# ConfValley: A Systematic Configuration Validation Framework for Cloud Services

Peng Huang*, William J. Bolosky†, Abhishek Singh‡, Yuanyuan Zhou*

University of California, San Diego*  Microsoft Research†  Microsoft‡

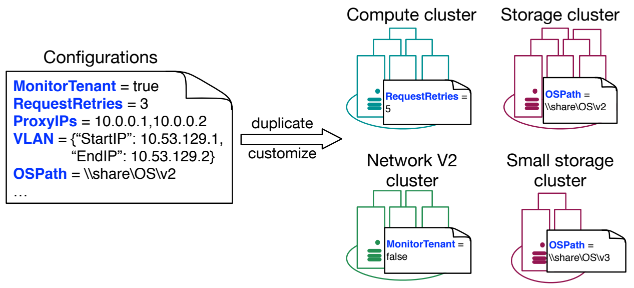
## Abstract

Studies and many incidents in the headlines suggest misconfigurations remain a major cause of unavailability in large systems despite the large amount of work put into detecting, diagnosing and repairing them. In part, this is because many of the solutions are either post-mortem or too expensive to use in production cloud-scale systems. Configuration validation is the process of explicitly defining specifications and proactively checking configurations against those specifications to prevent misconfigurations from entering production.

We propose a generic framework, *ConfValley*, to make configuration validation easy, systematic and efficient, and to allow configuration validation as an ordinary part of system deployment. ConfValley consists of a declarative language for practitioners to express configuration specifications, an inference engine that automatically generates specifications, and a checker that determines if a given configuration obeys its specifications. Our specification language expressed the configuration validation code from Microsoft Azure in 10x fewer lines, many of which were automatically inferred. Using expert-written and inferred specifications, we detected a number of configuration errors in the latest configurations deployed in Microsoft Azure.

## 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 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 [1, 4, 5, 9–11, 13, 14, 19], causing considerable financial cost [8].



```
Configurations
MonitorTenant = true
RequestRetries = 3
ProxyIPs = 10.0.0.1,10.0.0.2
VLAN = {"StartIP": 10.53.129.1,
        "EndIP": 10.53.129.2}
OSPath = \\share\OS\v2
...
```

Figure 1: Configuration data in cloud systems.

Misconfiguration is a thorny issue. Many solutions have been proposed to attack it ranging from misconfiguration detection [41, 42], diagnosis [22, 37–39] and repair [32, 36] to system resilience [30, 40]. These solutions greatly alleviate end users' pain in configuring unfamiliar software. However, they are often post-mortem 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.

*EuroSys '15*, April 21–24, 2015, Bordeaux, France.
Copyright © 2015 ACM 978-1-4503-3238-5/15/04. . . $15.00.
http://dx.doi.org/10.1145/2741948.2741963

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 [35]). 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 [12] 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.

## 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.

### 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 1, `MonitorNodeHealth` is a configuration class

```
1  <CloudGroup Name = "East1 Production">
2    <Setting Key = "MonitorNodeHealth" Value = "True">
3    <Setting Key = "ControllerReplicas" Value = "5">
4    <Cloud Name = "East1Storage1">
5      <Tenant Type = "A">
6        <Setting Key = "MonitorNodeHealth" Value = "False">
7      </Tenant>
8      <Tenant Type = "B" />
9    </Cloud>
10   <Cloud Name = "East1Storage2">
11     <Tenant Type = "A" />
12   </Cloud>
13 </CloudGroup>
14 <CloudGroup Name = "SSD Cluster">
15   <Setting Key = "MonitorNodeHealth" Value = "True">
16   <Setting Key = "ControllerReplicas" Value = "3">
17   <Cloud Name = "East1Compute1">
18     <Tenant Type = "A">
19       <Setting Key = "ControllerReplicas" Value = "5">
20     </Tenant>
21   </Cloud>
22 </CloudGroup>
```

Listing 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`.

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.

