UC San Diego
UC San Diego Electronic Theses and Dissertations

Title
Toward Understanding and Dealing with Failures in Cloud-Scale Systems

Permalink
https://escholarship.org/uc/item/6nn0x49p

Author
Huang, Peng

Publication Date
2016

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA, SAN DIEGO

Toward Understanding and Dealing with Failures in Cloud-Scale Systems

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Peng Huang

Committee in charge:

Professor Yuanyuan Zhou, Chair
Professor Tara Javidi
Professor Ranjit Jhala
Professor George Porter
Professor Stefan Savage

2016

Copyright

Peng Huang, 2016

All rights reserved.

The Dissertation of Peng Huang is approved and is acceptable in quality and form for publication on microfilm and electronically:

____

____

____

____

____
Chair

University of California, San Diego

2016

DEDICATION

To my parents, brother and fiancée for their unconditional love and support.

EPIGRAPH

Quis custodiet ipsos custodes? (But who can watch the watchmen?)

Juvenal

Anything that can go wrong, will go wrong.

Murphy's law

Those who fail to learn from the mistakes are doomed to repeat them.

George Santayana

In the middle of the night, [...] He would awaken and find himeself wondering if one of the machines had stopped working for some new, unknown reason. Or he would wake up thinking about the latest failure, the one whose cause they'd been looking for a whole week and sitll hadn't found. The bogeyman—la machine—was there in his bedroom.

Tracy Kidder, The Soul of a New Machine

TABLE OF CONTENTS

Signature Page iii

Dedication iv

Epigraph v

Table of Contents vi

List of Figures ix

List of Tables x

List of Listings xii

Acknowledgements xiii

Vita xv

Abstract of the Dissertation xvii

Chapter 1 Introduction 1
1.1 Understanding Cloud Service Failures 4
1.2 Misconfiguration in Cloud 6
1.3 ConfValley: A Systematic Configuration Validation Framework 7
1.4 Terminology 10
1.5 Organization 10

Chapter 2 Understanding Cloud Service Failure 12
2.1 Introduction 12
2.2 Note and Disclaimer 13
2.3 Case Studies 14
2.3.1 Case #1 Amazon Web Services Multi-Day Outage 16
2.3.2 Case #2 Microsoft Azure 2.5-Hour Outage 18
2.3.3 Case #3 Facebook 2.5-hour Outage 19
2.3.4 Case #4 Google Compute Engine Global Outage 20
2.4 Observations 21
2.4.1 Every failure is unique 22
2.4.2 Small changes have big impact 22
2.4.3 Single point of failure is rare 23
2.5 When Faults Were Not Tolerated 24
2.5.1 Fault Tolerance Techniques 25
2.5.2 A Taxonomy of Fault-Tolerance Failures 27

2.6 Were Faults Contained? . . . 33
2.6.1 Visualization with Impact Graph . . . 33
2.6.2 Fault propagation length . . . 34
2.7 What Caused These Failures? . . . 35
2.7.1 Root Cause Type . . . 35
2.7.2 Multiple Root Causes . . . 37
2.8 Zooming in on Misconfiguration . . . 38
2.8.1 What Components Were Misconfigured? . . . 38
2.8.2 What Introduced the Misconfiguration? . . . 39
2.8.3 What Constraints Were Violated? . . . 40
2.9 Conclusion . . . 43
2.10 Acknowledgements . . . 43

Chapter 3 CPL: A Configuration Specification Language . . . 45
3.1 Introduction . . . 45
3.2 Background and Motivation . . . 48
3.2.1 Configuration in cloud systems . . . 49
3.2.2 Configuration validation . . . 49
3.3 Design Considerations . . . 52
3.3.1 Language support . . . 52
3.4 Configuration Predicate Language . . . 56
3.4.1 Concepts . . . 57
3.4.2 Unified configuration representation . . . 59
3.4.3 Piping . . . 64
3.4.4 Commands . . . 66
3.4.5 Grammar and Examples . . . 66
3.4.6 Extending *CPL* . . . 66
3.5 Validation Policy and Runtime Information . . . 67
3.6 Error Messages . . . 68
3.7 Evaluation . . . 68
3.7.1 Baselines . . . 68
3.7.2 Rewriting Existing Validation Code . . . 69
3.8 Conclusion . . . 71
3.9 Acknowledgements . . . 71

Chapter 4 ConfValley: A Cloud Configuration Validation Framework . . . 72
4.1 System Design . . . 73
4.1.1 Overview . . . 73
4.2 Inference Engine . . . 74
4.3 Implementation . . . 76
4.3.1 Usage Scenarios . . . 76
4.3.2 Optimizations . . . 77
4.4 Evaluation . . . 78

4.4.1 Automatic Inference . . . 79
4.4.2 Preventing Configuration Errors . . . 81
4.4.3 Performance . . . 82
4.5 Conclusion . . . 84
4.6 Acknowledgements . . . 84

Chapter 5 Limitations and Future Work . . . 85
5.1 Limitations . . . 85
5.2 Future Work . . . 86

Chapter 6 Related Work . . . 87
6.1 Failure Studies . . . 87
6.2 Fault tolerance and recovery . . . 88
6.3 Characteristics of system misconfiguration . . . 88
6.4 Misconfiguration detection, diagnosis and fix . . . 88
6.5 System resilience to misconfiguration . . . 89
6.6 Configuration languages . . . 90
6.7 Configuration management . . . 90

Chapter 7 Concluding Remarks . . . 91

Bibliography . . . 92

LIST OF FIGURES

Figure 1.1. Event chain of a failure that happened in Microsoft Azure Storage service on December 28, 2012. 2

Figure 2.1. Fault tolerance in different layers of a simplified cloud storage service. 26

Figure 2.2. Taxonomy of why faults may not be not tolerated, resulting in failures. 28

Figure 2.3. A contrived example of impact graph for analyzing fault isolation. 34

Figure 2.4. Root cause distribution for the 34 public cloud service outages that we investigated 36

Figure 2.5. Patterns of how multiple root causes contribute together in an incident. 38

Figure 3.1. Configuration data in cloud systems. 46

Figure 3.2. The spectrum of typical configuration specifications. 51

Figure 4.1. Architecture of ConfValley. 73

Figure 4.2. Examples of *CPL* compiler optimizations. 78

Figure 4.3. Histogram of number of inferred constraints on a type of Microsoft Azure configuration data 79

LIST OF TABLES

Table 2.1. Dataset of 34 notable cloud service outages from 2009 to 2016 that we gathered using publicly available information. 14

Table 2.2. Example public cloud service outages in recent years. 15

Table 2.3. SLA compensations in Microsoft Azure. 18

Table 2.4. Representative fault tolerance techniques used by cloud practitioners. 25

Table 2.5. Operation that introduces misconfiguration 39

Table 2.6. Violated constraints in the studied misconfigurations 41

Table 3.1. Examples of configuration notations and their meanings 61

Table 3.2. Express validation code for three kinds of configuration data used inside Microsoft Azure into *CPL* specifications. 70

Table 3.3. Express validation code in two open-source cloud systems into *CPL* specifications. 70

Table 4.1. Driver code to convert different types of configuration data in Microsoft Azure into a unified representation. 76

Table 4.2. Validation constraint inference on three kinds of configuration data in Microsoft Azure. 80

Table 4.3. Breakdown on the types of inferred constrains in Table 4.2 80

Table 4.4. Running expert-written validation specifications on three latest configuration data branches in Microsoft Azure. 81

Table 4.5. Running inferred validation specifications on three latest configuration data branches in Microsoft Azure. 82

Table 4.6. Latency (in seconds) of sequential validation on three types of configuration data in Microsoft Azure. 82

Table 4.7. Latency (in seconds) of simple parallel validation on three types of configuration data in Microsoft Azure. 83

Table 4.8. Inference latency (in seconds) on three types of configuration data in Microsoft Azure. 83

LIST OF LISTINGS

Listing 3.1. A snippet that represents configurations (`Setting` elements) at different scopes using XML. 50

Listing 3.2. Validation snippet that operates on configuration instances rather than configuration classes. 53

Listing 3.3. An imperative validation snippet that prescribe the implementation details of a simple type constraint. 54

Listing 3.4. An imperative validation snippet that prescribe the implementation details of a simple value range constraint. 54

Listing 3.5. An imperative validation snippets that prescribe the implementation details of a simple uniqueness constraint. 55

Listing 3.6. *CPL* grammar 63

Listing 3.7. Example validation specifications in *CPL* 65

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere gratitude to my advisor, Professor Yuanyuan Zhou for inspiring me with her wisdoms, passion, and creativity, for teaching me how to do research, for sharing interesting stories about start-ups, and for supporting me at every stage of my PhD. Her encouragement, feedback, and honest critics have been a steady force that helped me keep improving. Obviously I would not have been able to go through the PhD grind without her help. But her inspiration to students is far beyond the PhD course. She will always be a role model and lighthouse for me.

I am thankful to my internship mentor Bill Bolosky for teaching me a lot of useful lessons on distributed systems and exposing me to the interesting war stories from operating real-world large-scale systems like Azure. They opened my eyes.

I would also like to thank the remaining members of my thesis committee: Professor Stefan Savage, Professor Ranjit Jhala, Professor George Porter, and Professor Tara Javidi for the helpful discussions and feedback that improved this dissertation.

I am indebted to members in the Opera research group. Ding Yuan, Xiao Ma, Soyeon Park, and Weiwei Xiong coached me on how to drive research projects in my junior years, which is tremendously helpful for me to conduct independent research later on. The other fellow Opera group members also gave me countless help along the way. I want to especially thank Xinxin Jin and Tianyin Xu for their friendship that kept me enlightened and inspired even in dark times. I am grateful. I also want to apologize to my co-authors on the failure analysis paper for being the victim of the publication withdrawl incident. I wish I could repay your efforts.

I feel lucky to be a student in UCSD, and the sysnet group. The faculty members and other students in sysnet set the bar for what a world-class research group should be. The various activities like SysLunch and CNS reviews are great opportunities for

students to stay connected.

My PhD would have been colorless without the love and support from my fianceé Rui. Even though we have been enduring long-distance relationship, she has always been the closet in my heart. She went through every bit of this journey together with me and painted it with endless joy. Words cannot express how grateful I am for having her as the love and sole mate in my life.

Finally, I owe my thanks forever to my parents and my brother for their unconditional love and support that they pour to me, withouth which none of the achivements that I have today is possible.

Chapter 2 contains material of a paper retracted from 11th USENIX Symposium on Operating Systems Design and Implementation 2014. Huang, Peng; Jin, Xinxin; Bolosky, Bill; Zhou, Yuanyuan. The dissertation author was the primary investigator and author of this paper.

Chapter 3, in part, is a reprint of the material as it appears in the Tenth European Conference on Computer Systems 2015. Huang, Peng; Bolosky, Bill; Singh, Abhishek; Zhou, Yuanyuan. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in part, is a reprint of the material as it appears in the Tenth European Conference on Computer Systems 2015. Huang, Peng; Bolosky, Bill; Singh, Abhishek; Zhou, Yuanyuan. The dissertation author was the primary investigator and author of this paper.

VITA

2010	B.S., Peking University, China
2010	B.A., Peking University, China
2014	M.S., University of California, San Diego
2016	Ph.D, University of California, San Diego

PUBLICATIONS

"DefDroid: Towards a More Defensive Mobile Os Against Disruptive App Behavior". Peng Huang, Tianyin Xu, Xinxin Jin, and Yuanyuan Zhou. In *Proceedings of the 14th International Conference on Mobile Systems Applications and Services (MobiSys)*, Singapore, June 2016.

"Saving Mobile App Developers from Network Disruptions". Xinxin Jin, Peng Huang, Tianyin Xu, and Yuanyuan Zhou. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys)*, London, UK, April 2016.

"ConfValley: A Systematic Configuration Validation Framework for Cloud Services". Peng Huang, Bill Bolosky, Abhishek Singh, and Yuanyuan Zhou. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys)*, Bordeaux, France, April 2015.

"Why Does a Cloud-Scale Service Fail Despite Fault-Tolerance?". Peng Huang, Xinxin Jin, Bill Bolosky, and Yuanyuan Zhou. Withdrawn from *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, CO, October 2014.

"Performance Regression Testing Target Prioritization via Performance Risk Analysis". Peng Huang, Xiao Ma, Dongcai Shen, and Yuanyuan Zhou. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, Hyderabad, India, May 2014

"Do Not Blame Users for Misconfigurations". Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, Farminton, Pennsylvania, November 2013.

"EDoctor: Automatically Diagnosing Abnormal Battery Drain Issues on Smartphones". Xiao Ma, Peng Huang, Xinxin Jin, Pei Wang, Soyeon Park, Dongcai Shen, Yuanyuan Zhou, Lawrence K. Saul, and Geoffrey M. Voelker. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, Lombard, IL, April 2013

“Be Conservative: Enhancing Failure Diagnosis with Proactive Logging”. Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, October 2012.

ABSTRACT OF THE DISSERTATION

Toward Understanding and Dealing with Failures in Cloud-Scale Systems

by

Peng Huang

Doctor of Philosophy in Computer Science

University of California, San Diego, 2016

Professor Yuanyuan Zhou, Chair

In cloud-scale systems, fault is a fact of life. To tolerate faults and provide highly-available service is arguably the single most important task for cloud builders. Yet, despite the considerable efforts into fault-tolerance and software engineering for reliability, all cloud scale services continue to experience costly failures. A natural question to ask is: *why do cloud-scale services still fail despite the abundant fault-tolerance and how we can further improve?* This thesis attempts to shed light on this question.

In the first part of this thesis, we study a set of 34 publically disclosed cloud ser-

vice outages that we gathered and consider them from the point of view of fault-tolerance mechanisms. We present a novel taxonomy to categorize why the mechanisms may be ineffective; it includes faults that cannot be handled by replication, insufficient redundancy, and undetected faults. We also explore the root causes of failures, and investigate the interactions of system components in failures that were caused by multiple faults.

We find that, in many cases, while cloud systems are robust to tolerate traditional faults, they are fragile under misconfiguration, which is a major source of service unavailability. To further improve cloud service quality, it is crucial to reduce misconfiguration.

In the second half of this thesis, we propose a framework, *ConfValley*, to systematically validate configuration and catch errors before production. At the core of ConfValley is a language called CPL to allow experts to express configuration specifications declaratively. To further reduce operators' burdens of writing configuration specifications, our framework also includes a component to automatically infer specifications.

We evaluate ConfValley in a leading cloud service provider, Microsoft Azure, on its various types of configuration data. We rewrite existing configuration validation code in Microsoft Azure in CPL with more than 10x fewer lines of code. The framework also automatically infers thousands of CPL specs with high accuracy. With the translated and automatically generated specifications, we prevented a number of configuration errors from rolling out in production in Microsoft Azure.

Chapter 1

Introduction

In 2006, Amazon launched a product that offers infrastructure services to third-party companies. At that time, it was not part of Amazon's core business but rather a side product that stemmed from Amazon's own struggle in building scalable infrastructure. Ten years later, this "side product", the Elastic Compute Cloud (Amazon Web Services), becomes a $10 billion dollar business with more than 60% yearly growth rate for Amazon that is even bigger than Amazon.com. More importantly, it has led a paradigm shift in the whole computing landscape—the rise of cloud computing.

For developers who want to build large-scale applications economically, instead of spending substantial resources operating their own infrastructure which may turn out to mismatch the growth requirement, now the *de facto* standard is to leverage the readily-available, *on-demand* computing resources offered in various abstractions such as virtual machine, batch jobs, and object storage. Many popular services today such as Netflix, Dropbox, and AirBnB are built in this way.

For end users who need convenient access to data and services, instead of going through the pain of installing, upgrading, and patching software in individual devices, moving to a centralized cloud that is available from a variety of devices has become a much simpler solution. The migration from traditional standalone office software to web application such as Google Docs for many users is such an example.

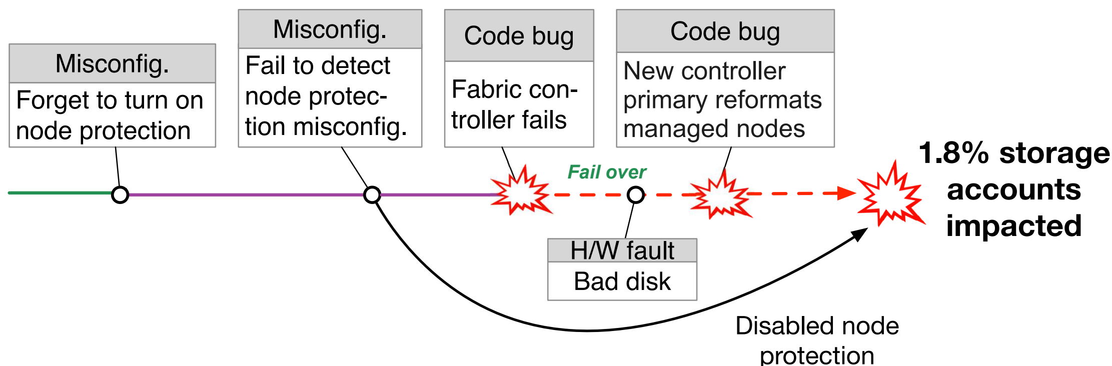

Figure 1.1. Event chain of a failure that happened in Microsoft Azure Storage service on December 28, 2012 [39]. Fabric controller is to Microsoft Azure as the kernel is to Windows operating system.

By definition, cloud service needs to be accessed anywhere, anytime. What this means for the underlying infrastructure is that the system availability[1] is of paramount importance [71, 77, 66]. Indeed, in almost all the cloud practitioners that we have interacted with have expressed similar emphasis that "availability is arguably the single most important KPI (Key Performance Indicator) now for cloud vendors". However, providing high availability at the scale and complexity of cloud systems is challenging: in an industrial-strength cloud, on any day, there is almost always some system component in a data center that is experiencing glitches [84, 114, 75, 83, 123]. For example, in a typical first year of a cluster, there can be thousands of hard drive faults [75]; a very rare Linux race condition bug was triggered 500 times per day inside Facebook [22]. Cloud builders often summarize this harsh reality as "fault is a norm rather than exception".

In response, cloud builders spend extensive efforts to reduce faults by improving software engineering, enforcing code reviews, training operators regularly, adopting safer programming languages, etc. Moreover, fault-tolerance is treated as a first-class citizen so that even if some component fails the system as a whole is still available. For example, RAID and erasure coding [91] cope with disk failures. Primary-backup

[1] Ratio of time the system is functional to the total time during an observation period.

and quorum-based replication [98, 69, 96, 105] defend against node failures. Data center networks [53, 88, 102] provide redundant paths and failover protocols in case of network failures. Software in the upper layer such as MapReduce [76] also expects failures and adds application specific failure handling logic. In addition, fast failure detection [70, 101] and cheap failure recovery [110, 68] expedite responses to failures.

Despite these efforts, *all* cloud-scale systems today continue to experience service disruptions or outages [7, 3, 16, 14, 38, 39, 40, 29, 27, 33, 46]. Figure 1.1 shows a cloud service failure in Microsoft Azure that is caused by a combination of hardware failure, code bug and misconfiguration.

When a cloud service failure occurs, the impact and cost is significant due to users' increasing reliance on cloud. The service disruption in Figure 1.1 lasted 77 hours, affecting 1.8% of the Microsoft Azure Storage accounts; a major outage in Amazon Web Services (AWS) in 2015 took down popular online services like Netflix, Reddit and AirBnB [17]; in a recent outage Google Compute Engine went down for all of its customers *everywhere* [32]. In these failure incidents, users are not the only ones who suffer. The service providers also sustain significant financial loss. Microsoft provided *100%* service credit to compensate the affected customers during the incident in Figure 1.1; an AWS outage can potentially result in a nearly $1 million-per-hour loss of sales to Amazon (based on its recent earnings statement [2]).

To systematically reduce the prevalence of cloud failures, it is crucial to understand 1). *why do cloud services fail?*, and 2). *what can be done to further improve?* This thesis aims to shed lights on the first question by presenting a study on real-world cloud failures to analyze their unique characteristics. Based on the analysis result and our interactions with cloud practitioners, configuration error is detrimental fault to cloud that calls for attention. Therefore, the second half of this thesis is devoted for the second question with a solution for reducing cloud configuration error.

Thesis Statement: *Analyzing failures in cloud needs to closely examine the fault tolerance mechanisms. Misconfiguration is a common cloud fault that is difficult to tolerate with existing techniques. Using a declarative specification language can effectively prevent misconfiguration.*

1.1 Understanding Cloud Service Failures

Failure analysis has long proven to be an indispensable part of reliability engineering. For example, the tragic Air France 447 accident was investigated with years of efforts to recover the black box and diagnose the root causes, which discovered a number of issues including the pitot tubes, side-stick control and pilot errors [1]. At the very least, learning from existing mistakes is an effective way to avoid making the same mistakes again in the future. More importantly, failures are usually a manifestation of some fundamental flaw that is applicable to a set of systems. Analyzing failures in depth can uncover such flaw and help improve design to eliminate errors. For example, the landmark study on Tandem mainframe system failures by Gray pointed out the lack of software fault-tolerance and inspired various techniques [87].

