HUAWEI ENTERPRISE A BETTER WAY

Cloud Reliability
Decreasing outage frequency using fault injection

9th International Workshop on Software Engineering for Resilient Systems
September 4-5, 2017, Geneva, Switzerland

Prof. Dr. Jorge Cardoso
University of Coimbra
Portugal

Chief Architect for Cloud Operations and Analytics
Huawei Research, Munich
Germany

enterprise.huawei.com
HUAWEI TECHNOLOGIES CO., LTD.

Executive Summary

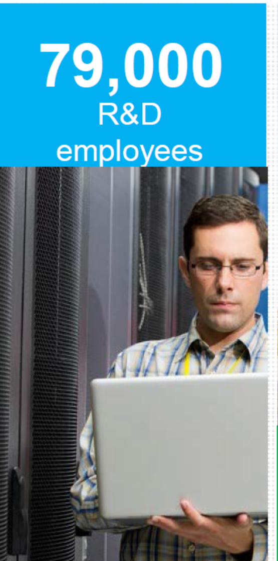
- *Title*: Cloud Reliability: Decreasing outage frequency using fault injection

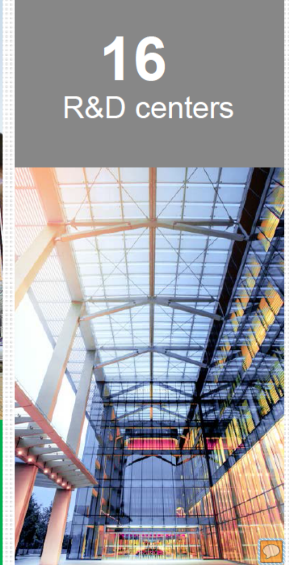
- *Abstract*: In 2016, Google Cloud had 74 minutes of total downtime, Microsoft Azure had 270 minutes, and 108 minutes of downtime for Amazon Web Services (see cloudharmony.com). Reliability is one of the most important properties of a successful cloud platform. Several approaches can be explored to increase reliability ranging from automated replication, to live migration, and to formal system analysis. Another interesting approach is to use software fault injection to test a platform during prototyping, implementation and operation. Fault injection was popularized by Netflix and their Chaos Monkey fault-injection tool to test cloud applications. The main idea behind this technique is to inject failures in a controlled manner to guarantee the ability of a system to survive failures during operations. This talk will explain how fault injection can also be applied to detect vulnerabilities of OpenStack cloud platform and how to effectively and efficiently detect the damages caused by the faults injected.

- *Acknowledgments*: This research was conducted in collaboration with Deutsche Telekom/T-Systems and with Ankur Bhatia from the Technical University of Munich to analyze the reliability and resilience of modern public cloud platforms.

- *Short CV*: Dr. Jorge Cardoso is Chief Architect for Cloud Operations and Analytics at Huawei's German Research Centre (GRC) in Munich. He is also Professor at the University of Coimbra since 2009. In 2013 and 2014, he was a Guest Professor at the Karlsruhe Institute of Technology (KIT) and a Fellow at the Technical University of Dresden (TU Dresden). Previously, he worked for major companies such as SAP Research (Germany) on the Internet of services and the Boeing Company in Seattle (USA) on Enterprise Application Integration. Since 2013, he is the Vice-Chair of the KEYSTONE COST Action, a EU research network bringing together more than 70 researchers from 26 countries. He has a Ph.D. in Computer Science from the University of Georgia (USA).

Huawei at a Glance





Winning Consumer Loyalty and Building Brand Influence

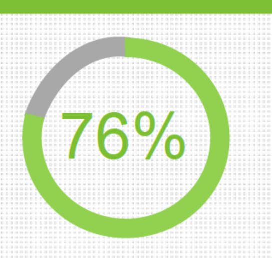
Interbrand's Top 100 Best Global Brands

No.88

Smartphone retail market share

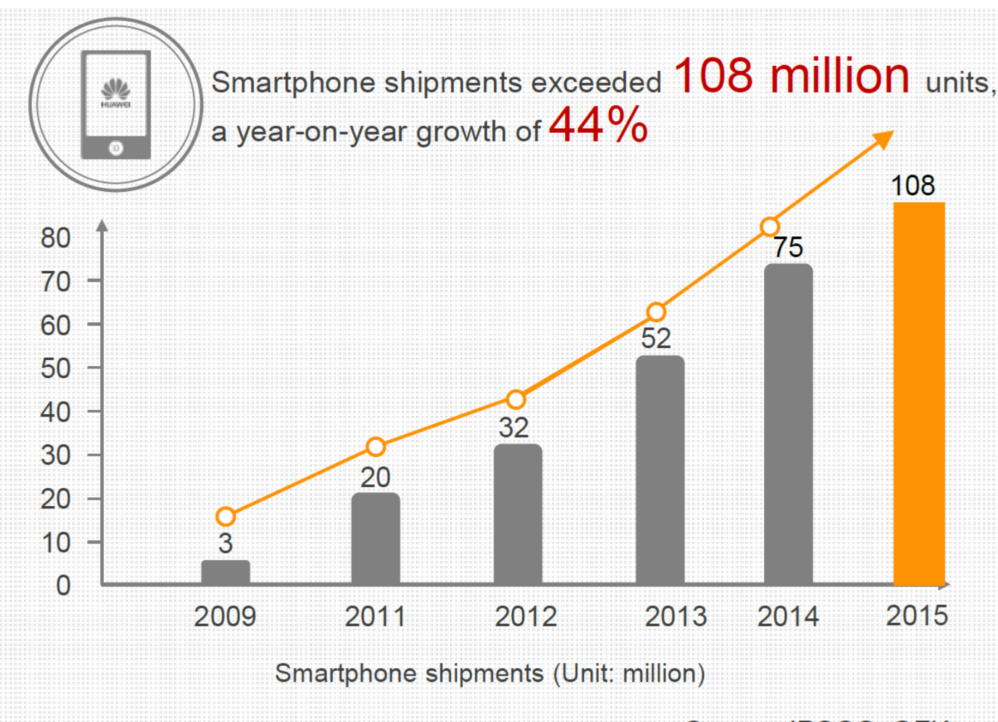
No.3

Increasing global brand awareness



NPS rose to 47%

No.3



Globalized Resource Deployment and Localized Business Operations

- Operations in 170+ countries; 176,000+ employees comprising 160+ nationalities worldwide; 72% localization rate.
- Huawei's global value chain allows the smooth transfer of capabilities around the world, develops and retains talent in local countries, and creates jobs and economic opportunities.

