ConfValley: A Systematic Configuration Validation Framework for Cloud Services

Ryan Huang, Bill Bolosky, Abhishek Singh, Yuanyuan Zhou

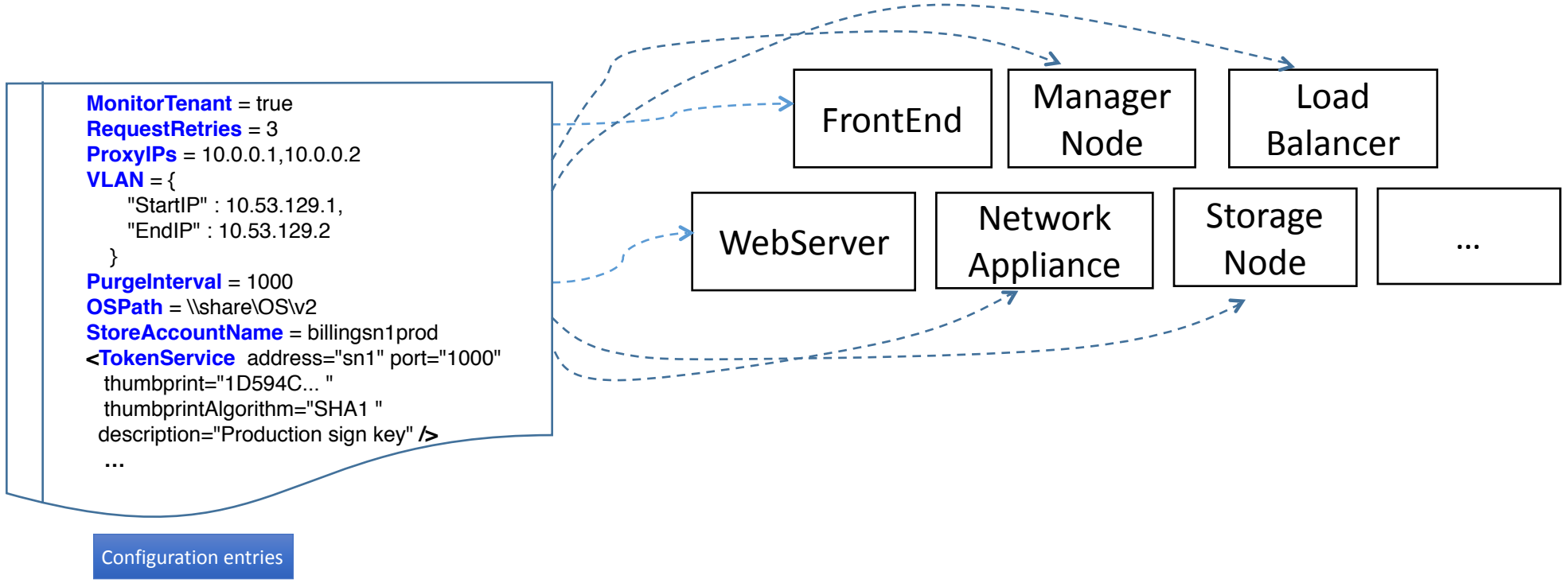
Misconfiguration is “expensive”

