many distinct reconcilers similarly, and display consistent data in UIs, CLIs and monitoring dashboards.

Kubernetes reconcilers conventionally report the actual state to the `status` field of the API object (for reference, the desired state is stored in the `spec` field). There are well-defined conventions for how to report `status` [21], [63]. Terraform, in turn, produces a *state file* [60] which is stored in a pluggable *backend* [49].

3 Declarative Configuration

The key takeaway from the previous chapter is that there is a need to describe the desired state of a target system (or *application* / app) declaratively. Thus, this chapter covers the declarative representation, i.e., the configuration ("config") of the application.

3.1 Complexity Clock

Initially in an application's lifecycle, before the app has a critical mass of users and features, parameters tend to be "Hard Coded". Over time, the need to change the defaults arise. Configuration in its simplest form tends to be just a set of scalars (e.g., strings, numbers or booleans) or key-value pairs (e.g., CLI flags or environment variables). However, once the set of knobs evolve and grow larger, there is a need for grouping and "more structured" config. Accordingly, the "simple" key-value pairs are turned into, e.g., JSON, YAML or XML documents ("Config Values"). Over time, the config becomes complex [29], which calls for "Rules Engine" features such as typed schemas, validation, defaulting, and boilerplate reduction (e.g., inheritance and overrides) [64].

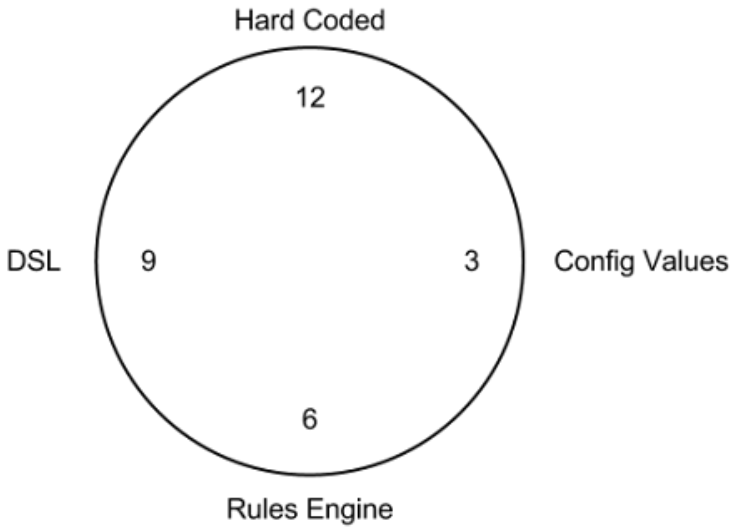

Figure 8: Hadlow, M. (2012). The Configuration Complexity Clock [64]. Accessed 2021-10-29. Retrieved from https://mikehadlow.blogspot.com/2012/05/configuration-complexity-clock.html. Licensed under the MIT license.

However, it is almost impossible to make a set of generic rules to govern the whole configuration space, and so a DSL is born to express the config through a custom programming language that is possibly even *Turing-complete*. However, building a custom programming language comes at a significant engineering cost,

and becomes *another* dependency to learn to get started. Moreover, the DSL now needs to be *executed to be understood*, unlike "plain" configuration. Unless a great amount of resources is invested in the DSL, it lacks features like unit testing and debugging, which are crucial for correctness over time. Finally, the DSL is replaced by a "real" programming language used for config generation, eventually leading to two programs: the app and the config generator for the app, where the latter is at the "Hard Coded" phase.

This phenomenon is common enough to have a well-established name: the *Configuration Complexity Clock* [64] (visualized in Figure 8). It is not a problem that can be easily solved; the authors of "Borg, Omega and Kubernetes" conclude: "Of all the problems we have confronted, the ones over which the most brainpower, ink, and code have been spilled are related to managing *configurations* — the set of values supplied to applications, rather than hard-coded into them" [8].

3.2 Data and Generation Separation

Configuration structure tend to grow larger and more complex over time but, unfortunately, too many configuration parameters ("knobs") can lead to cognitive overload, as the human brain capacity is limited [29]. The cognitive overload can be observed as the inability to find the right knob or understand what a knob does. Thus, the need for an *abstraction layer* arises (see Section 2.4), aiming to optimize the productivity of a certain type of user. This is often done through programmatic generation of a small, "high-level" config schema into the complex, "complete" config data.

This "config generation" method, is in fact what the authors of both [8] and [20, ch. 14] recommend: defining a clear contract between the configuration generation and resulting data. The config data could be expressed using a *data-only format* (like YAML or JSON) and the programmatic generation in a *real* programming language, similarly to how web pages clearly separate structure (HTML) from computation (JavaScript). This best-practice is illustrated in Figure 9.

In terms of Kowalski's equation `algorithm = logic + control` [38] (see Section 2.3), this flow could be described as [config] `data = high-level config + generator` (or `data = generator(high-level config)`). The `generator` function should be hermetic, i.e., *deterministic* and *side-effect free* [20, ch. 15]. Expressed in terms of functional programming, `generator` should be a *pure* function (see Section 2.3).

3.3 Data Formats

Overall, it seems like the cloud native community has converged on configuration data formats that support the *lingua franca* of elementary data types: unordered key-value maps, ordered arrays, and scalar values. Formats that support these basic primitives include JSON [65], YAML [66], TOML [67], XML [68], and Protobuf [69]. When config representation is limited to this "common denominator" of data types, the config can be represented equally well in any format, and thus converted between

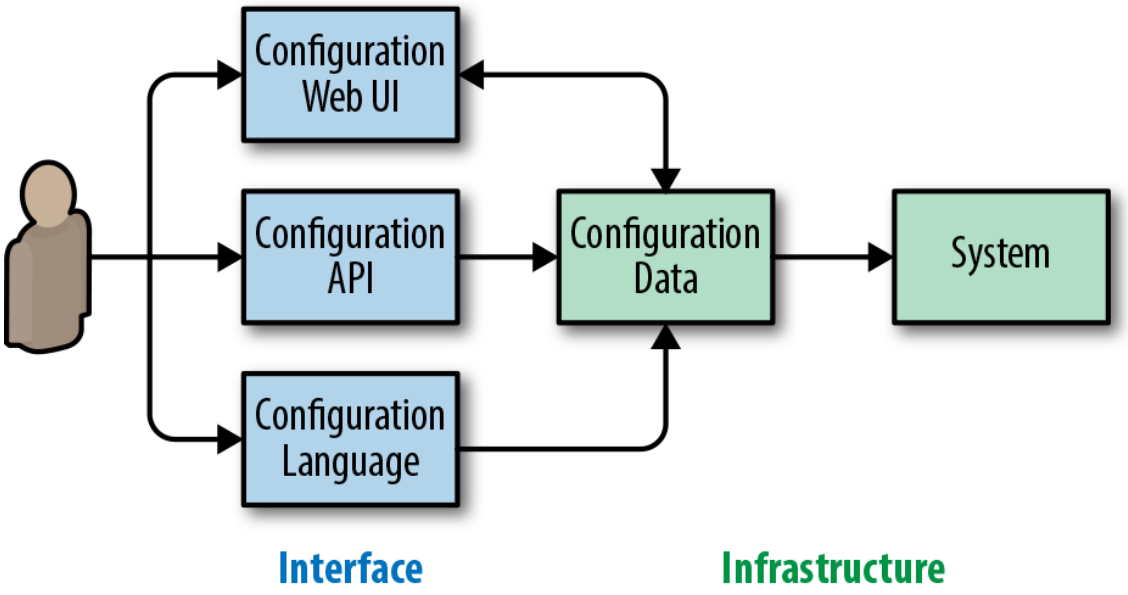

Figure 9: Beyer et al. (2018). The Site Reliability Workbook, Figure 14-2 [20, ch. 14]. Accessed 2021-11-24. Retrieved from https://sre.google/workbook/configuration-design/. Licensed under CC BY-NC-ND 4.0.
The Google SRE [20, ch. 14] and Kubernetes [8] best practice of separating the configuration high-level *logic* and the *control*: generation of config data.

formats without data loss. In fact, the format used becomes only a matter of taste for the user.

