Lifting Existing Applications to the Cloud: Abstractions, Separation of Responsibilities and Tooling Support

He Huang[1], Zhicheng Zeng[3] and Tu Ouyang[2]
[1]*University of Melbourne, Parkville VIC 3010, Australia*
[2]*Case Western Reserve University, CSDS department, U.S.A.*
[3]*Zilian Tech Inc., Shen Zhen, China*

Keywords: Cloud, Abstraction, DevOps, Software Engineering.

Abstract: The benefits from running applications on the cloud: easy to scale up, low cost from competition of many cloud vendors, and many others, are compelling reasons for more and more applications being developed and/or deployed against the cloud environment. This trend promptes application developers to rethink the structure of their applications, the runtime assumption of the applications, and what are the appropriate input/output abstractions. New generation of applications can be built from the scratch with the recent development of the cloud-native primitives (clo, nd). However there are many existing applications which were previously developed against single-machine environment. Some of them now are needed to be lift to the cloud so as to enjoy the benefits from cloud environment such as computation elasticity. What does a principled process look like to lift such applicaitons to the clcoud? In this paper, we present what we have learnt from helping our customers to lift their existing applications to cloud. We identify the key challenges from common questions being asked in practice, and we present our proposed methodologicl framework, to partially address these key challenges. The solution is comprised of various methods identifing right abstractions for cloud resources, separating the responsibilities between application developer and cloud DevOps, and how to leverage tooling to streamline the whole process. We try to design our methods as much cloud-vendor agnostic as possible. We use the lifting process of one application, a web crawler from our practice, to exemplify various aspects of the proposed methodological framewor.

1 INTRODUCTION

After the launch of AWS, recent fifteen years have seen a wave of moving applications to the cloud. Cloud technologies have evolved, from running applications on virtual machine, to running applications in a lightweight container, to the recent development of "serverless" Functions-as-a-Service paradigm (Akkus et al., 2018; Villamizar et al., 2016) that affects the program design deeply and directly. The serverless paradigm enables developers to write code without the need of planning distributed computation, simply upload the code to the cloud, the cloud does all the job including replicating the code to many machines and running all the code instances simultaneously. The new generation of applications being developed can immediately employ the new cloud primitives from the scratch and enjoy a simple and efficient deployment to the cloud. On the other hand, there are many existing applications that are still key to many organizations that are in the middle of moving things to cloud. It can be a long and costing journey to gradually educate original application developers of cloud knowledge so that they can deploy these applications to the cloud, and sometimes there are needs to rewrite part of the applications. Developing methods to speed up the lift process of these applications while minimizing man-hours asked from original developers can help both the developers and the organizations.

Cloud vendors and open source communities have devised many supporting tools and solutions (azu, nd; pod, nd; pup, nd; doc, nd; ter, ndb; aws, nd). Most of these solutions aim to solve specific challenges in one particular stage of the lifting process but not all. Overall it is still difficult for developers to choose the set of tools to use, and split up the work so to let cloud experts to come in to help.

Learned from our practices, in this study we summarize a few important challenges identified, and present a methodological framework to help practi-

Huang, H., Zeng, Z. and Ouyang, T.
Lifting Existing Applications to the Cloud: Abstractions, Separation of Responsibilities and Tooling Support.
DOI: 10.5220/0011109000003179
In *Proceedings of the 24th International Conference on Enterprise Information Systems (ICEIS 2022) - Volume 2*, pages 601-608
ISBN: 978-989-758-569-2; ISSN: 2184-4992

tioners reason about several important aspects of lifting an application to the cloud. This methodological framework also helps to evaluate and choose wisely the appropriate set of supporting tooling to help the lift process by matching with each applications' characteristics. The identified important challenges are presented in Section 2, and the methodological framework to these challenges, which can help to guide the implementation of artifacts such as requirement checklist, code, and team topologies. The methodological framework is detailed in Section 3.