Conducting cloud failure analysis is even more important. First, as mentioned earlier, reliability/availability has become the topmost priority for cloud, which is almost hard to improve without getting feedback from past failures. In our experiences, it is common for vendors to organize weekly or even daily internal meetings to discuss recent failure incidents and summarize the lessons. Second, despite a number of studies on failure characteristics in different environments such as personal computers [107], networked systems [127], high-performance computing systems [119], Internet services [108] and Hadoop clusters [116], to the best of our knowledge, there is no large-scale study of cloud service failures. Compared with the prior studied systems, cloud systems have a number of unique characteristics such as orders of mag-

nitudes more components, frequent component failures, highly dynamic environment, and abundant fault tolerance. Thirdly, a study on cloud failure can not only benefit cloud providers but also provide important lessons for the system research community to understand what major problems remains to be addressed. As cloud computing grows, the understandings and solutions derived from the study will benefit the design of future cloud systems.

But analyzing failures in a real-world cloud systems is not easy. Apart from the challenge of data collection, the irregularities of real-world failures also makes analysis difficult to generalize. The war stories from our industrial collaborators are typically incidents about cloud failing in a complex, unique and bizarre fashion (e.g., the data center gets too hot that renders disks to be flaky which triggers a software bug and in response the operators made some mistakes). It is the scenario where Murphy's law seems to constantly hold: "anything that can go wrong, will go wrong". When presented with these real-world failure cases, without a proper angle, failure analysis can quickly degrade to a case-by-case study, which sheds little light on how to fundamentally eliminate system flaws.

The conventional methodology of failure analysis as used in companies' internal incident reviews and prior research failure studies [87, 108, 119, 107, 116] put a lot of emphasis on analyzing root causes of component faults in a service failure. While this works fine for traditional systems, this root-cause based analysis is not enough for understanding cloud failures. This is because with hundreds of thousands of commodity components in a changing environment, component-level faults are just too many for developers to comprehend and react (otherwise, there is no point of adding fault-tolerance). The overarching design philosophy for cloud is to embrace faults and harden the system with abundant fault-tolerance mechanisms so that the system can continue to run even if some components fail.

Our insight on understanding cloud failure builds upon this unique nature of cloud systems: we should analyze not only *"why do cloud systems fail"* but also *"why do cloud systems fail* **despite the abundant fault-tolerance techniques** *that were supposed to prevent component faults from causing service disruptions"*. With this insight, we thought about the potential reasons for failures of the fault-tolerance techniques and develop a novel tree-based taxonomy to clarify our thinking. The taxonomy can be a useful tool to navigate a specific complex failure case as well as to aggregate multiple failure cases. In addition, we develop some other analysis methods to capture the unique patterns of cloud failures such as analyzing how faults propagated in different system components and how multiple root causes contribute to a failure.

To understand cloud failures quantitatively, we collected detailed failure data from the service incidents that happened in a major cloud service provider over a 13-month period. By using our methodology to analyze this failure data, we reveal findings that triggered wide discussion inside the cloud company. But due to confidentiality agreement, this thesis will mainly focus on discussion on our methodology, high-level observations, and independent analysis on a small set (34) of public cloud outages that we gathered from the Internet.

1.2 Misconfiguration in Cloud

Cloud systems today use an astonishingly large number of configuration artifacts of wide variety to control components across the stack: e.g., network routing, endpoint IP assignment, service authentication, recovery actions, resource locations, and caching. Compared to traditional distributed systems which get updated sparsely for stability, cloud environment is inherently dynamic. Along with the software, hardware and workload changes, configurations are frequently updated to adjust to the new need. For example, at Facebook, there are hundreds of configuration changes committed, a

throughput that is even higher than its source code commit rate [122].

Given the many configurations and frequent updates, which are mostly authored by developers and operators, it is inevitable to introduce configuration errors. And because configurations are used to control some of the most critical functionalities in a system, a seemingly small misconfiguration can affect the entire system. For example, misconfiguring the endpoints of load balancing components could cause all traffic to be directed to one server, overwhelming that server and effectively making the whole service unavailable; misconfiguring the backup options of a redundant service pair could cause the backup to be useless should the primary go down. It is not uncommon to see misconfiguration being the root cause for many public service disruptions [25, 3, 16, 14, 39, 40, 27, 50, 33].

In our cloud failure study, we find that configuration error is a dominant source of the studied cloud failures, especially for these high-severity incidents. From fault-tolerance analysis perspective, configuration error often causes failures that are difficult to be tolerated by the abundant fault-tolerance techniques used in cloud. Even worse, many fault-tolerance mechanisms themselves are controlled by configuration, and the any error in such configurations can make those mechanisms useless in case of hardware or software errors. Neither can traditional bug detection tools help catch the configuration error because the errors are external in the system.

This motivates us to further analyze these misconfigurations as part of our failure characteristics study. Chapter 2 discusses the high-level findings we summarized.

1.3 ConfValley: A Systematic Configuration Validation Framework

The problem of misconfiguration is not a unique in cloud. Many solutions have been proposed to attack misconfiguration in traditional systems ranging from misconfig-

uration detection [130, 131], diagnosis [125, 124, 126, 56] and repair [121, 97] to system resilience [94, 128]. These solutions greatly alleviate end users' pain in configuring unfamiliar software. However, they are often postmortem efforts or incur overhead (e.g., instrumentation, record and replay) that is too expensive to deploy for evolving cloud-scale production systems. For cloud services, the availability requirement dictates a *proactive* approach to prevent misconfiguration rather than reactively fix misconfiguration. The solution should also be efficient enough to handle frequent configuration changes at a large scale.

From our failure characteristics study, we find that a significant portion of the misconfiguration can be caught by explicitly validating the configuration against some constraints, e.g., a flag must be enabled, or a parameter must be a file that exists with write permission. But unfortunately the current practices for validating configuration is inefficient and *ad hoc*, by using manual configuration reviews that are time consuming, and/or bulky validation code that is tedious to write and hard to maintain. Developers and operators have no incentives to proactively write configuration checking code. As a result, there is insufficient validation to catch errors, and the validation often after service failures occurred.

The fundamental issue with the current practices for validating configuration is that there are no guiding principles and validation primitives to describe the kind of expectations developers/operators would like to express. Thus developers mindset of configuration validation is restricted to be an ad-hoc scripting at low-level using programming languages that are unfit for this task. We believe that by providing the proper abstractions and right tools to practitioners, configuration validation can be made an ordinary part of cloud-scale system deployment and can proactively prevent misconfiguring production services.

Toward this end, we take a language-based approach and design a declarative

language, *CPL*, to describe various configuration specifications for validation purpose. The language decouples the core validation logic from implementation details, allowing the specifications to be described compactly and independently of underlying configuration representations. Given a specification written in *CPL*, a running service will continuously validate configuration whenever configuration is updated, and report error if it finds violation of the specifications. In this way, misconfiguration can be prevented before rolling out to production. Chapter 3 describes the design of *CPL* in details.

Even with a compact language like *CPL*, writing all configuration specifications from scratch can be tiresome. Therefore, we also develop a component to automatically infer and generate *CPL* specifications. In this way, cloud practitioners can focus on writing complex and domain specific specifications that are hard to infer. Additionally, auto-inference can keep the specifications up to date as the systems evolve.

The specification language *CPL*, the validation service, an interactive console, and the specification-inference engine together form our framework ConfValley. ConfValley is designed for practitioners operating cloud-scale systems to efficiently conduct validate configuration and proactively prevent misconfiguration from causing service failures.

Our experience in using ConfValley inside Microsoft Azure [35] as well as on two open-source cloud systems shows that *CPL* makes configuration validation is much easier and compared to prior practice. For example, a previous validation module used in Microsoft Azure with more than 3000 lines of code became only 109 lines of code in 62 *CPL* specifications, out of which 27 specifications could be automatically inferred. With the inferred *CPL* specifications, we prevented a number of configuration errors in Microsoft Azure. Chapter 4 shows the evaluation in more details.

1.4 Terminology

In this thesis we use the word *fault* to mean a situation in which a system component did not behave as intended (or had a problem that would eventually result in incorrect behavior in the case of latent faults). This includes hardware stopping, software misbehaving, power loss, and any other problem with any component in the system. When the fault-tolerance mechanisms employed in the systems are insufficient a failure occurs. We use the word *failure* (or service disruption) to mean any event that resulted in a postmortem analysis (because there was a service failure of some sort). This is analogous to the usage of these words suggested by Patterson [111], where we restrict the system-level to components of cloud services that warrant postmortems.

1.5 Organization

The remaining of this thesis is organized as follows:

Chapter 2 discusses cloud service failures using cases publicly disclosed by various cloud vendors, presents several analysis angles in studying cloud failures, and a novel taxonomy for categorizing failures.

Chapter 3 presents the background, motivation and design of a declarative configuration specification language, *CPL*, that we developed to catch misconfiguration and prevent cloud failures. We compare writing configuration validation in *CPL* with existing *ad hoc* approach.

Chapter 4 describes the tool chain, ConfValley, that we developed along with *CPL* to make the configuration validation practices easier and more systematic. We show how the tool chain generates basic specifications written in *CPL*, and evaluate the effectiveness of the tool in a commercial cloud environment, Microsoft Azure.

Chapter 6 discusses the prior work that is related to this thesis and compare them

with our work.

Chapter 7 summarizes the contributions of this thesis and discusses future directions.

Chapter 2

Understanding Cloud Service Failure

2.1 Introduction

Cloud-scale systems [58] are built with hundreds of thousands of commodity servers [59]. Operating at this scale, faults are inevitable [84, 114, 75, 83, 123]. To cope with this fact of life, their designers harden them with abundant fault handling techniques such as RAID [112], erasure coding [91], primary-backup and quorum-based replication [98, 69, 96, 105], path-redundant and failover networking [53, 88, 102], and app-specific fault handling logic [76] to detect, tolerate and recover from various faults in different layers of the systems. Additionally, careful software engineering, extensive testing and gradual roll-out are widely adopted to catch problems before they manifest as failures. By and large, these techniques have been spectacularly successful, as evidenced by the fact that several cloud-scale services operate with good quality of service and only a handful of major outages. Nevertheless, all cloud-scale systems still encounter smaller-scale failures [7, 3, 16, 14, 38, 39, 40, 29, 27, 33, 46].

To reduce the prevalence of these failures in the future, we ask: *Why do failures occur even in systems designed for fault tolerance and equipped with many fault handling techniques?*; *What kind of faults are especially hard to tolerate?*; *Which parts are still lacking and need to be improved?*

We look into failures in cloud-scale systems along two independent dimensions. First, we ask why fault-tolerance did not prevent a fault from turning into a failure. This can be explained by reasons such as deterministic faults, insufficient redundancy, or failing to trigger the fault-tolerance when needed. In Section 2.5 we develop a novel taxonomy of why faults were not tolerated.

Second, when trying to build reliable services, it is useful both to tolerate faults and to remove them altogether. Understanding the types of faults (*e.g.*, contained faults versus propagating faults) and root causes of faults (*e.g.*, code bug, misconfiguration) is helpful for both. We therefore look at the set of faults that underlie failures and the set of root causes that give rise to faults. We answer questions like how faults propagated in different system components and how many failures were caused by multiple root causes. In Section 2.6 we visualize the fault propagation pattern with an impact graph.

While it would be ideal to gather fault and failure data from all cloud-scale services and investigate the common and specific patterns, because these services are operated as independent companies, we were unable to aggregate such confidential data.

Nevertheless, we conduct the first comprehensive study of failures inside a major cloud service, CloudA. In particular, we studied all service disruptions over a one year period for which a postmrtem was written by the service team. The reports contain sufficiently detailed information such as the fault events, root causes and impacts for us to investigate each case. CloudA had very few major outages (and no data losses) during our study, so we were unable to generalize about major problems. Rather, almost all of the failures we investigated are limited in scope and/or effect.

2.2 Note and Disclaimer

For confidentiality reason, this thesis will not report any quantitative findings drawn from the confidential failure data from CloudA. This thesis only focuses on the

Table 2.1. Dataset of 34 notable cloud service outages from 2009 to 2016 that we gathered using publicly available information.

Vendor	Examined outage cases
Google Cloud	9
AWS	8
Microsoft Azure	5
Rackspace	5
Other	7
Total	34

analysis methodology we developed during the study that is independent of the data in the hope that it can inspire future studies of similar kind. Some high-level observation will be discussed in the context of re-applying our independent analysis methodology on 34 publicly available failure outages that we gathered for cloud service outages from multiple vendors as shown in Table 2.1. The descriptions throughout this thesis are not specifically referring to issues in CloudA. Thus, this thesis does not reflect in any way, nor should it be used to imply by any means, the service quality of CloudA.

2.3 Case Studies

As shown in Table 2.1, we collected 34 notable cloud service outages from multiple cloud providers using publicly available information such as the official blog as a public dataset for use in this thesis. Table 2.2 lists some examples from this public dataset. The descriptions and discussions in this thesis are based on this dataset rather than the confidential dataset from CloudA. In the remaining of this section, we describe several cases from this dataset.

Table 2.2. Example public cloud service outages in recent years.

Service	**Date**	**Summary**	**Impact**
AWS	9/20/2015	Capacity pressure affected metadata service [13]	5 hrs; DynamoDB
AWS	8/26/2013	Networking device grey failure caused packet loss [12]	49 mins; one availability zone
AWS	12/24/2012	Deleted ELB state data caused load balancer misconfiguration [14]	22 hrs; 6.8% of ELB instances
AWS	10/22/2012	Memory leak due to stale DNS record causes EBS outage [16]	6 hrs; EBS, ELB, EC2, RDS
AWS	6/29/2012	Generator failed to transfer power during severe storm [15]	2 hrs; EC2, EBS, ELB, RDS
AWS	4/21/2011	Incorrect network traffic shift during configuration change [3]	3 days 18 hrs; EBS, EC2, RDS
Microsoft Azure	11/18/2014	Incorrect configuration change exposes infinite loop bug [37]	2 days 10 hrs; storage, VM
Microsoft Azure	2/22/2013	Storage service disruption due to expired certificate [40]	12 hrs; storage in all regions
Microsoft Azure	12/28/2012	Multiple issues caused storage service disruption [39]	3 days 5 hrs; 1.8% storage accounts
Microsoft Azure	7/26/2012	Improper throttle limit trigger bugs in hardware devices [38]	2 hrs; compute in one cluster
Microsoft Azure	2/29/2012	Leap day bug resulted in VM service outage [36]	34 hrs; compute in all clusters
Google Cloud	4/11/2016	Accidental IP blocks removal cause connectivity loss [32]	18 mins; all regions
Google Cloud	11/23/2015	New network link overloaded due to improper announcement [31]	1 hrs; GCE in one region
Google Cloud	2/18/2015	Stale routing information led to traffic loss [30]	2.7 hrs; majority of GCE instances
Google Cloud	1/24/2014	Invalid authentication server IPs caused service outage [27]	1.5 hrs; 100% API requests
Dropbox	1/10/2014	Script bug caused OS upgrade on production servers [21]	3 hrs; service offline

2.3.1 Case #1 Amazon Web Services Multi-Day Outage

Sources

Official blog: [3]. Other coverage: [4, 8, 5, 6, 11].

Summary

Description On April 21st, 2011, at around 1 AM PDT, multiple AWS customers experienced a large number of I/O errors. After 40 minutes, Amazon acknowledged that they were having problems with EBS (Elastic Block Store) services in US-East-1 region. The failures propagated to clusters across the region. By 12:04 PM, the outrage was contained to its source Availability Zone. But recovering the affected zone took more than two days, involving physical installation of new capacity to the cluster. Amazon RDS (Relational Database Service) service was also affected because of its dependency on EBS for storage.

Impact Many customers include Reddit, Quora, Heroku and Foursquare were affected. For example, Quora was completely down because of the incident. Reddit had to operate in read-only mode. "0.07% of the volumes in the affected Availability Zone for the customers could not be restored in consistent state". There is no further detail about the lost data.

Cost Amazon provides "a 10 day credit equal to 100% of their usage of EBS Volumes, EC2 Instances and RDS database instances that were running in the affected Availability Zone" to all customers in the affected Availability regardless of whether they were impacted or not.

Root Cause

Prior to the outage, Amazon performed a scaling activity in one US East Region Availability Zone to upgrade the primary EBS network (high bandwidth network in

normal operation) capacity. This involved a network configuration change step to re-route the traffic to the redundant routers in the primary network to allow the upgrade. But it's mistakenly configured to route to the low capacity secondary network, which cannot handle such high load. As a result, both the primary and secondary network were disrupted, with many EBS nodes isolated from its replicas.

When a node could not connect to the node it's replicating data to, it will try to find another node to "re-mirror". Once Amazon the correct the configuration mistake and restored the network, the re-mirror operations began to soar dramatically as the number of isolated nodes is too big and "re-mirror" is performed in peer-to-peer fashion. The surge created a "re-mirror storm" that quickly exhausted the cluster's free space.

This problem has a dramatic cascading effect to impact not only that Availability Zone but also other Zones in the EBS control plane. In EBS, a regional thread pool is allocated to serve API requests. Since the exhausted cluster cannot accept volume creation API requests, the large number of pending requests quickly absorbed all threads in the pool and caused API requests for other Availability Zones in that region to fail as well. This "ripple effect" kept going. What exacerbated the situation is that "re-mirror storm" reveals a rare concurrency bug that could crash the nodes, further reducing the number of available nodes.

After about 10 hours, Amazon managed to contain the outage to the source failure zone by disabling its API accesses to the EBS control plane. But recovering the initial failure zone was very time consuming. First, failed nodes cannot be reused until every data replica is re-mirrored. New capacity has to be provided for these nodes to find free space for re-mirroring and then reusing them. Physical relocation and installation of such capacity took about one day, which brought back 97.8% of the "stuck" volumes. Second, when the team started to restore the frozen API accesses to EBS control plane, there were a large number of operations in the backlog to be synced between

the failure zone and EBS control plane. To avoid another communication storm, restoring was carried out gradually in more than one day. After opening up the API accesses, the team spent another day to restore the remaining 2.2% data from early snapshots of these volumes in S3. In the end, 0.07% of the data could not be restored.

2.3.2 Case #2 Microsoft Azure 2.5-Hour Outage

Sources

Official blog: [38]. Others: [42, 41, 43], etc.

Summary

Description On July 26th, 2012, a network device misconfiguration caused Microsoft Azure service disruption in Western Europe sub-region at 11:09 AM GMT. The disruption lasts around 2.5 hours.

Impact It triggered a burst of Tweets about the incident. For example, SoundGecko, a text-to-audio translation service, tweeted "*SoundGecko is currently unavailable due to data centre outage on Windows Azure. Apologies for inconveniences*"[1].

Cost Like other cloud vendors, Microsoft compensates the customers for SLA credits. According to the official document [52], the SLA compensations are:

Table 2.3. SLA compensations in Microsoft Azure.

Monthly Uptime Percentage	Service Credit
<99.95%	10%
<99%	20%

[1] https://twitter.com/SoundGecko/status/228467305112293378

Root Cause

In Microsoft Azure network infrastructure, to prevent network failure cascading, datacenter network devices has a “safety valve” mechanism to limit the connections it can accept. When such limit is exceeded, there will be management message exchanges between the device and some manager nodes (probably to redirect requests, exact mechanisms unknown).

Before the incident, Microsoft added new capacity to the West Europe sub-region. But the safety-valve setting was not adjusted accordingly. A rapid increase of traffic to the cluster exceeded the connection limit. This resulted in a considerable amount of network management messages. The surge exposed some hardware device driver bug that hogs the CPU to 100% utilization, preventing these machines from accepting connections. The main fix is to increase the limit and patch the exposed bugs. Microsoft also says it will improve the network monitoring systems to prevent such outage.

2.3.3 Case #3 Facebook 2.5-hour Outage

Sources

Official blog: [25]. Other coverage: [23, 26, 24].

Summary

Description On September 23rd 2010, Facebook went down at around 11:30 a.m. PST. The site did not come back online for most users until around 3 p.m. PST.

Impact This is Facebook’s “worst outage in over four years”.

Cost There is no direct cost to compensate users. But the incident did cause a lot of user dissatisfaction and hurt Facebook’s reputation.

Root Cause

Facebook uses automatic configuration management system. Clients (internal system component) will query and store configuration from the system. There is also a caching layer that sits between the persistent storage and client. Whenever the configuration management system detects there is a problem in the configuration cache, it will query the persistent storage in hope it has a valid, new configuration.