For example: Kubernetes supports JSON, YAML and Protobuf. YAML is preferred over JSON by users writing "raw config data" by hand. JSON is preferred by HTTP clients of the API server. Protobuf is preferred for inter-component cluster traffic, because it is a more efficient binary wire format with lower CPU and memory overhead. Consequently, Kubernetes is able to achieve desired scalability targets without forcing users to deal with binary formats [70].

3.4 Generation

Regardless of how the configuration "compilation" or generation is achieved, it should – according to best practices – "support configuration health, engineer confidence, and productivity via tooling for managing the config files (linters, debuggers, formatters, IDE integration, etc.)" [20, ch. 15].

Multiple config generators can be combined and put into a pipeline; there does not have to be just a single generator. Although this can be tempting, [20, ch. 15] reminds us that with every new step of generation, the *complexity toil* increases. That said, in some cases, more than one generation steps are needed, e.g., to combine generators written in distinct programming languages.

HashiCorp chose to develop a DSL for their software: HCL [71]. HCL can

be thought of as an extension to JSON, with additional features such as formula interpolation. Users of the generic HCL format like Terraform [18] establishes HCL schemas and conventions. This is quite similar as how Kubernetes specifies API conventions [21] for its use of JSON and how BCL, the configuration DSL for Borg, [6] relates to GCL, a general-purpose configuration DSL used at Google [72].

It is worth noting that one of the primary authors of BCL, Marcel van Lohuizen, is the creator of the open-source CUE [73] configuration language, which aims to strike a balance between providing usability while enforcing order independence and hermiticity while purposefully not being Turing-complete. In other words, much like Kubernetes draws on lessons learned from Borg and Omega, CUE applies lessons learned from the Turing-complete BCL.

It shall be noted that this kind of config generation abstraction layer by definition builds in trade-offs by focusing on a specific user. Consequently, to meet all users' demand, multiple distinct generators need to be made. For instance, the Kubernetes community includes over 120 known tools to help with configuration complexity, including, but not limited to: kustomize, Helm, kpt, CUE, ksonnet and Carvel.

3.5 Encoding and Decoding

Before two entities can communicate, they need to agree on what language to speak, or at least know what language the other entity is using. *Encoding* refers to the process of transforming a message, idea or data into a representation that can be stored or transmitted, using some *encoding format*. *Decoding* is the opposite process which reconstructs the stored or transmitted representation back into the original message, idea or data. In order to decode, at least the encoded data and its format must be known. However, although the receiver is able to *technically* decode the message, they might not have enough *context* to know how to *understand* it. Examples of context include, but are not limited to: version/dialect of encoding format, message kind, or entity ID.

> **Example 9:** Person A wants to describe their new home to person B. A encodes their memory of their home into Finnish speech. B hears the speech, and decodes it such that B now has an impression of A's home too. The *encoding format* is Finnish speech. If B does not understand any Finnish, B might not even know what language A is speaking, thus the encoding format is unknown. If B does not understand Finnish, B is unable to decode the speech.
>
> Persons C and D exchange messages online. C speaks Swedish and D Finnish. They negotiate to use written English as their *lingua franca*. C is accustomed to UK English and D US English, and because there is a mismatch in the encoding format *version*, some words are misunderstood. Furthermore, when D sends messages about their cat E without any prior introduction, C can *interpret*, but not *understand*, the messages.

Example 9 highlights that for decoding messages successfully, the message *metadata* is crucial. Message metadata includes the encoding format (to *interpret* the message) and the context (to *understand* the purpose of the message). Some communicators rely entirely, sometimes quite successfully even, on *auto-detection* of the format and/or the context. Some communicators hard-code assumptions about the format and context.

> **Example 10:** One can try to identify the format of a file by investigating its *file extension*, repeatedly trying to parse the file using many parsers until success, or looking for magic numbers (e.g., [74]). One can try to assert the purpose of the file by looking at existence of certain data points, e.g., JSON fields. However, all of these implicit approaches risk making wrong decisions too often in users' view, or have security implications.
>
> HTTP URLs and headers often communicate both encoding format (using the standardized `Content-Type` header) and related context between client and server. For instance, a HTTP POST to DigitalOcean's `/v2/droplets` explicitly communicates "create a VM (droplet) using the parameters specified in the attached JSON object of schema v2".

From real life, most people should agree that a "good" communicator swiftly negotiates the format (language) and version (dialect or level) with their audience and clearly expresses all required context before presenting their case.

In this light, Kubernetes could be considered a well-behaved communicator, as the set of formats is fixed (JSON, YAML and Protobuf) and specifying the context is *required*. Kubernetes config objects are characterized by three key context properties: *API version*, *API group*, and *kind*. The *API version* expresses the object schema version used, *kind* is a name for the object schema, whereas the *API group* is a classification mechanism for grouping related but distinct *kinds* together. Collectively, this triple is referred to as a *GroupVersionKind* (GVK). The Kubernetes encoder always adds GVK information, which is then used by the decoder on the way back.

> **Example 11:** A recent Kubernetes Deployment object has `kind=Deployment`, `group=apps`, and `version=v1`, but a previous (now removed) API structure used `group=extensions` and `version=v1beta1`. However, there is automation to upgrade the config structure from the previous to current version.
>
> The presence of API groups avoids ambiguity between projects and use-cases, e.g. `kind=Cluster` could refer to a database, container, or storage cluster. Furthermore, API groups allow several kinds to be versioned, documented and evolved as a group. Thus, a kind is unique only within its API group, not globally.

An alternative to auto-detection and explicit context specification is the "Convention over Configuration" practice utilized in Ruby on Rails [75]. CoC trades some "uniqueness" or "individuality" in favor of consistency and less cognitive overhead.

> **Example 12:** Many computer programs use file names and directory structure as a way to express format and context in a CoC-like way, similar to HTTP URLs and headers. For example, given a file extension, the format is asserted and given a file name, the config kind is asserted. However, users with workflow requirements in conflict with the convention are unable to use the program.

Once type information is part of the encoding step, and always written in serialized form, a user can use any file/folder structure, the version is clear to reviewers, automated schema linters can be used for pre-merge validation and reconciler loops operating on that data can know directly if they can operate on the object. For example, if an engineering team uses Git for application development, they might also want to use Git for a consistent application deployment to desired target environments. Due to that the desired state checked in to a Git repository is *self-describing*, they can utilize any file and folder structure, and put multiple related objects into the same file by using e.g. YAML document separation.

Kubernetes enforces strong API conventions [21], as a lesson learned from Borg and Omega [8]. These API conventions result in *resource uniformity*, and allow for great *extensibility*. For example, all resources are encoded with their GVK metadata, and express uniform metadata information in the top-level `metadata` field. This allows for any reader (e.g., a CLI, UI or reconciler loop) to make sense out of an object, although it is unknown.