Rich literature exist where many methods have been described in regard to lifting applications to the cloud, in almost every stage of the lifting process, numerous open-source or commercial tools are available to choose. In this paper, we provide a methodological framework that summarizes valuable practical lessons we learned from lifting several applications to the cloud, we also discuss some practical implementation strategies we adopted, selection criteria to choose tools, and how to efficiently integrate human specialists in the process. Additionally , we survey some of the most popular tools when describing the learned lessons.

For the purpose of clear illustration, we use one application as an example throughout this paper: a web crawler. This web crawler application was originally developed against and previously ran on a single machine. The number of web pages to crawl later increases dramatically, renders the single-machine application incapable to handle the workload. Lifting this application to the cloud, to leverage the parallel computation power there, became clear way to go to the developer. The web crawler application takes a text file as input that contains a list of web page URLs to crawl, the application outputs the content of URLs to individual files whose file system paths are defined by an input parameter to the application. We note that there are certainly many more complex applications than a web crawler. In this position paper, we focus on this simple web crawler example, exploring methodological framework to help lifting more complex applications is a compelling future work.

2 KEY CHALLENGES

Application developers, who initially designed and implement their application without any assumption of cloud environment, when the needs come to migrate such application to the cloud, unavoidably they would have many questions in mind. Some typical questions are: what benefits would come with cloud lifting? How much code needs to be added? How much effort for the developers? Could the application be lifted with only a small amount of adjustments and have most of the business logic untouched? How much must application developers understand the target cloud environment during the lifting process? Should it be the application developer to do the lifting, or should it be someone else who knows the cloud better to do it, is there a systematical guideline to help to define and separate the work?

Stemmed from numerous questions we were asked in practice, we identify several key challenges:

C.1. How to identify what application can be lift to cloud with little to no adjustment work.

C.2. What are appropriate set of building blocks(i.e., primitives) to provide to the application developers, help them to remove the common road blocks in the lifting process? The goal is to encapsulate the application from the "new" cloud runtime and environment as much as possible, resulting in less adjustment work. For an application that meets certain property criteria, the amount of adjustment work could be close zero for the developer, which means the lifting is transparent to the application and all the extra work could be done by specialists.

C.3. How *not* to assign too much extra responsibilities to the application developer? It helps reduce anxiety resulted from uncertainty and also helps to brings additional help to accelerate the lift. Some responsibilities, for instance, understanding cloud environment assumptions, managing cloud resources, monitoring applications running status in the cloud, could ideally be assigned to the domain specialists.

C.4. What are the tooling support to enable automation of the lifting? And importantly, to support overcoming the other challenges aforementioned. Tooling helps to reduce effort and human error in many lifting stages, for example, provisioning cloud resources such as computation container, storage and network, automatic deploying the applications to cloud and scaling up cloud resources appropriately, and monitoring running applications.

3 METHODOLOGICAL FRAMEWORK

We present a methodological framework derived from the author's real-world practices, to help, partly address the challenges listed in previous section.

3.1 Identifying Application Properties through Checklist

This facet is to address challenge C.1. We are interested to figure out a *checklist* to identify such application properties, so as to follow the checklist then provide confident answers to application developer if his/her application falls into one of these easy-to-lift categories defined by our checklist. Certain properties of application would greatly reduce the amount of the adjustment need in corresponding parts.

These application properties and their implications are summarized as a *checklist*, in practical, we examine the application that need to be lifted through this checklist and identify properties that enable certain effortless migration.

- A monotonic application, could have multiple instances scheduled at the same time without the need of explicit coordination. Stated by CALM theorem (Ameloot et al., 2013), the output of application is deterministic and monotonic code can be run free without any need for locking, barriers, commit, consensus, etc.
- An associative and communicative application could be transparent to out-of-order scheduling. This is similar to the map function in map-reduce computing paradigm (Dean and Ghemawat, 2004)
- An idempotent application (Hummer et al., 2013) does not need to be aware of a recovery strategy that performs full restart facing application failure, since running the application multiple times produces the same output.
- An application using file-system based input and output. Such application can be migrated to cloud without modification of I/O part, given abundance in cloud virtual file abstractions.