In the incident, an invalid configuration was pushed to the persistent storage. When every client attempts to read the invalid configuration from cache, it will send a query to the database to "fix" the invalid cache. This caused thousands of queries to database per second, which quickly overwhelm the database system. Furthermore, when the client gets no result from the database, it interprets the value as invalid and will delete the cache entry and retry. This means as long as some clients do not get results, the database will get more and more queries even when the bad configuration is fixed.

2.3.4 Case #4 Google Compute Engine Global Outage

Sources

Official blog: [32]. Other coverage: [51, 28].

Summary

Description On April 11th, 2016, at around 19:09 Pacific Time, Google Compute Engine instances in all regions lost connectivity for a total of 18 minutes.

Impact Even though the 18 minutes connecitivity does not sound terribly long, this incident impacted *all* customers *everywhere*, which is very rare for cloud.

Cost Google issued 10% to 25% service credits to the impacted customers. These

credits are more than the defined service level agreement to keep "with the spirit of those SLAs and our ongoing intention to provide a highly-available Google Cloud product suite to all our customers".

Root Cause

Google Compute Engine use IP blocks for external network routing. The IP blocks are announced from different locations using standard BGP protocol. In the incident, the engineers removed an unused IP block from the network configuration. During this configuration change, the configuration management software detected inconsistency of this configuration in different files and decided to revert to the old configuration. Unfortunately, due to a bug in the software, instead of reverting, the software removed all IP blocks from the new configuration and pushed this change.

What is even more interesting about this incident is that like other companies Google employs a number of safety mechanisms to prevent bad changes causing wide impact. In particular, a configuration change will typically go through a canary stage to be deployed at a single site. The canary stage in this incident indeed caught the bag change. But unfortunately, a second software bug here did not send the canary result back to stop the deployment. As a result, the invalid configuration is propagated to all regions.

2.4 Observations

From the above case studies and the public dataset, we made a few high-level observations about cloud failure characteristics.

2.4.1 Every failure is unique

In the failure cases that we examined, there are no two cases that happen in identical or even similar fashions. Every case fails because of some seemingly random combination of unexpected factors: temperature getting too high, device driver hit a bug, recovery protocol overly aggressive, backup component having insufficient capacity, canary phase ignoring alert signals, operator redirecting the traffic to a wrong route, etc. Although this uniqueness characteristics could be due to the different architecture and implementations for the systems we investigated, even within the same system, it still applies.

The failure uniqueness characteristics in part reflects the scale and complexity of cloud systems: with a large number of complex components, from probability point of view, it is very likely different failures are caused by unique combinations of component faults. We can also view the failure uniqueness as a positive sign in the vendors' system quality assurance process. When a failure happens, if a vendor learn from the mistake and take actions, it will help prevent the same failures from happening again, On the contrary, if failures are only dismissed as a discrete accident without being carefully analyzed, it is likely that the failure "will repeat itself".

But when cloud failures are happening in a unique way, it also raises the challenge for how to analyze past failures and proactively prevent new failures. It is tempting to treat failures as random events and analyze them case by case. Being able to treat failures from novel perspectives and identify underlying patterns is a rewarding but difficult problem.

2.4.2 Small changes have big impact

Change is a constant theme in various cloud system components, e.g., rolling out software feature, fixing glitches, upgrading capacity, and adjusting workload. This

is one of the defining characteristics of commercial cloud systems. But frequent changes also become the dominant trigger of failures. Most of the incidents that we examined happened because the developers/operators made some changes to the system. For example, Facebook similarly found that incidents happen much less frequent in weekends than in weekdays, and much less frequent in holiday months like December compared to other months [104]. This fact should not be used for discouraging developers from making changes. For systems like cloud, it is impractical to develop and maintain in a traditional conservative and slow fashion. However, how to better assess changes before pushing them to global scope requires solutions beyond traditional testing.

Interestingly, as observed in our dataset, the changes that lead to large-scale failures are often small changes, especially those changes that developers assumed to be safe. But when rolled out, these "safe" small changes caused catastrophic impact. The most common type of this kind of small change with big negative impact is misconfiguration, e.g., the removal of a single IP block [32], traffic shift [3], change message sampling rate. Part of the reason that many failure-inducing changes are small is that big changes often go through more thorough code review and testing while small changes are often some activities that are so frequently performed that developers become less cautious about them. For example, in Google's recent global outage, "*by itself, this sort of change was harmless and had been performed previously without incident*" [32]; in AWS's catastrophic EBS outage, "*a network change was performed as part of our normal AWS scaling activities*" [3].

2.4.3 Single point of failure is rare

In traditional systems, failures are often caused by a single defect, e.g., a null pointer dereference bug crashing MySQL server, Nasdaq halt due to data link failure [49]. In cloud-scale systems, at component-level, single-defect failure is still com-

mon. However, at service-level, it is to see widespread failures caused by a single defect. The level of resilience comes from cloud system fault-tolerance by design philosophy: with modularity, redundancy and coordination, the impact of a single fault is restricted. Instead, many of the cloud failures that we examined result from a combination of multiple faults, e.g., disk flakiness plus incorrect recovery plus misconfigured protection mechanism [39].

Misconfiguration is an exception, though. We see from many failure incidents that while the examined industrial-strength cloud systems are robust against single hardware fault or software bug, they are fragile when facing misconfiguration: in the dataset it is rare to see a single software bug bringing down the entire service across regions; and almost all of the high-impact failure incidents that caused *all-region* outages such as the recent Google one [32] are caused by some configuration error. In these cases, the system designers did not foresee that misconfiguration was the single point of failure in the system. For example, even though the storage systems are replicated and geo-distributed to tolerate fail-stop faults, all storage servers might be using the same service certificate and therefore a single misconfiguration in the certificate can impact all HTTPS traffic [40].

2.5 When Faults Were Not Tolerated

Faults are the norm in a cloud environment. For example, thousands of disk faults can occur in the first year of a new cluster [75]. Therefore, fault-tolerance is necessary for high availability. Towards this end, a wide body of work has been proposed to tolerate faults in different layers (*e.g.*, [63, 112, 99, 69, 64, 96, 53, 93, 91, 105]).

Table 2.4. Representative fault tolerance techniques used by cloud practitioners.

Technique	Type	Example
Redundancy	Physical	redundant PDUs, routers, servers
	Temporal	timeout and retry
	Information	data replicas, checksum, erasure coding
Protocol	Primary/backup	DNS servers
	Paxos RSM	fabric controller, database
Load Balance	ECMP, VLB	VL2 [88], B4 [92]
	S/W load balancer	Ananta [109], Maglev [80]

2.5.1 Fault Tolerance Techniques

A fault tolerance mechanism is usually designed for a particular fault type. There are two common fault types: *fail-stop* and *Byzantine* faults. The fail-stop model assumes that components either behave correctly or stop. A Byzantine fault is when the faulty component behaves arbitrarily due to hardware errors [7], race conditions and/or malicious software. Byzantine faults are harder to tolerate than fail-stop. Other fault types include *omission faults*, in which the component fails to respond to requests due to crash(es) or lost messages, *response faults*, in which the components respond incorrectly, and *fail-slow*, in which the component responds correctly but too slowly.

Table 2.4 lists some representative fault tolerance techniques used by cloud practitioners. We include load balancing techniques for discussion about fail-slow. Although load balancing does not guarantee fault tolerance, proactive load balancing can alleviate fail-slow by offloading the requests to less loaded replicas.

Redundancy is typically key to achieving fault tolerance. Three forms of redundancy exist: *physical redundancy* that duplicates functionality in multiple resources; *temporal redundancy* for tolerating transient or intermittent faults by repeating requests in case of faults (*e.g.*, TCP retransmission); and, *information redundancy* that provides

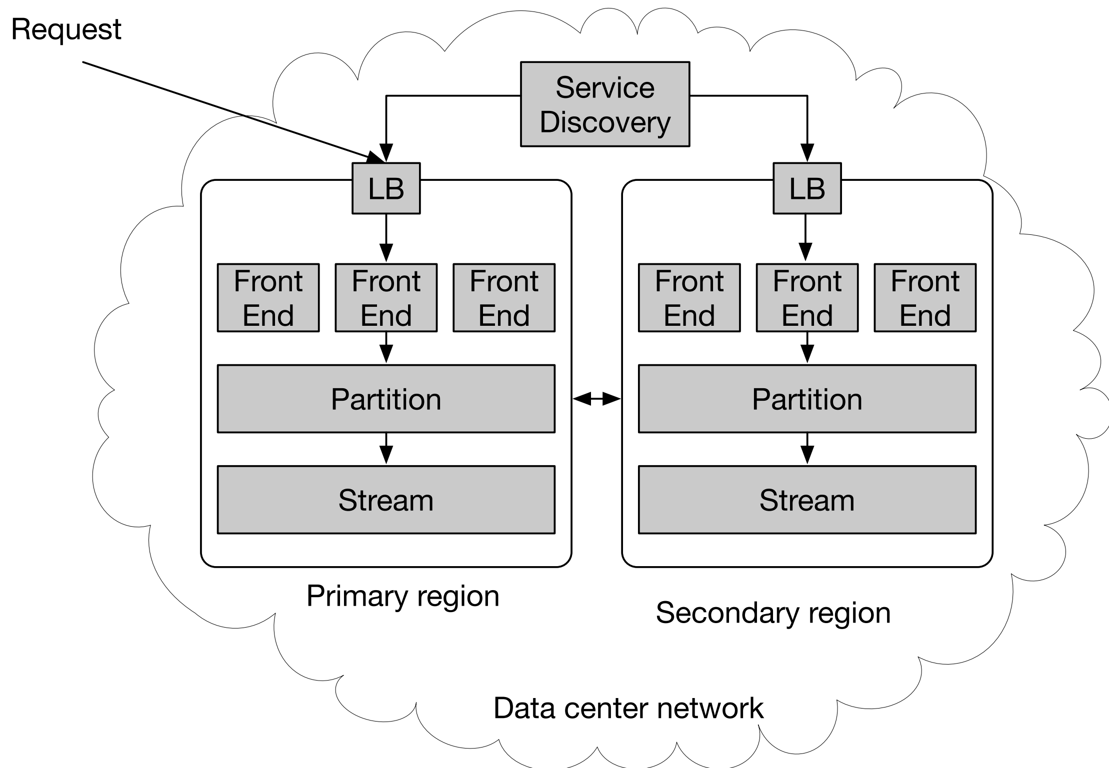

Figure 2.1. Fault tolerance in different layers of a simplified cloud storage service.

redundant data or information related to data (*e.g.*, error correcting codes). With redundant components, coordinating replicas is necessary. Paxos [99] and Byzantine fault tolerance [100, 69, 96, 105] are common consensus protocols that tolerate fail-stop and Byzantine faults respectively. A system is said to be t fault tolerant if it can continue providing service if no more than t replicas fail.

These tolerance techniques are used in different layers. For example, in a typical cloud storage service, a request is usually routed via the network to the load balancer in the primary region and dispatched to one of the front end servers. The request then goes through the storage partition and stream layers. In this scenario, redundancy exists in the network, load balancer, front end, partition layer and stream layer to defend against possible faults. The service can also be failed over to another region when there is a region-wide failure (Figure 2.1).

2.5.2 A Taxonomy of Fault-Tolerance Failures

With abundant fault tolerance techniques in place, why do failures still occur? There are three top level reasons:

1. Faults occur in all or most replicas, so replication-based fault tolerance simply cannot handle them.
2. The faults can be tolerated but the fault tolerance mechanism is not triggered.
3. Fault tolerance is triggered but is ineffective.

Different improvement strategies are needed for different cases, which also depend on the sub-categories within each case. Based on the studied service disruptions, we present our taxonomy and summarize the distribution in Figure 2.2.

Untolerable faults

Even though fault tolerance techniques can defend against various faults, some are not tolerable no matter how high the redundancy. In replication-based fault tolerance systems, deterministic faults or faults that happen too frequently or with no way for replicas to tell that they are faults are *untolerable*.

From the public reference dataset, we find a significant number of cloud failures occurred due to untolerable faults. These failures cannot be mitigated by restarting, fail-over or retrying. The faulty components must be repaired, *e.g.*, with a bug or misconfiguration fix.

For these untolerable faults, we can further break them down by different fault types as shown in Figure 2.2. The fault type could be a repeated omission fault (*i.e.*, failure to respond), specifically a persistent connectivity issue, repeated crash or hang. The persistent connectivity issues, often due to misconfiguration, can affect all redundant paths. For repeated crashes/hangs, even if the system automatically fails over in

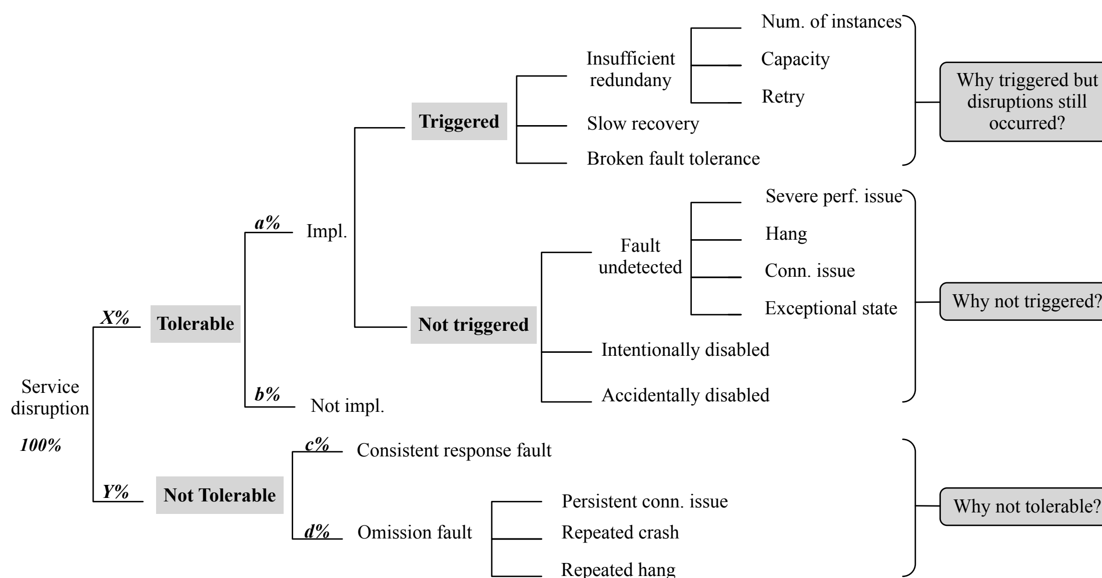

Figure 2.2. Taxonomy of why faults may not be not tolerated, resulting in failures. The branches can annotated with occurrence rates. "Response fault" means responding incorrectly. "Omission fault" means not responding. "Exceptional state" means an exception event (*e.g.*, error message) was generated but not leveraged by the failure detector. Note that there were probably several orders of magnitude more faults that are handled and do not result in failures, but we have no visibility into them and in any case they do not belong in this taxonomy.

the face of a crash/hang, the same fault will soon be triggered in the new primary and cause frequent fail-overs, resulting in a failure.

Except for repeated omission faults, the remaining untolerable faults are consistent response faults (*i.e.*, responding/behaving incorrectly). Examples of this type include the Active Directory service returning wrong tokens and IPs getting reallocated. Since these faults happen deterministically due to permanent defects like code bugs or misconfigurations, even Byzantine fault tolerant systems could not tolerate them: each replica behaves consistently and incorrectly. N-version Programming [57] might make some untolerable faults tolerable, but we are not aware of its use in any cloud service.

In terms of root causes for untolerable faults, we find from the public service outage cases that misconfigurations were a common source. This is because configuration is often applied to all replicas and the effect of misconfiguration is persistent. For example, when the same SSL certificates for the storage service were used in all servers across regions, misconfigured certificates could impact all HTTPS traffic [40]. As another example, an erroneous configuration change made to all front ends can result in crash of all web server processes. The misconfiguration must be corrected to recover the service. In a way, the misconfiguration poses a single point of failure in the system. In comparison, only 1 case was caused by permanent hardware faults in all replicas.

Since almost all fault tolerance techniques employed today are replication based, they cannot tolerate those faults that are deterministic and duplicated to all replicas. The primary way to deal with untolerable faults is to eliminate the defects before they're able to manifest as faults. To further reduce their incidence, it may be necessary either to increase diversity [57, 118] (*e.g.*, different communications, mechanisms, implementations), or to further improve software engineering or process (even better tooling, still more careful development practices, increased thoroughness in testing and/or rollouts that are staged more gently than they already are) to catch the problems (both in bina-

ries and configurations) before they become faults [90].

Fault tolerance not triggered

Within the tolerable faults, service disruption could still occur because the redundancy that could have helped was not in place. But since redundancy and replication is an overreaching theme in cloud system design, the case of no redundancy in place rarely happens. The other reason for service disruption to still occur in spite of tolerable faults is that fault tolerance was not triggered as shown in Figure 2.2.

Improper failure detection was often the reason fault tolerance was not triggered. Failure detection is an important, yet often overlooked [101], part of fault tolerance and recovery. The inability to detect component health correctly or in a timely manner prevents fault tolerance from taking corrective action. One consequence we see from public cloud service outage cases is that when an unhealthy node was incorrectly reported as healthy, the same unhealthy node would be used to service certain functionality again and again, each time resulting in failures.

The standard way to gather component health information is through heartbeats and leases [86]. If these failure detectors are not run frequently enough, they may generate false negatives. For example, an improper configuration change in a DNS server may result in NIC restarts, which would cause intermittent network failures. But if the NIC restarts were frequent enough, the heartbeats may not be missed. A failure detector can also be too-coarse grained: the components are in a bad state but still appear healthy to the detector. For example, in a typical cloud storage system, faults were detected both by the table master maintaining a heartbeat to the table servers and the table server maintaining lease with a lock service. When either the heartbeat or the lease is consistently lost, the system will kill the table server and offload the partitions to other table servers. It could be the case that a table server was in degraded states but still appeared

healthy because it maintained both heartbeats and leases. Fine grained detection [101], or determining health from different layers could be useful to detect these faults.

A common reason that faults went undetected is the failure was severe performance problems (*e.g.*, 95% CPU utilization). The faulty components in these failure cases would exhibit high resource usage. To monitor these faults, performance counters need to be proactively collected. When high resource usage is detected, to quicken recovery, consider fail-fast (*e.g.*, reboot) and automatic shifting of load rather than just throttling requests.

It is crucial to prevent cascade failures in fault-tolerant systems. These kinds of failures happen when fault-tolerance actions induce more faults, which in turn trigger more fault-tolerance actions. Cascade failures can turn a small failure into a complete service outage. One technique to prevent them is a "circuit-breaker" that disables fault-tolerance when it has been too active, and asks for help from the operations staff. For example, the circuit breaker technique was helpful in making the Netflix API more resilient [34]. However, if the circuit breaker engages, it disables fault-tolerance and can result in tolerable faults turning into failures. These cases would fall into the "Intentionally disabled" category in Figure 2.2.

Besides, fault tolerance in some failures was not triggered because of code bugs or misconfigurations.

Fault tolerance triggered but ineffective

In a industrial-strength cloud service such as AWS, Microsoft Azure or Google Cloud, the fault tolerance mechanisms in place were usually triggered for tolerable faults, but they were unsuccessful in preventing the failures mainly due to three reasons.

Slow recovery: In some cases, the fault tolerance in place succeeded in mitigating the failures. The systems self-recovered without manual intervention (*e.g.*, automatic fail-

over, quorum reestablished), but the detection and recovery process took too long to prevent the failures from being visible. With enough number of such self-recovered cases, an analysis on the statistics of the recovery time for may expose the systematic flaw in the service design. A long recovery time could be due to the large number of impacted replicas, intermediate stale data, or inefficient protocols. For these cases, a more efficient fault tolerance implementation or better protocols were needed to automatically restore the services without the user noticing the faults.

Insufficient redundancy: In some cases, the redundancy level in the employed fault tolerance was insufficient to tolerate the number of faults that occurred. For example, in a 5-instance quorum, faults happening in 3 instances exceeds the level of tolerance. Insufficient redundancy also includes insufficient retries or capacity. Figure 2.2 shows the breakdown of what was lacking for these cases due to insufficient redundancy. Determining appropriate redundancy levels entails effective capacity planning and analysis of historical failure rates and patterns.

Broken fault tolerance: In the remaining cases, the tolerance mechanisms failed to work properly. Code bugs and misconfigurations in the fault tolerance path are often the root causes. Figure 1.1 is an example of misconfiguration breaking fault tolerance: when the fabric controller incorrectly carried out a reformat action for a node, a mechanism should protect the node provided that it is configured so. If a node is mistakenly configured not to be protected, the mechanism will not prevent the problematic reformat action. As expected, fault tolerance related code or configuration is usually much less well tested because it would require extensive fault injection and recovery testing [89].

2.6 Were Faults Contained?

Cloud systems consist of many dependent and interactive components that function together to provide services. Faults in one component can impact others. Therefore, understanding failures in a cloud environment requires a whole system view instead of a per-component view as taken in prior studies [85, 120, 123, 83]. Such a whole system view of whether/how faults propagate is important for enhancing fault containment of the system. In this section, we look into the fault propagation pattern of cloud service disruptions.