3.6 Versioning

Configuration is bound to evolve over time, as a consequence of the general entropy increasing (see Chapter 2) and the application (requirements) evolving. For example, Xu et al. [29] found that on average, 8 config parameters were added, renamed or deleted *per application version*, for four applications investigated. This highlights a need for up- and downgrading config schemas between versions.

Let us consider a case where app A supports only one config schema version, for each possible app version. An up- or downgrade of A possibly requires a *simultaneous* config up- or downgrade. This requires the administrator to closely keep track of tightly coupled app and config versions when up- or downgrading. Mathematically, if function f maps app version to config version, f is *surjective*, possibly *bijective* if the config is always upgraded when the app is. CLI parameters are an example of this kind of app-config version relation.

> **Example 13:** In Kubernetes, CLI parameters have been criticized by users for this tight coupling, making it harder to run Kubernetes safely in production. For instance, say that user U is using the `-insecure-port` *kubeadm* flag at v1.21. This flag is removed in v1.22. Thus, the automation that is executing kubeadm, along with related CLI flags, needs to be

aware of the kubeadm version and adjust the CLI flags accordingly. In contrast, if `-insecure-port` would still be passed at v1.22, an error would be thrown. Furthermore, if U first upgrades to v1.22 (with correctly adjusted flag values), and then wants to roll back, the upgrade should not have affected the config supplied at v1.21. Thus, `-insecure-port` should be passed if and only if app version is equal to or below v1.21.

A better approach, is to let each app version be compatible with N *previous* config schemas, where $N > 0$. The app will then internally upgrade any "old" schema it sees, within the supported interval determined by N. This allows an admin to first upgrade the app version, and first when there is *enough confidence a rollback will not be needed*, the config is upgraded to the current version. Some Kubernetes components, kubeadm included, support this feature called ComponentConfig [76] [77].

In fact, The Kubernetes design goes even further. Some systems are designed with the assumption that a new config version makes up a new, independent "config object", such that one can say "config of version X in the system". Kubernetes, however, treats versions just as different "views" of the *same* config object.

Thus, many Kubernetes API resources are both fully upgradable *and* downgradable, in other words, *roundtrippable*. Roundtrippability, in this context, means the ability to convert between versions without any data loss, such that, e.g., `config = down(up(config)) = up(down(down(up(config))))` holds, where `up` and `down` are up- and downgrade procedures, respectively.

Example 14: Git is an example of a system where every version is potentially a new "object". A Git commit represents changes in file data, and thus file contents before and after are two separate objects with distinct data.

Kubernetes v1.22 supports three schema versions of the Horizontal Pod Autoscaling (HPA) resource, `v1`, `v2beta1`, and `v2beta2`. This means a user can create a new HPA called H using schema version `v2beta2`, yet still request H from the API server at version `v1`, and vice versa.

3.7 Storage

Let us discuss state stores referred to in the reconciliation model described in Section 2.9. The first crucial property of the state store is that it does not modify the target system in any way in such a model: it should be *side-effect free*, just like the pure functions discussed in Section 2.3. That might sound obvious, but several state stores are in fact deeply entangled with the control flow of the system, e.g., Docker Swarm whose API server operate in an imperative and side-effectful manner. Kubernetes API server, on the other hand, is a stateless, side-effect free REST API server in front of some private data storage, such as etcd [78].

The second property is *symmetry* [79], which follows quite easily if the store is side-effect free. If a store is symmetric, it contains all the necessary state to

express the target system; there is no "hidden" or "internal" state, i.e., the state is symmetrically shared between the store and its user. An asymmetric system typically contains critical portions of target system state private to the store and its related "system components", which makes it impossible for the user to see, verify, change or backup the complete state. Kubernetes is symmetric and thus, *readily extensible* by its users [8]. Docker Swarm, however, is asymmetric.

Finally, the store may be *versioned*. Versioned stores can be beneficial as users can easily roll state back to previous points in time and audit state evolution. In general, the Kubernetes API server is not versioned, although the underlying storage etcd is [80]. However, workarounds exist for common cases, e.g., to allow rollbacks of workload upgrades. Auditing [81] is supported as an alternative mechanism to following state evolution. Swarm is not versioned. However, Git is an example of a versioned state store that provide a clear ways of tracing state evolution, through commits.

4 Kubernetes Operators

The Industrial Revolution during the 18th and 19th centuries marks one of history's many significant turning points. The Industrial Revolution is characterized by the transition from resource production by hand to machine-produced resources. With the advent of the machine, did the need for humans in factories disappear? Certainly not. Conversely, it did allow for a massive increase in efficiency, production volume and factory employment.

For a company to be successful in taking advantage of the benefits the industrial revolution brings, however, they need at least capital to buy the machines and employees skilled at operating the machines. If the production volume or more generally scale is small, the cost of the machine and operator salaries might outweigh the benefits of the increase in efficiency.

This is analogous to server systems, as well. If the scale is small, it might not, on capitalistic terms at least, be worth to invest in refined server automation and employee education. But the larger the scale, the more likely it gets that sophisticated server automation is beneficial or even the only choice. This is always worth remembering when evaluating deploying a system like Kubernetes. However, operating at a small scale does not free you from the underlying dynamics discussed in Chapter 2, e.g., constant entropy increase, randomness and finite speed of information transfer. Thus, it is worth thinking twice before disregarding automation designed to cope with these dynamics.

The shift from relying on humans to manage servers "by hand" to *designing automation whose purpose is to manage servers* is analogous to the Industrial Revolution. And yet, the need for humans does not disappear although this automation is utilized. It just takes a different, more efficient form, which allows for a higher human-to-server ratio (or some similar, more appropriate measure). The role of humans is now to create, oversee and maintain the automation, just as the role of humans in a factory of today's standards is to make sure the machines keep producing and perform maintenance. *Site Reliability Engineer* is the job title popularized by Google for a person operating this kind of server automation [45].

A *Kubernetes Operator*, coined by Brandon Philips at CoreOS in 2017 [19], exactly represents this kind of server automation [22]. A Kubernetes operator, or just *operator* for short, codifies domain-specific knowledge about how to operate a given application in production. The "application" mentioned here can be any process, server, API or resource that requires custom handling during its lifecycle. This pushes the boundary of human limits forward, by delegating "repetitive human activities that are devoid of lasting value" [22] to the operator.

The topics in Chapter 2 and 3 are not random, they cover the challenges to be dealt with and the foundational design principles of Kubernetes operators. Due to space constraints, there is no room for a detailed walkthough on *how* these relate. However, the end result of combining declarative configuration, the state reconciler model, separation of desired and actual state, control through choreography, and the other discussed topics, is a novel *programming model* for the next generation of server systems.

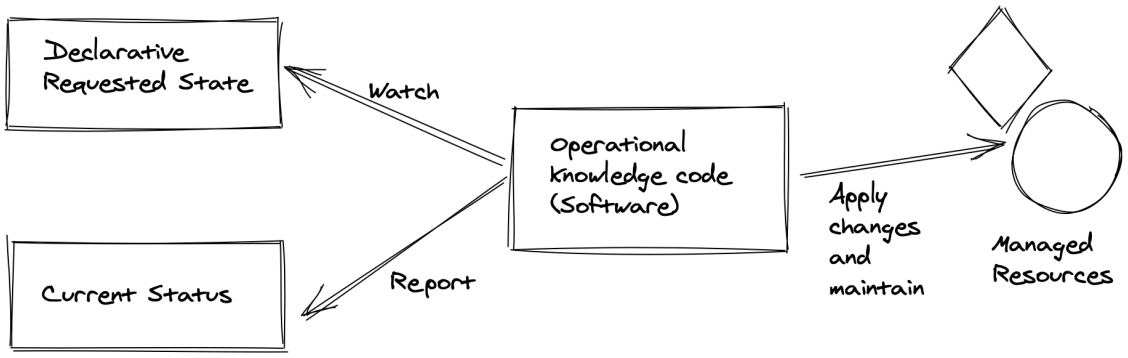

Figure 10: CNCF WG App Delivery. (2021). Operator pattern [22]. Accessed 2021-09-24. Retrieved from https://github.com/cncf/tag-app-delivery/blob/49271fe4d63b5330150c96529b1ef8ad4dc8fa20/operator-wg/whitepaper/Operator-WhitePaper_v1-0.md. Licensed under Apache 2.0.
Operators extend the generic reconciler model depicted in Figure 7.