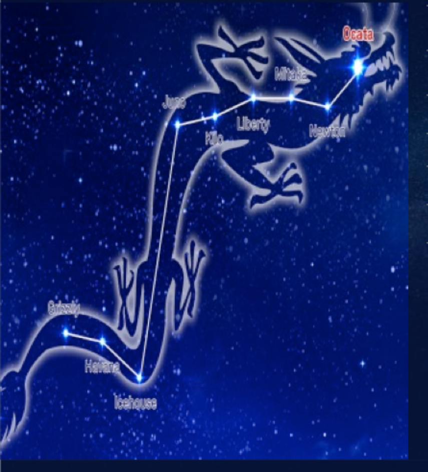
Huawei Public Cloud Products and Services

Category	Services
Enterprise Application	Workspace; SAP Hana/SAP Suite (IaaS)
Database	Relational Database Service; MapReduce
Security	Anti-DDoS
Solution	SAP Hana/SAP Suite (IaaS)
Management & Deployment	Cloud Eye; Identity and Access Management; MaaS
Compute	Elastic Cloud Server; Auto Scaling; Image Mgmt Service; Container Service; Dedicated Cloud Service
Storage	Elastic Volume Service; Volume Backup Service; Object Storage Service
Network	Elastic IP; Virtual Private Cloud; Elastic Load Balance; Direct Connect; DNS

Global Deployment of Public Cloud Services

- ◆ Huawei Enterprise Cloud (HEC) public cloud regions include Langfang and Suzhou in China.
- ● Open Telefonica Cloud worldwide regions include Mexico, Brazil, Chile, USA, Spain, Peru, Argentina, and Colombia.
- ■ China Telecom Cloud (CTC) public cloud regions include Guizhou and Beijing in China.
- ▲ Orange Cloud Business (OBS) public cloud regions include France, Singapore, USA, Holland, and Africa.
- ◆ Open Telekom Cloud (OTC), Germany

Strong Technical Leadership



Metric	Definition	Degree of difficulty	Rank of Ocata
Completed Blueprints	Feature number of upstream	*****	TOP 1
Resolved Bugs	Number of Resolved bugs	****	TOP 4
Reviews	Number of Reviews	***	TOP 6
Lines of Code	Lines of code contribution	**	TOP 7
Commits	Number of submissions	*	TOP 6

Key technical seats:

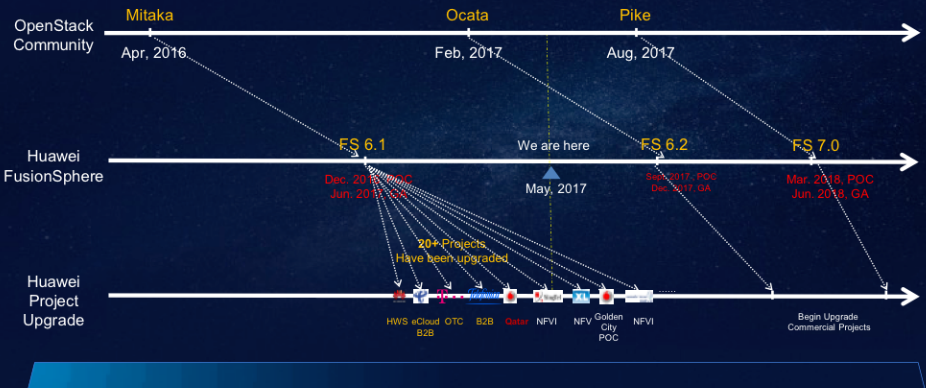
- 2 TC(Technical committee)
- 8 Project Leaders (including Nova, Cinder)
- 20 Core Members in 11 projects
- **800+ Developer over 3 continents**

Contribution Summary:

- Completed Blueprints: 232
- Resolved Bugs: 2144
- Lines of Code: 2,545,896
- Commits: 8165
- Initiated projects: 6

Note: numbers are based on Ocata release

Huawei Closely Follow the Latest Community Release



FusionSphere will be synchronize with the latest community release within ~6 months

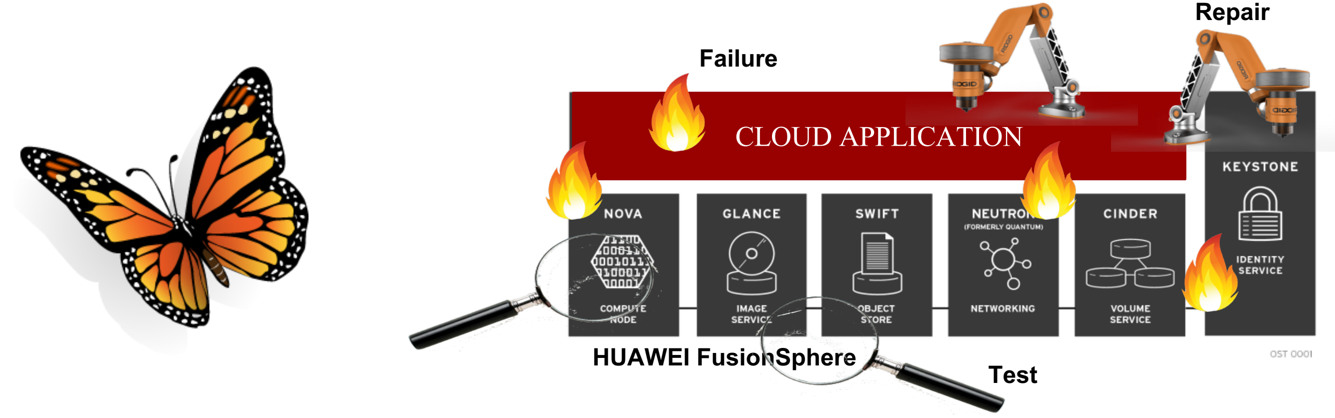
Customer Example: Public Cloud

Integrate with Adjacent Innovative Open Source Projects

- Huawei is committed to open source and is one of the leading members in the 15+ main communities
- Huawei aims to use and integrate various innovative open source projects to meet customer needs