The web crawler meets all criteria in the checklist, it is monotonic, associative, communicative and idempotent, its I/O model is the simplest file based, all these properties together allow this application to be lift with almost zero effort from the developer under our proposed framework's support.

3.2 Abstractions for Hiding Complexity

This facet is similar to a software engineering goal raised in many engineering contexts, that is *WORA(Write once, run anywhere*), a acronym originally coined for advertising Java and the Java virtual machine. It highlights the benefits brought by elegant separation of the code and the runtime, via right abstractions of the underlying layers thus hiding irrelevant detail from the upper layers. By doing that the upper layer implementation requires little to no modification in case some lower layers implementation changes. The same strategy can apply to lifting existing application to cloud, that is to hide the detail of runtime environment changes from existing applications through a right set of abstractions.

We acknowledge that switching a cloud environment is way more complicated than a language runtime, thus the list of desired abstractions can grow long. Some of the cloud environment elements might not be feasible at all to abstract away. Below we list a few fundamental abstractions that are common to many applications, to tackle challenge C.2:

- Abstractions to cloud runtime environment. Concrete examples include, docker for container (doc, nd) that present a whole virtual hardware and software runtime to the application, and Kubernetes technologies (k8s, nd) that provide a set of frameworks and technologies to orchestrating and scheduling many containers within same cloud environment, they are designed such that most applications do not need to be aware of their existing.
- Input/Output abstractions. Virtual file system, is one of the simplest abstraction that could facilitate an effortless and seamless migration of application, built of file-system-based I/O, to the cloud. To many client libraries, accessing virtual file system is no different from accessing a local file system. Database is another common means as Input/Output, database could be installed on a virtual machine in the cloud and similar access interface is provided to applications. A SQL-supporting database abstraction provided by the cloud, is considered an enabler for a quick lift of existing application to the cloud. In addition, many cloud vendors provide managed services of selected database to simplify database management in cloud, e.g., Microsoft Azure provides Azure SQL Managed Instance for its SQL server that usually requires management by individual enterprise's IT department (azu, nd), solution marketplace are thriving with plenty of cloud-vendor and third-party database migration service options available to choose from (aws, nd; dat, ndb). The I/O abstraction provided by cloud vendors usually come with different SLA(service level agreement) on different properties, most important ones are consistency, latency and availability. Cloud vendors price different in multiple tiers, offering differential combination of SLAs of properties to suite diversified user cases. One example is that Microsoft Azure offers four tiers of share File services, where the most expensive pre-

mium tier guarantees low latency, as well as high throughput, the second most expensive tier implements same throughput guarantee, but not optimized for low latency (mic, nd).

The web crawler application's input/output is based on UNIX-like file system. In practice, the target cloud vendor for this application is Microsoft Azure, we choose to use the Azure shared file service to simulate a local UNIX-like file system to the application, so that there is no modification needed at all in the I/O layer of the application. Docker container (doc, nd) is used to provide the runtime abstraction.

3.3 Separation of Responsibilities via Code

This facet is to reduce the amount of required effort and the anxiety associated with it from the application developers, Software framework and associated processes need to devised to separates as many responsibilities as possible from developers, and route them to cloud specialists who can solve one responsibility efficiently. Several key responsibilities to separate from developers are listed below:

- Separate application scheduling, so that application logic is independent of job scheduling strategies as much as possible
- Separate cloud resource management from the application
- Separate deploy-to-cloud process from application
- Separate application monitoring in Cloud from application