4.1 Maturity Levels

Depending on the application needs, Kubernetes operators can be implemented in different degrees of sophistication. What applications the operator should deploy (and how) is configured through a user-friendly, application-specific, declarative, and often quite high-level API. In Kubernetes, this mechanism is called *custom resources* [82]. One operator maturity model [83] (e.g., level 3 includes everything in 1 and 2) defines the following levels:

1. Basic Install: The desired application is provisioned and configured through a high-level, app-tailored operator configuration API. Also referred to as a "compiler" as it "compiles" high-level API resources into lower-level ones.

2. Seamless Upgrades: Safe version up- and (possibly) downgrades, often with automatic rollback, are supported through its declarative API.

3. Full Lifecycle: All stages of the application's lifecycle are managed, e.g., clustering, backups and restores, storage handling, failure recovery or migration.

4. Deep Insights: Application telemetry data such as metrics and logs is processed, and critical alerts and warnings are sent to humans with a high signal-to-noise ratio (SNR).

5. Autopilot: The application is horizontally and vertically scaled with demand, configuration is tuned at runtime, and abnormal behavior is detected.

4.2 Use-cases

High-level objectives for a Kubernetes operator thus are to:

- keep server infrastructure in control, continuously minimizing drift between the desired and actual state (thus, operators are a superset of "plain" reconcilers),
- codify and automate "repetitive human activities that are devoid of lasting value" [22], by encoding domain-specific knowledge (*resource scalability*),
- observe application health, metrics and logs, such that configuration can be adaptively tuned and alerts of any abnormal behavior can be sent seldom but with high importance (*monitoring scalability*, to avoid SRE burnout), and
- provide a high-level abstraction interface such that the application can be operated by engineers without the domain-specific knowledge otherwise required (*knowledge scalability*).

Use-cases of operators include:

- application data backups, e.g., mariadb-operator [84], K8up [85], and etcd-operator [86],
- application up- or downgrades, or clustering, e.g., the etcd-operator [86], mariadb-operator [84], and minio-operator [87],
- adaptive configuration tuning, e.g., Horizontal [88] or Vertical [89] Pod Autoscaling, and prometheus-operator [90],
- provide a pluggable abstraction layer, e.g., Cluster API [91] and Istio [92],
- server operating system administration, e.g., Machine Config Operator [93],
- advanced deployment rollout techniques, e.g., Flagger [94],
- GitOps workflows, e.g., Flux [95] and Argo [96].

4.3 Example: GitOps

GitOps has evolved as an industry best-practice during recent years [97] after the coining blog post from Alexis Richardson at Weaveworks [98] in 2017. Unlike traditional mechanisms for delivering software (industry term: *continuous delivery*) that tend to be of imperative nature [99] ($algorithm = control$), GitOps provides for a fully declarative ($algorithm = control + logic$ [38], see Section 2.3) delivery flow.

GitOps is the practice of storing declarative desired state in an *immutable, versioned and auditable* storage (see Section 3.7), combined with automated software agents that continuously observe the actual state and attempt to apply the most recently pulled desired state [100]. Notably, despite its name and popular usage, GitOps does not mandate a specific source nor target. That said, the most popular source is Git and delivery target Kubernetes, through the Flux [95] and Argo [96] software agents.

On a high-level, the only difference between a GitOps software agent and a reconciler in Section 2.9, is that GitOps requires an immutable, versioned and

auditable desired state store, unlike the reconciler. Flux and Argo are Kubernetes operators that whose target system is, in fact, Kubernetes itself. In other words, they keep Kubernetes in desired control through syncing configuration from Git to the Kubernetes API server.

5 Discussion

Kubernetes has allowed organizations to scale to unpredecented levels [9]–[14] through encoding human-like operational knowledge with regards to operating and scheduling containerized workloads. With Kubernetes, users can operate on a higher abstraction layer, and focus on their application needs, not on underlying server-related details.

The Kubernetes operator *programming model* has allowed subject matter experts (that have authored, e.g., [6]–[8]) to *encapsulate knowledge* in a way that lets tens of thousands of Kubernetes users "stand on the shoulders of giants" (*knowledge scalability*). Furthermore, the declarative abstraction layer allows for optimizations in the implementation (e.g., scheduling), leading to *resource scalability*. Finally, through continuous reconciliation "repairing" the system as often as possible, fewer error situations need human intervention, leading to *monitoring scalability*.

While most people associate Kubernetes with containers, Kubernetes can be fundamentally seen as an extensible chaos management engine, helping humans in the race against inevitable entropy and randomness. Kubernetes' API machinery [21], [101] (whose foundations are covered in Chapter 3 and [102]) and its framework for building operators are the true innovations [103]. The patterns built into these form the backbone of the cloud native patterns, and could apply more widely beyond "just" server management. This is prototyped in, e.g., kcp [104] and libgitops [105].

Let us recap the key features of the Kubernetes operator model.

1. **Explicitly driven by user intent:** The users focus on what the end state should look like, freeing them from thinking about state diagrams or internal details, leading to higher development productivity and code re-use.

2. **Counters unwanted inevitable state fluctuations:** Operators reconcile periodically, even during partial failures, correcting emergent changes caused by, e.g., entropy and randomness, as well as user updates to the desired state.

3. **Readily extensible and transparent:** The *complete* system state is split up into loosely coupled resources, usually paired with a reconciler loop. Custom resources [82] can extend the system, and resources build upon each other.

4. **APIs are versioned and self-describing "plain" config**: API versioning provides a clear evolution path. Self-describing APIs allow for seamless data exchange. Separation of data and generation provide for relatively simple decoding yet a rich user experience ecosystem.

However, the Kubernetes model also has shortcomings:

1. **A steep learning curve:** An influential factor to Kubernetes being perceived as "too complex" is that it introduces a lot of new terminology and concepts to learn, leading to cognitive overload [106]–[108].

2. **"The Tyranny of Choice" [109]:** Kubernetes as a fully transparent and extensible system, not including critical aspects of its operation by default (e.g.,

isolation, networking and storage subsystems) and exposing all knobs needed for operation, can negatively affect perception and satisfaction [110].

3. **Hard to debug "too many moving parts":** Although the *control through choreography* is beneficial for, e.g., minimizing MTTR and scaling the system, it can be non-trivial to debug many loosely coupled resources reconciling in parallel [111]. In other words, it is a problem of tracability. It is even hard to understand what happens behind the scenes when a single workload is run, warranting long explanations like [112].

4. **Dependencies are implicit, not explicit:** "Kubernetes doesn't know the difference between 'the system has converged successfully' and 'a control loop is wedged and is blocking everything else.'" [111]. In other words, it is not straightforward to comprehend how state will evolve without a defined execution order.