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 entries

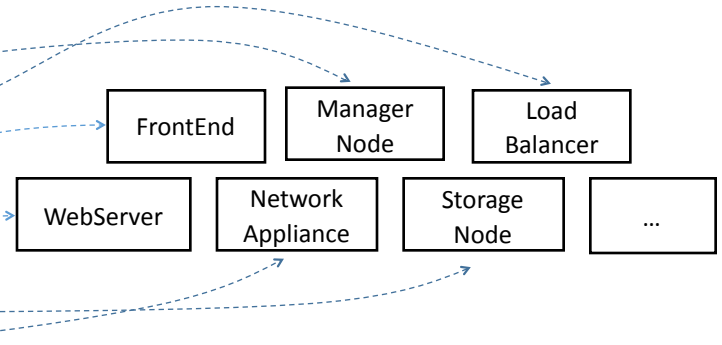
Configuration in cloud systems

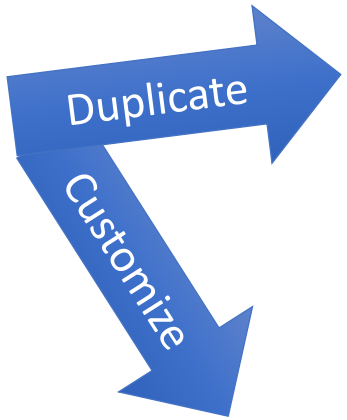


Configuration in cloud systems

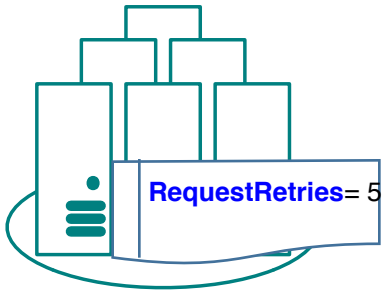
```
MonitorTenant = true
RequestRetries = 3
ProxyIPs = 10.0.0.1,10.0.0.2
VLAN = {
      "StartIP" : 10.53.129.1,
      "EndIP" : 10.53.129.2
   }
PurgeInterval = 1000
OSPath = \\share\OS\v2
StoreAccountName = billingsn1prod
<TokenService  address="sn1" port="1000"
  thumbprint="1D594C... "