2.6.1 Visualization with Impact Graph

We find a useful tool in understanding fault isolation and which component interaction needs hardening is to visualize the fault propagation information in an impact graph of internal cloud components. Figure 2.3 is a *contrived* impact graph that is not specific to a particular cloud service provider.

Such impact graph can be extracted manually from postmortem reports of cloud service failures. Automatically constructing such graph with failure-path inference techniques as proposed by Candea et al. [67] is a more efficient way to continuously generate the graph via fault injection.

The impact graph can be applied to a subset of services, coarse-grained components (e.g., storage and load balancer), fine-grained sub-components (e.g., front ends in storage), or even a particular type of failure. The differences in the resulted graphs may reveal specific fault propagation patterns. For example, we can draw an impact graph for just misconfiguration related failures. Compared to the impact graph for all failures, this impact graph may make some component more predominant in propagating faults while leaving other components less evident.

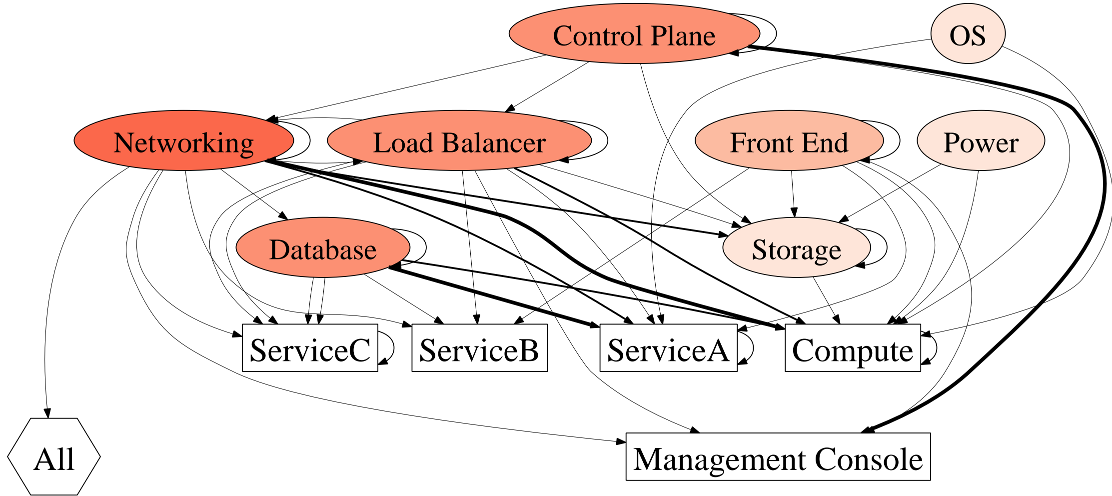

Figure 2.3. A contrived example of impact graph for analyzing fault isolation. Each vertex in the graph represents a major component of an interested service. The darker the color of a component, the more peers it affected. An edge $A \rightarrow B$ means failures in A impacted B. Weight can be added to the edge to denote the percentage or number of cases. A self-pointing edge means in some failure, the failed component only impacted itself.

Based on the public reference dataset, we find that while cloud system builders take careful efforts of enforcing fault isolation in design (e.g., divide data center resources into logical clusters that are managed by different instances of controller components), faults in a major component still frequently propagated to other components.

The cross-component impact happens due to dependencies on *data* (*e.g.*, a compute service uses a database), *control* (*e.g.*, storage needs the lock service for coordination), *connectivity and utility* (*e.g.*, power). For service disruptions contained within a component, the component is usually an external service.

2.6.2 Fault propagation length

Except for the impact graph, the analysis can further zoom into the distribution of number of major components impacted across the service disruption cases. We find from our public service failure dataset that most faults only impacted a small number of major

components, including the component where the fault originated. This means while faults propagated to other components in the system, the length of such propagation was limited. A short propagation path usually indicates a system is carefully decoupled and thus reduces unexpected cross-component interactions which are common in tightly coupled systems [113].

To be more effective in confining faults, further reducing dependencies among components (*e.g.*, stateless services) is one way. But if the dependency is inevitable, hardening the interaction boundaries and having cross-component monitoring will be helpful [90] so that when the dependent component(s) are down, the depending component(s) can take some corrective actions (*e.g.*, fallback to temporary storage) [34]. In terms of testing, major components are often managed by different teams and tested independently. Reducing cross-component failures calls for more joint testing of dependent components.

2.7 What Caused These Failures?

When building reliable services, it is useful both to tolerate faults and to remove them altogether. Understanding the root caus(es) that give rise to faults is helpful for both. In this section, we investigate what caused the faults that underlie these studied failures.

2.7.1 Root Cause Type

Common root causes for failures in traditional systems include software bugs, misconfigurations, hardware faults and human errors. Other root causes that arise in cloud environments are power losses and lacks of capacity.

A failure can be the result of multiple root cause types. In our public reference dataset the majority of multi-cause cases involve either software bugs or misconfigu-

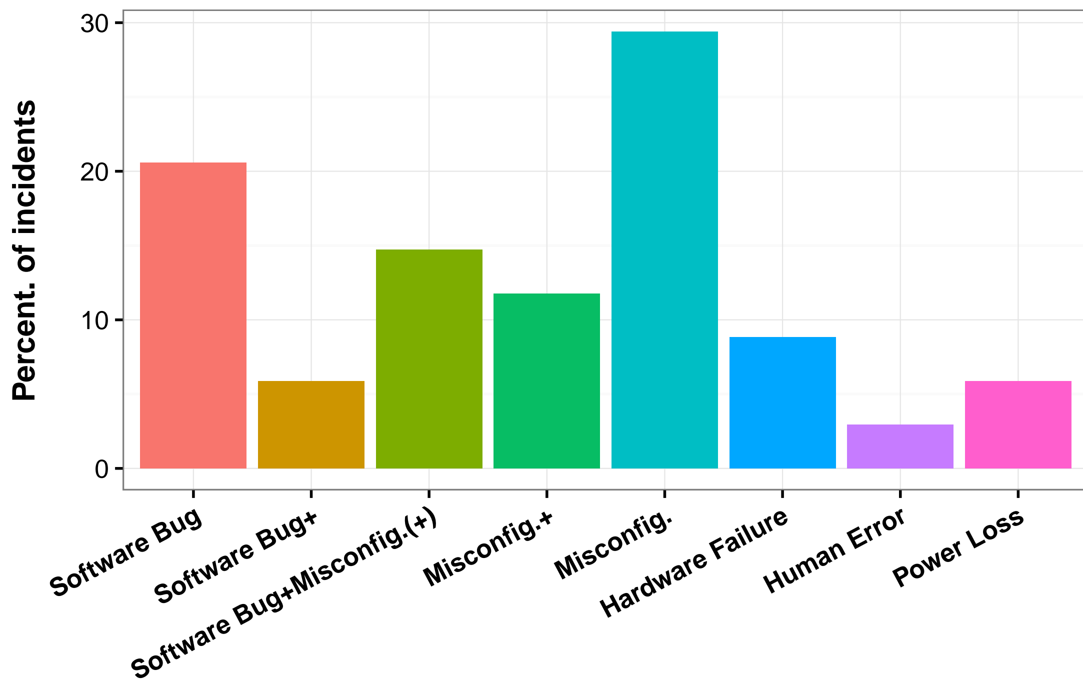

Figure 2.4. Root cause distribution for the 34 public cloud service outages that we investigated. Categories with "+" in their name represent incidents with multiple root cause types.

rations. Therefore, we simplify counting the combination of cause types using only software bugs and misconfigurations. To be specific, we first categorize cases involving both software bugs and misconfigurations, and optionally other cause types into "*Software Bug+Misconfig.(+)*". The remaining multi-cause cases will be put into "*Software Bug+*", "*Misconfig.+*" or "*Other*". Based on this counting method, Figure 2.4 shows the root cause type distribution for the 34 public cloud service outages.

We call a cause type a *single-contributor* if it is the only cause type for a failure, and *co-contributor* if it is at least one cause type for a failure. The first and fifth bars in Figure 2.4 are the single-contributor software bugs and misconfigurations, respectively. The cases where a software bug is a co-contributor are the sum of the first to the third bars. Similarly misconfiguration as a co-contributor is the sum of the third to fifth bars.

Based on the public reference dataset, we find misconfiguration is a dominant single-contributor and co-contributor to the collected public cloud service outages, especially for the outages with servere impact. This suggests that misconfiguration is an important issue in the cloud environment. Hardware faults were responsible for only a small number of the failures. This reflects the resilience in industrial-strength cloud systems.

2.7.2 Multiple Root Causes

One characteristic of fault-tolerant systems such as aircraft is that a system-level incident is often a result of multiple root causes. This is also the case for industrial-strength cloud systems. A significant number of cloud failures today are a result of multiple cause types. Even for those cases with a single cause type, there can be multiple causes (*e.g.*, two code bugs).

There are two common patterns of multi-cause failure. *Concurrent causes* are separate causes (*e.g.*, improper local preference and wrong community strings in BGP routers) that in combination lead to the failure. Concurrent causes need to be investigated and addressed independently. In *chained causes*, multiple causes form an ordered sequence. An initial cause leads to or exposes another cause.

For chained causes, fixing one of the causes can mitigate the issue. A more tricky pattern is a *compound* of concurrent and chained causes. These three patterns are illustrated in Figure 2.5. Our experiences in reviewing the public cloud service outages suggest that the majority of multi-cause incidents are chained and compound causes, partially due to the interactive nature of cloud systems. This may indicate opportunities to model and prevent multi-cause incidents.

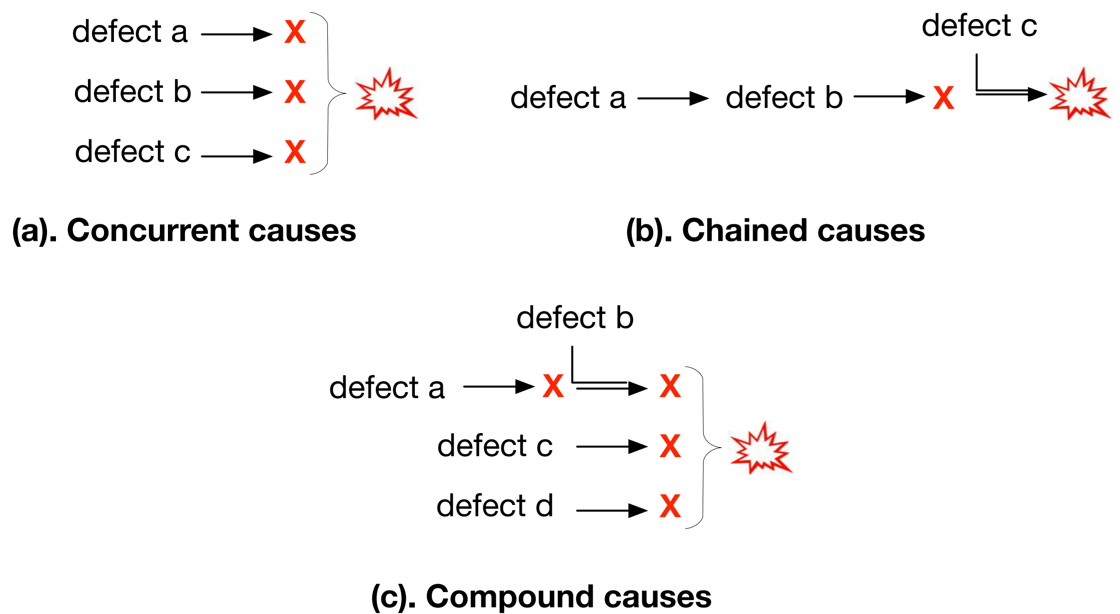

Figure 2.5. Patterns of how multiple root causes contribute together in an incident.

2.8 Zooming in on Misconfiguration

After understanding why faults were not tolerated and what caused the failures, it is clear that misconfiguration was a major source of the untolerable faults, of high-severity outages cases, and of the unavailability incurred by service outages. These facts motivate us to further look into misconfiguration in cloud service failures.

2.8.1 What Components Were Misconfigured?

From the publically disclosed cloud service disruptions, we can find that a significant percentage of the misconfigured components were network elements such as routers and software load balancers. For example, most of the public Google cloud failures are caused by misconfiguration in networking.

This phenomenon comes from the fact that configuring network elements relies heavily on the orchestration of correct configuration for various aspects (*e.g.*, interface, routing, ACL) and other network elements. In a way, configuring a network is like

Table 2.5. Operation that introduces misconfiguration

<table>
<tr><td colspan="2">Update</td></tr>
<tr><td rowspan="2">None</td><td>Missing update</td></tr>
<tr><td>Long-standing</td></tr>
</table>

writing a distributed program. Yet the available network configuration primitives are often low level (*e.g.*, route maps), which makes configuring a large-scale network a daunting task even for trained operators [62]. Moreover, compared to other components, network elements often have low redundancy but high impact on connectivity.

Misconfigurations in management nodes affect their functioning (*e.g.*, broadcast, recovery). Although more configuration errors lay in regular nodes than management nodes, misconfigurations in management nodes can have large-scale impact. For example, a misconfiguration in the network manager could cause incorrect DNS suffixes to be issued to all rebooted VMs.

An often overlooked type of cloud components that can get misconfigured are the monitoring nodes, which are used for end-to-end failure detection. Misconfiguration in these nodes can prolong the detection times, which in turn delayed mitigation and recovery.

2.8.2 What Introduced the Misconfiguration?

The tediousness of conducting similar configuration tasks has motivated many automation tools that generate and manage configuration entities. Bugs in these tools then become another source for misconfigurations. For example, a recent service outage in Google was caused by a bug in the configuration generation system [33]. We suspect with more configuration automation tools, the percentage of bug-induced misconfigurations may go higher (though the total misconfigurations may be reduced). How to detect bug-induced misconfiguration remains to be explored.

Another aspect of looking at what introduced misconfiguration is through the operations as listed in Table 2.5. Often cloud misconfiguration happen due to both configuration changes that occur frequently in the dynamic cloud environment. For example, when there are workload increases or hardware updates, a session limit parameter may be adjusted. To detect changes related misconfigurations, tools analyzing configuration change impact will be useful. It is also necessary to track configuration change and ownership so when failures happen, they can be fixed quickly. For example, failures due to problematic network configuration changes usually can be quickly fixed by reverting the changes. Knowing who made the changes and what the change was will help expedite failure handling.

Except for change-induced misconfiguration, cloud misconfigurations can also be long standing or occur due to missing update(s). Long-standing misconfigurations exist in the system for a while but do not cause immediate impact when they were introduced (*e.g.*, protection misconfigurations in Figure 1.1). For the missing update category, the configuration is initially good but becomes invalid when the environment changes or its validity expires (*e.g.*, a temporary timeout increase to accommodate planned maintenance downtime persists after the maintenance). These misconfigurations emphasize the need to automatically detect what configuration should be changed when the environment changes as well as to periodically clean up legacy configurations.

2.8.3 What Constraints Were Violated?

Misconfigurations happen because some inherent constraints are violated. Based on the public cloud service outages as well as misconfiguration from prioer studies, we identified several constraints commonly violated as described in Table 2.6.

The simplest type of misconfigurations were format violations. These misconfigurations are easily eliminated with operator training, configuration review and regular

Table 2.6. Violated constraints in the studied misconfigurations

Constraint	Description	Example of Constraint Violation
Format	Basic lexical and syntax requirements.	`partition = 1` (extra spaces)
Consistency	When similar configuration entries exist in multiple places, the settings should be in agreement.	Cluster A in some config. file was incorrectly considered as part of cluster B while in other files it was not. This caused wrong (B's) certifate to be picked up in deploying cluster A.
Dependency	A configuration depends on other configuration(s) that may lie in different components.	New IPs were added to cluster. But ACLs in edge routers were not updated.
Scope	The configuration is only applicable to a particular environment or scope.	Standby node mistakenly inherited some settings from active node, causing the standby to also process requests and silently drop the result.
Time	The configuration is legitimate initially. But it has a validity time limit.	A scheduled maintenance to migrate circuits exposed the legacy config. in a router, causing intermittent failures.
Global	The effect of configuration is only meaningful in the context of peers' settings.	Local pref. of a subnet address was set to 400, while other addresses had it set to 100. It attracted overwhelming traffic.
Intention	Default constraint: the effect of configuration matches what is expected.	All tenant VLANs were configured with DHCP relay pointing to a few management nodes, causing all DHCP requests to be forwarded to these nodes.

testing. But even with these practices, the exact effect of a configuration setting can still be difficult to grasp. This is why the biggest violation is the intention violation. Checking intention violation requires radically new configuration languages and systems to allow expressing high-level intention [79, 78].

A common type of misconfigurations in cloud environment is a violation of some consistency constraints. The consistency constraints refer to consistency of the same configuration entries in multiple instances (*e.g.*, BGP export policies among edge routers should be consistent) as well as the consistency of seemingly different configuration entities that contain the same information (*e.g.*, a host name appearing in both the database connection string and the hosts file should be consistent for the name to be resolved). Inconsistency can be reduced with better configuration management that centralizes and keeps configurations up-to-date in all instances/entities and monitors these configurations in case of out-of-band edits.

A constraint related to consistency constraints is cross-component configuration dependency constraint. These dependency requirements often lay across components. For instance, the rule for bypassing NAT to support direct communication between private IPs depends on the ACLs in the destination components. Tracking configuration dependencies across components demands extensions to existing techniques [56, 128].

At cloud scale, it is inevitable to have different software versions, hardware (*e.g.*, router models), and cluster types (*e.g.*, storage and compute clusters). As a result, the configuration that gets deployed might be for a different environment or software version, i.e., violating scope constraint. In one case, the configuration for software in version A was applied to version B binaries, which enabled a feature with a known bug in the version B binaries resulting in a crash. These cases require configuration tagging and versioning and tooling enforcement to check that the configuration is compatible with the target environment.

Other violated constraints include time and global constraints. A certificate is a typical time-constrained configuration. Temporary configuration workarounds also have validity time limits. An example of a global constraint is the local preference configuration in BGP. The effect of this setting is determined by peers. The route with higher local preference will be preferred over other routes.

2.9 Conclusion

Cloud service disruptions are costly to both end users and vendors. As more systems evolve towards cloud scale, studying cloud service disruptions becomes more important. This chapter makes an attempt to examine 34 notable cloud failure incidents that we collected using public information. We take a novel perspective of revisiting the pervasively used fault tolerance mechanisms in cloud and analyze why failures still occurred despite abundant fault tolerance.

We find that many cloud failures occurred due to some faults that cannot be tolerated by existing replication-based fault tolerance techniques. A major cause of these "untolerable" faults were misconfigurations. For these faults that could have been tolerated, fault tolerance in place may not be triggered often because the failures were not detected. For the remaining cases where fault tolerance was triggered, but still did not successfully prevent the service disruptions, the reasons could be that the self-recovery was too slow, redundancy was insufficient or the fault tolerance mechanisms were broken. We also zoomed into misconfiguration as it was often a major cause of the untolerable faults and service unavailability in failures from multiple providers.

2.10 Acknowledgements

Chapter 2 contains material of a paper retracted from 11th USENIX Symposium on Operating Systems Design and Implementation 2014. Huang, Peng; Jin, Xinxin;

Bolosky, Bill; Zhou, Yuanyuan. The dissertation author was the primary investigator and author of this paper.

Chapter 3

CPL: A Configuration Specification Language

3.1 Introduction

Cloud-scale systems today use a wide variety of configuration entities to control different features and components. These configurations are further duplicated and customized for different deployment environments (Figure 3.1), creating a large volume of configuration data. A misconfiguration can affect the entire system. For example, misconfiguring the endpoints of load balancing components could cause all traffic to be directed to one server, overwhelming that server and effectively making the whole service unavailable; misconfiguring the backup options of a redundant service pair could cause the backup to be useless should the primary go down. It is not uncommon to see misconfiguration(s) being the root cause(s) for service disruptions in today's highly fault-tolerant systems [25, 3, 16, 14, 39, 40, 27, 50, 33], causing considerable financial cost [20].