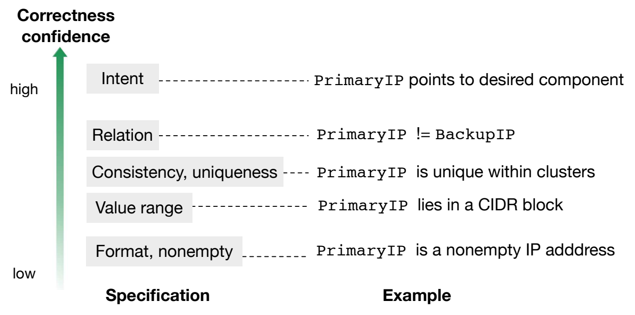
### 2.2 Configuration validation

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 $C$.

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 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 [15] and CloudStack [3] is mixed with other code and scattered in different source regions.



Correctness confidence
high
low
Intent
Relation
Consistency, uniqueness
Value range
Format, nonempty
Specification
Example
PrimaryIP points to desired component
PrimaryIP != BackupIP
PrimaryIP is unique within clusters
PrimaryIP lies in a CIDR block
PrimaryIP is a nonempty IP adddress

Figure 2: The spectrum of typical configuration specifications.

## 3. Design Considerations

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

### 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 configuration 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 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 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 shows 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 last snippet in Listing 3. 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 configuration validation code and adding correctness constraints on top of existing diverse configurations.

In the context of cloud-scale systems, the characteristics of the configurations (§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

```
// 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;
  }
}
```

```
// 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);
  }
}
```

```
// 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: Three imperative validation snippets that prescribe the implementation details for each constraint.

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.

### 3.2 Tooling support

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.

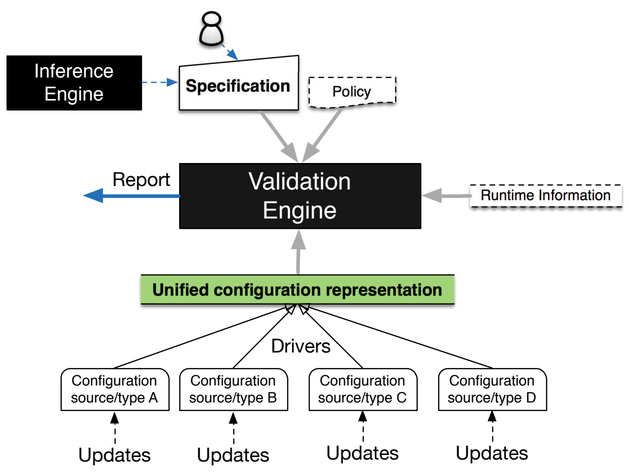

Inference Engine
Specification
Policy
Report
Validation Engine
Runtime Information
Unified configuration representation
Drivers
Configuration source/type A
Configuration source/type B
Configuration source/type C
Configuration source/type D
Updates
Updates
Updates
Updates

Figure 3: Architecture of ConfValley.

## 4. System Design

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

### 4.1 Overview

Figure 3 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. 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 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

#### 4.2.1 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 $\boldsymbol{r}(x)$, 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.*, $\boldsymbol{r}(x)$, $\boldsymbol{r}(y)$, and $\boldsymbol{r}(z)$, 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 (§4.2.2). 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 (§4.2.2) 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.

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

Table 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

| 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 |

Table 1: Examples of configuration notations and their meanings

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, *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.

### 4.2.3 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 `$_`.

### 4.2.4 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`

```
⟨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 4: *CPL* grammar

```
/* 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 5: Example validation specifications in *CPL*

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).

### 4.2.5 Grammar and Examples

Listing 4 shows the main CPL grammar. A simple validation statement in CPL is ⟨*domain*⟩ → ⟨*predicate*⟩, For example, `$OSBuildPath → path & exists` checks if each instance of configuration `OSBuildPath` is an extant path. Listing 5 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.

### 4.2.6 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 §4.2.1). 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.

## 4.3 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.

## 4.4 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.

## 4.5 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 [34, 40]. Black-box approaches mine constraints from configuration data [31, 37, 38, 42]. 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 [31, 37, 38, 42], 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).

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

Table 2: Driver code to convert different types of configuration data in Microsoft Azure into a unified representation.