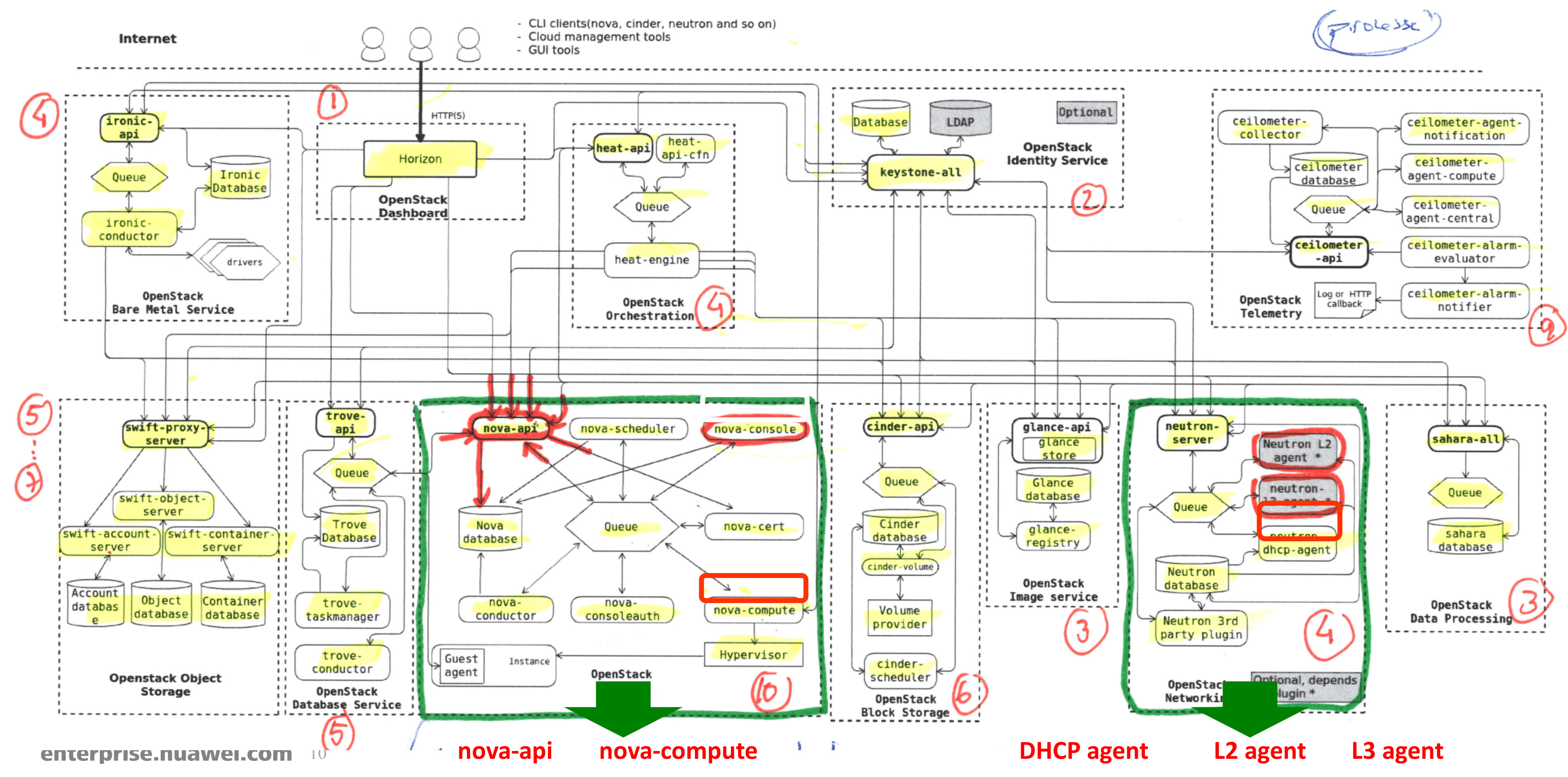
Fault Injection into Clouds

Enables to Automatically Test and Repair OpenStack and Cloud Applications

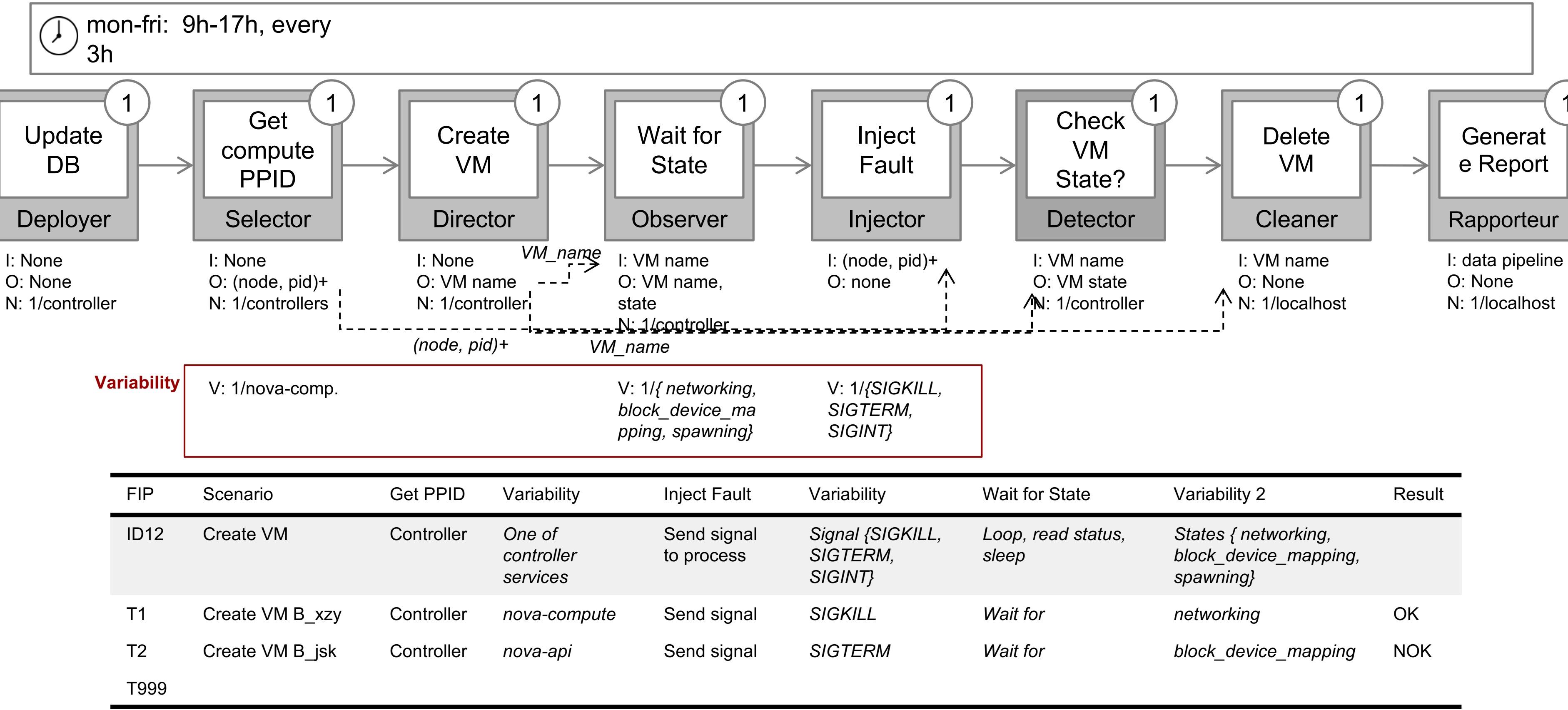


The system works by intentionally injecting different failures, test the ability to survive them, and learn how to predict and repair failures preemptively

OpenStack Architecture



Fault Injection Plans



FIP	Scenario	Get PPID	Variability	Inject Fault	Variability	Wait for State	Variability 2	Result
ID12	Create VM	Controller	*One of controller services*	Send signal to process	*Signal {SIGKILL, SIGTERM, SIGINT}*	*Loop, read status, sleep*	*States { networking, block_device_mapping, spawning}*	
T1	Create VM B_xzy	Controller	*nova-compute*	Send signal	*SIGKILL*	*Wait for*	*networking*	OK
T2	Create VM B_jsk	Controller	*nova-api*	Send signal	*SIGTERM*	*Wait for*	*block_device_mapping*	NOK
T999								

HUAWEI

Background - Requirement

- With improvements in processing power, network and storage technologies, cloud platforms have witnessed an unprecedented growth in **complexity**.
- Due to the increase in complexity, the need to **efficiently diagnose failures** in cloud platforms has also risen.
- Because of these challenges, cloud operators often develop new sets of tests to diagnose failures.
- But as mentioned before, cloud platforms are continuously evolving. They undergo modification and frequent updates, and have periodic release cycles. Hence, the tests developed become outdated and there is a constant need to modify them when a new release is available. Therefore, this approach is costly for the cloud operators.

Work done In collaboration with Ankur Bhatia from Technical University of Munich (TUM), Germany

Solution

- As with most software, the validation of all the modules of a cloud platform is done through a **test suite** containing a large number of unit tests. It is a part of the software development process where the smallest testable part of an application, called unit, along with associated control data are individually and independently tested.
- Executing unit tests is a very effective way to test code integration during the development of software. They are often executed to validate changes made by developers and to guarantee that the code is error free.
- Although unit tests are extremely useful for the purpose of development and integration, they are not meant to diagnose failures.

Fault Injection Plans

mon-fri: 9h-17h, every 3h

Step	Component	Details
1. Update DB	Deployer	I: None O: None N: 1/controller
1. Get compute PPID	Selector	I: None O: (node, pid)+ N: 1/controllers
1. Create VM	Director	I: None O: VM name N: 1/controller