5. **Configuration is still an unsolved problem:** In 2016, Burns et al. [8] concluded that configuration is an unsolved problem, and that still holds today [113]–[116]. Of over 120 solutions tools in [113], there has not yet been any convergence to *de facto* best-practice solutions [117].

6. **Lack of explicit field ownership:** As multiple actors can operate on overlapping sets of resources, they can over time become "fighting" by flipping between mutually exclusive states without any *errors being reported* [111]. Field ownership is also needed to avoid that several reconcilers overwriting each others' data when patching resources [118].

7. **It is non-trivial for humans to write JSON and YAML:** When humans write JSON or YAML, it is easy to make mistakes in the syntax (e.g., spacing or colons), misspell field names, add duplicated fields, forget quotation marks, or otherwise craft an invalid config [117]. Without good linters, schema validators, encoders and decoders, an invalid config can travel quite a long way in the system before an error is caught. Furthermore, it is also hard to compute and apply differences of two objects.

However, Kubernetes is *not* designed to be used directly "by hand". It is a "platform for building platforms" [119], meaning it operates at a high enough level to be a useful, but low enough level to "work anywhere". Thus, problems 1, 2, 7, and maybe 5 are purposefully meant to be solved by something built at a higher layer, providing pre-set *opinions* and a tailored, more focused "view" of the system. The community is trying to address problem 3 and partially 4 through, e.g., kspan [120] and kstatus [63]. Problem 6 is being addressed, in part, by Server Side Apply [118], [121].

Over time, the foundational could native patterns described in this thesis will hopefully be employed even more concretely outside of the "obvious" server system domain as well. For instance, Kubernetes operators that do not use any of Kubernetes is a sign of success of the pattern. In other words, as the underlying physical dynamics described and the reconciler solution are quite generic, reconciliation applications (like GitOps) could extend way beyond container orchestration.

6 Conclusion

With Kubernetes operators, cumbersome and repetitive server-related work items can be taken care of autonomously without human intervention. This allows for scaling systems beyond human cognitive limits in many dimensions, analogously to how the Industrial Revolution in the 19^{th} century allowed production to scale beyond the rate of production by hand. There are multiple dimensions of scaling systems, including, but not limited to, *resource, knowledge and monitoring scalability.*

Resource scalability can be achieved through operators computing as optimal decisions for actions as possible, through inspecting both the user-defined desired state, and the actual state of the system. Knowledge scalability can be achieved through the operator defining a declarative API for users, both humans and higher-level automation, to utilize for controlling the underlying system. Monitoring scalability can be achieved through the operator periodically inspecting the actual state, taking action for minor "known" deviation or error conditions, and reporting to humans only for unexpected states.

At small scale of server infrastructure systems, the impact of the inevitable entropy increases, randomness, Hyrum's Law and finite speed of information transfer might be hard to notice and thus easy to overlook in system design. Learning from the experience and "lessons learned" from engineers running large-scale systems can hence be an effective way of designing systems reliably from their inception. Finally, once a critical mass of large-scale engineers have come to similar conclusions, a set of *patterns and practices* emerge fundamentally requiring a new *mindset* to effectively cope with problems faced.

The Kubernetes operator patterns have indeed emerged as a consequence of the underlying dynamics of server systems noticed at large scale. Compared to alternatives, the Kubernetes philosophy can be summarized as "control through choreography", rather than more monolithic management found in, e.g., Swarm and Terraform. Declarative configuration can be understood without execution, can be compared, is side-effect free, and naturally forms an abstraction layer. Periodic, pull-based reconciler loops form an effective control mechanism for systems naturally and randomly evolving to more chaotic states. Reconciler bidirectionality and level-triggeredness provide for more reliable operation. The principles of separation of desired and actual state, and of configuration data and its generation provide for "best-practice" designs.

6.1 Future Work

Currently, operators are mostly written in Go and targeted at Kubernetes / server-related environments. Expanding this operator control mechanism further to other programming languages and environments would provide benefits to other scenarios as well, just as *control theory* generally has provided benefits to many kinds of automation. Likewise, expanding the GitOps concept further beyond Git and Kubernetes is an interesting area to follow. This, however, requires more education on the "fundamentals" of cloud native and the operator pattern for the implementers.

Still, there is much work to be done in how to cope with configuration complexity of declarative files when stored in, e.g., Git, Kubernetes and generally in files. Having an even more *unified* API to encode, decode, list, diff, patch, and up- and downgrade declarative configuration stored in pluggable storages would help with that problem [105].

Finally, as the "control through choreography" pattern comes with a relatively high understandability cost, there is a need for better tracability and visualization of automated operations being carried out. For instance, kspan [120] uses the OpenTelemetry [62] specification to transform Kubernetes events into *traces*, tracable timelines of multiple correlated actors and their actions executed over time.

References

[1] CNCF Technical Oversight Committee, *CNCF cloud native definition v1.0*, Sep. 23, 2021. [Online]. Available: https://github.com/cncf/toc/blob/c0bee0289989c3dc4b5105755859d6564d02a075/DEFINITION.md (visited on 09/23/2021).

[2] CNCF Executive Staff. "New Cloud Native Computing Foundation to drive alignment among container technologies," [Online]. Available: https://www.cncf.io/announcements/2015/06/21/new-cloud-native-computing-foundation-to-drive-alignment-among-container-technologies/ (accessed: 12/01/2021).

[3] ——, "CNCF Annual Report 2020," [Online]. Available: https://www.cncf.io/cncf-annual-report-2020/ (accessed: 10/14/2021).

[4] The Kubernetes Authors. "kubernetes.io: Production-Grade Container Orchestration," [Online]. Available: https://kubernetes.io/ (accessed: 10/08/2021).

[5] C. Aniszczyk. "Update on CNCF and Open Source Project Velocity 2020," [Online]. Available: https://www.cncf.io/blog/2021/08/02/update-on-cncf-and-open-source-project-velocity-2020/ (accessed: 10/08/2021).

[6] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, Large-scale cluster management at Google with Borg, in *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015. [Online]. Available: https://research.google/pubs/pub43438.pdf (visited on 10/11/2021).

[7] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, Omega: flexible, scalable schedulers for large compute clusters, in *SIGOPS European Conference on Computer Systems (EuroSys)*, Prague, Czech Republic, 2013, pp. 351–364. [Online]. Available: http://eurosys2013.tudos.org/wp-content/uploads/2013/paper/Schwarzkopf.pdf (visited on 09/29/2021).

[8] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *ACM Queue*, vol. 14, no. 1, Mar. 2, 2016. DOI: 10.1145/2898442.2898444. [Online]. Available: https://dl.acm.org/doi/10.1145/2898442.2898444 (visited on 09/23/2021).

[9] The New Stack. "Apple Plans to Run Most of Its 'Compute Management' on Kubernetes," [Online]. Available: https://thenewstack.io/apple-plans-to-run-most-of-its-compute-management-on-kubernetes/ (accessed: 12/01/2021).

[10] The Kubernetes Authors. "CERN Case Study," [Online]. Available: https://kubernetes.io/case-studies/cern/ (accessed: 12/01/2021).

[11] ——, "JD.com Case Study," [Online]. Available: https://kubernetes.io/case-studies/jd-com/ (accessed: 12/01/2021).

[12] ——, "Spotify Case Study," [Online]. Available: https://kubernetes.io/case-studies/spotify/ (accessed: 12/01/2021).