Misconfiguration is a thorny issue. Many solutions have been proposed to attack it ranging from misconfiguration detection [130, 131], diagnosis [125, 124, 126, 56] and repair [121, 97] to system resilience [94, 128]. These solutions greatly alleviate end users' pain in configuring unfamiliar software. However, they are often post-mortem

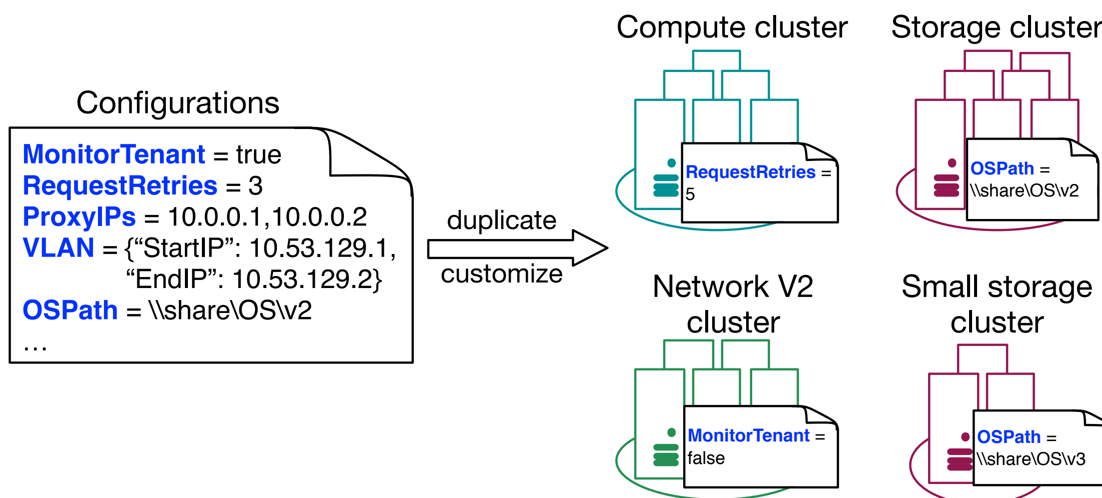

Figure 3.1. Configuration data in cloud systems.

efforts or incur overhead (*e.g.*, instrumentation, record and replay) that is too expensive to deploy for evolving cloud-scale production systems.

Handling misconfigurations in cloud systems needs to be both proactive to prevent service downtime, and lightweight to continuously operate as systems undergo frequent changes. Deployment testing is a proactive solution that can prevent misconfiguratons from being introduced into production. But due to resource constraints, not all configuration updates will be tested before rolling into production. Additionally, some misconfigurations are latent (*e.g.*, misconfiguration in fault-tolerance options) and therefore can escape testing. Even when a misconfiguration is uncovered during deployment testing, the setup and roll-out of the bad deployment at cloud scale is costly.

Configuration validation checks if configurations satisfy some specification, *e.g.*, a file path that should exist, an IP range that should not overlap with others, a timeout that should be consistent with others. The earlier and more thorough the validation, the less likely misconfigurations would damage production services. Moreover, when a misconfiguration is detected, the pre-defined specifications and validation results can help pinpoint which part of the configuration is problematic. Configuration validation is

complementary to deployment testing.

Configuration validation is a feasible practice for cloud services because the systems are operated by dedicated, trained staff who collaborate with developers (*cf.* DevOps movement [117]). These practitioners have expertise to understand the constraints of some configurations, *e.g.*, proxy endpoints should be HTTPS if the SSL option is enabled; disabling security token and setting token service endpoints could cause an authentication outage. Practitioners use their expertise to validate configurations by constantly reviewing configuration changes before the changes are applied. They also write plenty of code and scripts to programmatically validate configurations.

However, the current configuration validation practice is inefficient and *ad hoc*. The manual configuration reviews are time consuming. Validation code is scattered in different code regions and sometimes invoked too late at runtime. The validation code is bulky and hard to maintain. Practitioners often waste time writing similar checks. Writing validation code becomes a reactive effort, *e.g.*, after incidents occurred.

We believe that by providing the right tools to practitioners, configuration validation can be made an ordinary part of cloud-scale system deployment and can proactively prevent misconfiguring production services. Towards this end, we present *ConfValley*, a systematic framework for practitioners operating cloud-scale systems to efficiently conduct configuration validation.

The challenges in building such a validation framework lie in how to allow practitioners to express configuration specifications easily and precisely; how to minimize tedious manual efforts so practitioners have incentives to conduct validation; how to avoid investing repeated efforts on similar validations; and, how to run validation efficiently on a large volume of diverse configurations in evolving environments.

At the core of ConfValley is a simple, declarative configuration validation language, *CPL*, to describe various specifications easily. The language decouples the core

validation logic from implementation details, allowing the specifications to be described compactly and independently of the underlying configuration representations. The benefits are that validation code become maintainable, modular, parallizable, and adaptable to various configuration sources.

Even with a compact language, writing all configuration validation specifications from scratch can be tiresome. Therefore, ConfValley contains a component to automatically infer and generate specifications. In this way, experts can focus on writing complex or domain specific specifications that are hard to infer. Additionally, auto-inference makes it feasible to keep the specifications up to date as the systems evolve.

Our experience in using ConfValley inside Microsoft Azure [35] as well as on two open-source cloud systems shows that using the framework for configuration validation is much easier and systematic compared to prior practice. To be specific, the *ad hoc* configuration validation code that was used in Microsoft Azure could be expressed in our new language with more than a 10x reduction in lines of code. For example, a previous validation module with more than 3000 lines of code became only 109 lines of code in 62 *CPL* specifications, out of which 27 specifications could be automatically inferred. The new concise validation code is more declarative and easier to read. Second, the inference component in ConfValley infers thousands of specifications with high accuracy. With the inferred specifications, we validated the latest configuration snapshot in Microsoft Azure and reported 43 violations, 32 of which were true configuration errors. With specifications written by experts, ConfValley reported 8 configuration errors, all of which were confirmed.

3.2 Background and Motivation

In this section, we introduce characteristics of configurations in cloud systems, practices of configuration validation, and the issues in these practices.

3.2.1 Configuration in cloud systems

A wide variety of configurations are used in cloud systems to control features, endpoints (*e.g.*, cache server addresses), security, fault tolerance, tunable behaviors (*e.g.*, timeouts, throttling limits) and so on. These configurations lie in different system components and software stacks. Their representations also vary (*e.g.*, XML, key-value, .INI files). In Microsoft Azure, there are many thousands of configuration entities in tens of different representations.

Configurations in cloud systems are intertwined. Improper changes to a configuration in one place can affect the correctness of configuration(s) in other places. In these cases, cross-validating configurations across different sources is useful. For instance, account configurations need to be consistent across controller and authentication components.

Additionally, configurations in cloud systems are heavily replicated and customized for different deployment environments to tailor them to heterogeneous infrastructure, services, workloads and customer needs. This replication and customization creates notions of *configuration class* and *configuration instance*, which are comparable to class definition and instantiation in object-oriented programming. In Listing 3.1, `MonitorNodeHealth` is a configuration class that has instances in each of the 4 `Tenant` scopes (line 5, 8, 11, 18). The ratio of configuration instance to configuration class is as high as 80:1 to 14,000:1 in the repository of static configuration data in Microsoft Azure.

3.2.2 Configuration validation

Configuration validation is the process of checking a configuration against some explicit specifications. The specifications impose constraints on configurations, *e.g.*, parameter *A* should be a file path that exists, parameter *B* should be smaller than parameter

```
<CloudGroup Name = "East1 Production">
  <Setting Key = "MonitorNodeHealth" Value = "True">
  <Setting Key = "ControllerReplicas" Value = "5">
  <Cloud Name = "East1Storage1">
    <Tenant Type = "A">
      <Setting Key = "MonitorNodeHealth" Value = "False">
    </Tenant>
    <Tenant Type = "B" />
  </Cloud>
  <Cloud Name = "East1Storage2">
    <Tenant Type = "A" />
  </Cloud>
</CloudGroup>
<CloudGroup Name = "SSD Cluster">
  <Setting Key = "MonitorNodeHealth" Value = "True">
  <Setting Key = "ControllerReplicas" Value = "3">
  <Cloud Name = "East1Compute1">
    <Tenant Type = "A">
      <Setting Key = "ControllerReplicas" Value = "5">
    </Tenant>
  </Cloud>
</CloudGroup>
```

Listing 3.1. A snippet that represents configurations (`Setting` elements) at different scopes using XML. For example, `MonitorNodeHealth` is inherited by all `Tenant` scopes, some of which override the value to be `False`.

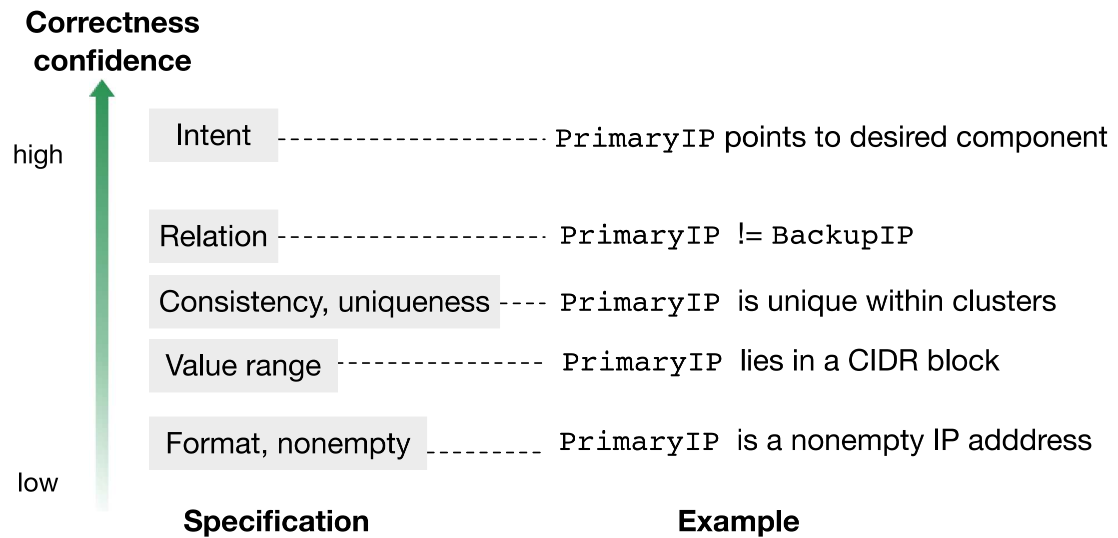

Figure 3.2. The spectrum of typical configuration specifications.

C.

Like other validation approaches, configuration validation is unsound in that it is unable to reject all invalid configurations. In other words, a configuration that fails any specification is invalid but a configuration that passes all specifications is not necessarily correct. For example, a parameter for the VM image path can pass the validation of the data type (file path), matching pattern (ends with `.vhd`), existence (path exists), consistency (same values across clusters), *etc.*, while still being the wrong version of the VM image.

Despite the unsoundness, having configurations validated with respect to various specifications can increase confidence in the correctness of a configuration in the same way as testing provides confidence in code quality. Typically, the overall value space for a configuration parameter is large, within which the space of correct values is small (and often just one). Validating configurations against various specifications shrinks the invalid value space and increases the correctness confidence. Figure 3.2 shows some typical types of specifications and examples.

The existing practice of configuration validation is often inefficient and *ad hoc*. There are time-consuming manual configuration reviews in which practitioners look for errors in each configuration update. There is also programmatic configuration checking, but the checking code is interspersed with other code in different regions. When invoked at runtime, some checks can be too late to prevent misconfigurations. Practices that use separate code and scripts to check configurations are often *ad hoc*. The validation code buries essential validation logic in details that are not very relevant to the validation. Consequently, the validation code becomes bulky and hard to maintain. For example, in Microsoft Azure, there are thousands of lines of validation code with high redundancy. The validation code in OpenStack [44] and CloudStack [10] is mixed with other code and scattered in different source regions.

3.3 Design Considerations

In this section, we examine the design trade-offs in making configuration validation an efficient and systematic activity.

3.3.1 Language support

The lack of efficiency and systematicness in existing configuration validation practices can be addressed by software engineering processes such as refactoring. But we observe that the issues are often due to the lack of better language support for configuration validation. In bulky *ad hoc* validation code, the validation logic is entangled with tedious implementation details and tailored for a specific configuration source or validation scenario.

For example, *ad hoc* validation code operates on configuration instances, while the validation logic essentially applies to configuration classes. This means the core validation code will be interspersed with code that discovers all instances for a configu-

ration class. For example, to check a simple requirement that `MonitorNodeHealth` is a boolean type, existing code will first find all instances of `MonitorNodeHealth` in all scopes as shown in Listing 3.2. In a cloud system, a configuration class can have a high number of instances in various scopes. The code to discover instances becomes tedious to write and obscures the essential validation logic, and may even harbor bugs itself.

```
// Check if MonitorNodeHealth class is boolean
Settings = Parse("setting.xml");
foreach (CloudGroup in Settings.CloudGroups) {
  foreach (Cloud in CloudGroup.Clouds) {
    foreach (Tenant in Cloud.Tenants) {
      if (!CheckBoolean(Tenant.MonitorNodeHealth))
        return false;
    }
  }
}
```

Listing 3.2. Validation snippet that operates on configuration instances rather than configuration classes.

Additionally, *ad hoc* validation code is fairly imperative and needs to prescribe *how* to implement each constraint. Listing 3.3, 3.4 and 3.5 show three imperative validation snippets taken from existing cloud systems to check simple constraints. For example, to check that a parameter is a list of IP addresses involves splitting the value and check if each part is an IP address. Also, to check uniqueness of different parameters, a set needs to be created and tested for each parameter like the snippet in Listing 3.5. With many configuration parameters and properties to be checked, imperative validation becomes clumsy.

While designing a completely new configuration language that provides inherent validation capabilities (*e.g.*, type-safety) would be revolutionary, the cost of such a solution, *e.g.*, changes to existing infrastructure to support it, makes it an elusive goal. We chose the less ambitious direction of refining the language used for writing config-

```
// Check if IpRanges is a list of IP ranges
bool passed = true;
string [] ranges = IpRanges.Split(';');
foreach (string range in ranges) {
  if (!IsIPRange(range)) {
    passed = false;
    break;
  }
}
```

Listing 3.3. An imperative validation snippet that prescribe the implementation details of a simple type constraint.

```
// Check if parameters are positive integers
configForValidation = new HashSet<String>();
configForValidation.add("event.purge.interval");
configForValidation.add("alert.wait");
Class<?> type = config.getType();
if (type.equals(Integer.class) &&
  configForValidation.contains(config.name)) {
  try {
    int val = Integer.parseInt(config.value);
    if (val <= 0) {
      throw new InvalidParameterValueException(
        "Enter a positive value for:" + config.name);
      }
  } catch (NumberFormatException e) {
    throw new InvalidParameterValueException(
      "Error parsing integer value for:" + config.name);
  }
}
```

Listing 3.4. An imperative validation snippet that prescribe the implementation details of a simple value range constraint.

```
// Check if address and location are unique
HashSet<string> ipList = new HashSet<string>();
HashSet<string> locationList = new HashSet<string>();
foreach (LoadBalancer loadBalancer in loadBalancers) {
  if (!ipList.Add(loadBalancer.Address)) {
    Console.WriteLine("LoadBalancer address {0} is " +
        "not unique: \t", loadBalancer.Address);
    DumpList(ipList);
  }
  if (!locationList.Add(loadBalancer.Location)) {
    Console.WriteLine("LoadBalancer location {0} is " +
        "not unique: \t", loadBalancer.Location);
  }
}
```

Listing 3.5. An imperative validation snippets that prescribe the implementation details of a simple uniqueness constraint.

uration validation code and adding correctness constraints on top of existing diverse configurations.

In the context of cloud-scale systems, the characteristics of the configurations (see Section 3.2.1) dictate several desired properties for a refined configuration validation language:

- **Scalable**: fast and easy to evaluate over a large volume of configuration instances
- **Representation-independent**: the validation logic is not tied to specific configuration representations
- **Expressive**: the ability to specify various configuration constraints easily
- **Precise**: the ability to precisely refer to the intended scope for a constraint
- **Modular**: easy to group constraints and compose constraints from existing modules

- **Extensible**: the ability to add new constraints
- **Debuggable**: debugging support when a validation fails

There are existing languages that provide some of these properties but fall short in others. For example, C# LINQ and XQuery provide convenient ways to query data, which is useful for finding target configurations to validate. But they lack inherent support for validation and domain knowledge of configurations. XML Schema Definition provides constructs to write rules to which XML documents must conform. But the types of validation that can be expressed are very limited (mainly formats). The language is also tied to XML documents and is complicated to use. While it is possible to add extensions to these languages to achieve the missing properties, designing a new domain-specific language specially for configuration validation is a cleaner approach.

3.4 Configuration Predicate Language

CPL is a simple, domain-specific language that makes validation specifications easy to write, systematic and maintainable, and which in turn encourages practitioners in cloud systems to put more validation in place, which in turn increases their confidence in overall system correctness.

The general design principle of *CPL* is to focus on validation logic rather than implementation. In particular, *CPL*:

- Refers to configurations conveniently
- Describes constraints declaratively
- Describes the scope of validation precisely
- Covers common constraint primitives

- Allows extensions to the language
- Encourages modular validation specifications
- Supports convenient debugging constructs

3.4.1 Concepts

Before describing the details of *CPL*, we first explain several of its key concepts.

Predicate. The essential construct in *CPL* is a predicate. A predicate is used to characterize a boolean property of some entity. For example, “X is an IP address”, “X lies in the range from 1 to 10”, “X is consistent”, “A is greater than B”, and “X has read-only permission” are simple predicates described in natural language. In *CPL*, we provide a set of primitives for common properties such as data types, relation among multiple entities, pattern matching, value ranges, consistency, and uniqueness.

We denote predicates using lower case bold letters, *e.g.*, ***r***, ***s***, ***t***. When evaluating a predicate over some entity, we use the form of a function over arguments like ***r***(x), in which ***r*** is a predicate and x is the argument.

With logical operators such as `AND`, `OR` and `IMPLIES`, a predicate can be defined recursively from other predicates. Therefore, ***s*** can be defined from (***r*** `AND` ***t***), (`NOT` ***r*** `OR` ***t***), or (***r*** `IMPLIES` ***t***). *CPL* also allows shorthand notations that are familiar to programmers. For example, (***r*** `IMPLIES` ***s***) can be written as (`IF` ***r*** `THEN` ***s***); (***r*** `IMPLIES` ***s***) `AND` (`NOT` ***r*** `IMPLIES` ***t***) is the same as (`IF` ***r*** `THEN` ***s*** `ELSE` ***t***).

Domain. A predicate describes properties of an entity. When there are multiple related entities to be tested for the same properties, *e.g.*, x, y, and z, instead of associating a predicate with each entity, *e.g.*, ***r***(x), ***r***(y), and ***r***(z), we can group these related entities into a domain and associate the domain with the predicate. A domain is the source that provides entities to evaluate one or multiple predicates.

In our specific context, a domain mainly refers to the values of a group of related configuration instances. For example, configuration class C is a domain and $\boldsymbol{r}(C) := x \in C \mid \boldsymbol{r}(x)$ is a predicate over all the instances of C. Users of *CPL* would be mainly concerned with defining domains like configuration class C using unified notation (see Section 3.4.2). The system will automatically try to find all elements that belong to the specified domain.

Transformation. It is often useful to transform an entity and then evaluate a predicate on the transformed entity. For example, suppose that predicate $\boldsymbol{r}(x)$ represents "x ends with .xml". But we may need to perform the test on values of x of mixed cases. To support these cases, we can apply a lowercase transformation function f over x first and then evaluate $\boldsymbol{r}$, *i.e.*, $\boldsymbol{r}(f(x))$. Without transformation, we would have to redefine a new predicate $\boldsymbol{s}$ that represents "x ends with .xml case-insensitively", even though the main logic between $\boldsymbol{r}$ and $\boldsymbol{s}$ are the same. Transformation improves the modularity and extensibility of predicates.

In addition to specific entities, transformation functions are also applicable to domains. There are two styles when applying a transformation function to a domain: the "map-like" style applies the transformation to each member in the domain (*e.g.*, split each member with a comma) and the "reduce-like" style applies the transformation to all members in the domain as a whole (*e.g.*, the union of all range-type members).

In addition to a single domain, transformation functions also may be applied to multiple domains. For example, transformation functions can be standard binary operators like + and - to connect two domains as a typical arithmetic expression. By default, a transformation over multiple domains will be applied to the Cartesian product of the members in these domains. We also provide a construct (see Section 3.4.2) to override this behavior. The transformed domain(s) form a new domain, which can be tested using predicates.

Quantifier. While a domain provides necessary arguments to a predicate, the quantity of elements in a domain that should satisfy the predicate can vary. For example, some cases require every possible argument value to satisfy the predicate while others require only one possible argument to satisfy the predicate. *CPL* provides quantifier construct to describe quantification for a predicate. $\exists$ means that there exists at least one argument in the domain that satisfies the predicate. $\forall$ enforces that every argument in the domain should satisfy the predicate, which is the default quantifier in *CPL*. $\exists!$ denotes that there should exist exactly one argument in the domain that satisfies the predicate.

3.4.2 Unified configuration representation

Since components in cloud systems are developed and typically managed by different teams, configurations for these components can be in diverse representations. For example, some use standard INI or YAML format, others use customized XML hierarchies, key-value stores or REST APIs. In *ad hoc* validation code, the validation logic is tied to the underlying configuration representations, which makes it painful to maintain and adapt to representation changes. To avoid such entanglement, our framework uses a set of drivers to abstract the diverse representations of configuration sources into a unified representation to expose to the validation engine and the validation language.

