# Efficient Failure Diagnosis of OpenStack Using Tempest

**Ankur Bhatia**
**Michael Gerndt**
Technical University of Munich

**Jorge Cardoso**
Huawei Munich Research Center
University of Coimbra

While cloud computing continues to be popular in the IT world, companies offering cloud solutions are under pressure to provide the most reliable solutions to users. In this article, we describe an innovative approach to diagnose service failures in an OpenStack-based cloud using Tempest as a starting point.

## SERVICE FAILURE DIAGNOSIS OF OPENSTACK: EARLIER WORK

OpenStack[1] is an open-source cloud-operating system for building public and private clouds. It can manage large pools of compute, storage, and networking resources in data centers. It is a continuously evolving system with a major release cycle every six months.

To reach high availability levels, the failures of OpenStack services need to be continuously monitored. Avižienis *et al.*[2] characterize a service failure as "an event that occurs when the delivered service deviates from correct service. A service failure is a transition from correct service to incorrect service, i.e., to not implementing the system function."

The OpenStack community curates a Wiki page[3] with over 50 different tools that can monitor and diagnose failures.[4] However, most of them are generic solutions. The ones specific to OpenStack typically show usage metrics or monitor if a certain process is running or not. Other tools require users to have expert knowledge of systems and to interpret large amounts of log information to diagnose failures. DevOps can also use Rally, an OpenStack tool capable of managing complex workflows which orchestrate benchmarking and evaluation experiments.

Over the years, several approaches have been proposed to detect anomalies in software systems. The three most relevant techniques that have yield good results are log, performance, and trace analysis.


Published by the IEEE Computer Society
1089-7801/18/$33.00 ©2018 IEEE

- Log analysis[5] uses clustering or heuristics, as well as templates to mine logs into events or flows. Features/graphs are extracted to model the normal and abnormal behavior of systems.
- Performance analysis[6] identifies resource consumption models (e.g., from CPU, memory, disk I/O) to establish patterns of normality and abnormality.
- Trace analysis[7] instruments code to enable the generation of traces (i.e., sequences of correlated events) at run time to identify the normal and abnormal behavior of systems.

The main advantage of our approach is that it copes well with the periodic new releases of OpenStack code base. There is no need to reparameterize algorithms (e.g., the *k* parameter of *k*-means clustering), update heuristics, or rules as frequently needed by log analysis. For cloud platforms such as Openstack, the variability of cloud workloads and multitenancy makes performance analysis and resource modeling a difficult task. Finally, trace analysis also breaks when new instrumentation points are added, removed, or changed in the code base.

## CAN TEMPEST TESTS BE USED TO DIAGNOSE FAILURES IN OPENSTACK?

The OpenStack community also developed a test suite named Tempest. It is used for the validation of all the modules of OpenStack during the development cycle to guarantee that the code is error free. It is the official integration test suite containing more than 1500 tests for API and scenarios validation. Due to the value of this large set of tests, one interesting question is whether or not it would be possible to also use it to diagnose service failures in OpenStack.

Although Tempest tests are extremely useful for the purpose of development and integration, their use for service failure diagnosis presents a set of challenges. First, they do not provide any information about the nonresponsive or failed services in cloud platforms. The execution of Tempest 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 Tempest tests and services running on OpenStack. Second, Tempest contains more than 1500 tests. With every new release of OpenStack, the number of tests also increases. Thus, it takes a considerable amount of time (3–4 h) to execute them. Thus, cloud operators often develop custom tests to diagnose failures in OpenStack. However, as mentioned previously, OpenStack is a continuously evolving cloud platform with a release cycle of six months. Hence, the tests developed become outdated and there is a constant need to modify them. Therefore, the approach is costly for cloud operators.

A different approach is, therefore, needed to diagnose service failures in OpenStack. The solution should be efficient (fast) and should be able to establish relationships between Tempest tests and the services they are capable of testing. The solution should also be able to cope up with the fast release cycle of OpenStack.