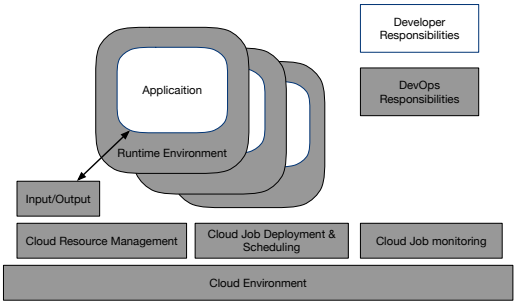
Recent years have seen a new specialist of software engineering workers emerging, namely DevOps (Mueller, 2010). While DevOps is not limited to cloud environment, Such specialist does blooms in recent years with the thriving trend of moving applications to cloud. It is also important enabling force for operating cloud applications efficiently and effectively because now cloud vendor could run such a DevOps organization to handle the needs of individual enterprises. Cloud DevOps' cloud operation expertises render them the best personnels to handle many of the responsibilities we want to separate out from developers, particularly those related to cloud runtime. We suggest achieving the separation with appropriate team design, team process and tooling support. Figure 1a illustrates an overview of many responsibilities that can be handled by different teams to support running an application in a cloud environment. Ideally, application developer only needs to take care of the application's logics, while leaves everything else, from runtime configuration, cloud deployment, to job monitoring to other specialist teams, for example, DevOps.

Operation teams should be treated as essentially engineering team, engineers are trained and best in expressing intended actions in code. Therefore it's preferred to execute process as running a program in stead of human manual steps following instruction in a documentation. We propose a key paradigm for separation of responsibilities: **Responsibility-as-Code**. We apply this paradigm to tackle the challenge C.3.

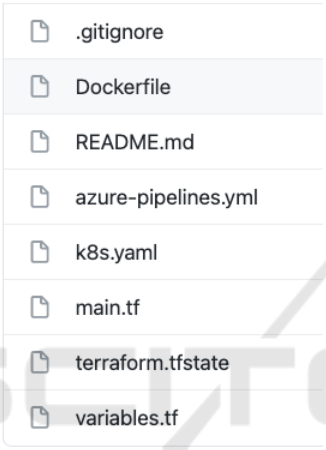
We deem setting up framework for coding up various responsibilities as effective ways to bring in experts from DevOps team who are experts to individual responsibility, code is something that is a familiar vehicle to both ops and application teams, a place to encode and store different expertise knowledge in a precise way, a means where tasks(and responsibilities) can be split up, boundaries can be set, while collaboration between teams still can happen if needed, through established engineering practices such as pull request, code review and branches management.

We find several discussion topics in software engineering literature that are closely related to what we call *responsibility-as-code*: Examples include *infrastructure-as-code* (Hüttermann, 2012; ter, nda; Morris, 2016) in the field of DevOps engineering, and *executable playbooks* (exe, nd) in "chaos engineering" field, Of note many of these concepts discussed in the literature rely on tooling to realize the full potential, that we discuss in section refsec:tooling.

Research and practice suggests code for expressing infrastructure or process are better constructed with a declarative language, tooling could compile the process code into action plans and then execute. (Schwarz et al., 2018; Morris, 2016). Figure 1b shows a list of configuration files for the web crawler application, each of the files aims to express one category of responsibilities for setting up and running the application in a cloud environment. Application developer needs to know little about what are these configuration files for. These files are added later by DevOps specialists. In this example list, *Dockerfile* is for cloud runtime setup with docker container, *k8s.yaml* is the configuration code for orchestrating various cloud resources for running application via Kubernetes. *main.tf*, *variables.tf* and *terraform.tfstates* contain code for cloud resource definitions and management, specified in terraform scripting language. *azure-pipelines.yaml* is to specify the deployment process of the application to the Microsoft Azure cloud These configuration files are created by DevOps and executed by tooling.



(a) The mapping of responsibilities to components for running application instances in the cloud, in ideal case, application developer need to only focus on the application logic and assign other tasks to DevOps.



(b) A sample list of responsibility-as-code files, except *.gitignore* and *README.md*(both for other code repository management intends), each of these code files expresses one particular category of responsibilities, and the code can be executed by corresponding tooling to perform specified tasks.

Figure 1: The conceptual responsibility layout for running applications in the cloud, and *responsibility-as-code* sample files.

3.4 Tooling for Automation

Building toolchains, as stated in challenge C.4, is the most important aspect to save the cost of every application lift, It is also the enabler to realize various concepts in the other proposed methods. We list a few tooling support categories we consider as the most essential. In every of these category we have seen many offerings from both open source communities and commercial companies, some of the solutions are opinionated. It is challenging to compare these offerings' pros and cons, and choose the ones that fit the need.