1. Wait for State	Observer	I: VM name O: VM name, state N: 1/controller
1. Inject Fault	Injector	I: (node, pid)+ O: none
1. Check VM State?	Detector	I: VM name O: VM state N: 1/controller
1. Delete VM	Cleaner	I: VM name O: None N: 1/localhost
1. Generate Report	Rapporteur	I: data pipeline O: None N: 1/localhost

VM_name

Failure diagnosis system. Diagnoses failures in cloud platforms making use of unit tests and helps to detect nonresponsive and failed services.

Unit tests. It is a part of the software development process where the smallest testable part of a cloud platform, called unit, along with associated control data are individually and independently tested.

Unit tests for failure diagnosis. Reuse the already developed unit tests to test a cloud platform at runtime.

Challenges

Using unit tests for failure diagnosis presents a set of challenges

- Unit tests do not provide any information about the nonresponsive or failed services in cloud platforms.
  - The execution of unit tests generates a list of passed and failed tests. This list can help to locate software errors or to find issues with individual modules of the code but cannot diagnose failures as there are no relationships between unit tests and services running on a cloud platform.
- With the increase in codebase of cloud platforms, the number of unit tests also increases. Thus, it takes a considerable amount of time to execute them.

Experiments

- OpenStack has more than 1500 unit tests
- Used for development and integration
- Only validate APIs
- Time consuming
  - E.g., unit tests to create a VM, uploading a large operating system image, etc., need a few minutes to execute.
  - It can take up to 3 to 4 hours to execute all unit tests. Considering the reliability requirements of 99.95%, cloud platforms can have a downtime of only 21.6 minutes per month.
  - Hence, the time required to execute unit tests is considerably high.
- Not able to directly detect services that are not functioning as expected

General Approach

The approach of 3 phases efficiently diagnoses a cloud platform

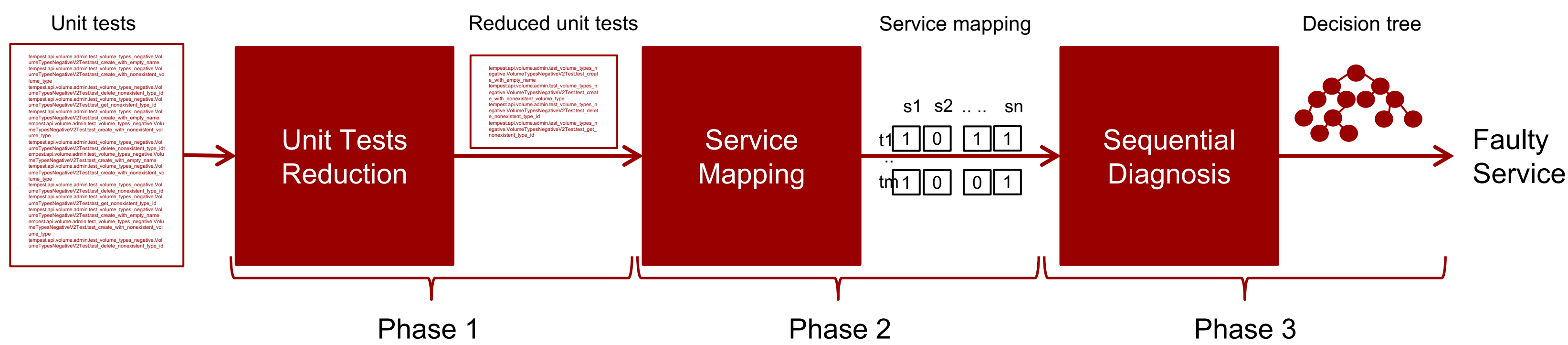
- 1. Reduce the number of unit tests using the set cover algorithm
- 2. Establish relationships between the reduced unit tests and the services running on a cloud platform. These relationships help to determine nonresponsive or failed services. This is done by simulating failure of services in cloud platforms and running unit tests in this environment.
- 3. Construct a decision tree based on these relationships to select the unit tests that are most relevant to diagnose failures. The ID3 algorithm is used to construct a decision tree.