## FAILURE DIAGNOSIS OF OPENSTACK USING TEMPEST TESTS

Diagnosing failures in OpenStack is not a simple task. Although there has been research in this field,[8–10] there is no solution that uses Tempest tests for the purpose of failure diagnosis. We have already mentioned the challenges associated with using Tempest for failure diagnosis. However, there are a few benefits of using Tempest as well. First, Tempest tests are developed along with OpenStack. Hence, there is no additional cost of development involved. Second, they can be executed with minimal effort as they are automated. Finally, with every new release, Tempest tests are updated. Therefore, there is no need to modify the tests for a new release.

Having these benefits in mind, we developed a new method to diagnose service failures in OpenStack using Tempest. The method has three main phases detailed in the following sections. The overall achievement is that it can diagnose failures by running only 4–5% of the reduced Tempest tests.

## Phase 1: Tempest Test Suite Reduction

Phase 1 of the solution deals with Tempest test reduction. This task is handled by the Tempest test manager. Figure 1 shows the architecture of this component. It consists of the following five subsystems.

### Identify Modules

The first task of this phase is to identify all the modules where the Tempest tests are implemented. These modules are identified using a naming convention followed by OpenStack, for example, tempest.api.compute.flavors.test_flavors.FlavorsV2TestJSON.test_get_flavor yields the following information:

- path to the module: tempest.api.compute.flavors.test_flavors;
- class name: FlavorsV2TestJSON;
- test method: test_get_flavor.

### Construct Abstract Syntax Tree (AST)

Every Tempest test has one test method defined in a module. The test methods make calls to other methods that are known as support methods. These methods call various functionalities provided by OpenStack. The main task of this subsystem is to perform code analysis to identify the test methods and their support methods. This is done by constructing an AST[11] to represent the source code in the form of a tree. Each node of an AST represents a construct (module, class, test method, or a support method) occurring in the source code, as shown in Figure 2(b).

### Filter AST

In the previous step, we constructed ASTs to identify test methods and support methods. However, there are some support methods that are not relevant. They are language specific and do not play a direct role in testing any functionality of OpenStack. Therefore, these support methods have to be eliminated. This is achieved using the inverse document frequency (IDF).[12] This technique is based on the notion that words with low IDF are present across multiple documents and are usually not considered important. The same logic is used to determine if a support method is relevant or not. If a support method is present across most of the modules, it is considered to be irrelevant. The IDF of a support method is calculated by the following expression:

$$\mathrm{IDF}(x) = \log\,(\#\text{modules} / \#\text{modules with support method } x).$$

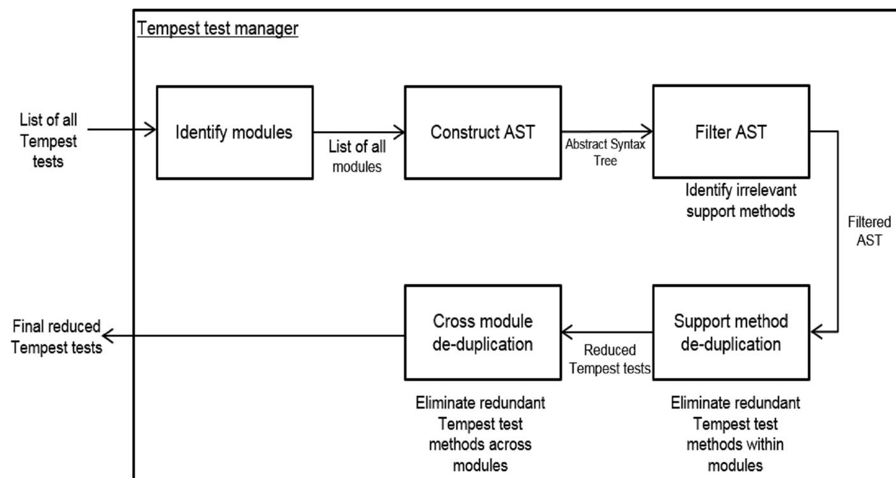



Figure 1. Tempest test manager. This is the architectural diagram of the Tempest test manager that consists of five modules explained in detail in the text.