Consequently, domains in *CPL* are referred to with a consistent, representation-independent notation. In the simplest form of the notation, a configuration class is represented with a single key such as *SecurityConfigFile*. This key refers to all configuration instances in the underlying configuration source(s). Many configuration parameters in cloud-scale systems are organized and categorized based on factors like the components or features for which these parameters are used. Therefore, a more generic form of configuration notation in *CPL* attaches a scope to a key. This *qualified notation* allows practitioners to easily specify target configurations. For example, *Fabric.RecoveryAttempts*

represents the configuration class *RecoveryAttempts* in the component *Fabric*.

To map the language-level scope notation to the underlying configuration sources, our framework gets the scope information for configuration data in three ways. First, if the configuration data already encodes scopes in the configuration parameter name, out framework will directly extract the scope information. Second, if the configuration data is in a hierarchical format, our framework uses domain knowledge to encode the hierarchy. For example, the configuration parameter *MonitorNodeHealth* in Listing 3.1 can be parsed into a qualified notation *CloudGroup.Cloud.MonitorNodeHealth* based on its tree path. Third, the configuration source loading statement in *CPL* allows users to provide an optional scope to place before all the parameter names in the configuration source. For example, if users indicate that a configuration source comes from the *Fabric* component, the parameters in the configuration source will be prefixed by *Fabric*.

Similar to configuration keys, scopes in qualified notations can refer to multiple instances. For example, the scope *Fabric* can have multiple instances because of multiple *Fabric* components. *CPL* specifications primarily deal with configuration and scope classes. But there are cases where it is necessary to check for particular instance(s). To allow this precise specification of target configurations, *CPL* supports *fully qualified notation* with named styles (*e.g.*, *Fabric::inst1.RecoveryAttempts*) and numbered styles (*e.g.*, *Fabric[1].RecoveryAttempts*). During internal processing in our framework, we assign a unique fully qualified key for each configuration instance from the underlying sources.

To add to the expressiveness of configuration references, *CPL* also supports wildcard patterns as well as substitutable variables in a notation (both in the scope and key parts). During evaluation, the validation engine in our framework will process these notations in the specifications, substitute variables in any part of the notations and perform

Table 3.1. Examples of configuration notations and their meanings

Notation	**Refers to**
Cloud.Tenant.SecretKey	*SecretKey* in all tenants in all clouds
Cloud::CO2test2.Tenant.SecretKey	*SecretKey* in all tenants in cloud CO2test2
Cloud::$CloudName.Tenant.SecretKey	*SecretKey* in all tenants in clouds named with values of $CloudName
Cloud[1].Tenant::SLB.SecretKey	*SecretKey* in tenant SLB in the first cloud
*.SecretKey	*SecretKey* under any top-level scope
*IP	Any parameter with a key that ends with IP in any scope

pattern matching to find all relevant instances in the underlying configuration sources to validate.

Table 3.1 shows examples of supported configuration notations in *CPL*. In our experience, we find that another benefit of the unified configuration notation is that it makes it easy to cross-validate different configuration sources, which is commonly needed for configuration validation. For example, we easily can validate that the secret keys for the controller component are consistent with the configuration data in a component providing authentication services.

Namespace. Since scopes are commonly used in configuration notations, *CPL* supports a namespace concept that is very similar to namespaces in modern languages like C++ or C#. But rather than avoiding name collisions, the `namespace` keyword in *CPL* is syntactic sugar to avoid repeatedly writing long scopes for a configuration key. For example, instead of referring to $r.s.k1$, $r.s.k2$, and $r.s.k3$, we can just use $k1$, $k2$, and $k3$ when inside `namespace` $r.s$. Multiple namespaces can be assigned in one predicate block.

When resolving a configuration notation in a predicate block with namespaces,

we try to prefix the reference for each specified namespace in order and stop upon finding its existence. For example, inside a namespace n, we resolve the notation $a.k1$ by first trying to prefix the notation with n, *i.e.*, $n.a.k1$, and if it does not exist then look for $a.k1$.

Compartment. In *CPL*, a predicate is evaluated iteratively on all instances of a domain. When a predicate is defined over multiple domains, the instances to be evaluated, by default, will be the Cartesian product of the instance sets for these domains. This might be unwanted. For example, the predicate ***r***(*VLAN.StartIP*, *VLAN.EndIP*) $:=$ *VLAN.StartIP* $\leq$ *VLAN.EndIP* represents the assertion that *StartIP* should be smaller than *EndIP* under the *VLAN* scope. If there are 5 instances for each of the two domains, *VLAN[1].StartIP*, *VLAN[1].EndIP*, *VLAN[2].StartIP*, *VLAN[2].EndIP*, etc., the default evaluation will check the predicate on 25 pairs like (*VLAN[1].StartIP* $\leq$ *VLAN[2].EndIP*). But what is often needed is instead to check the predicate on *StartIP* and *EndIP* instances that appear under the same *VLAN* instance. In other words, only 5 pairs should be checked.

CPL provides the construct `compartment` to override the default evaluation behavior for predicates involving multiple domains. A compartment is similar to a namespace in that resolving configuration notations inside a compartment will try to prefix the notation with the compartment name. But unlike namespace, every instance of the compartment name is treated as an isolated scope when evaluating any predicate. In other words, a predicate in a compartment will repeatedly be evaluated the same number of times as the number of compartment instances, each time with the configuration keys being under a specific compartment instance. If inside a particular compartment instance, some domain in the predicate does not have any instance predicate evaluation will skip this compartment instance and continue to next compartment instance.

Using the previous example, we can put ***r***(*StartIP*, *EndIP*) $:=$ *StartIP* $\leq$ *EndIP* in `compartment` *VLAN*. During evaluation, the predicate is evaluated 5 times,

```
⟨statement⟩ ::= ⟨predicate⟩ | ⟨command⟩

⟨predicate⟩ ::= ⟨domain⟩ '→' ⟨predicate⟩
             |  'if' '(' ⟨predicate⟩ ')' ⟨predicate⟩
             |  'if' '(' ⟨predicate⟩ ')' ⟨predicate⟩ 'else' ⟨predicate⟩
             |  ⟨quantifier⟩ ⟨predicate⟩
             |  ⟨predicate⟩ '&' ⟨predicate⟩
             |  ⟨predicate⟩ '|' ⟨predicate⟩
             |  '∼' ⟨predicate⟩
             |  'namespace' ⟨qid⟩ '{' ⟨predicate⟩ '}'
             |  'compartment' ⟨qid⟩ '{' ⟨predicate⟩ '}'
             |  ⟨primitive⟩
             |  ...

⟨primitive⟩ ::= ⟨type⟩ | ⟨relation⟩ | ⟨match⟩ | ⟨range⟩ | ⟨consistent⟩ | ⟨unique⟩ | ⟨order⟩ |
                '@' ⟨id⟩ | ...

⟨quantifier⟩ ::= ∃ | ∀ | ∃!

⟨domain⟩  ::= '$' ⟨qid⟩
           |  ⟨transform⟩ '(' ⟨domain⟩ ')'
           |  ⟨domain⟩ '→' ⟨transform⟩
           |  ⟨domain⟩ ⟨binary_op⟩ ⟨domain⟩
           |  ⟨unary_op⟩ ⟨domain⟩
           |  '#' ⟨compartment⟩ ⟨domain⟩ '#'
           |  ...

⟨qid⟩     ::= ⟨qid⟩ '.' ⟨wid⟩
           |  ⟨qid⟩ '::' '$'? ⟨wid⟩
           |  ⟨qid⟩ '[' '$'? ⟨wid⟩ ']'
           |  ⟨qid⟩ '[' ⟨int⟩ ']'
           |  ⟨wid⟩

⟨wid⟩     ::= ⟨wid⟩ ⟨wsym⟩ | '_' | '*' | ⟨letter⟩

⟨wsym⟩    ::= '_' | '*' | ⟨letter⟩ | ⟨digit⟩

⟨command⟩ ::= ⟨let⟩ | ⟨load⟩ | ⟨get⟩ | ⟨include⟩ | ...
```

Listing 3.6. *CPL* grammar

VLAN[1].StartIP $\leq$ *VLAN[1].EndIP*, *VLAN[2].StartIP* $\leq$ *VLAN[2].EndIP*, *etc.*. If a *VLAN* instance does not have *StartIP* or *EndIP* keys, the predicate will skip this instance (if needed, a predicate can easily be written that assures that appearances of *StartIP* in a *VLAN* implies an *EndIP*).

Compartments are also useful for predicates defined over a single domain. For example, suppose that we need to validate that the location identifier assigned to a blade is unique in the rack it belongs to. The qualified configuration notation is *Rack.Blade.Location*. But we cannot simply write a predicate that validates that this class is unique because the location identifier is allowed to overlap *across* different racks. In other words, the uniqueness should be enforced only *within* a rack. With a compartment and the way we resolve the configuration notation, we can write a predicate ***r*** that represents "*Blade.Location* is unique" and put it under `compartment` *Rack*. The framework will then find all instances for the compartment and check if uniqueness for instances of *Blade.Location* under each compartment instance.

3.4.3 Piping

Domains in *CPL* provide the data instances to be validated in constraints. In some scenarios, constraints are enforced on only part of the data values or transformed data values. To avoid superfluous temporary assignments, *CPL* allows domains to go through a data pipeline and be applied in a final constraint at the end of the pipeline. Each step of the pipeline can be a transformation function or predicated transformations. The result values of a step, if any, will be passed on as implicit arguments to the transformation functions in the next step, iteratively or as a whole based on transformation functions. A step can also access the result of its prior step explicitly with the variable `$_`.

```
/* Prepare configuration sources for (cross-)validation, define
   macros */
load 'runninginstance' '10.119.64.74:443'
load 'cloudsettings' '/path/to/settings'
load 'assets' 'example.com/resources'
include 'type_checks.prop'
let UniqueCIDR := unique & cidr

// machinepool in cluster is
// one of the defined machinepool names
$Cluster.MachinePool → {$MachinePool.Name}

// threshold is a nonempty integer in range
$Fabric.AlertFailNodesThreshold → int & nonempty & [5,15]

// consistent fill factors within a data center
#[Datacenter] $Machinepool.FillFactor# → consistent

compartment Cluster {
  // IP is in range within each cluster
  $ProxyIP → [$StartIP, $EndIP]
  // either empty or unique CIDR notation
  $IPv6Prefix  → ~nonempty | @UniqueCIDR
}

// if any gateway points to loadbalancer
// a loadbalancer device should exist
if (∃ $RoutingEntry.Gateway ==
  'LoadBalancerGateway')
  $LoadBalancerSet.Device → nonempty

// if not a type of cloud, TenantName in the
// corresponding fabric starts with UfcName
if ($CloudName → ~match('UtilityFabric')) {
  $Fabric::$CloudName.TenantName
    → split(':') → at(0) → $_ == $UfcName
} else {
  $Fabric::$CloudName.TenantName → ~nonempty
}

// VipRanges value is like 'ip1-ip2;ip3-ip4'
// each item within should be in range
$MachinPoolName → foreach($MachinPool::$_.LoadBalancer.VipRanges)
    → if (nonempty) split('-')
    → [at(0), at(1)] → ∃ [$StartIP, $EndIP]
```

Listing 3.7. Example validation specifications in *CPL*

3.4.4 Commands

CPL defines a set of commands to prepare or facilitate validation. For example, the `load` command is used to provide configuration sources for a validation session; the `include` command adds existing specifications to the current session, which is useful for making specifications modular; the `let` command defines common constraints as a "macro", *e.g.*, `let UniqueIP := unique & ip`, which can be used later in a predicate (with `@` symbol).

3.4.5 Grammar and Examples

Listing 3.6 shows the main *CPL* grammar. A simple validation statement in *CPL* is $\langle domain \rangle \rightarrow \langle predicate \rangle$, For example, `$OSBuildPath` $\rightarrow$ `path & exists` checks if each instance of configuration `OSBuildPath` is an extant path. Listing 3.7 shows more *CPL* code examples, written to replace some *ad hoc* validation code snippets. These examples are much more concise in *CPL* than their original counterparts. The examples also show cross-validation of different configuration sources in *CPL* does not require cumbersome handling.

3.4.6 Extending *CPL*

CPL provides a common set of predicate primitives and the recursive construction of predicates from other predicates to cover typical validation requirements. Yet it can be the case that some predicates cannot be described in *CPL*.

There are two approaches to extend *CPL* to support more validation logic. The first one is to add predicates into *CPL* language primitives (*e.g.*, keyword `reachable`). This requires modifying the *CPL* compiler, which is written on top of a modern framework. It is relatively straightforward to extend the compiler. We provide base classes in the compiler for extending new types of predicates. On average, the implementation of

existing predicates that inherit the provided interfaces takes about 70 lines of C# code.

The second approach is to leverage new transformation functions to transform the domain to be validated into a new domain so that it is easy to validate the new domain without new language primitives (see Section 3.4.1). We allow user-defined transformation functions to be added as plug-ins. In this way, there is no need to modify the syntax or compiler of *CPL*.

We also plan to allow predicates to be added as plug-ins without modifying the *CPL* compiler. A current workaround to achieve this is to define a predicate as a binary transformation function. In this way, the predicate can be added as a plug-in. But users need to explicitly test that the transformation result is TRUE, which is not needed for predicates that are *CPL* language primitive.

3.5 Validation Policy and Runtime Information

In addition to the main validation logic, a separate policy can be provided during evaluation to control the validation behaviors. We currently allow policies to describe violation severity, violation handling (*e.g.*, stop on first violation, continue on violations), failed actions and validation priority (i.e., assigning priorities for configuration parameters so that specifications involving critical parameters are evaluated first).

The validation engine may also collect some runtime information such as the host environment to evaluate predicates that require this information. For example, the OS name of a host or date time can be used in predicates. The current support for such dynamic validation involving runtime information is limited in *CPL*. We plan to extend the support for dynamic validation in future work.

3.6 Error Messages

An error message is commonly used in validation to help understand why the target configuration failed the validation. In existing configuration validation code, an explicit error message is manually crafted after a check is violated. When there are many such checks, writing error messages becomes tedious. In addition, these error messages are usually just a repetition to the semantic of a check plus information about the configuration. In *CPL*, we automatically generate the error messages based on the checks and configuration key values. For instance, if the predicate is a range, the error message is that a value for the key is out of the range. We also allow overriding this default error message for an individual check.

3.7 Evaluation

In this section, we evaluate *CPL* inside Microsoft Azure as well as on two open-source cloud systems, OpenStack and CloudStack. We seek to answer how expressive and easy is it to write configuration validation specifications in our proposed language *CPL*.

3.7.1 Baselines

Our main evaluation subject is Microsoft Azure. Microsoft Azure practitioners currently typically use stand-alone validation tools and scripts written mainly in C# and PowerShell to validate different kinds of configurations. The high-level languages that are used to write these tools have features that are suitable for certain configuration validation tasks, *e.g.*, LINQ in C# for querying configuration data. But the core validation part is imperative and is tightly coupled with individual validation needs. This creates repeated validation development costs and hurts code maintainability. We compare writ-

ing validation in *CPL* with the *ad hoc* code in these validation tools.

We also compare with configuration validation practices in two major open-source cloud systems, OpenStack and CloudStack. Configuration validation in OpenStack is lacking. Its developers recently raised a discussion to improve this situation [19] and took initial efforts, *e.g.* adding type validation. But a concrete solution is missing. A third-party tool called Rubick [47] that is written in Python exists to provide configuration validation for OpenStack. Therefore, we compare writing validation in *CPL* with the Python code in Rubick. In CloudStack, the validation code is embedded in its Java source code files and is very imperative. We compare writing validation in *CPL* with the scattered Java validation code in CloudStack.

3.7.2 Rewriting Existing Validation Code

One criterion to test the expressiveness of *CPL* is to rewrite existing *ad hoc* validation code in *CPL*. With the background knowledge of the baseline systems and configurations, we manually went over existing validation code and expressed the validation requirements in *CPL* specifications. We measured the code size reduction as well as the development time.

Table 3.2 shows the result for the Microsoft Azure evaluation. The validation specifications in *CPL* are substantially more concise than the original validation code, as much as a 30x reduction. For example, an existing module of validation code of more than 3300 lines was rewritten in 109 lines of *CPL* code that consisted of 62 specifications. The *CPL* specifications are also much easier to read according to anecdotal feedback from practitioners. The time it takes to write these specifications ranges from half an hour to 6 hours, a large portion of which is spent on extracting the precise validation requirements from the original imperative code. Table 3.2 also shows that on average one third of the translated specifications can be automatically inferred by our inference

Table 3.2. Express validation code for three kinds of configuration data used inside Microsoft Azure into *CPL* specifications. *: time it takes to understand the original code and then write specifications in *CPL*.

Config.	**Orig. code**	**Specs in *CPL***			**Dev. time** *
	LOC	LOC	Count	Inferable	(man-hour)
Type A	800+	**50**	**17**	6	1
Type B	3300+	**109**	**62**	27	6
Type C	180+	**14**	**6**	1	0.5

Table 3.3. Express validation code in two open-source cloud systems into *CPL* specifications. *: time it takes to understand the original code and write specifications in *CPL*.

System	**Orig. code**	**Specs in *CPL***		**Dev. time***
	LOC	LOC	Count	(man-hour)
OpenStack	480	40	19	1
CloudStack	340	18	15	1.5

components.

Table 3.3 shows the result for two open-source systems, OpenStack (Rubick) and CloudStack. We can see similar code reduction and small development time from writing specifications in *CPL*.

The high code reduction and ease-of-writing mainly come from three advantages of *CPL*. First, validation requirements are described declaratively in *CPL* with predicates without prescribing the implementations. Second, configurations are referred by concise unified notations in *CPL*, which disentangles the essential specifications from details of handling specific diverse configuration instances. Third, *CPL* makes it easy to reuse specifications because of the modular nature of the specifications and language constructs like `let`.

We also encountered cases where some parts of the existing validation code were

hard to express in *CPL* (they are not counted in the results in Table 3.2 and Table 3.3). These are mainly validations that require complex dynamic workflow or multiple components to be running, *e.g.*, verifying with authentication services that certificates are acceptable. We plan to extend *CPL* to improve support for dynamic validation requirements. But it is not a design goal of *CPL* to be all-encompassing, which may make it hard to use. Instead, these few complex scenarios can still be handled by old-style imperative, hand-written validation code.

3.8 Conclusion

Configuration validation is an important quality assurance activity to catch configuration errors before production. But the current practice for configuration validation is *ad-hoc* and error-prone, eventually leaving practitioners with little incentives to add comprehensive validation code.

We design a specification language, called *CPL*, for developers to write configuration validation code in a declarative way. The language provides several essential abstractions to relieve developers' burden of prescribing tedious validation details and allow developers to focus on the core validation logic.

Using *CPL*, we can express the existing validation code in Microsoft Azure and two open-source cloud systems with 10x to 30x fewer lines of code and much smaller development efforts.

3.9 Acknowledgements

Chapter 3, in part, is a reprint of the material as it appears in the Tenth European Conference on Computer Systems 2015. Huang, Peng; Bolosky, Bill; Singh, Abhishek; Zhou, Yuanyuan. The dissertation author was the primary investigator and author of this paper.

Chapter 4

ConfValley: A Cloud Configuration Validation Framework

An improved validation language is not a panacea for the bad practices of configuration validation. We also aim to improve the tooling support for configuration validation. For example, configuration validation can be carried out at different stages of the configuration life cycle: while editing configurations, before checking-in to the repository, before deployment or at runtime. These validation scenarios require different tools such as an IDE that provides auto-completion and quick warnings for simple mistakes, a validation service that runs continuously on the configuration repository, and an interactive console for operators to validate production configurations on the fly. As another example, as the configuration data and services are constantly evolving, keeping the validation specifications up to date calls for effective tools.

In this chapter, we describe our framework, *ConfValley*, that aims to make configuration validation in cloud systems easy, systematic and efficient.

