ConfValley: A Systematic Configuration Validation Framework for Cloud Services

Ryan Huang, Bill Bolosky, Abhisheek Singh, Yuanyuan Zhou
Misconfiguration is “expensive”
Configuration in cloud systems

**MonitorTenant** = true
**RequestRetries** = 3
**ProxyIPs** = 10.0.0.1, 10.0.0.2
**VLAN** = {
  "StartIP" : 10.53.129.1,
  "EndIP" : 10.53.129.2
}
**PurgeInterval** = 1000
**OSPath** = \share\OS\v2
**StoreAccountName** = billingsn1prod

<**TokenService** address="sn1" port="1000"
  thumbprint="1D594C... "
  thumbprintAlgorithm="SHA1 "
  description="Production sign key" />

...
Configuration in cloud systems

MonitorTenant = true
RequestRetries = 3
ProxyIPs = 10.0.0.1,10.0.0.2
VLAN = {
  "StartIP" : 10.53.129.1,
  "EndIP" : 10.53.129.2
}
PurgeInterval = 1000
OSPath = \share\OS\v2
StoreAccountName = billingsn1prod

<TokenService address="sn1" port="1000"
  thumbprint="1D594C..."
  thumbprintAlgorithm="SHA1"
  description="Production sign key"/>

...
Configuration in cloud systems

MonitorTenant = true
RequestRetries = 3
ProxyIPs = 10.0.0.1,10.0.0.2
VLAN = {
  "StartIP" : 10.53.129.1,
  "EndIP" : 10.53.129.2
}
PurgeInterval = 1000
OSPath = \share\OS\v2
StoreAccountName = billingsn1prod
<TokenService address="sn1" port="1000"
thumbprint="1D594C..."
thumbprintAlgorithm="SHA1"
description="Production sign key" />
...
Life of configuration in cloud environment

- editing
- review
- integration
- testing
- production
- diagnosis
- fix

Reactive, expensive
Proactive, lightweight
Proactive method – configuration validation

• **What:** check if configuration satisfies some explicit specs
  • e.g., LockboxPath should be an existing directory, LBAddress should be a unique IP

• **When:**

• **Benefits:** prevent damages to system, save diagnosis, fix efforts
Configuration validation in practice

• Inefficient, ad-hoc and late
  • Manual reviews of configuration changes
  • Bulky scripts and code scattered in different places
  • Invoked late at runtime

Cloud systems have many configurations that undergo frequent changes

• Consequences
  • Time-consuming 😞
  • Repeated efforts to write similar validations 😞
  • Insufficient validations and service disruptions 😞
Bad practice (1): imperative validation

```csharp
bool passed = true;
string[] ranges = IpRanges.Split(';');
foreach (string range in ranges) {
    string[] cidr = range.Split('/');
    if ((cidr.Length != 1 && cidr.Length != 2) ||
        !IsIPAddress(cidr[0])) {
        passed = false;
        break;
    }
    if (cidr.Length == 2) {
        UInt32 mask;
        if (!UInt32.TryParse(cidr[1], out mask) ||
            mask > 32) {
            passed = false;
            break;
        }
    }
}
configForValidation = new HashSet<string>();
configForValidation.Add("event.purge.interval");
configForValidation.Add("alert.wait");
...
Class<Type> 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(         
                "Please enter a positive value for parameter:" + config.Name);
        }
    } catch (NumberFormatException e) {
        throw new InvalidParameterValueException(         
            "Error trying to parse the integer value for:" + config.Name);
    }
}
```