  thumbprintAlgorithm="SHA1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

Configuration entries

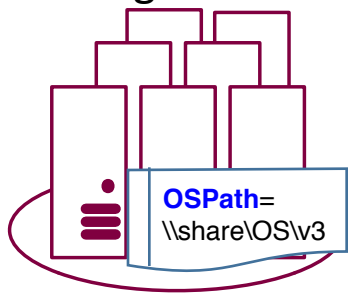




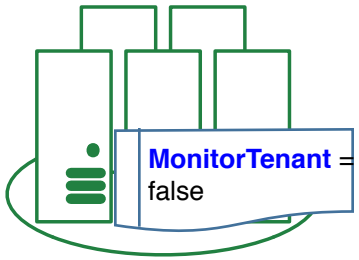
Computer Cluster



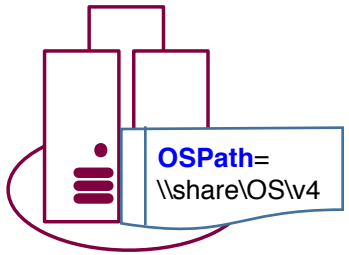
Storage Cluster



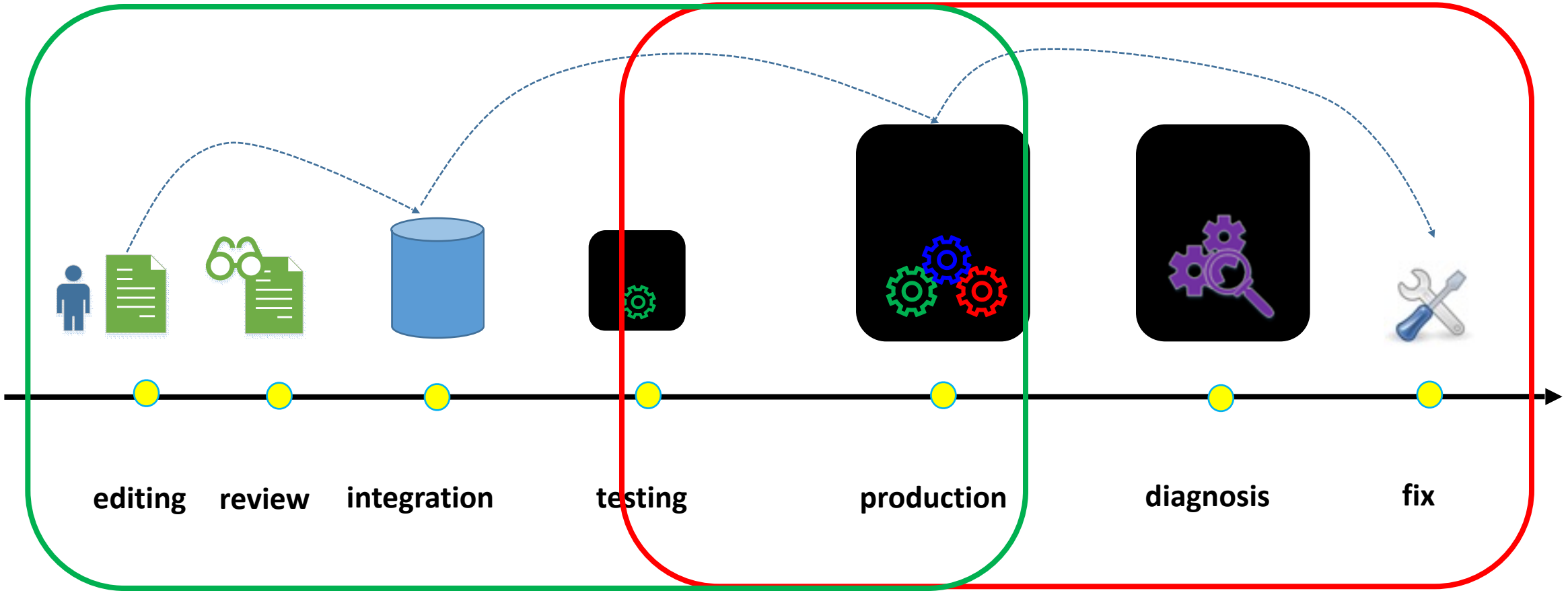
Next Gen Network Cluster



Small Storage Cluster



Life of configuration in cloud environment

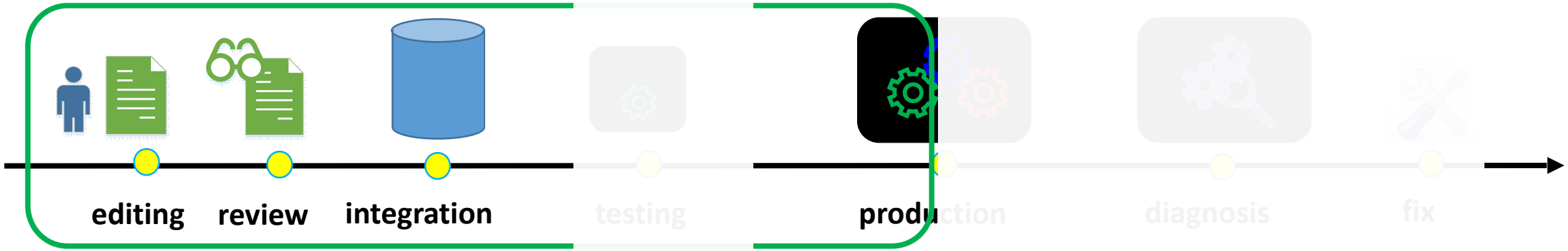


Proactive, lightweight

Reactive, expensive

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