```
           $s.k1 → ip
           compartment s{
             $k1 → unique
             $k1 ≤ $k2
           }
                   ⇩
     compartment s{
       $k1 → ip & unique & $_ ≤ $k2
     }
```
(a). aggregate predicates with same domains to avoid repeated instance discovery

```
     $s.k1 → ip & unique & [$range]
     $s.k2 → ip & unique & [$range]
                 ⇩
   $s.k1,$s.k2 → ip & unique & [$range]
```
(b). aggregate domains with same predicates to reuse internal predicate memory objects

```
$k1 → string & nonempty & {'compute','storage'}
                 ⇩
       $k1 → {'compute','storage'}
```
(c). omit constraints implied by others to avoid unnecessary checking

Figure 4: Examples of CPL compiler optimizations.

## 5. 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 [2]. 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 2 shows the code size of such drivers for common configuration formats used in Microsoft Azure.

### 5.1 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.

### 5.2 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 (§4.2.2) 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 domain. 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 shows examples of these compiler optimizations.

| **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: 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.

## 6. 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 following questions: First (§6.2), how expressive and easy is it to write configuration validation specifications in our proposed language CPL? Second (§6.3), how effective is the inference engine in ConfValley? Third (§6.4), how effective are human-written and inferred validation specifications in catching configuration errors? Lastly (§6.5), 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.

### 6.1 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 writing validation in CPL with the *ad hoc* code in these validation tools. In addition, we evaluate the effectiveness and efficiency of the inference and validation engines in ConfValley using various kinds of configuration data in Microsoft Azure systems.

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 [7] and took initial efforts, *e.g.* adding type validation. But a concrete solution is missing. A third-party tool called Rubick [17] 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.

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

Table 4: Express validation code in open source cloud systems into CPL specifications. *: time it takes to understand the original code and write specifications in CPL.

### 6.2 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 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 also shows that on average one third of the translated specifications can be automatically inferred by our inference components.

Table 4 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

| Config. | # of config. analyzed | | # of constraints inferred | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Class | Instance | Type | Nonempty | Range | Equality | Consistency | Uniqueness | Total |
| Type A | 1,391 | 67,231 | 1,026 | 317 | 203 | 367 | 722 | 71 | 2,706 |
| Type B | 162 | 2,306,935 | 126 | 114 | 62 | 1 | 29 | 43 | 375 |
| Type C | 95 | 2,253 | 93 | 75 | 18 | 0 | 75 | 0 | 261 |

Table 5: Validation constraint inference on three kinds of configuration data in Microsoft Azure. *: parenthesized values are the corresponding number of configuration instances analyzed.

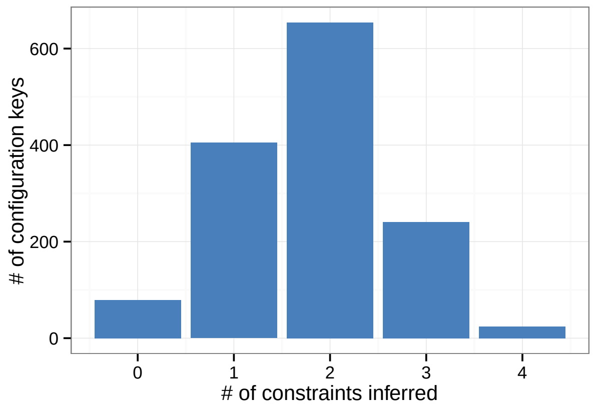


Figure 5: Histogram of number of inferred constraints on a type of Microsoft Azure configuration data with 1,391 configuration keys and 67,231 instances.

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 and Table 4). 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.

## 6.3 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 5 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`.

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

Table 6: Running expert-written validation specifications on three latest configuration data branches in Microsoft Azure.

Table 5 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 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.

## 6.4 Preventing Configuration Errors

The ultimate goal of a configuration validation framework is to prevent configuration errors from being introduced into

| Config. | Instances | Specs | | Time | | | |
|---|---|---|---|---|---|---|---|
| | | Count | Source | Sequential | P10.Min | P10.Median | P10.Max |