[13] Vuk Gojnic, *KubeCon Keynote: How Deutsche Telekom Technik Built Das Schiff for Sailing the Cloud Native Seas.* (2021), Accessed: 10/14/2021, [Online Video]. Available: https://www.youtube.com/watch?v=s0UKWiNNFTM.

[14] The Kubernetes Authors. "Nokia Case Study," [Online]. Available: https://kubernetes.io/case-studies/nokia/ (accessed: 12/01/2021).

[15] P. Kornstädt. "Deutsche Telekom rolls out 5G on Kubernetes with Weaveworks," [Online]. Available: https://www.telekom.com/en/media/media-information/archive/deutsche-telekom-rolls-out-5g-on-kubernetes-with-weaveworks-641936 (accessed: 12/15/2021).

[16] H. Mfula, A. Ylä-Jääski, and J. K. Nurminen, "Seamless kubernetes cluster management in multi-cloud and edge 5g applications: International conference on high performance computing & simulation," *International Conference on High Performance Computing & Simulation (HPCS 2020)*, Aug. 2021.

[17] Docker Inc. "Swarm mode overview," [Online]. Available: https://docs.docker.com/engine/swarm/ (accessed: 12/15/2021).

[18] HashiCorp. "Terraform," [Online]. Available: https://www.terraform.io/ (accessed: 10/29/2021).

[19] B. Philips. "Introducing Operators: Putting Operational Knowledge into Software," [Online]. Available: https://web.archive.org/web/20170129131616/https://coreos.com/blog/introducing-operators.html (accessed: 09/24/2021).

[20] B. Beyer, N. R. Murphy, D. K. Rensin, K. Kawahara, and S. Thorne, Eds., *The Site Reliability Workbook*, O'Reilly Media, Inc., 2018, ISBN: 978-1-4920-2950-2. [Online]. Available: https://sre.google/workbook/table-of-contents/ (visited on 11/24/2021).

[21] The Kubernetes Authors, *Kubernetes API conventions*, Oct. 12, 2021. [Online]. Available: https://github.com/kubernetes/community/blob/74ae2f97f7f57949505d4636b4b037ec31079e97/contributors/devel/sig-architecture/api-conventions.md (visited on 10/12/2021).

[22] CNCF WG App Delivery. "Operator White Paper," [Online]. Available: https://github.com/cncf/tag-app-delivery/blob/49271fe4d63b5330150c96529b1ef8a0operator-wg/whitepaper/Operator-WhitePaper_v1-0.md (accessed: 09/24/2021).

[23] B. Burns, J. Beda, and K. Hightower, *Kubernetes: Up and Running*, 2nd edition. O'Reilly Media, Inc., 2019, ISBN: 978-1-4920-4653-0. (visited on 09/23/2021).

[24] M. Burgess. "Deconstructing the 'CAP theorem' for CM and DevOps: Part 2: The greatest distributed system of them all," [Online]. Available: http://markburgess.org/blog_cap2.html (accessed: 10/22/2021).

[25] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, Jul. 1948, Conference Name: The Bell System Technical Journal, ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1948.tb01338.x. (visited on 11/10/2021).

[26] E. Sorenson. Portland, OR, USA, *Cloud Native Configuration Management 2020 and beyond*. (2019), Accessed: 11/11/2021, [Online Video]. Available: https://www.youtube.com/watch?v=GMMPBZIrqqE Presentation slides: https://speakerdeck.com/ahpook/cloud-native-configuration-management.

[27] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, Feb. 1, 2013, ISSN: 0001-0782. DOI: 10.1145/2408776.2408794. [Online]. Available: https://doi.org/10.1145/2408776.2408794 (visited on 11/11/2021).

[28] P. Garraghan, X. Ouyang, P. Townend, and J. Xu, Timely Long Tail Identification through Agent Based Monitoring and Analytics, in *2015 IEEE 18th International Symposium on Real-Time Distributed Computing*, ISSN: 2375-5261, Apr. 2015, pp. 19–26. DOI: 10.1109/ISORC.2015.39.

[29] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software, in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, Bergamo Italy: ACM, Aug. 30, 2015, pp. 307–319, ISBN: 978-1-4503-3675-8. DOI: 10.1145/2786805.2786852. [Online]. Available: https://dl.acm.org/doi/10.1145/2786805.2786852 (visited on 10/29/2021).

[30] R. C. Dorf and R. H. Bishop, *Modern Control Systems*, 13th ed. Harlow, England: Pearson, 2017, ISBN: 978-1-292-15297-4. [Online]. Available: https://primo.aalto.fi/permalink/358AALTO_INST/1g8mond/alma998090244406526 (visited on 11/22/2021).

[31] P. Laplante and S. Ovaska, *Real-Time Systems Design and Analysis: Tools for the Practitioner*. Wiley, 2011, ISBN: 978-0-470-76864-8. [Online]. Available: https://books.google.fi/books?id=Ez6-aSfbqtsC.

[32] CNCF Events Team. "KubeCon + CloudNativeCon North America | Linux Foundation Events," [Online]. Available: https://events.linuxfoundation.org/kubecon-cloudnativecon-north-america/ (accessed: 10/22/2021).

[33] T. Hockin. "Kubernetes: What is "reconciliation"?" [Online]. Available: https://speakerdeck.com/thockin/kubernetes-what-is-reconciliation (accessed: 09/29/2021).

[34] N. N. Taleb, *The black swan: The impact of the highly improbable*, 2nd ed. Penguin Books Ltd, 2010, ISBN: 978-0-14-103459-1. [Online]. Available: https://www.amazon.com/Black-Swan-Impact-Highly-Improbable-ebook/dp/B002RI99IM.

[35] The Merriam-Webster Dictionary. "Declarative: Definition," [Online]. Available: https://www.merriam-webster.com/dictionary/declarative (accessed: 09/21/2021).

[36] ——, "Declare: Definition," [Online]. Available: https://www.merriam-webster.com/dictionary/declare (accessed: 10/12/2021).

[37] J. W. Lloyd, "Practical advantages of declarative programming," *Joint Conference on Declarative Programming*, pp. 3–17, 1994. [Online]. Available: http://scholar.googleusercontent.com/scholar?q=cache:gf5E3N94C_8J:scholar.google.com/+Practical+Advantages+of+Declarative+Programming&hl=sv&as_sdt=0,5 (visited on 09/21/2021).

[38] R. Kowalski, "Algorithm = logic + control," *Communications of the ACM*, vol. 22, no. 7, pp. 424–436, Jul. 1, 1979, ISSN: 0001-0782. DOI: 10.1145/359131.359136. [Online]. Available: https://doi.org/10.1145/359131.359136 (visited on 10/12/2021).

[39] Haskell.org. "Evaluation order and state tokens - HaskellWiki," [Online]. Available: https://wiki.haskell.org/Evaluation_order_and_state_tokens (accessed: 11/03/2021).

[40] R. Rusanu. "On SQL Server boolean operator short-circuit," [Online]. Available: http://rusanu.com/2009/09/13/on-sql-server-boolean-operator-short-circuit/ (accessed: 12/15/2021).

[41] Haskell.org. "Haskell Language," [Online]. Available: https://www.haskell.org/ (accessed: 10/22/2021).

[42] *Systems and software engineering — Vocabulary*, ISO/IEC/IEEE 24765:2017, Sep. 2017.

[43] The Kubernetes Authors. "Components," [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/ (accessed: 12/15/2021).

[44] H. Wright. "Hyrum's Law," [Online]. Available: https://www.hyrumslaw.com/ (accessed: 11/19/2021).