```
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 > 32) {
         passed = false;
         break;
      }
   }
}
```

`IpRanges` is a list of IP range

```
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);
    ...
    lueException(
    or parameter:" + config.name);
    }
  } catch (NumberFormatException e) {
    throw new InvalidParameterValueException(
      "Error trying to parse the integer value for:" + config.name);
  }
}
```

`event.purge.interval`,… are positive integers

Wanted: validate in *declarative* fashion

Bad practice (2): validate instances


```
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}", bladeId);
                                                    d {0}", bladeId);
   }
}
```

Wanted: validate *classes* of configuration

Finding instances of `Blade.Id` is tied with the checking logic

Bad practice (3): validate too late

```
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

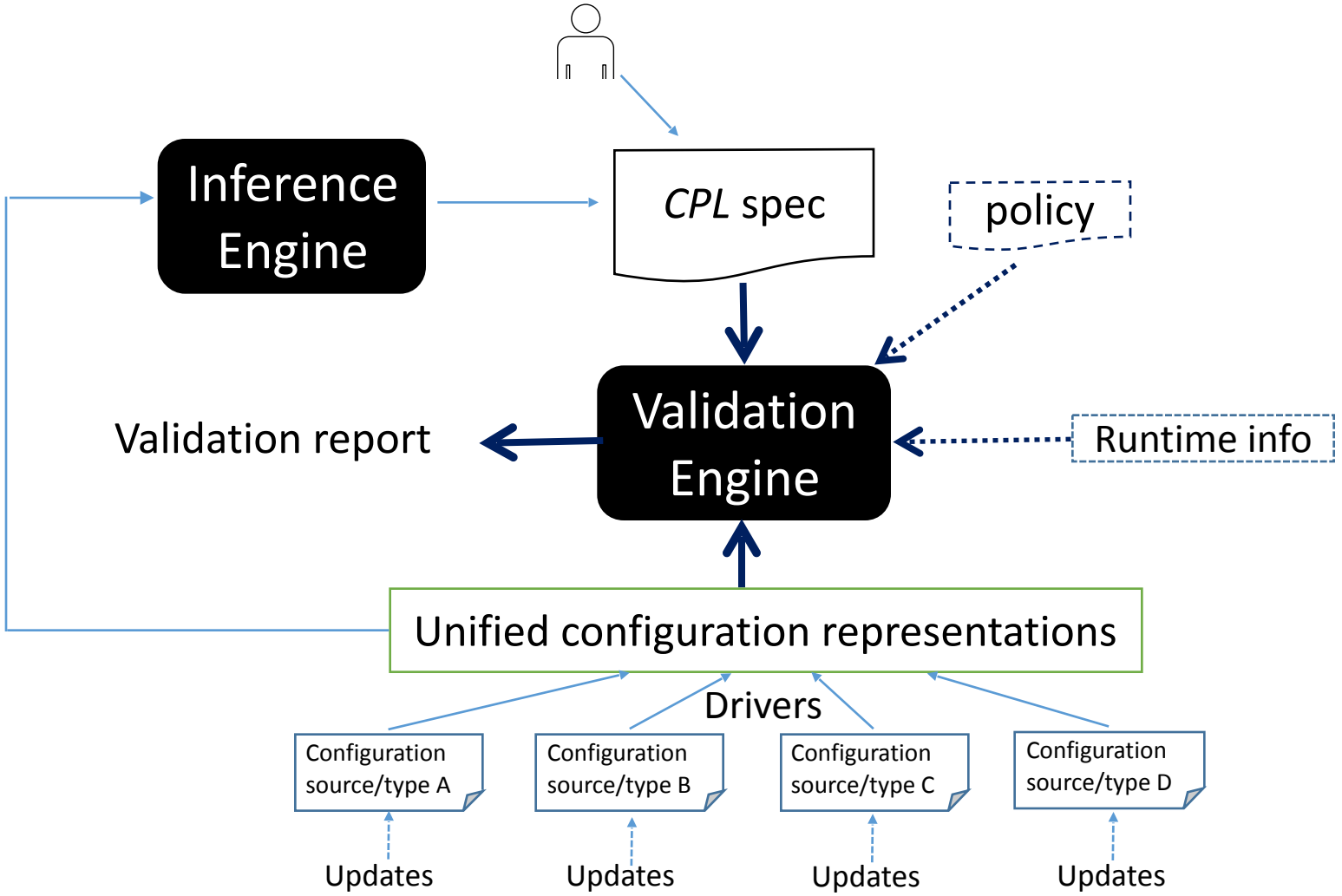
Check `restore_directories` right before restoring archive

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



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

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

- Recursive construction of predicates in CPL

⟨predicate⟩ ::= ⟨domain⟩ '→' ⟨predicate⟩
| 'if' '(' ⟨predicate⟩ ')' ⟨predicate⟩
| ⟨quantifier⟩ ⟨predicate⟩
| ⟨predicate⟩ '&' ⟨predicate⟩
| ⟨predicate⟩ '|' ⟨predicate⟩
| '~' ⟨predicate⟩
| ...