Wanted: validate in **declarative** fashion

- **IpRanges is a list of IP range**
- **event.purge.interval, ... are positive integers**
Bad practice (2): validate instances

```csharp
Config configs = ParseConfigs(...);
...
foreach (Config.Datacenter datacenter in configs.Datacenters) {
    List<Config.Rack> racks = datacenter.GetRacks();
    foreach (Config.Rack rack in racks) {
        HashSet<string> idList = new HashSet<string>();
        List<Config.Blade> blades = rack.GetBlades();
        foreach (Config.Blade blade in blades) {
            string bladeId = blade.GetId();
            if (!IsGuid(bladeId)) {
                Console.WriteLine("ERROR: Invalid Blade Id: {0}\n", bladeId);
            } else if (!idList.Add(id)) {
                Console.WriteLine("ERROR: Duplicated Blade Id {0}\n", bladeId);
            }
        }
    }
}
```

**Wanted:** validate classes of configuration

Finding instances of `Blade.Id` is tied with the checking logic.
Bad practice (3): validate too late

```java
public void maybeRestoreArchive() {
    restoreDirectories = getProperty("restore_directories");
    if (Strings.isNullOrEmpty(restoreDirectories))
        return;
    for (String dir : restoreDirectories.split(",")) {
        File[] files = new File(dir).listFiles();
        if (files == null) {
            throw new RuntimeException("Unable to list directory " + dir);
        }
        for (File fromFile : files) {
            String command = restoreCommand.replace("%from", fromFile.getPath());
            command = command.replace("%to", toFile.getPath());
            try {
                exec(command);
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

**Wanted:** separate, early validation activity

**Check** `restore_directories` right before restoring archive
ConfValley validation framework

Goal

- A simple language (CPL) to write validation specs
  - Easy to write, read

- Infer many specs automatically
  - Reduce manual efforts

- Separate validation policy
  - Flexible
  - Assign priorities to validate critical parameters first
  - Actions for failed validation

- Support validation in different scenarios
  - Comprehensive
  - Edit-time: instant validation in configuration IDE
  - In production: interactive console to quick check with “one-liner” spec
  - Continuous service: (re)validate with given spec as configuration is updated
Overview of ConfValley

Inference Engine → CPL spec

CPL spec → Validation Engine

Validation Engine → Runtime info

Runtime info → policy

Inference Engine → Validation report

Validation report → Unified configuration representations

Unified configuration representations → Drivers

Drivers:
- Configuration source/type A
- Configuration source/type B
- Configuration source/type C
- Configuration source/type D

Updates for:
- Configuration source/type A
- Configuration source/type B
- Configuration source/type C
- Configuration source/type D
Design goals of CPL

• Describes constraints declaratively
• Refers to configurations conveniently
  • Independent of underlying representations
  • Classes of configurations
• Specifies the validation scope precisely
• Allows extensions to the language
• Encourages modular validation specifications
• Supports convenient debugging constructs
Declarative constraints in *CPL*: predicate

• A predicate is used to characterize a boolean property

  • X is an IP address
  • X lies in the range from 1 to 10
  • X is consistent
  • X is greater than Y
  • X has read-only permission

• CPL provides common predicate primitives