Results

- The approach diagnoses failures by running only 4-5% of the original unit tests.

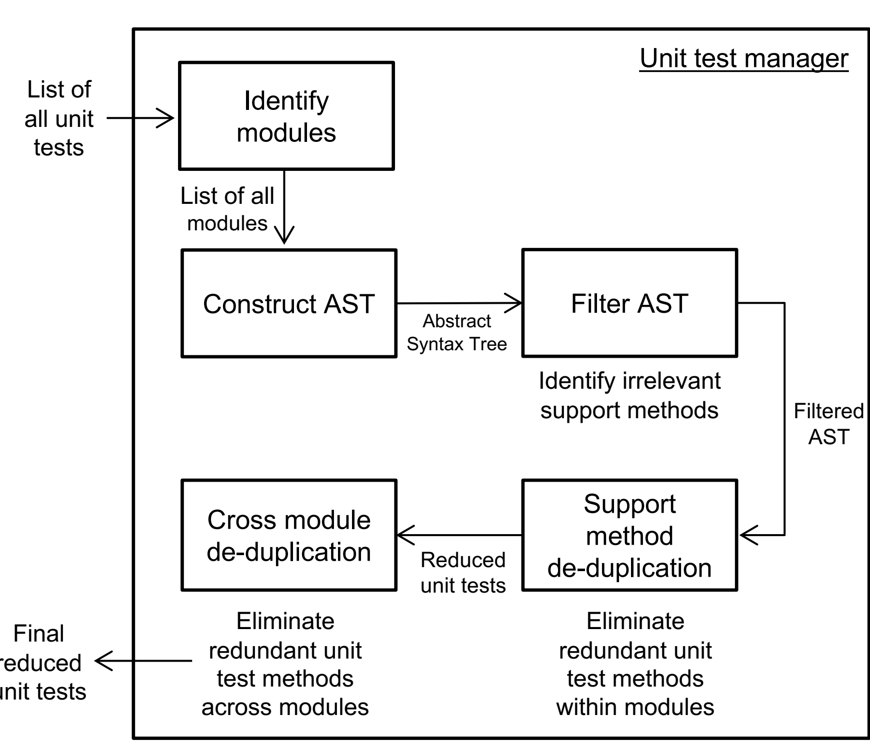
General Approach

- **Unit Tests Reduction** (Phase 1)**.** Reduce the number of unit tests written during the code development
- **Service Mapping** (Phase 2)**.** Establish relationships between the reduced unit tests and services. It further reduces the number of unit tests based on these relationships
- **Sequential Diagnosis** (Phase 3)**.** Use relationships to construct a decision tree to select the most relevant unit tests to diagnose failures



Phase 1 - Unit Test Reduction

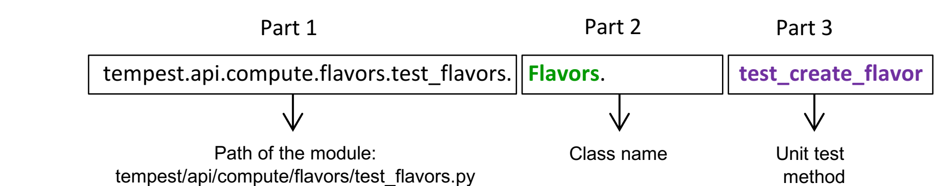
- Identify modules (1/5)
  - Identify the modules available in the unit test framework
- Construct Abstract Syntax Tree (2/5)
  - For each module, an AST is constructed to establish a relationship between unit test methods and support methods
- Filter AST (3/5)
  - All irrelevant support methods are filtered out from the AST using a Inverse Document Frequency
- Support method deduplication (4/5)
  - Unit test methods that perform redundant operations are eliminated by finding the minimum set cover. Time to execute an unit test is the cost of a set
- Cross module deduplication (5/5)
  - Unit test methods from one module are compared with methods of other modules and are further reduced



Phase 1 (1-2/5)

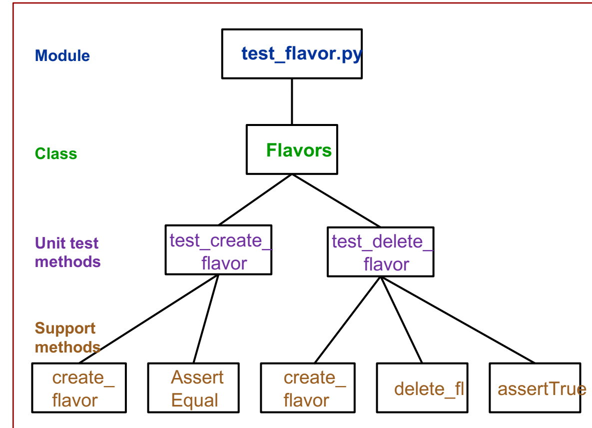
Identify Module & Construct AST

- Identify modules (1/5)
  - A module is the source file which contains the definition of the unit test method
- Construct AST (2/5)
  - For all the modules construct an AST
- Traversed the AST to identify unit test methods and their support methods
- For each unit test method, a set of its support methods is created:
  - **test_create_flavor**: {**create_flavor**, **assertEqual**}
  - **test_delete_flavor**: {**create_flavor**, **delete_fl**, **assertTrue**}




```
class Flavors (base.Test):

    def test_create_flavor(self):

        new_flavor_id = self.create_flavor()

        self.assertEqual(new_flavor_id, flavor_id)

    def test_delete_flavor(self):

        flavor = self.create_flavor(flavor_name)

        del_flavor = self.delete_fl(flavor)

        self.assertTrue(some_statement)
```