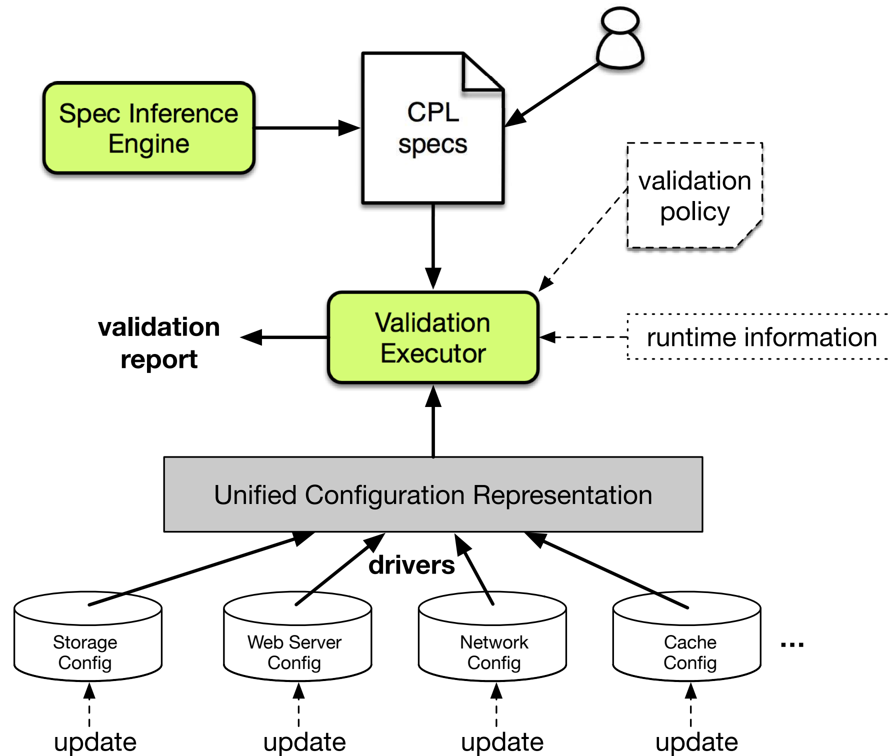

Figure 4.1. Architecture of ConfValley.

4.1 System Design

4.1.1 Overview

Figure 4.1 shows an overview of ConfValley. The core validation engine in ConfValley takes validation specifications and an optional validation policy (*e.g.*, actions for failed validations and the priorities for different specifications). It will find all the instances of target configuration entities according to the specifications, gather some runtime information if necessary, and then evaluate whether the constraints in the specifications are satisfied.

The validation specification is written in our new language *CPL* (cf. Chapter 3). The specification and validation engines interact with unified configuration representations that are abstracted from diverse configuration sources by a set of drivers. In this

way, the validation core logic is disentangled from tedious details.

Since there are a considerable number of configuration parameters in cloud systems, writing all the validation code from scratch can be time-consuming. ConfValley contains a component to automatically infer as many specifications as possible, especially basic ones. Automatic inference also helps address the issue of how to keep specifications up-to-date when the properties of some configurations change, *e.g.*, new value ranges of timeout parameters.

4.2 Inference Engine

Manually writing all the validation code can be time-consuming even for experts using a concise language like *CPL*. Therefore, ConfValley provides an inference engine to automatically generate as much *CPL* code as possible, especially basic checks such as data types and value ranges. In this way, experts can focus on writing advanced validation code.

Since validation specifications essentially depend on configuration constraints, the inference engine needs to extract configuration constraints. There are two options. White-box approaches use static analysis to infer configuration constraints from source code [128, 115]. Black-box approaches mine constraints from configuration data [131, 125, 124, 95]. White-box approaches usually have higher inference accuracy compared to black-box approaches. But their static analysis is difficult to scale to multi-component cloud-scale systems.

The inference engine in ConfValley follows the black-box approach to provide scalability, and leverage the fact that a configuration parameter has many instances in a cloud system, from which it is possible to extract useful information. The inference engine runs on samples of configuration data that are considered good (*e.g.*, the configurations have been scrutinized carefully and caused few incidents in the past). It infers

a constraint when there is enough evidence based on the samples. For example, to infer the data type of parameter A, if all the instances for parameter A can be parsed as integers, we infer the integer-type constraint for parameter A. The constraints we can currently infer include data types, non-emptiness, value range, enumeration elements, equality among multiple parameters, uniqueness, and consistency.

With the inferred constraints, the inference engine generates *CPL* specifications to evaluate on new configuration data. The inference does not need to be re-run each time configuration data is updated. This is because the *properties* of a configuration parameter can remain stable even though the values assigned to that parameter change frequently. In general, the inference is re-run when there are major changes to the system.

To improve inference accuracy, the inference engine tolerates irregularity and noise in the input configuration data. For example, some instances of parameter A may be integer values while other instances are comma-separated list of integers. In this case, we define an ordering on types and infer the type constraint of parameter A to be the highest-order type (list of integer). Like other black-box solutions [131, 125, 124, 95], we also use heuristics for noise-filtering. For example, we determine an enumeration constraint of a configuration class if $\ln(values.size) \geq value_set.size \wedge value_set.size \leq MAX_ENUM_VALS$; in determining the equality constraints, we ignore configuration values whose string-lengths are smaller than 6 and configuration classes that have fewer than 20 instances to avoid over-clustering (*e.g.*, irrelevant boolean configurations).

As another example, we infer the equality constraint by clustering configuration classes based on the value set and enforce a constraint that configuration classes in the same group should be equal. We take into consideration of the scopes in which the value instances lie in and may enforce the equality constraint under a certain compartment (see Section 3.4.2).

Table 4.1. Driver code to convert different types of configuration data in Microsoft Azure into a unified representation.

Config. format	Driver (LOC)
Generic XML settings	400
Type A	150
Type B	30
Type C	80
Type D	30
Type E	50

4.3 Implementation

We have implemented our validation framework, ConfValley, with 9,000 lines of C# code. The compiler component for our validation language *CPL* is built on top of a popular framework, ANTLR [9]. In the current implementation, *CPL* provides 19 predicate primitives and 13 transformation functions. We wrote driver code to convert different types of configuration data to a unified representation. Table 4.1 shows the code size of such drivers for common configuration formats used in Microsoft Azure.

4.3.1 Usage Scenarios

Three usage scenarios are supported in ConfValley. In the first scenario, we extend configuration editors to support *CPL* specifications and perform validation as configuration data is edited. The instant feedback can help correct simple errors (*e.g.*, incorrect type or format) before the wrong data is committed. In the second scenario, we provide an interactive console to allow practitioners to write short (one-liner) specifications and validate production data on-the-fly. The main usage scenario is a batch validation mode where ConfValley takes an input specification file and (re)validates it continuously as configuration specifications or data are updated.

4.3.2 Optimizations

Performance is an important factor for continuous configuration validation. We discuss two performance optimizations that have been implemented in ConfValley.

The first optimization is in the instance discovery component, *i.e.*, processing domain notations (see Section 3.4.2 in Chapter 3) in *CPL* specifications to find corresponding configuration instances in the underlying data sources for validation. Internally in ConfValley, each configuration instance is assigned a key that uniquely identifies this instance. The key consists of multiple segments describing the scope of the configuration instance, *e.g.*, `a::inst.b[1].c`. To find appropriate configuration instances, we need to match the configuration domain keys in *CPL* specifications with the instance keys. Such matching may not be a literal match. For example, domain key `a` in *CPL* matches all more specific instance keys such as `a::inst1`, `a::inst2`. Also, any wildcards in domain keys require pattern matching.

In our initial implementation of the instance discovery, we got all instance keys that had the same number of segments as the domain key, and then iterated segment-by-segment to gradually filter out instance keys whose segment did not approximately match the corresponding segment of the domain key. But this implementation was inefficient in handling the high load of discovery queries, which is typical in large systems like Microsoft Azure (in some runs we see more than 5 million configuration instance discovery queries). Therefore, our initial implementation became a bottleneck in the validation process. We rewrote the this part with better data structures (*e.g.*, trie) and caching support. The optimizations improved the processing time by 5x to 40x.

The second optimization is in the *CPL* compiler. Manually written validation code can contain inefficiencies. For example, multiple predicates for the same domain may be used by multiple specifications, causing repeated instance queries for that do-

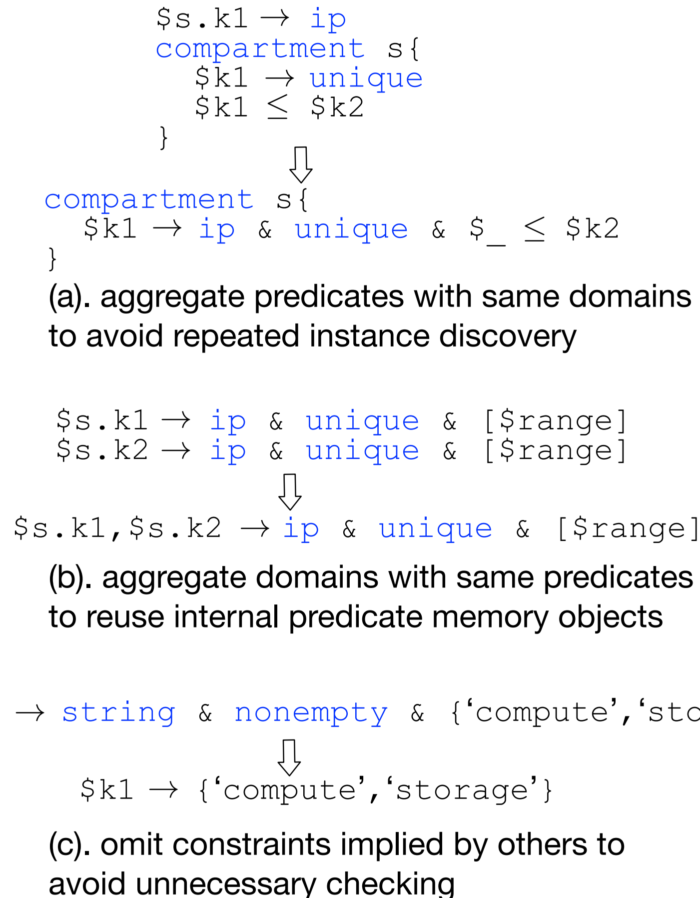

Figure 4.2. Examples of *CPL* compiler optimizations.

main. Conversely, a number of domains for the same predicate may be in separate specifications, causing excessive predicate memory objects to be created during validation. Additionally, some constraints in a specification can be implied by others, causing unnecessary checking. Our compiler rewrites these types of inefficient specifications by aggregating predicates, aggregating domains or omitting implied constraints. Figure 4.2 shows examples of these compiler optimizations.

4.4 Evaluation

In this section, we evaluate ConfValley inside Microsoft Azure as well as on two open-source cloud systems, OpenStack and CloudStack. We seek to answer the

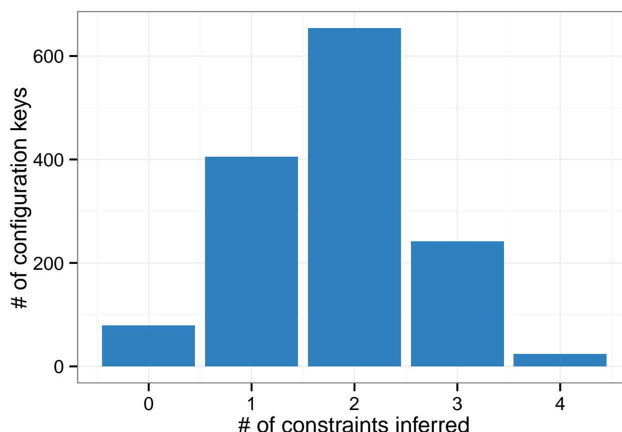

Figure 4.3. Histogram of number of inferred constraints on a type of Microsoft Azure configuration data with 1,391 configuration keys and 67,231 instances.

following questions: First (Section 4.4.1), how effective is the inference engine in ConfValley? Second (Section 4.4.2), how effective are human-written and inferred validation specifications in catching configuration errors? Lastly (Section 4.4.3), how efficient is the validation and inference process? All the experiments were carried out on a single machine with a 2.8 GHz Intel Core i7 CPU and 8 GB RAM running Windows.

4.4.1 Automatic Inference

An effective inference component should automatically generate many constraints, especially tedious ones, leaving experts to focus on complex constraints. Our inference engine can mine constraints from a large volume of stable configuration data.

On a type of configuration data with 1,391 configuration keys, and 67,231 total instances, Figure 4.3 shows the histogram of inferred constraints. We can see that the majority of the configuration keys had at least 2 constraints inferred. There were 79 configuration keys that had no constraints inferred. They were configuration parameters that did not have much associated semantics or constraints by nature, *e.g.*, `IncidentOwner`, `ClusterName`.

Table 4.2. Validation constraint inference on three kinds of configuration data in Microsoft Azure.

Config.	# of config. analyzed		Inferred
	Class	Instance	constraints
Type A	1,391	67,231	2,706
Type B	162	2,306,935	375
Type C	95	2,253	261

Table 4.3. Breakdown on the types of inferred constrains in Table 4.2

Config.	Type	Nonempty	Range	Equality	Consistency	Uniqueness
Type A	1,026	317	203	367	722	71
Type B	126	114	62	1	29	43
Type C	93	75	18	0	75	0

Table 4.2 and Table 4.3 shows the result of running our inference component on three kinds of configuration data in Microsoft Azure. Most configuration data had at least data type (we only count data types other than default `string`) or nonemptiness constraints inferred. Inference of other constraints such as value range and equality depends on whether the constraint is applicable to the particular configuration.

We manually examined the inferred constraints. The accuracy is around 80%, which is acceptable. The inaccuracies (*e.g.*, incorrect range inferred) come from insufficient samples for a configuration and from suboptimal heuristics for certain inferences. With more configuration data and tuning, the accuracy can be improved. We also plan to explore whether the heavy-weight white-box solutions can be efficiently combined in our inference component to improve accuracy.

In practice, we find that inaccurately inferred constraints are usually easy to spot after running the inferred specifications on real configuration data. Practitioners can first inspect the validation reports generated by ConfValley, which can group failed

Table 4.4. Running expert-written validation specifications on three latest configuration data branches in Microsoft Azure.

Config. branch	Reported errors
Trunk	4
Branch 1	2
Branch 2	2

validations by constraint. If many configuration instances fail a constraint, it is likely that constraint is problematic because it is rare that configuration data in an enterprise environment has a large error percentage.

4.4.2 Preventing Configuration Errors

The ultimate goal of a configuration validation framework is to prevent configuration errors from being introduced into production. We used both manually crafted and automatically inferred validation code to run on the latest configuration data in Microsoft Azure.

With manually-written validation specifications, Table 4.4 shows that ConfValley reported 8 errors in total, all of which are confirmed. The reported errors included that "the VIP range of a load balancer set is not contained in VIP range of its cluster" , "bad BladeID", and "inconsistent number of addresses in MAC range and IP range".

With inferred validation specifications, Table 4.5 shows that ConfValley reported 43 errors, among which 11 are false positives. Examples of the true errors are "empty `FccDnsName`" and low `ReplicaCountForCreateFCC` (these errors have previously caused deployment incidents). The false positives are due to the inaccurately inferred specifications. For example, the value range inferred from the input configuration is incomplete; the type seen in the input data is in a simplified form, *e.g.*, configuration instances in input are a single IP address but their true types are a list of IP address.

Table 4.5. Running inferred validation specifications on three latest configuration data branches in Microsoft Azure.

Config. branch	Reported errors	False positives
Trunk	12	3
Branch 1	15	5
Branch 2	16	3

Table 4.6. Latency (in seconds) of sequential validation on three types of configuration data in Microsoft Azure.

Config.	Instances	Specs		Time
		Count	Source	
Type A	44,102	182	Inferred, optimized	10
Type B	1,969,588	62	Human-written	518
Type C	1,529	95	Inferred	0.4

4.4.3 Performance

The performance of the validation framework affects whether it can be practically adopted to run continuously. We evaluate the efficiency of both the validation and automatic inference processes.

Table 4.6 shows that the validation time has wide variation, depending on the number and types of specifications and configuration data. We show the performance of the single-threaded implementation of ConfValley on a single machine. The maximum time in these sequential experiments is less than 9 minutes, which is acceptable given that the validation happens before deployment and therefore will not interfere with online services.

Since each specification in *CPL* is independent, the validation process can be made parallel. We demonstrate the potential speedup with parallel validation by simply

Table 4.7. Latency (in seconds) of simple parallel validation on three types of configuration data in Microsoft Azure by splitting the specifications into 10 pieces, and validating each piece in parallel. (Min, Median, Max) measures the (min, median, max) validation time of the 10 jobs.

Config.	**Instances**	**Specs**		**Time**		
		Count	Source	Min	Median	Max
Type A	44,102	182	Inferred, optimized	2	2	4
Type B	1,969,588	62	Human-written	49	52	208
Type C	1,529	95	Inferred	0.3	0.3	0.3

Table 4.8. Inference latency (in seconds) on three types of configuration data. *: the time it takes to parse all configuration sources and convert into unified representations.

Config.	**Instance**	**Time**		
		Total	Parsing*	Inference
Type A	67,231	19.7	19.5	0.2
Type B	2,306,935	82	75	7
Type C	2,253	0.09	0.08	0.01

splitting the specifications into 10 partitions and running 10 validation jobs in parallel. We can see from Table 4.7 that the maximum time reduces to 3.5 minutes. The speed-ups are not always linear because some specifications are more complex than others, and also each validation job parses configuration sources independently.

Table 4.8 shows the automatic inference time. The total elapsed time is within 2 minutes for all evaluated configuration data. The break-down shows that the bottleneck lies in parsing the configuration data into a unified representation, while the actual inference time is fairly small. Because inference only runs at specification-creation time, it would be acceptable even if it were orders of magnitude slower.

4.5 Conclusion

Configuration errors continue to haunt practitioners of large-scale systems. We believe that making configuration validation an ordinary part of system deployment is crucial to prevent misconfigurations from impacting production services. The validation should go beyond just employing *ad hoc* checking code that is added and invoked reactively.

We present a generic framework, ConfValley, to allow experts operating production cloud services to easily, systematically, and efficiently conduct validation for different configuration data in various scenarios. Evaluating ConfValley inside Microsoft Azure and two open-source cloud systems showed that with our declarative validation language, we can express the existing validation code in a much more concise and readable form. Using both the rewritten and automatically inferred validation specifications, our framework detected 40 configuration errors in the latest configuration data to be deployed inside Microsoft Azure.

4.6 Acknowledgements

Chapter 4, in part, is a reprint of the material as it appears in the Tenth European Conference on Computer Systems 2015. Huang, Peng; Bolosky, Bill; Singh, Abhishek; Zhou, Yuanyuan. The dissertation author was the primary investigator and author of this paper.

We thank the anonymous reviewers, and the OPERA research group members for their valuable feedback. We are grateful to the Microsoft Azure teams for their help and support. This work was partially supported by NSF CNS-1017784, NSF CNS-1321006 and Microsoft CNS Research Fund.

Chapter 5

Limitations and Future Work

5.1 Limitations

While the evaluation demonstrates the advantages of ConfValley, our solution has the following limitations. First, our proposed validation language is not able to express certain types of validation requirements, especially those involving dynamic, complex requirements. We are continuing to work with the developers and operators to refine the language to support more validation needs.

Second, we target generic configuration data and thus the tool is not optimized towards a particular sub-type of configuration. Therefore, our framework has limited support for validation methods targeting domain-specific configurations such as network configurations.

Third, like any other validation approach, ConfValley is not a verification tool and therefore does not guarantee that validated configurations are fault free. Furthermore, not all types of parameters benefit much from validation. For example, there are fewer validation benefits for tunable parameters like `ipc.timeout` compared to other types of parameters. This is because tunable parameters have a large space of correct values and require constant tuning to determine the optimum. Validation can only eliminate a portion of obviously incorrect values.

5.2 Future Work

There are several directions that we would like to work on in the future along the line of dealing with cloud configuration error. First, we are interested in enhancing *CPL* to be able to validate not only static constraints of configurations but also dynamic system states. For example, developers often have some expectations on a particular configuration change's effect on some dynamic metric (e.g., load distribution). Providing primitives in *CPL* for developers to express such expectations would be useful.

Second, most of state of the art solutions (including ConfValley!) for tackling misconfiguration mainly deal with configuration errors that violate some clear constraints. But in many misconfiguration-inducing failure incidents, the misconfiguration is some seemingly normal settings that are subtle to detect. For example, Facebook reported that 36% of the configuration issues they encountered belong to this type [122]. We would like to further investigate how to effectively deal with such subtle misconfiguration.

Third, in the ConfValley project, our choice of designing a configuration validation language instead of a new configuration language was based on practical reasons: convincing tens of teams to replace their configuration and libraries with a brand new languages faces significant cultural acceptance challenge while *CPL* can support all the teams' existing configuration languages. But moving forward, we still believe changing the interfaces and languages of how large systems are configured is the ultimate way to fundamentally attack the notorious configuration problem. We are interested in building upon our experience of designing *CPL* to design a much better configuration language.

Chapter 6

Related Work

6.1 Failure Studies

Since Gray's pioneering study [87] of failures in mainframes and how to make software fault tolerant, a number of studies have been conducted to understand failure characteristics in different environments such as personal computers [107], networked systems [127], high-performance computing systems [119], Internet services [108] and Hadoop clusters [116]. We present the first large-scale study of cloud service disruptions. Compared with the prior studied systems, our study subject has many more components, frequent component failures and abundant fault tolerance in place.