[45] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, Eds., *Site Reliability Engineering*, O'Reilly Media, Inc., 2016, ISBN: 978-1-4919-2912-4. [Online]. Available: https://sre.google/sre-book/table-of-contents/ (visited on 11/24/2021).

[46] The Kubernetes Authors. "Backoffs should have jitter · Issue #87915," [Online]. Available: https://github.com/kubernetes/kubernetes/issues/87915 (accessed: 12/15/2021).

[47] M. McIlroy, E. Pinson, and B. Tague, "UNIX time-sharing system: Forward," *Bell System Technical Journal*, vol. 57, pp. 1899–1904, Jul. 8, 1978. [Online]. Available: http://archive.org/details/bstj57-6-1899 (visited on 11/20/2021).

[48] HashiCorp. "Resource Graph," [Online]. Available: https://www.terraform.io/docs/internals/graph.html (accessed: 12/15/2021).

[49] ——, "Backend Overview," [Online]. Available: https://www.terraform.io/docs/language/settings/backends/index.html (accessed: 11/23/2021).

[50] Docker Inc. "SwarmKit Nomenclature," [Online]. Available: https://github.com/docker/swarmkit/blob/7956265ead9931dcea6357415d75080e95d9e7ed/design/nomenclature.md (accessed: 11/01/2021).

[51] *Unknown analogy author; probably observed during some KubeCon conference gathering from some Kubernetes contributor. Please let me know if you have context on who coined this (quite popular) saying.*

[52] The Kubernetes Authors. "API Concepts," [Online]. Available: https://kubernetes.io/docs/reference/using-api/api-concepts/ (accessed: 11/01/2021).

[53] ——, "KEP-589: Efficient Node Heartbeats," [Online]. Available: https://github.com/kubernetes/enhancements/blob/11a976c74e1358efccf251d4c7611d05ce27f
keps/sig-node/589-efficient-node-heartbeats/README.md (accessed: 11/01/2021).

[54] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," *2014 USENIX Annual Technical Conference*, pp. 305–319, 2014. [Online]. Available: https://www.usenix.org/system/files/conference/atc14/atc14-paper-ongaro.pdf (visited on 10/21/2021).

[55] D. Inc. "Raft consensus in swarm mode," [Online]. Available: https://docs.docker.com/engine/swarm/raft/ (accessed: 11/01/2021).

[56] HashiCorp. "State Locking," [Online]. Available: https://www.terraform.io/docs/language/state/locking.html (accessed: 11/23/2021).

[57] The Kubernetes Authors. "Controller Re-entrancy - The Cluster API Book," [Online]. Available: https://cluster-api.sigs.k8s.io/reviewing#controller-reentrancy (accessed: 11/02/2021).

[58] T. Hockin. "Edge vs. Level triggered logic," [Online]. Available: https://speakerdeck.com/thockin/edge-vs-level-triggered-logic (accessed: 10/13/2021).

[59] G. Arbezzano. "Extend Kubernetes via a shared informer," [Online]. Available: https://www.cncf.io/blog/2019/10/15/extend-kubernetes-via-a-shared-informer/ (accessed: 11/01/2021).

[60] HashiCorp. "Purpose of Terraform State," [Online]. Available: https://www.terraform.io/docs/language/state/purpose.html (accessed: 11/23/2021).

[61] The OpenMetrics Authors. "The OpenMetrics project — Creating a standard for exposing metrics data," [Online]. Available: https://openmetrics.io/ (accessed: 12/15/2021).

[62] The OpenTelemetry Authors. "OpenTelemetry," [Online]. Available: https://opentelemetry.io/ (accessed: 12/16/2021).

[63] The Kubernetes Authors. "kstatus specification," [Online]. Available: https://github.com/kubernetes-sigs/cli-utils/tree/4c8a09e14248d6a479d535f3f92d63efa pkg/kstatus (accessed: 11/08/2021).

[64] M. Hadlow. "The Configuration Complexity Clock," [Online]. Available: https://mikehadlow.blogspot.com/2012/05/configuration-complexity-clock.html (accessed: 10/29/2021).

[65] *The JavaScript Object Notation (JSON) Data Interchange Format*, IETF RFC 2610, Dec. 2017, [Online]. Available: https://rfc-editor.org/rfc/rfc8259.txt.

[66] YAML Language Development Team. "YAML Ain't Markup Language (YAML™) revision 1.2.2," [Online]. Available: https://yaml.org/spec/1.2.2/ (accessed: 12/16/2021).

[67] T. Preston-Werner and P. Gedam. "TOML: English v1.0.0," [Online]. Available: https://toml.io/en/v1.0.0 (accessed: 12/16/2021).

[68] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. "Extensible Markup Language (XML) 1.0 (Fifth Edition)," [Online]. Available: https://www.w3.org/TR/xml/ (accessed: 12/16/2021).

[69] Google. "Protocol Buffers - Google's data interchange format," [Online]. Available: https://github.com/protocolbuffers/protobuf (accessed: 12/16/2021).

[70] The Kubernetes Authors. "Protobuf serialization and internal storage," [Online]. Available: https://github.com/kubernetes/community/blob/b9266ed1a2365d4f83b559c521a2b6ee73bbe20f/contributors/design-proposals/api-machinery/protobuf.md (accessed: 10/29/2021).

[71] HashiCorp. "The HCL specification," [Online]. Available: https://github.com/hashicorp/hcl/blob/e84201c45df4fce4e9dfaba9e8aaa8730d24dd25/spec.md (accessed: 10/29/2021).

[72] I. Bokharouss, "GCL viewer: A study in improving the understanding of GCL programs," Master's Thesis in Computer Engineering, Eindhoven University of Technology, Eindhoven, 2008. [Online]. Available: https://pure.tue.nl/ws/portalfiles/portal/46927079/638953-1.pdf (visited on 10/21/2021).

[73] The CUE Authors. "CUE," [Online]. Available: https://cuelang.org/ (accessed: 10/29/2021).

[74] G. Roelofs. "PNG Specification: Rationale," [Online]. Available: http://www.libpng.org/pub/png/spec/1.0/PNG-Rationale.html#R.PNG-file-signature (accessed: 12/16/2021).

[75] D. H. Hansson. "Ruby on Rails Doctrine," [Online]. Available: https://rubyonrails.org/doctrine/ (accessed: 11/26/2021).

[76] The Kubernetes Authors. "Create a k8s.io/component-base repo," [Online]. Available: https://github.com/kubernetes/enhancements/tree/aa523b838782a14ca9de82d7f727627c4b1be17e/keps/sig-cluster-lifecycle/wgs/783-component-base#related-proposals--references (accessed: 11/24/2021).

[77] L. Käldström. Shanghai, China, *Configuring Your Kubernetes Cluster on the Next Level.* (Nov. 14, 2018), Accessed: 11/24/2021, [Online Video]. Available: https://www.youtube.com/watch?v=klHBzISZkCw Presentation slides: https://speakerdeck.com/luxas/configuring-your-kubernetes-cluster-on-the-next-level.

[78] The etcd Authors. "etcd," [Online]. Available: https://etcd.io/ (accessed: 10/11/2021).

[79] *A term and concept coined by Clayton Coleman; did not find any online references from him. Please let me know if there are some resource I can refer to.*

[80] M. Gasch. "Onwards to the Core: etcd," [Online]. Available: https://www.mgasch.com/2021/01/listwatch-part-1/ (accessed: 12/13/2021).