| Type A | 44,102 | 182 | Inferred, optimized | 10 | 2 | 2 | 4 |
| Type B | 1,969,588 | 62 | Human-written | 518 | 49 | 52 | 208 |
| Type C | 1,529 | 95 | Inferred | 0.4 | 0.3 | 0.3 | 0.3 |

Table 8: Validation latency (in seconds) on three types of configuration data in Microsoft Azure. P10.(Min, Median, Max) means splitting the specifications into 10 pieces, validating each piece in parallel, and measuring the (min, median, max) validation time of the 10 jobs.

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

Table 7: Running inferred validation specifications on three latest configuration data branches in Microsoft Azure.

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 6 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 7 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.

## 6.5 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 8 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

| 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 |

Table 9: Inference latency (in seconds) on three types of configuration data. *: the time it takes to parse all configuration sources and convert into unified representations.

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 splitting the specifications into 10 partitions and running 10 validation jobs in parallel. We can see from Table 8 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 9 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.

## 7. Limitations

While the evaluation demonstrates the advantages of our validation framework, 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.

## 8. Related Work

**Misconfiguration detection, diagnosis and fix.** A wide body of work has been done to detect and troubleshoot misconfigurations [21–23, 26, 29, 29, 36–39, 41, 42]. For example, ConfAid [22] and X-ray [20] use dynamic information-flow tracking to find possible configuration errors that cause failures or performance anomalies; AutoBash [36] speculatively executes processes and tracks causality to automatically fix misconfigurations; Strider [38] and PeerPressure [37] leverage a set of configuration settings from different machines to narrow down the problematic configuration on a sick machine; EnCore [42] learns configuration rules and exploits environment information to detect misconfigurations.

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.

**System resilience.** Research has been done to test system resilience to misconfigurations [24, 30, 40]. ConfErr [30] uses a human error model from psychology and linguistics to inject misconfigurations into systems. SPEX [40] 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.

**Configuration languages.** There is growing interest in new configuration languages [25, 27, 28, 33] 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 [28] automates the generation of device-native configurations with *configlets* in a template language. Loo et al. [33] adopt Datalog to express routing protocols in a declarative fashion. COOLAID [25] 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.

**Configuration management.** A variety of tools such as Puppet [16], Chef [6] and Salt [18] 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.

## 9. 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.

## 10. Acknowledgements

We thank our shepherd, Julia Lawall, 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.

## References

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

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

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

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

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

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

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

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

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

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

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

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

[13] 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.

[14] 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.

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

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

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

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

[19] 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.

[20] ATTARIYAN, M., CHOW, M., AND FLINN, J. 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* (Hollywood, CA, USA, 2012), OSDI'12, pp. 307–320.