Recent studies investigate failures in data center hardware [120, 83, 123], networks [85], storage systems [83] and service events [58]. But these failure events are mainly at the component level, most of which are successfully tolerated and do not correspond to service level disruptions. We study visible service disruptions and focus on why the service disruptions occur despite fault tolerance and the characteristics of misconfiguration.

6.2 Fault tolerance and recovery

Fault tolerance design is an extensively studied topic in different areas such as hardware (*e.g.*, [112, 60]), networks (*e.g.*, [73]), and distributed systems (*e.g.*, [63, 99, 69, 64, 96, 93, 105]). Fast failure detection [70, 101] and cheap recovery [110, 68] are also important for system availability. The proposed techniques are adopted in different layers of large-scale distributed systems to provide high availability. Complementary to these works, we examine why component faults escape these mechanisms and lead to service disruptions.

6.3 Characteristics of system misconfiguration

Studies over the past decades have shown operator mistakes are a common cause of system unavailability [87, 108, 106, 129, 116, 58]. Among these works, Yin *et al*. [129] and Rabkin *et al*. [116] investigate further about misconfiguration characteristics.

We find misconfiguration has similar dominance in cloud service disruptions. This motivates us to zoom into these configuration errors. Compared with the in-depth misconfiguration studies by Yin's and Rabkin's studies, we study misconfiguration in cloud-scale systems, which consist of many components interacting with each other and in which the environment is undergoing frequent changes. Additionally, the configuration practitioners in our studied system are trained operators rather than end users/customers.

6.4 Misconfiguration detection, diagnosis and fix

A wide body of work has been done to detect and troubleshoot misconfigurations [125, 126, 124, 82, 121, 74, 82, 61, 55, 130, 56, 131]. For example, Con-

fAid [56] and X-ray [54] use dynamic information-flow tracking to find possible configuration errors that cause failures or performance anomalies; AutoBash [121] speculatively executes processes and tracks causality to automatically fix misconfigurations; Strider [125] and PeerPressure [124] leverage a set of configuration settings from different machines to narrow down the problematic configuration on a sick machine; EnCore [131] learns configuration rules and exploits environment information to detect misconfigurations. KarDo [97] automates configuration tasks on a computer by searching a solution database to replay traces from other computers.

Compared to systems that diagnose and fix misconfigurations, we target configuration validation, which is a complementary direction that aims to proactively prevent configuration errors from being introduced into production services. Compared to the misconfiguration detectors, which attack a specific type of misconfiguration or follow some heuristics, our work proposes a generic validation framework that provides a compact language for practitioners to explicitly specify the validation requirements based on their expertise and experience.

6.5 System resilience to misconfiguration

Research has been done to test system resilience to misconfigurations [94, 128, 65]. ConfErr [94] uses a human error model from psychology and linguistics to inject misconfigurations into systems. SPEX [128] takes a white box approach to automatically extract configuration parameter constraints from source code and generates misconfigurations to test systems by violating these constraints.

Making systems gracefully handle misconfigurations and eliminating configuration errors are two orthogonal directions. The former helps improve system robustness and eases diagnosis. This is especially important for software that will be widely distributed to end users. Our work belongs to the latter case, which is useful to prevent

errors in the first place.

6.6 Configuration languages

There is growing interest in new configuration languages [79, 103, 81, 72] to reduce configuration errors induced by fundamental deficiencies in existing languages (*e.g.*, untyped, too low-level). This is especially the case in the network configuration management area, where it is an onerous task to configure diverse network devices and protocols to support evolving service scenarios. PRESTO [81] automates the generation of device-native configurations with *configlets* in a template language. Loo et al. [103] adopt Datalog to express routing protocols in a declarative fashion. COOLAID [72] proposes a language to describe domain knowledge about network devices and services to ease network reasoning and management.

Different from this work, it is not our goal to replace existing configuration languages, which would require extensive changes in the software stack that manages and uses existing configuration data. Our proposed language, *CPL*, is for writing validation code for existing configuration data. Moreover, our framework provides more than just a new language but also other tool chains such as an inference component and an iterative validation console.

6.7 Configuration management

A variety of tools such as Puppet [45], Chef [18] and Salt [48] have been developed to ease configuration management in cloud-scale infrastructure. These tools focus on the resource-arrangement aspect of configuration, whereas our work, like most others in literature, target the system-options aspect of configuration.

Chapter 7

Concluding Remarks

Cloud services have brought, and will continue to bring, significant disruptions to the computing landscapes and real life with a plethora of new application domains. For system researchers, how to design and operate systems as complex as cloud to provide high-availability to enable the new applications is a billion-dollar question. This thesis attempts to shed some lights on this question by taking a bottom-up approach to first study carefully how existing, state-of-the-art cloud systems fail and then present a solution that is inspired by the study to tackle a type of plague in cloud, misconfiguration. The common theme of both parts centers on the classic question in system research: *what is the right level of abstraction?* For cloud failure analysis, we believe that, different from traditional failure analysis, the level uniquely lies in the fault-tolerance mechanisms rather than the original root cause. For attacking cloud misconfiguration, we demonstrate that with a declarative specification language instead of *ad-hoc*, low-level scripting, configuration validation can be made significantly more efficient.

Bibliography

[1] Air france flight 447 accident final report. http://www.bea.aero/docspa/2009/f-cp090601.en/pdf/f-cp090601.en.pdf.

[2] Amazon earnings statement for 2015. http://www.sec.gov/Archives/edgar/data/1018724/000101872416000170/amzn-20151231xex991.htm.

[3] Amazon EC2 and RDS service disruption on April 21st, 2011. http://aws.amazon.com/message/65648.

[4] Amazon EC2 outage downs Reddit, Quora. http://money.cnn.com/2011/04/21/technology/amazon_server_outage/index.htm.

[5] Amazon EC2 outage explained and lessons learned. http://www.infoq.com/news/2011/04/Amazon-EC2-Outage-Explained.

[6] Amazon EC2 outage: summary and lessons learned. http://blog.rightscale.com/2011/04/25/amazon-ec2-outage-summary-and-lessons-learned.

[7] Amazon S3 availability event on July 20th, 2008. http://status.aws.amazon.com/s3-20080720.html.

[8] Amazon's trouble raises cloud computing doubts. http://www.nytimes.com/2011/04/23/technology/23cloud.html.

[9] ANTLR tool. http://www.antlr.org.

[10] Apache CloudStack. http://cloudstack.apache.org.

[11] The AWS outage: The cloud's shining momen. http://broadcast.oreilly.com/2011/04/the-aws-outage-the-clouds-shining-moment.html.

[12] AWS service disruption on August 26th, 2013. http://www.geekwire.com/2013/vine-instagram-stop-working-time-users-freak-twitter.

[13] AWS service disruption on September 20th, 2015. https://aws.amazon.com/message/5467D2.

[14] AWS service outage on December 24th, 2012. http://aws.amazon.com/message/680587.

[15] AWS service outage on June 29th, 2012. http://aws.amazon.com/message/67457.

[16] AWS service outage on October 22nd, 2012. https://aws.amazon.com/message/680342.

[17] AWS service outage on September 20th, 2015. https://aws.amazon.com/message/5467D2.

[18] Chef software. http://www.getchef.com/chef.

[19] Discussions on configuration validation in OpenStack. http://lists.openstack.org/pipermail/openstack-dev/2013-November/018557.html.

[20] Downtime, outages and failures - understanding their true costs. http://www.evolven.com/blog/downtime-outages-and-failures-understanding-their-true-costs.html.

[21] Dropbox service outage on January 10th, 2014. https://blogs.dropbox.com/tech/2014/01/outage-post-mortem.

[22] Facebook and the kernel. http://lwn.net/Articles/591780.

[23] Facebook downtime explained. http://mashable.com/2010/09/23/facebook-downtime-explained.

[24] Facebook gives a post-mortem on worst downtime in four years. https://techcrunch.com/2010/09/23/facebook-downtime.

[25] Facebook outage on September 23rd, 2010. https://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919.

[26] Facebook outage was its bigger ever. http://www.cnn.com/2010/TECH/social.media/09/24/facebook.outage.

[27] Google API infrastructure outage on April 30th, 2013. http://googledevelopers.blogspot.com/2013/05/google-api-infrastructure-outage_3.html.

[28] Google apologizes for cloud outage that one person describes as a 'comedy of errors'. http://www.businessinsider.com/google-apologizes-for-cloud-outage-2016-4.

[29] Google App Engine outage on October 26th, 2012. http://googleappengine.blogspot.com/2012/10/about-todays-app-engine-outage.html.

[30] Google compute engine incident #15045. https://status.cloud.google.com/incident/compute/15045.

[31] Google compute engine incident #15064. https://status.cloud.google.com/incident/compute/15064.

[32] Google compute engine incident #16007. https://status.cloud.google.com/incident/compute/16007?post-mortem.

[33] Google service outage on January 24th, 2014. http://googleblog.blogspot.com/2014/01/todays-outage-for-several-google.html.

[34] Making the Netflix API more resilient. http://techblog.netflix.com/2011/12/making-netflix-api-more-resilient.html.

[35] Microsoft Azure. https://azure.microsoft.com/en-us.

[36] Microsoft Azure service disruption on February 29th, 2012. http://blogs.msdn.com/b/windowsazure/archive/2012/03/09/summary-of-windows-azure-service-disruption-on-feb-29th-2012.aspx.

[37] Microsoft Azure service disruption on November 18th, 2014. https://azure.microsoft.com/en-us/blog/final-root-cause-analysis-and-improvement-areas-nov-18-azure-storage-service-interruption.

[38] Microsoft Azure service interruption in western Europe on July 26th, 2012. https://azure.microsoft.com/en-us/blog/root-cause-analysis-for-recent-windows-azure-service-interruption-in-western-europe.

[39] Microsoft Azure storage disruption in US south on December 28th, 2012. http://blogs.msdn.com/b/windowsazure/archive/2013/01/16/details-of-the-december-28th-2012-windows-azure-storage-disruption-in-us-south.aspx.

[40] Microsoft Azure storage disruption on February 22nd, 2013. http://blogs.msdn.com/b/windowsazure/archive/2013/03/01/details-of-the-february-22nd-2013-windows-azure-storage-disruption.aspx.

[41] Microsoft blames last week's Azure outage on a configuration error. http://www.pcworld.com/article/260336/microsoft_blames_last_weeks_azure_outage_on_a_configuration_error.html.

[42] Microsoft explains last weekâĂŹs Azure outage: Whoops! http://allthingsd.com/20120803/microsoft-explains-last-weeks-azure-outage-whoops.

[43] Microsoft investigates Azure outage in europe. http://www.informationweek.com/cloud-computing/infrastructure/microsoft-investigates-azure-outage-in-e/240004433.

[44] OpenStack. http://www.openstack.org.

[45] Puppet software. http://puppetlabs.com/.

[46] Rackspace ORD service interruption on June 20th, 2013. https://blog.managebac.com/2013/06/21/rackspace-downtimes-full-incident-reports.

[47] Rubick project for OpenStack. https://wiki.openstack.org/wiki/Rubick.

[48] Salt software. http://www.saltstack.com.

[49] Server crash spurs 3-hour nasdaq halt as data link lost. http://www.bloomberg.com/news/articles/2013-08-26/nasdaq-three-hour-halt-highlights-vulnerability-in-market.

[50] Twilio billing incident post-mortem: Breakdown, analysis and root cause. https://www.twilio.com/blog/2013/07/billing-incident-post-mortem-breakdown-analysis-and-root-cause.html.

[51] Why Google's cloud went dark for 18 minutes. http://www.ciodive.com/news/why-googles-cloud-went-dark-for-18-minutes/417404.

[52] Windows Azure SLA. http://www.windowsazure.com/en-us/support/legal/sla.

[53] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, pages 63–74, New York, NY, USA, 2008. ACM.

[54] M. Attariyan, M. Chow, and J. Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 307–320, Hollywood, CA, USA, 2012.

[55] M. Attariyan and J. Flinn. Using causality to diagnose configuration bugs. In *Proceedings of the 2008 USENIX Annual Technical Conference*, ATC'08, pages 281–286, Boston, Massachusetts, 2008.

[56] M. Attariyan and J. Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–11, Vancouver, BC, Canada, 2010.

[57] A. Avizienis. The N-version approach to fault-tolerant software. *Software Engineering, IEEE Transactions on*, SE-11(12):1491–1501, Dec 1985.

[58] L. Barroso and U. Hölzle. *The Data Center as a Computer: An Introduction to the Design of Warehouse- Scale Machines*. Synthesis lectures on computer architecture, ISSN 1935-3235. Morgan & Claypool Publishers, 2009.

[59] L. A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, Mar. 2003.

[60] J. F. Bartlett. A NonStop kernel. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, SOSP '81, pages 22–29, New York, NY, USA, 1981. ACM.

[61] L. Bauer, S. Garriss, and M. K. Reiter. Detecting and resolving policy misconfigurations in access-control systems. In *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies*, SACMAT '08, pages 185–194, Estes Park, CO, USA, 2008.

[62] T. Benson, A. Akella, and D. Maltz. Unraveling the complexity of network management. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 335–348, Berkeley, CA, USA, 2009. USENIX Association.

[63] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 1–11, New York, NY, USA, 1995. ACM.

[64] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[65] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, San Diego, California, 2008.

[66] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, New York, NY, USA, 2011. ACM.

[67] G. Candea, M. Delgado, M. Chen, and A. Fox. Automatic failure-path inference: A generic introspection technique for internet applications. In *Proceedings of*

the The Third IEEE Workshop on Internet Applications, WIAPP '03, pages 132–, Washington, DC, USA, 2003. IEEE Computer Society.

[68] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot – a technique for cheap recovery. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

[69] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, Nov. 2002.

[70] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, Mar. 1996.

[71] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[72] X. Chen, Y. Mao, Z. M. Mao, and J. Van der Merwe. Declarative configuration management for complex and dynamic networks. In *Proceedings of the 6th International Conference*, Co-NEXT '10, pages 6:1–6:12, Philadelphia, Pennsylvania, 2010.

[73] D. Clark. The design philosophy of the DARPA internet protocols. In *Symposium Proceedings on Communications Architectures and Protocols*, SIGCOMM '88, pages 106–114, New York, NY, USA, 1988. ACM.

[74] T. Das, R. Bhagwan, and P. Naldurg. Baaz: A system for detecting access control misconfigurations. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, pages 11–11, Washington, DC, 2010.

[75] J. Dean. Designs, lessons and advice from building large distributed systems. Keynote from LADIS 2009, Oct. 2009.

[76] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[77] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[78] T. Delaet and W. Joosen. Podim: A language for high-level configuration management. In *Proceedings of the 21st Conference on Large Installation System Administration Conference*, LISA'07, pages 21:1–21:13, Dallas, 2007.

[79] J. DeTreville. Making system configuration more declarative. In *Proceedings of the 10th Conference on Hot Topics in Operating Systems*, HOTOS'05, pages 11–11, Santa Fe, NM, 2005.

[80] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A fast and reliable software network load balancer. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 523–535, Berkeley, CA, USA, 2016. USENIX Association.

[81] W. Enck, P. McDaniel, S. Sen, P. Sebos, S. Spoerel, A. Greenberg, S. Rao, and W. Aiello. Configuration management at massive scale: System design and experience. In *Proceedings of the 2007 USENIX Annual Technical Conference*, ATC'07, pages 6:1–6:14, Santa Clara, CA, 2007.

[82] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation*, NSDI'05, pages 43–56, Boston, Massachusetts, 2005.

[83] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–7, Berkeley, CA, USA, 2010. USENIX Association.

[84] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[85] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 350–361, New York, NY, USA, 2011. ACM.

[86] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, pages 202–210, New York, NY, USA, 1989. ACM.

[87] J. Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[88] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, pages 51–62, New York, NY, USA, 2009. ACM.

[89] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur. FATE and DESTINI: A framework for cloud recovery testing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 18–18, Berkeley, CA, USA, 2011. USENIX Association.

[90] J. Hamilton. On designing and deploying Internet-scale services. In *Proceedings of the 21st Conference on Large Installation System Administration Conference*, LISA'07, pages 18:1–18:12, Berkeley, CA, USA, 2007. USENIX Association.

[91] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in Windows Azure storage. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[92] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 3–14, New York, NY, USA, 2013. ACM.

[93] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. All about eve: Execute-verify replication for multi-core servers. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 237–250, Berkeley, CA, USA, 2012. USENIX Association.

[94] L. Keller, P. Upadhyaya, and G. Candea. ConfErr: A tool for assessing resilience to human configuration errors. In *Proceedings of the 38th International Conference on Dependable Systems and Networks*, DSN'08, pages 157–166, Anchorage, Alaska, USA, 2008.

[95] E. Kiciman and Y.-M. Wang. Discovering correctness constraints for self-management of system configuration. Technical Report MSR-TR-2004-22, Microsoft Research, March 2004.

[96] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 45–58, New York, NY, USA, 2007. ACM.

[97] N. Kushman and D. Katabi. Enabling configuration-independent automation by non-expert users. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–10, Vancouver, BC, Canada, 2010.

[98] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[99] L. Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[100] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[101] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish. Detecting failures in distributed systems with the falcon spy network. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 279–294, New York, NY, USA, 2011. ACM.

[102] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson. F10: A fault-tolerant engineered network. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 399–412, Berkeley, CA, USA, 2013. USENIX Association.

[103] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative routing: Extensible routing with declarative queries. In *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '05, pages 289–300, Philadelphia, Pennsylvania, USA, 2005.

[104] B. Maurer. Fail at scale. *Queue*, 13(8):30:30–30:46, Sept. 2015.

[105] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 358–372, New York, NY, USA, 2013. ACM.

[106] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and dealing with operator mistakes in Internet services. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, OSDI'04, pages 5–5, San Francisco, CA, 2004.

[107] E. B. Nightingale, J. R. Douceur, and V. Orgovan. Cycles, cells and platters: An empirical analysisof hardware failures on a million consumer pcs. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 343–356, New York, NY, USA, 2011. ACM.

[108] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do Internet services fail, and what can be done about it? In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, pages 1–1, Berkeley, CA, USA, 2003. USENIX Association.

[109] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud scale load balancing. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 207–218, New York, NY, USA, 2013. ACM.

[110] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery Oriented Computing (ROC): Motivation, definition, techniques,. Technical report, Berkeley, CA, USA, 2002.

[111] D. A. Patterson. An introduction to dependability. *;login:*, 27(4):pp. 61–65, 2002.

[112] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, pages 109–116, New York, NY, USA, 1988. ACM.

[113] C. Perrow. *Normal accidents: living with high-risk technologies*. Basic Books, 1984.

[114] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, FAST '07, pages 2–2, Berkeley, CA, USA, 2007. USENIX Association.

[115] A. Rabkin and R. Katz. Static extraction of program configuration options. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 131–140, Waikiki, Honolulu, HI, USA, 2011.

[116] A. Rabkin and R. Katz. How Hadoop clusters break. *Software, IEEE*, 30(4):88–94, 2013.

[117] B. Rockwood. The DevOps transformation. http://www.usenix.org/events/lisa11/tech/slides/rockwood.pdf.

[118] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstraction to improve fault tolerance. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 15–28, New York, NY, USA, 2001. ACM.

[119] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proceedings of the International Conference*

on Dependable Systems and Networks, DSN '06, pages 249–258, Washington, DC, USA, 2006. IEEE Computer Society.

[120] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM errors in the wild: A large-scale field study. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, pages 193–204, New York, NY, USA, 2009. ACM.

[121] Y.-Y. Su, M. Attariyan, and J. Flinn. AutoBash: Improving configuration management with operating system causality analysis. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 237–250, Stevenson, Washington, USA, 2007.

[122] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl. Holistic configuration management at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 328–343, New York, NY, USA, 2015. ACM.

[123] K. V. Vishwanath and N. Nagappan. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 193–204, New York, NY, USA, 2010. ACM.

[124] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, OSDI'04, pages 17–17, San Francisco, CA, 2004.

[125] Y.-M. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, C. Yuan, and Z. Zhang. Strider: A black-box, state-based approach to change and configuration management and support. In *Proceedings of the 17th USENIX Conference on System Administration*, LISA '03, pages 159–172, San Diego, CA, 2003.

[126] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, OSDI'04, pages 6–6, San Francisco, CA, 2004.

[127] J. Xu, Z. Kalbarczyk, and R. Iyer. Networked Windows NT system field failure data analysis. In *Dependable Computing, 1999. Proceedings. 1999 Pacific Rim International Symposium on*, pages 178–185, 1999.

[128] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 244–259, Farminton, Pennsylvania, 2013.

[129] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 159–172, Cascais, Portugal, 2011.

[130] D. Yuan, Y. Xie, R. Panigrahy, J. Yang, C. Verbowski, and A. Kumar. Context-based online configuration-error detection. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, ATC'11, pages 28–28, Portland, OR, 2011.

[131] J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou. En-Core: Exploiting system environment and correlation information for misconfiguration detection. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 687–700, Salt Lake City, Utah, USA, 2014.