A Python module (test_flavor.py) containing the implementations of unit tests



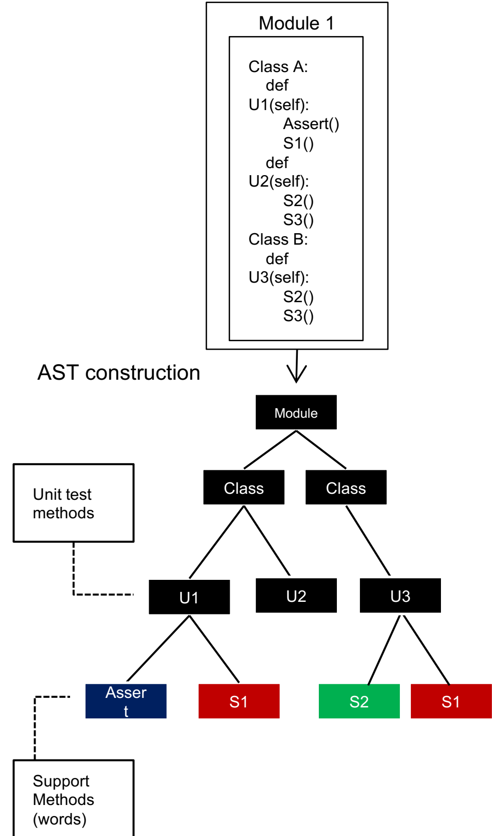
A simple AST

Note: Apart from the nodes shown in the figure, an AST contains other nodes like variables, constructors, etc. Since they are not relevant to us, they are excluded for simplicity.

Phase 1 (3/5)

Filter AST

- Support methods are eliminated
  - Some of the support methods are specific to the unit test framework
  - E.g., **assertEquals**
- Use Inverted Document Frequency (IDF) technique
  - Higher occurrence -> lower IDF
- Calculated IDF for support methods
  - Set of documents = ASTs of modules
- Eliminate support methods having a IDF below a threshold *alpha*
  - These support methods identified do not play any role in the reduction of the unit tests
  - They are irrelevant



Phase 1 (3/5)

Document 1 Document 2 Document 3 Document 4

Module
Class Class
Unit test methods
U1 U2 U3 U4 U5 U6 U7 U8 U9 U10 U11 U12
Assert S1 S2 S1 | Assert S3 S2 S3 | Assert S5 S5 S5 | Assert S6 S7 S18
Support Methods (words)

IDF(x) = log (# documents/ # documents with word x)

- The support methods **Assert** is present in all the modules -> its IDF is log(4/4) = 0
- Similarly, IDF for **S1** is log(4), **S2** is log(2)
- Thus, Assert is irrelevant. It is eliminated from the list of support methods.

Phase 1 (4/5)

Support method deduplication

- Each module has unit *test methods* which call *support methods*
- In most cases, a **subset** of the unit test methods call **all** the support methods
- Thus, some unit test methods are redundant
- Prune the ASTs constructed
  - Pruning is done using the minimum set cover
- Two important parameters: cost and coverage
  - Cost is the time a unit test takes to execute
  - Coverage is the percentage of support methods to consider in the set cover

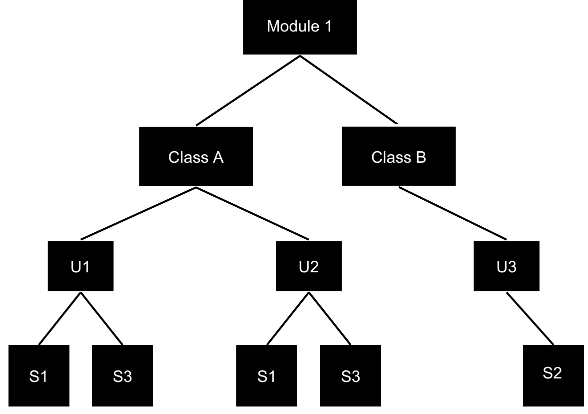
Algorithm 1 Set Cover with Coverage

1: $C \leftarrow 0$
2: **while** $|C| < \lceil((|U| * c)/100)\rceil$ **do**
3: Find the Set whose cost effectiveness is smallest, say S
4: Let $\alpha = \dfrac{c(S)}{|S - C|}$
5: For each e ε S-C, set price(e) $= \alpha$
6: $C \leftarrow C \cup S$
7: Output the picked sets

Phase 1 (4/5)

Support method (S) and time in seconds for each Unit test method (U)

U_1 : {S_1 , S_3}, 15
U_2 : {S_1 , S_3,} ,8
U_3 : {S_2}, 9



In case 1, minimum coverage is100 %, so all the support methods must be present. U2 and U3 covers all the distinct support methods and has the least cost. Hence, U1 is eliminated.

Minimum coverage requested = 100 %
U = {S_1 , S_2 , S_3}
Required set = all elements of U

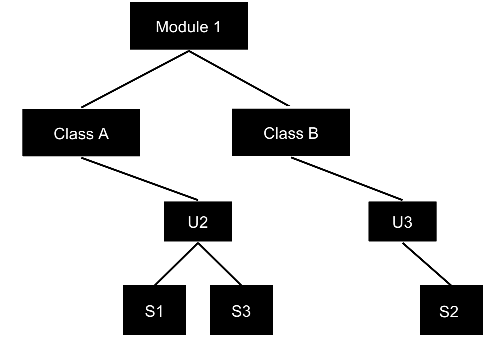
Loop 1:
$\alpha_1 = 15/2$
$\alpha_2 = 8/2$
$\alpha_3 = 9/1$

I = {S1, S3}

Loop 2:
$\alpha_1 = 15/0$
$\alpha_2 = 8/0$
$\alpha_3 = 9/1$

I = {S1, S2, S3}

Coverage obtained: 100 %



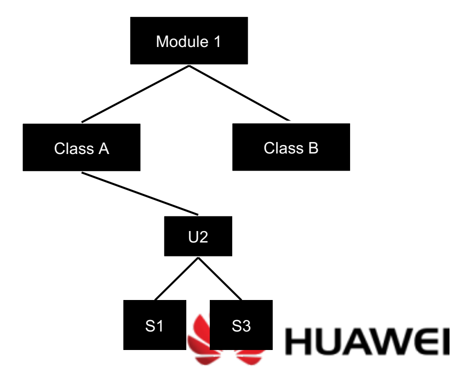
In case 2, the minimum requested is 50% so any 2 support methods out of 3 are sufficient. U2 contains 2 distinct support methods and has the least cost. Hence, U1 and U3 are eliminated.