[21] ATTARIYAN, M., AND FLINN, J. Using causality to diagnose configuration bugs. In *Proceedings of the 2008 USENIX Annual Technical Conference* (Boston, Massachusetts, 2008), ATC'08, pp. 281–286.

[22] ATTARIYAN, M., AND FLINN, J. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada, 2010), OSDI'10, pp. 1–11.

[23] BAUER, L., GARRISS, S., AND REITER, M. K. Detecting and resolving policy misconfigurations in access-control systems. In *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies* (Estes Park, CO, USA, 2008), SACMAT '08, pp. 185–194.

[24] CADAR, C., DUNBAR, D., AND ENGLER, D. 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* (San Diego, California, 2008), OSDI'08, pp. 209–224.

[25] CHEN, X., MAO, Y., MAO, Z. M., AND VAN DER MERWE, J. Declarative configuration management for complex and dynamic networks. In *Proceedings of the 6th International Conference* (Philadelphia, Pennsylvania, 2010), Co-NEXT '10, pp. 6:1–6:12.

[26] DAS, T., BHAGWAN, R., AND NALDURG, P. Baaz: A system for detecting access control misconfigurations. In *Proceedings of the 19th USENIX Conference on Security* (Washington, DC, 2010), USENIX Security'10, pp. 11–11.

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

[28] ENCK, W., MCDANIEL, P., SEN, S., SEBOS, P., SPOEREL, S., GREENBERG, A., RAO, S., AND AIELLO, W. Configuration management at massive scale: System design and experience. In *Proceedings of the 2007 USENIX Annual Technical Conference* (Santa Clara, CA, 2007), ATC'07, pp. 6:1–6:14.

[29] FEAMSTER, N., AND BALAKRISHNAN, H. Detecting BGP configuration faults with static analysis. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation* (Boston, Massachusetts, 2005), NSDI'05, pp. 43–56.

[30] KELLER, L., UPADHYAYA, P., AND CANDEA, G. ConfErr: A tool for assessing resilience to human configuration errors. In *Proceedings of the 38th International Conference on Dependable Systems and Networks* (Anchorage, Alaska, USA, 2008), DSN'08, pp. 157–166.

[31] KICIMAN, E., AND WANG, Y.-M. Discovering correctness constraints for self-management of system configuration. Tech. Rep. MSR-TR-2004-22, Microsoft Research, March 2004.

[32] KUSHMAN, N., AND KATABI, D. Enabling configuration-independent automation by non-expert users. In *Proceed-*

*ings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada, 2010), OSDI'10, pp. 1–10.

[33] LOO, B. T., HELLERSTEIN, J. M., STOICA, I., AND RAMAKRISHNAN, R. Declarative routing: Extensible routing with declarative queries. In *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (Philadelphia, Pennsylvania, USA, 2005), SIGCOMM '05, pp. 289–300.

[34] RABKIN, A., AND KATZ, R. Static extraction of program configuration options. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA, 2011), ICSE '11, pp. 131–140.

[35] ROCKWOOD, B. The DevOps transformation. http://www.usenix.org/events/lisa11/tech/slides/rockwood.pdf.

[36] SU, Y.-Y., ATTARIYAN, M., AND FLINN, J. AutoBash: Improving configuration management with operating system causality analysis. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA, 2007), SOSP '07, pp. 237–250.

[37] WANG, H. J., PLATT, J. C., CHEN, Y., ZHANG, R., AND WANG, Y.-M. Automatic misconfiguration troubleshooting with PeerPressure. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation* (San Francisco, CA, 2004), OSDI'04, pp. 17–17.

[38] WANG, Y.-M., VERBOWSKI, C., DUNAGAN, J., CHEN, Y., WANG, H. J., YUAN, C., AND ZHANG, Z. Strider: A blackbox, state-based approach to change and configuration management and support. In *Proceedings of the 17th USENIX Conference on System Administration* (San Diego, CA, 2003), LISA '03, pp. 159–172.

[39] WHITAKER, A., COX, R. S., AND GRIBBLE, S. D. Configuration debugging as search: Finding the needle in the haystack. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation* (San Francisco, CA, 2004), OSDI'04, pp. 6–6.

[40] XU, T., ZHANG, J., HUANG, P., ZHENG, J., SHENG, T., YUAN, D., ZHOU, Y., AND PASUPATHY, S. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania, 2013), SOSP '13, pp. 244–259.

[41] YUAN, D., XIE, Y., PANIGRAHY, R., YANG, J., VERBOWSKI, C., AND KUMAR, A. Context-based online configuration-error detection. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference* (Portland, OR, 2011), ATC'11, pp. 28–28.

[42] ZHANG, J., RENGANARAYANA, L., ZHANG, X., GE, N., BALA, V., XU, T., AND ZHOU, Y. EnCore: 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* (Salt Lake City, Utah, USA, 2014), ASPLOS '14, pp. 687–700.