- Tooling for automated creation of specific runtime environment for the application. An ideal runtime environment should provider an higher-level abstraction to specific underlying hardwares and the operating system. We choose docker (doc, nd) as runtime packaging system in our practice. The configuration files manifesting the runtime parameters using docker, for the web crawler application, are shown in figure 1b. Other alternative tooling include Podman (pod, nd), a daemonless container engine running on Linux; and LXC (lxc, nd) that is implemented within Linux facilitating an operating-system-level virtualization.
- Tooling for cloud resource definition and management. One example of such tools is puppet (pup, nd), it includes its own declarative language to describe system configuration and it actually supports systems beyond cloud. Chef (che, nd) uses a pure-Ruby, domain-specific language (DSL) for writing system configuration "recipes", it can streamline tasks in company servers, as well as clouds like AWS, Microsoft Azure, IBM cloud. SaltStack is another open source option that provides configuration management in its automation engine (sal, nd). Config files shown in figure 1b include terraform config files that terraform tooling understand and execute to manage the cloud resources for web crawler application. Terraform (ter, ndb) is a software tool that allows users to define and configure cloud resources using a high level script language, which in turn can be executed to run management routines of these resources against all popular cloud vendors, One useful feature of Terraform is that it provides a layer of resource abstraction to hide as much as possible the differences between cloud vendors.
- Tooling for observability, for monitoring application status running in the cloud, , and managing alerts when something needs attention. In addition to native tooling provided by almost every cloud vendor, there are commercial products available covering across cloud vendors from Datadog (dat, nda), and Lightstep (lig, nd), using a vendor-agnostic tooling is useful if the application targets multi-cloud deployments.
- Tooling for incident response, to provide frameworks and templates to standardize generic incident mitigation and simplify producing executable playbook code to cope with specific application incidents. (ser, nd; sta, nd) are example products for this category.

4 RELATED WORK

Infrastructure-as-code (IaC) is the practice to automatically configure the system and to provision resources on remote instances (Hüttermann, 2012; Morris, 2016). It has become a highly advocated practice for day-to-day work for DevOps (Artac et al., 2017) since it allows system configuration tasks to be executed repeatably, safely, and efficiently. Some research on IaC focuses on identifing and veriing the properties of IaC code, such as idempotency and convergence, as well as others, to detect possibly harmful effects in the code that might lead to serious consequences (Van der Bent et al., 2018; Rahman and Williams, 2018). Hummer et al. (Hummer et al., 2013) applied model-driven testing techniques to Chef configuration cookbooks, and were able to find some of the non-idempotent ones. Rehearsal (Shambaugh et al., 2016), a configuration verification tool for Puppet, uses static analysis to verify determinism and idempotency. We believe many proposed methods from Infrastructure-as-Code research can be applied to our proposed Responsibilities-as-Code paradigm to guide the code design and verification.

The "Separation of concerns" principle is one of the essential principles in software engineering. It states software should be decomposed in such a way that well-separated modules address different "concerns" or aspects of the problem. (Hürsch and Lopes, 1995) Concern can be "the details of the hardware for an application", or "elements to present the information on the user interface". Many research has explored ways to apply this principle in constructing software systems, through code patterns (Gamma et al., 1995; Sarcar, 2016), programming paradigms (Stutterheim et al., 2017; Kiczales and Mezini, 2005), development process (Panunzio and Vardanega, 2014; Castellanos Ardila and Gallina, 2020), and so on. Win et al, state the importance of applying this principle to design and implement secure software (De Win et al., 2002). The *concern* concept in SoC overlaps with *responsibility* used in this paper, though in our context, responsibility also implies human factor: not only does software need to be decoupled to be addressed as independently as possibly, but also decoupled software aspects should be grouped such that different aspect groups can be handled by differen human specialists as independently as possible, to maximize the payoff of human expertise.