Minimum coverage requested = 50 %
U = {S_1 , S_2 , S_3}
Required set = $\lceil (50/100) * 3 \rceil = 2$

Loop 1:
$\alpha_1 = 15/2$
$\alpha_2 = 8/2$
$\alpha_3 = 9/1$

I = {S1, S3}

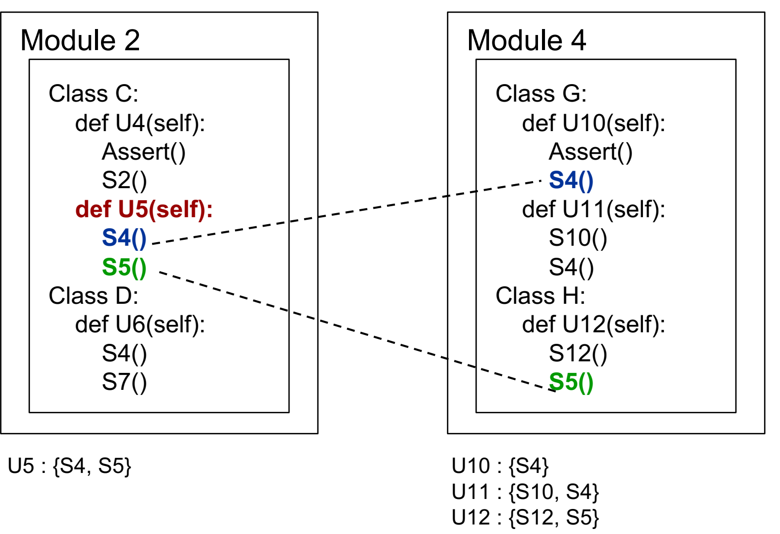
Coverage obtained: 66.6 %



Phase 1 (5/5)

Cross module deduplication

- The AST pruning reduces the unit test methods by finding the minimum set cover of the support methods in the same module. However, some unit test methods are covered in other modules
- Thus, the reduction can be improved without losing the coverage
- Cross module deduplication compares the support methods of a unit test method from one module with the universe of the support methods of other modules



- The support method de-duplication subsystem guarantees that **{S4, S5, S10, S12}** are called by the subset of the unit test methods in Module 4
- All the support methods called by U5 (i.e., **{S4, S5}**) are already covered in Module 4.
- Hence, U5 is redundant and can be eliminated

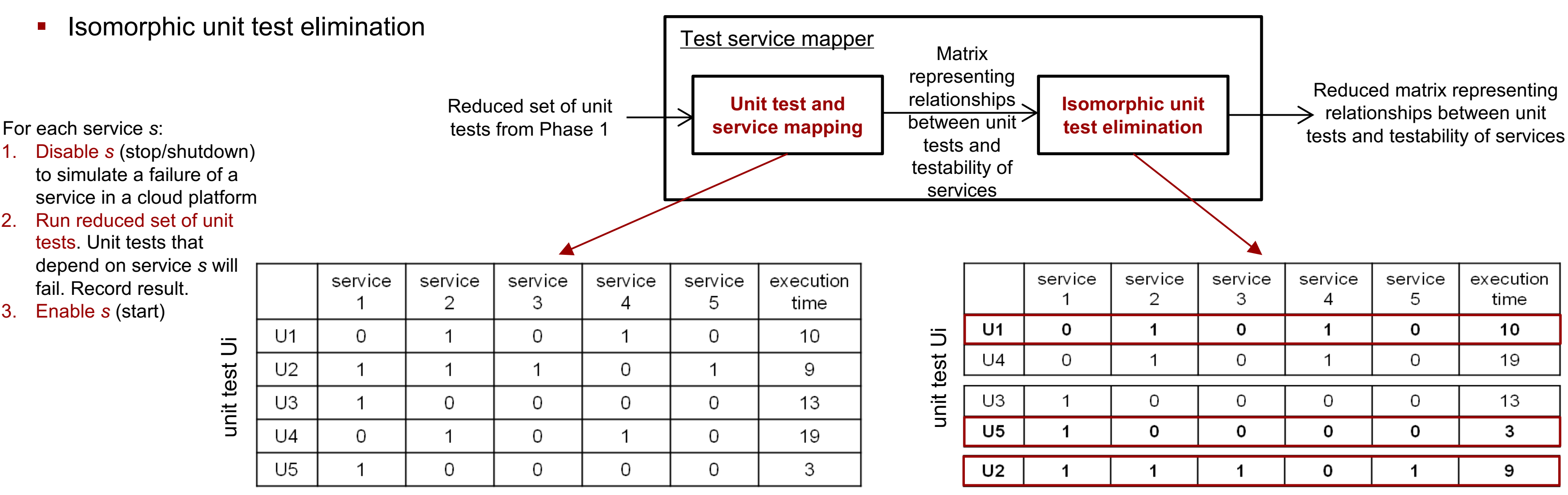
Our experiment with OpenStack enabled us to reduce 1391 unit tests created by developers to 538 tests with 100% coverage

Phase 2

Service Mapping

One of the major challenges of using unit tests for failure diagnosis is the lack of relationships between unit tests and services running on cloud platforms. The execution of unit tests outputs a list of passed, failed and skipped tests.

- Establish a relation between the unit tests and the services they can test
- Isomorphic unit test elimination



For each service *s*:

1. Disable *s* (stop/shutdown) to simulate a failure of a service in a cloud platform
2. Run reduced set of unit tests. Unit tests that depend on service *s* will fail. Record result.
3. Enable *s* (start)

unit test Ui	service 1	service 2	service 3	service 4	service 5	execution time
U1	0	1	0	1	0	10
U2	1	1	1	0	1	9
U3	1	0	0	0	0	13
U4	0	1	0	1	0	19
U5	1	0	0	0	0	3

unit test Ui	service 1	service 2	service 3	service 4	service 5	execution time
U1	**0**	**1**	**0**	**1**	**0**	**10**
U4	0	1	0	1	0	19
U3	1	0	0	0	0	13
U5	**1**	**0**	**0**	**0**	**0**	**3**
U2	**1**	**1**	**1**	**0**	**1**	**9**

The execution time for unit tests U1 is 10 seconds, U2 is 9 seconds, U3 is 13 seconds, U4 is 19 seconds and U5 is 3 seconds. Hence, **U1, U5 and U2 are selected**

Experiment: 20 OpenStack services running and 538 unit tests after reduction from Phase 1. At the end of Phase 2, the isomorphic test elimination procedure reduced the number of test to 25.