  \( \langle \text{primitive} \rangle ::= \langle \text{type} \rangle | \langle \text{relation} \rangle | \langle \text{match} \rangle | \langle \text{range} \rangle | \langle \text{consistent} \rangle | \langle \text{unique} \rangle | \langle \text{order} \rangle | \sim \langle \text{id} \rangle \ldots \)
Abstract configuration instances: domain

• A *domain* is the source that provides instances for predicates

• Example:
  • Domain $C = \{x, y, z\}$, predicate $r$ (is an integer)

    $r (C) := r (a) \mid a \in C$

  • Predicate $s$ (smaller than 10), $t := r \& s$

    $t (C) := r (a) \& s (a) \mid a \in C$

• Domain in *CPL* is mainly an abstraction for a group of related configuration instances

    $\langle\text{domain}\rangle := \$\langle\text{qid}\rangle$
Domain notation in \textit{CPL}

\textbf{Basic form}: (optional) scope + configuration key

\textbf{Advanced form}: fully qualified scope and key, wild cards

<table>
<thead>
<tr>
<th>Notation</th>
<th>Refers to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cloud.Tenant.SecretKey</td>
<td>\textit{SecretKey} in all tenants in all clouds</td>
</tr>
<tr>
<td>Cloud::CO2test2.Tenant.SecretKey</td>
<td>\textit{SecretKey} in all tenants in cloud CO2test2</td>
</tr>
<tr>
<td>Cloud::$\text{CloudName}.Tenant.SecretKey</td>
<td>\textit{SecretKey} in all tenants in clouds named with values of $\text{CloudName}}</td>
</tr>
<tr>
<td>Cloud[1].Tenant::SLB.SecretKey</td>
<td>\textit{SecretKey} in tenant SLB in the first cloud</td>
</tr>
<tr>
<td>*.SecretKey</td>
<td>\textit{SecretKey} under any top-level scope</td>
</tr>
<tr>
<td>*IP</td>
<td>Any parameter with a key that ends with IP in any scope</td>
</tr>
</tbody>
</table>
Other core constructs in CPL

• **Transformation**: transform values in domain to apply to a predicate

  Predicate \( r(x) \): \( x \) is equal to “eurosys”
  
  But \( x \) can be in mixed-cases...

  Define new predicate \( s(x) \): \( x \) is case-insensitively equal to “eurosys”?

  Use to-lower-case function \( f \) to transform domain, then \( r \) on \( f(x) \)!

  **Reuse predicates without defining new ones!**

• **Quantifier**: the quantity of elements in a domain that should satisfy a predicate.

  \( \exists \): at least one configuration instance in the domain should satisfy the predicate

  \( \forall \): every configuration instance in the domain should satisfy the predicate

  \( \exists! \): exactly one configuration instance in the domain satisfies the predicate
\textit{CPL: Configuration Prediction Language}
CPL example

/* 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.cpl'
let UniqueCIDR := unique & cidr

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

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

// consistent fill factors within a data center
#\{Datacenter\} $Machinepool.FillFactor# \to\ consistent
CPL example

CPL example

/* 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.cpl'
let UniqueCIDR := unique & cidr

// machinepool in cluster is
// one of the defined machinepool names

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

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

Original	
  | imperative	
  | validation	
  | code
One-line CPL spec

HashSet<string> machinePoolList = new HashSet<string>();
foreach (Datacenter datacenter in Datacenters)
{
  foreach (MachinePool machinePool in datacenter.MachinePools)
  {
    machinePoolList.Add(machinePool.Name);
  }
}
foreach (Cluster cluster in Datacenter.Clusters)
{
  foreach (MachinePool machinePool in cluster.MachinePools)
  {
    if (!MachinePoolList.Contains(machinePool.Name))
    {
      Console.WriteLine("ERROR: Cluster contains unknown "+"MachinePool: {0}", machinePool.Name);
      passed = false;
    }
  }
}
More CPL examples