5 CONCLUSION AND DISCUSSION

We provide several important lift-to-cloud challenges and propose a methodological framework that consists of cloud-vendor-agnostic reasoning and methods to help addressing the challenges. Our hope is to shield lights into what are the common road-blocks in the lifting process and What are the systematical way to reason about how to overcome these road-blocks. Hopefully what is presented here can foster the related discussion and help the community to devise new methods and tools to accelerate the lifting process of many of existing applications.

There are more complex applications, than the web crawler example used throughout this study, which we believe will benefit from a similar systematical methodological framework in the process of lifting-to-cloud. Many of the reasoning aspects presented in our methodological framework are mostly applicable for the more complex applications, but with new challenges that are worthwhile further explorations. For example, some complex applications can not be lift to the cloud without any modification. In those cases, how to identify the responsibility boundary between the original developer and the cloud DevOps? Can we devise tools to help quickly identify code blocks that potentially require changes from DevOps?

As to the future development directions of supporting application lifting, Several new responsibility categories are worth further considerations, In addition what have been described in Section 3.3. First is how to manage security responsibilities. Some cloud and third party vendors have been providing mechanisms and tooling to support requirements on security, some of these vendors providing very fine-grained access control (gcp, nd). If choosing native security mechanisms, application developers can leave the responsibilities to the cloud providers. Second is responsibilities of automated dynamic resource allocation and resource optimization, achieving it asks for platform support as well as appropriate set of tooling. Cheung et al. (Cheung et al., 2021) has proposed a concept called "targets for dynamic optimization" as one of four new facets for cloud programming to abstract such resource optimization responsibilities. Databricks spark, a distributed compute engine, has introduced an autoscaling technique, that removes the need for developers to state the amount of resource (for instance, CPU and memory) provisions automatically. Databricks claims that this technique is able to reduce cloud costs by up to 30%. However, this technique is specific to the Databricks platform. We hope

to see more new vendor-agnostic techniques and tooling being developed to provide similar capabilities of automated cloud resource allocation. Furthermore, co-developing tooling for the cloud resource allocation optimization purpose and tooling for cloud application performance monitoring is a possible direction for tooling advance.

ACKNOWLEDGMENTS

We thank Dr. Zhimin Zeng from Zilian Tech, and Prof. Longpeng Zhang from University of Electronic Science and Technology of China, for their constructive discussion on related topics, and their support for this work. This work is partially supported by National Social Science Foundation of China(Grant No. 20CJY009).

REFERENCES

(n.d.). Ansible executable playbooks. https://docs.ansible.com/ansible/latest/cli/ansible-playbook.html.

(n.d.). Aws database migration service. https://aws.amazon.com/dms/.

(n.d.). Azure files pricing. https://azure.microsoft.com/en-us/pricing/details/storage/files/.

(n.d.). Chef infra, a powerful automation platform. https://github.com/chef/chef.

(n.d.). Cloud native compute foundation. https://www.cncf.io/.

(n.d.a). Datadog: Cloud monitoring as a service. https://www.datadoghq.com/.

(n.d.). Docker: Empowering app development for developers. https://www.docker.com/.

(n.d.). Gcp security and identity. https://cloud.google.com/products/security-and-identity.

(n.d.a). Infrastructure as code: What is it? why is it important? https://www.hashicorp.com/resources/what-is-infrastructure-as-code.

(n.d.). Kubernetes: Production-grade container orchestration. https://kubernetes.io/.

(n.d.). Lightstep - monitor and improve performance. https://lightstep.com/.

(n.d.). Linux containers. https://linuxcontainers.org/.

(n.d.b). Migrating workloads to aws with datadog. https://www.datadoghq.com/resources/aws-migration-ebook.

(n.d.). Podman: A tool for managing oci containers and pods. https://github.com/containers/podman.

(n.d.). Puppet: Powerful infrastructure automation and delivery. https://puppet.com/.

(n.d.). Salt project. https://saltproject.io/.

(n.d.). Servicenow: The simplicity your company craves. https://www.servicenow.com/.

(n.d.). Stackpulse - reliability platform. https://stackpulse.com/.