[81] The Kubernetes Authors. "Auditing," [Online]. Available: https://kubernetes.io/docs/tasks/debug-application-cluster/audit/ (accessed: 12/01/2021).

[82] ——, "Custom Resources," [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/ (accessed: 11/30/2021).

[83] Red Hat Inc. "Understanding Operators," [Online]. Available: https://docs.openshift.com/container-platform/4.1/applications/operators/olm-what-operators-are.html (accessed: 12/01/2021).

[84] M. Dhanorkar. "mariadb operator," [Online]. Available: https://operatorhub.io/operator/mariadb-operator-app (accessed: 11/02/2021).

[85] VSHN. "K8up - Kubernetes Backup Operator," [Online]. Available: https://k8up.io/k8up/2.0/index.html (accessed: 11/02/2021).

[86] The etcd Authors. "etcd-operator," [Online]. Available: https://operatorhub.io/operator/etcd (accessed: 11/02/2021).

[87] MinIO, Inc. "MinIO operator," [Online]. Available: https://operatorhub.io/operator/minio-operator (accessed: 11/02/2021).

[88] The Kubernetes Authors. "Horizontal Pod Autoscaler," [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/ (accessed: 11/02/2021).

[89] ——, "Vertical Pod Autoscaler," [Online]. Available: https://github.com/kubernetes/community/blob/7c6d1d31322e4ca0a15f1d6bbe3320fe19530159/contributors/design-proposals/autoscaling/vertical-pod-autoscaler.md (accessed: 11/02/2021).

[90] The Prometheus operator authors. "Prometheus operator," [Online]. Available: https://operatorhub.io/operator/prometheus (accessed: 11/02/2021).

[91] Kubernetes SIG Cluster Lifecycle. "Kubernetes Cluster API: Introduction," [Online]. Available: https://cluster-api.sigs.k8s.io/ (accessed: 10/14/2021).

[92] The Istio Authors. "Istio," [Online]. Available: https://istio.io/latest/ (accessed: 11/02/2021).

[93] Red Hat Inc. "machine-config-operator," [Online]. Available: https://github.com/openshift/machine-config-operator (accessed: 12/01/2021).

[94] The Flux authors. "Flagger," [Online]. Available: https://docs.flagger.app/ (accessed: 11/02/2021).

[95] ——, "Flux - the GitOps family of projects," [Online]. Available: https://fluxcd.io/ (accessed: 11/02/2021).

[96] The Argo authors. "Argo CD," [Online]. Available: https://argo-cd.readthedocs.io/en/stable/ (accessed: 11/02/2021).

[97] CNCF GitOps Working Group. "About OpenGitOps," [Online]. Available: https://opengitops.dev/about (accessed: 11/03/2021).

[98] A. Richardson. "GitOps - Operations by Pull Request," [Online]. Available: https://www.weave.works/blog/gitops-operations-by-pull-request (accessed: 11/03/2021).

[99] I. Dmitrichenko. "Kubernetes anti-patterns: Let's do GitOps, not CIOps!" [Online]. Available: https://www.weave.works/blog/kubernetes-anti-patterns-let-s-do-gitops-not-ciops (accessed: 12/16/2021).

[100] CNCF GitOps Working Group. "GitOps Principles v1.0.0," [Online]. Available: https://github.com/open-gitops/documents/blob/release-v1.0.0/PRINCIPLES.md (accessed: 12/01/2021).

[101] The Kubernetes Authors. "apimachinery," [Online]. Available: https://github.com/kubernetes/apimachinery (accessed: 12/03/2021).

[102] M. Hausenblas and S. Schimanski, *Programming Kubernetes*. O'Reilly Media, Inc., 2019. (visited on 09/23/2021).

[103] The Kubernetes Authors. "Introduction - The Kubebuilder Book," [Online]. Available: https://book.kubebuilder.io/ (accessed: 12/03/2021).

[104] The kcp Authors. "kcp," [Online]. Available: https://github.com/kcp-dev/kcp (accessed: 12/03/2021).

[105] L. Käldström. "Introducing libgitops: A programming model and Go runtime for any GitOps automation," [Online]. Available: https://github.com/luxas/cfps/blob/main/2021/gitopscon/README.md (accessed: 12/17/2021).

[106] T. St. Clair. "For folks that view #Kubernetes as complicated," [Online]. Available: https://twitter.com/timothysc/status/1451236903672590340 (accessed: 12/02/2021).

[107] J. Beda. "Re: "Kubernetes is too complex"," [Online]. Available: https://twitter.com/jbeda/status/993978918196531200 (accessed: 12/03/2021).

[108] J. Moiron. "Is K8s Too Complicated?" [Online]. Available: http://jmoiron.net/blog/is-k8s-too-complicated/ (accessed: 12/03/2021).

[109] A. Roets, B. Schwartz, and Y. Guan, "The tyranny of choice: A cross-cultural investigation of maximizing-satisficing effects on well-being," *Judgment and Decision Making*, vol. 7, no. 6, p. 689, 2012. [Online]. Available: http://works.swarthmore.edu/cgi/viewcontent.cgi?article=1002&context=fac-psychology.

[110] The Kubernetes Authors. "Extending Kubernetes," [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/ (accessed: 12/03/2021).

[111] D. Anderson. "A better Kubernetes, from the ground up · blog.dave.tf," [Online]. Available: https://blog.dave.tf/post/new-kubernetes/ (accessed: 12/02/2021).

[112] J. Hannaford, *What happens when i type kubectl run?* Dec. 3, 2021. [Online]. Available: https://github.com/jamiehannaford/what-happens-when-k8s (visited on 12/03/2021).

[113] B. Grant. "Declarative application management in Kubernetes," [Online]. Available: https://github.com/kubernetes/community/blob/b9266ed1a2365d4f83b559c5contributors/design-proposals/architecture/declarative-application-management.md (accessed: 10/29/2021).

[114] The Kubernetes Authors. "Issues related to Declarative App Management," [Online]. Available: https://docs.google.com/document/d/e/2PACX-1vSlhRgKFgkAEgWhHyeto3ltgU0c5EOXKSh8mlUj5o1LLhcIXQCK28eBT9A--hM9-vTfTdTG6P8lbhoK/pub (accessed: 10/29/2021).

[115] G. Rushgrove. "Developer tooling for Kubernetes configurations," [Online]. Available: https://speakerdeck.com/garethr/developer-tooling-for-kubernetes-configurations (accessed: 10/29/2021).

[116] E. Sorenson. "Cloud Native Configuration Management," [Online]. Available: https://speakerdeck.com/ahpook/cloud-native-configuration-management (accessed: 10/29/2021).

[117] J. Beda, *Nightmares On Cloud Street.* (Nov. 5, 2020), Accessed: 12/03/2021, [Online Video]. Available: https://www.youtube.com/watch?v=8PpgqEqkQWA.

[118] The Kubernetes Authors. "Server-Side Apply," [Online]. Available: https://kubernetes.io/docs/reference/using-api/server-side-apply/ (accessed: 12/03/2021).

[119] K. Hightower. "Kubernetes is a platform for building platforms. It's a better place to start; not the endgame.," [Online]. Available: https://twitter.com/kelseyhightower/status/935252923721793536 (accessed: 12/03/2021).

[120] Weaveworks. "kspan - Turning Kubernetes Events into spans," [Online]. Available: https://github.com/weaveworks-experiments/kspan (accessed: 12/03/2021).

[121] The Kubernetes Authors. "Structured Merge and Diff," [Online]. Available: https://github.com/kubernetes-sigs/structured-merge-diff (accessed: 12/03/2021).