```cpl
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')) {  // match and split
    $Fabric::$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]
```
Use a light-weight black-box approach: Mine large samples of configuration instances, apply inference.

<table>
<thead>
<tr>
<th>Intent</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrimaryIP points to correct component</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrimaryIP (!=) BackupIP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Consistency, uniqueness</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrimaryIP is unique within a cluster</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value range</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrimaryIP lies in a CIDR block</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Format, type, nonempty</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrimaryIP is a nonempty IP address</td>
</tr>
</tbody>
</table>

Output: CPL specs
Implementations
ConfValley prototype and CPL

• 9,000 lines of C# code for ConfValley
• 19 predicate primitives, 13 built-in transformation functions in CPL

<table>
<thead>
<tr>
<th>Predicate primitive</th>
<th>Transformation function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>split</td>
</tr>
<tr>
<td>Nonempty</td>
<td>foreach</td>
</tr>
<tr>
<td>Range</td>
<td>union</td>
</tr>
<tr>
<td>Match</td>
<td>at</td>
</tr>
<tr>
<td>Relation</td>
<td>replace</td>
</tr>
<tr>
<td>Unique</td>
<td>lower</td>
</tr>
<tr>
<td>Consistent</td>
<td></td>
</tr>
<tr>
<td>Expires</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Drivers to parse existing configurations

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Driver code (LOC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic XML</td>
<td>400</td>
</tr>
<tr>
<td>Type* A</td>
<td>30</td>
</tr>
<tr>
<td>Type B</td>
<td>30</td>
</tr>
<tr>
<td>Type C</td>
<td>150</td>
</tr>
<tr>
<td>Type D</td>
<td>80</td>
</tr>
<tr>
<td>Type E</td>
<td>50</td>
</tr>
</tbody>
</table>

*: Different types of configurations are in different representations used by different components
Evaluation
Rewrite existing validation code in CPL (1)

<table>
<thead>
<tr>
<th>System</th>
<th>Config</th>
<th>Original code</th>
<th>Specs in CPL</th>
<th>Dev. time (man-hour)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft</td>
<td></td>
<td></td>
<td>LOC</td>
<td>LOC</td>
</tr>
<tr>
<td>Azure</td>
<td>Type A</td>
<td>800+</td>
<td>50</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>Type B</td>
<td>3300+</td>
<td>109</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>Type C</td>
<td>180+</td>
<td>14</td>
<td>6</td>
</tr>
</tbody>
</table>

Expressed in 10x fewer lines of specs from Microsoft Azure
Rewrite existing validation code in CPL (2)

<table>
<thead>
<tr>
<th>System</th>
<th>Original code</th>
<th>Specs in CPL</th>
<th>Dev. time (man-hour)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LOC</td>
<td>LOC</td>
<td>Count</td>
</tr>
<tr>
<td>OpenStack</td>
<td>480</td>
<td>40</td>
<td>19</td>
</tr>
<tr>
<td>CloudStack</td>
<td>340</td>
<td>18</td>
<td>15</td>
</tr>
</tbody>
</table>

Expressed in 10x fewer lines of specs

from open-source systems
Automatic inference

Inference on several types of configuration data inside **Microsoft Azure**

<table>
<thead>
<tr>
<th>Config.</th>
<th># of config. analyzed</th>
<th># of specs inferred</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Keys</td>
<td>Instances</td>
</tr>
<tr>
<td>Type A</td>
<td>1391</td>
<td>67,231</td>
</tr>
<tr>
<td>Type B</td>
<td>162</td>
<td>2,306,935</td>
</tr>
<tr>
<td>Type C</td>
<td>95</td>
<td>2,253</td>
</tr>
</tbody>
</table>

70-80% accuracy
Preventing real-world misconfigurations (1)

<table>
<thead>
<tr>
<th>Config.</th>
<th>Reported errors</th>
<th>False positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch* A</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>Branch* B</td>
<td>15</td>
<td>5</td>
</tr>
<tr>
<td>Branch* C</td>
<td>16</td>
<td>3</td>
</tr>
</tbody>
</table>

Using inferred CPL specs on latest configuration data in Microsoft Azure

Example error: empty `ReplicaCountForCreateFCC` which caused deployment incidents before.

*: different branches are for different deployment environments