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)

$$\mathrm{r}(C) := \mathrm{r}(a) \mid a \in C$$

  - Predicate s (smaller than 10), t: $= \mathrm{r}\ \&\ \mathrm{s}$

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

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

⟨domain⟩ ::= ‘$’ ⟨qid⟩

Domain notation in *CPL*

Basic form: (optional) scope + configuration key

Advanced form: fully qualified scope and key, wild cards

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

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 n… to "eurosys" ?

Reuse predicates without defining new ones!

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

- **Quantifier**: the quantity of elements in a domain that should satisfy a predicate.
  - ∃ : at least one configuration instance in the domain should satisfy the predicate
  - ∀ : every configuration instance in the domain should satisfy the predicate
  - ∃!: exactly one configuration instance in the domain satisfies the predicate

CPL: Configuration Predicate Language

⟨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⟩ '#'
| ...
...

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
$Cluster.MachinePool → {$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
```

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
$Cluster.MachinePool → {$MachinePool.Name}

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

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

One-line *CPL* spec

VS

```
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;
  }
 }
}
```

Original imperative validation code

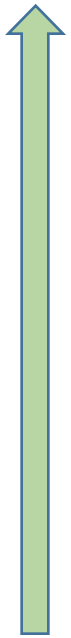
More CPL examples