(n.d.b). Terraform. https://www.terraform.io/.

(n.d.). What is azure sql managed instance? https://docs.microsoft.com/en-us/azure/azure-sql/managed-instance/sql-managed-instance-paas-overview.

Akkus, I. E., Chen, R., Rimac, I., Stein, M., Satzke, K., Beck, A., Aditya, P., and Hilt, V. (2018). {SAND}: Towards high-performance serverless computing. In *2018 {Usenix} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 923–935.

Ameloot, T. J., Neven, F., and Van den Bussche, J. (2013). Relational transducers for declarative networking. *Journal of the ACM (JACM)*, 60(2):1–38.

Artac, M., Borovssak, T., Di Nitto, E., Guerriero, M., and Tamburri, D. A. (2017). Devops: introducing infrastructure-as-code. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 497–498. IEEE.

Castellanos Ardila, J. P. and Gallina, B. (2020). Separation of concerns in process compliance checking: divide-and-conquer. In *27th European & Asian System, Software & Service Process Improvement & Innovation EuroAsiaSP2'20, 09 Sep 2020, Düsseldorf, Germany*. Springer International Publishing.

Cheung, A., Crooks, N., Hellerstein, J. M., and Milano, M. (2021). New directions in cloud programming. *arXiv preprint arXiv:2101.01159*.

De Win, B., Piessens, F., Joosen, W., and Verhanneman, T. (2002). On the importance of the separation-of-concerns principle in secure software engineering. In *Workshop on the Application of Engineering Principles to System Security Design*, pages 1–10. Citeseer.

Dean, J. and Ghemawat, S. (2004). Mapreduce: Simplified data processing on large clusters.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., and Patterns, D. (1995). Elements of reusable object-oriented software. *Design Patterns. massachusetts: Addison-Wesley Publishing Company*.

Hummer, W., Rosenberg, F., Oliveira, F., and Eilam, T. (2013). Testing idempotence for infrastructure as code. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 368–388. Springer.

Hürsch, W. L. and Lopes, C. V. (1995). Separation of concerns.

Hüttermann, M. (2012). Infrastructure as code. In *DevOps for Developers*, pages 135–156. Springer.

Kiczales, G. and Mezini, M. (2005). Separation of concerns with procedures, annotations, advice and pointcuts. In *European Conference on Object-Oriented Programming*, pages 195–213. Springer.

Morris, K. (2016). *Infrastructure as code: managing servers in the cloud.* " O'Reilly Media, Inc.".

Mueller, E. (2010). What'sa devop? the agile admin. *URL (consulted 2019): http://theagileadmin.com/2010/10/08/whats-a-devop*.

Panunzio, M. and Vardanega, T. (2014). A component-based process with separation of concerns for the development of embedded real-time software systems. *Journal of Systems and Software*, 96:105–121.

Rahman, A. and Williams, L. (2018). Characterizing defective configuration scripts used for continuous deployment. In *2018 IEEE 11th International conference on software testing, verification and validation (ICST)*, pages 34–45. IEEE.

Sarcar, V. (2016). *Java design patterns*. Springer.

Schwarz, J., Steffens, A., and Lichter, H. (2018). Code smells in infrastructure as code. In *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 220–228. IEEE.

Shambaugh, R., Weiss, A., and Guha, A. (2016). Rehearsal: A configuration verification tool for puppet. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 416–430.

Stutterheim, J., Achten, P., and Plasmeijer, R. (2017). Maintaining separation of concerns through task oriented software development. In *International Symposium on Trends in Functional Programming*, pages 19–38. Springer.

Van der Bent, E., Hage, J., Visser, J., and Gousios, G. (2018). How good is your puppet? an empirically defined and validated quality model for puppet. In *2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER)*, pages 164–174. IEEE.

Villamizar, M., Garces, O., Ochoa, L., Castro, H., Salamanca, L., Verano, M., Casallas, R., Gil, S., Valencia, C., Zambrano, A., et al. (2016). Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 179–182. IEEE.