Preventing real-world misconfigurations (2)

Using manual-written CPL specs on latest configuration data in Microsoft Azure

Example error: length of MACRanges ≠ length of IPRanges; inconsistent MuxJumboPacketSize, MonitorIfSessionsHung; missing IDnsFqdn;
Conclusion

- Misconfiguration is an expensive issue for cloud services
- We present a framework to easily and systematically validate configurations with a simple validation language CPL
- CPL expressed the ad-hoc validation code from Microsoft Azure and open-source cloud systems in 10x fewer lines
- Using CPL specs, we detected a number of misconfigurations in the latest configuration data in Microsoft Azure

Configuration validation should and can be made an ordinary part of cloud service life cycle!
Thanks!

Q&A
Related work

- Misconfiguration detection
  - CODE [USENIX ‘11], EnCore [ASPLOS ‘14]

- Misconfiguration diagnosis
  - STRIDER [LISA ‘03], PeerPressure [OSDI ‘04], Chronous [OSDI ‘04], ConfAid [OSDI ‘10]

- Misconfiguration fix
  - AutoBash [SOSP ‘07], KarDo [OSDI ‘10]

- System resilience
  - ConfErr [DSN ‘08], SPEX [SOSP ‘13]

- Configuration Language
  - PRESTO [USENIX ‘07], COOLAID [CoNEXT ‘10]
FAQ

• How fast is the validation and inference?
• What kind of requirements are hard to express in CPL?
• How to extend CPL?
• How about a new configuration language?
• Is it feasible to assume that users of CPL have expertise to write validation specs?
• How severe are the detected misconfigurations?
Limitations

• CPL has limited ability to express complex, dynamic validation requirements
• CPL is validating generic configuration files and has limit support for domain-specific configurations, e.g., network configurations
• Passing validation does not guarantee configuration error-free
• Not all types of configurations benefit a lot from validation
## Validation performance

<table>
<thead>
<tr>
<th>Config.</th>
<th>Instances</th>
<th>CPL specs</th>
<th>Time (second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type A</td>
<td>44,102</td>
<td>182</td>
<td>10</td>
</tr>
<tr>
<td>Type B</td>
<td>1,969,588</td>
<td>62</td>
<td>518</td>
</tr>
<tr>
<td>Type C</td>
<td>1,529</td>
<td>95</td>
<td>0.4</td>
</tr>
</tbody>
</table>

Running CPL specs on configuration data in Microsoft Azure

*: P10 is the splitting the CPL specs in 10 folds and running in parallel
## Inference performance

<table>
<thead>
<tr>
<th>Config.</th>
<th>Instances</th>
<th>Time (second)</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Total</td>
<td>Parsing</td>
<td>Inference</td>
<td></td>
</tr>
<tr>
<td>Type A</td>
<td>67,231</td>
<td>19.7</td>
<td>19.5</td>
<td>0.2</td>
<td></td>
</tr>
<tr>
<td>Type B</td>
<td>2,306,935</td>
<td>82</td>
<td>75</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Type C</td>
<td>2,253</td>
<td>0.09</td>
<td>0.08</td>
<td>0.01</td>
<td></td>
</tr>
</tbody>
</table>
Automatic inference: histogram

On config. data A:
1391 keys, 67231 instances

E.g., IncidentOwner = “Deployment Engineering”
Performance optimizations

• Finding instances for configuration notation query
  • In critical path: a moderate-size validation => 4,600,000+ queries
  • Cache + Trie => 5x-40x improvement

• Optimizer to re-write specification file

  $$s.k1 \rightarrow ip$$
  $$compartment s\{$$
  $$\quad k1 \rightarrow unique$$
  $$\quad k1 \leq k2$$
  $$\}$$

  $$s.k1 \rightarrow ip \& unique \& [range]$$
  $$s.k2 \rightarrow ip \& unique \& [range]$$

• Re-validation on updates: validate only dependent specs and configurations
Extending CPL

• Adding predicate primitives to CPL (e.g., keyword `reachable`)
  • The compiler is written in a modern compiler framework, easily extensible
  • Provided base classes of predicates to extend new predicates
    • On average 70 LOC for existing predicates

• Leverage transformation functions
  • User-defined transformation function as plug-ins without modifying the compiler
Feasibility of configuration validation

• Feasible for cloud environment: trained practitioners have expertise and experiences!
  • If SSL option enabled, the proxy URL be https
  • Empty FccDNSName caused incidents before
  • Disable ActiveDsts and set HomeDsts for storage cluster cause authentication outage
  • In Microsoft Azure, more than thousands of lines of validation code!