```
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]
```

Automatic inference

Use a light-weight black-box approach:
Mine large samples of configuration instances, apply inference.



Intent	`PrimaryIP` points to correct component
Relation	`PrimaryIP != BackupIP`
Consistency, uniqueness	`PrimaryIP` is unique within a cluster
Value range	`PrimaryIP` lies in a CIDR block
Format, type, nonempty	`PrimaryIP` is a nonempty IP adddress

constraint

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

Predicate primitive
Type
Nonempty
Range
Match
Relation
Unique
Consistent
Expires
…

Transformation function
split
foreach
union
at
replace
lower
…

Drivers to parse existing configurations

Configuration	Driver code (LOC)
Generic XML	400
Type* A	30
Type B	30
Type C	150
Type D	80
Type E	50

*: Different types of configurations are in different representations used by different components

Evaluation

Rewrite existing validation code in CPL (1)

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

from Microsoft Azure

Expressed in 10x fewer lines of specs

Rewrite existing validation code in CPL (2)

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

from open-source systems

Expressed in 10x fewer lines of specs

Automatic inference

Config.	# of config. analyzed		# of specs inferred						
	Keys	Instances	Type	Nonempty	Range	Equality	Consistency	Unique	Total
Type A	1391	**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**

Inference on several types of configuration data inside Microsoft Azure

70-80% accuracy

Preventing real-world misconfigurations (1)

Config.	Reported errors	False positives
Branch* A	12	3
Branch* B	15	5
Branch* C	16	3

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)

Config.	Reported errors
Branch A	4
Branch B	2
Branch C	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

Config.	Instances	*CPL* specs	Time (second)			
			Sequential	P10.Min	P10.Median	P10.Max
Type A	44,102	182	10	2	2	4
Type B	1,969,588	62	518	49	52	208
Type C	1,529	95	0.4	0.3	0.3	0.3

Max 9min

Max 3.5min

Running *CPL* specs on configuration data in Microsoft Azure

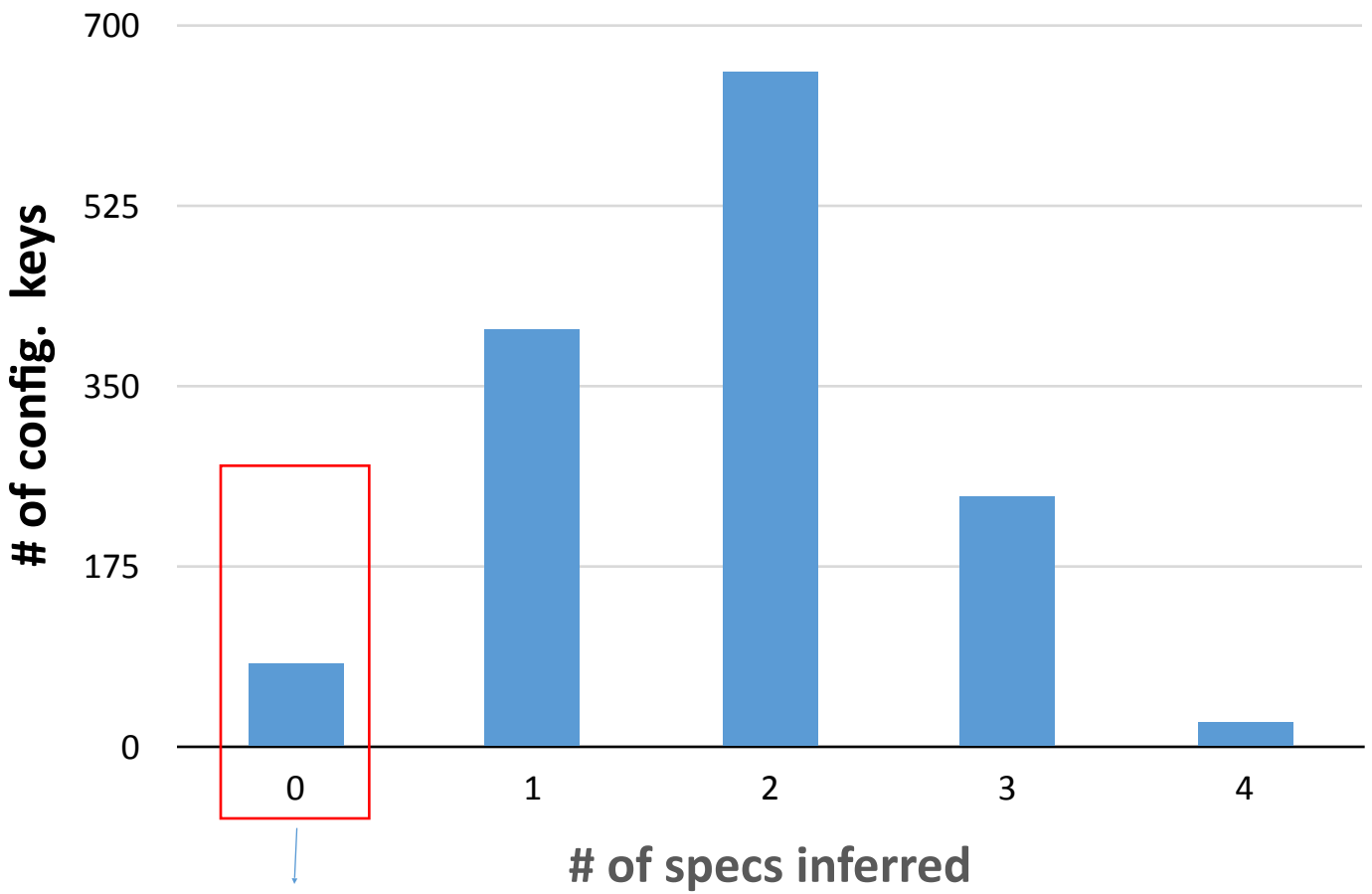
*: P10 is the splitting the CPL specs in 10 folds and running in parallel

Inference performance

Config.	Instances	Time (second)		
		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

Automatic inference: histogram

On config. data A:
1391 keys, 67231 instances

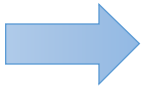


e.g., IncidentOwner = "Deployment Engineering"

Performance optimizations

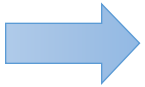
- Finding instances for configuration notation query
  - In critical path: a moderate-size validation => 4,600,000+ queries
  - Cache + Trie => 5x-40x improvement
- Optimizer to re-write specification file

```
$s.k1 → ip
compartment s{
    $k1 → unique
    $k1 ≤ $k2
}
```



```
compartment s{
    $k1 → ip & unique & $_ ≤
$k2
}
```

```
$s.k1 → ip & unique & [$range]
$s.k2 → ip & unique & [$range]
```



```
$s.k1,$s.k2 → 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!**