HUAWEI

Phase 3

Sequential Diagnosis

The main task of this phase is to analyze these relationships by constructing a decision tree and use the tree to detect failed service(s) in a cloud platform in operation. Moreover, the failed services are detected without executing all the unit tests.

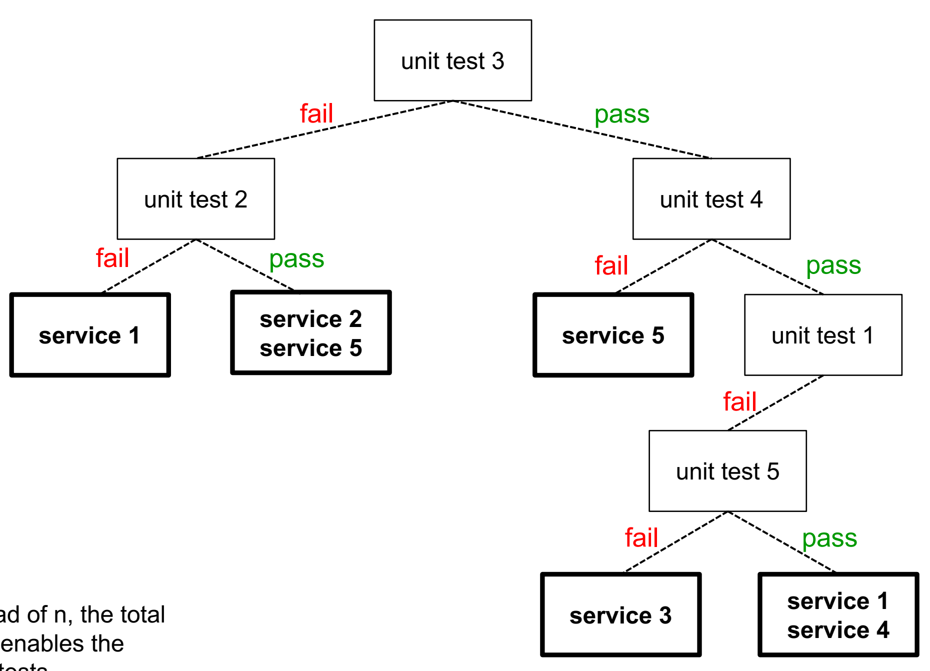
- Construction of the *decision tree*
- Execution of unit tests based on the *decision tree*

The tree of the Figure (right)

- Execute unit test 3
- If it fails, unit test 2 is executed
  - The failure of unit test 2 indicates that service 1 is in a failed state

In some cases, there might be more than one possible service in a failed state

- If unit test 2 passes, then either service 2 or service 5 or both are in a failed state



A failed service(s) can be detected by running log(n) number of unit tests instead of n, the total number of unit tests after eliminating the isomorphic unit tests. Hence, the tree enables the users to detect the failed service(s) automatically without executing all the unit tests.

HUAWEI

Results

We evaluated the approach with OpenStack = 1391 unit tests

Phase 1. Reduce the number of unit tests to 538 with 100% coverage

Phase 2. Established relationships between the reduced set of unit tests and the OpenStack services. There were 20 services. Eliminated isomorphic unit tests. At the end of Phase 2, we were able to reduce the number of unit tests to 25.

Phase 3. Build decision tree using the relationships established. 5 unit tests on average.

Open Positions and CFP

- **FTE, PostDoc, PhD Students**
  - Fault injection, fault models, fault libraries, fault plans, brake and rebuild systems all day long, …
  - Cloud Operations and Analytics for High Availability
  - AI, machine learning, data mining, and time-series analysis for Cloud operations

IEEE.ORG COMPUTER SOCIETY

IEEE Software

ABOUT HISTORY BACK ISSUES WRITE FOR US SUBSCRIBE SE-RADIO BLOG JOIN

Cloudware Engineering—Call for Papers

JULY 19, 2017

CALLS FOR PAPERS

Submission deadline: 1 Apr. 2018

Publication: Nov./Dec. 2018

Today, more than a decade since the launch of Amazon Web Services (AWS), on-demand computing platforms are pervasive in industry. Recent years have seen the emergence of many concepts and technologies to implement and support cloud-native applications: from full cloud application platforms, to serverless architectures, to docker containers, to microservices, and to unikernels, including chaos engineering and continuous integration. These advancements constitute a significant body of knowledge and make up a new professional engineering domain that's at the intersection of software development, IT operations, and the cloud.

Cloudware engineering employs systematic, disciplined, and quantifiable approaches to the development, operation, and maintenance of a specific type of software: cloud-native applications. Delivering such applications requires us to revisit traditional software engineering theories, methods, architectures, technologies, and tools to guarantee the delivery of applications that are scalable, elastic, portable, resilient, and adaptable for the cloud.

Cloudware engineering's importance and timeliness are highlighted by recent initiatives such as the Cloud Native Computing Foundation to advance cloud-native technology and services, cloud-native libraries such as Netflix Open Source Software, Microsoft's cloud design patterns, and Heroku's twelve-factor app.

Building, packaging, and launching cloud-native applications is far more complex than just lifting and shifting monolithic applications into virtual machines running in AWS, Azure, Google Cloud, SoftLayer, or OpenStack. Cloud applications

HUAWEI ENTERPRISE ICT SOLUTIONS A BETTER WAY

Copyright©2015 Huawei Technologies Co., Ltd. All Rights Reserved.
The information in this document may contain predictive statements including, without limitation, statements regarding the future financial and operating results, future product portfolio, new technology, etc. There are a number of factors that could cause actual results and developments to differ materially from those expressed or implied in the predictive statements. Therefore, such information is provided for reference purpose only and constitutes neither an offer nor an acceptance. Huawei may change the information at any time without notice.