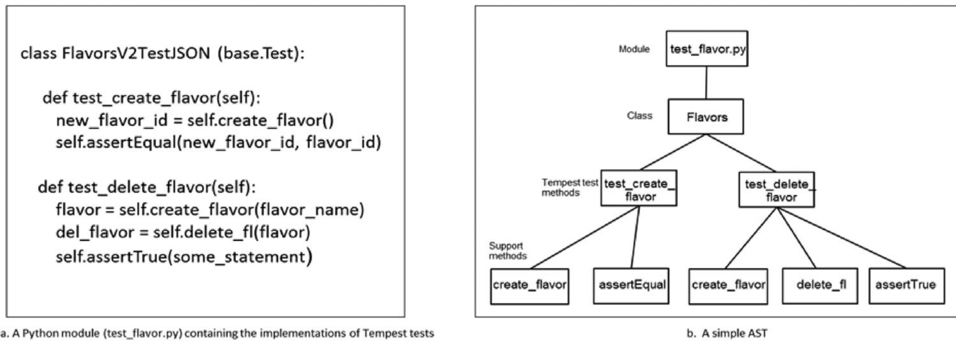



Figure 2. AST. The figure shows a simple AST constructed for a Python module.

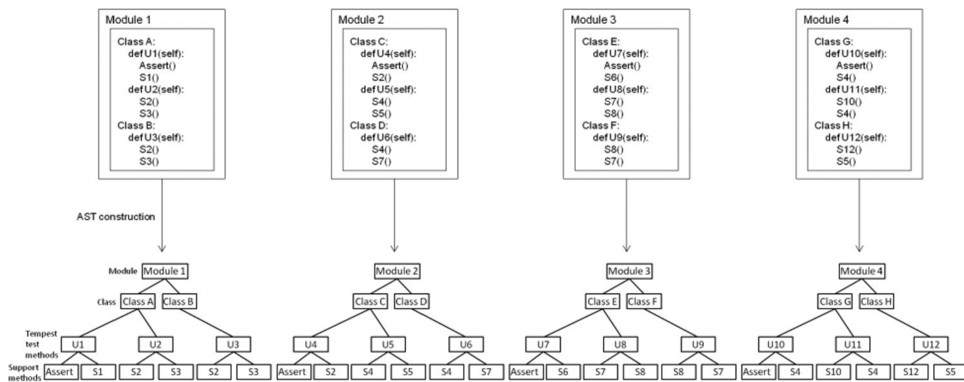



Figure 3. IDF calculation. The figure shows the construction of ASTs for Python modules to calculate the IDF of all the support methods in the module.

Let us consider the example shown in Figure 3.

Module 1 calls 4 support methods: Assert, S1, S2, and S3. The IDF of these methods is calculated as follows:

IDF (Assert) = log (4 / 4) = log (1) = 0

IDF (S1) = log (4 / 1) = log (4) = 2

IDF (S2) = log (4 / 2) = log (2) = 1

IDF (S3) = log (4 / 1) = log (4) = 2.

As mentioned, the support methods with low IDF are not relevant and hence are eliminated. From the above example, Assert is labeled as irrelevant, while S1, S2, and S3 are relevant.

## Support Method Deduplication

In the previous step, we eliminated the irrelevant support methods from modules. The next task is to eliminate the redundant test methods from modules. The analysis of the code enabled us to discover that in most cases a subset of the test methods calls all support methods in a module. Therefore, within most modules, there are redundant test methods that can be eliminated.

This task is accomplished using the set cover algorithm.[13] The algorithm selects the minimum number of test methods from the module such that the selected test methods call all the support methods present in the module. For example, in Figure 3, in Module 3, the subsets of test methods {U7, U8} and {U7, U9} call all the support methods {S6, S7, S8} present in the module. Hence, any one of these subsets can be selected to call all the support methods. However, the algorithm selects the subset with a lower execution time. The following two parameters are considered in the algorithm.

- Time is the time taken for each Tempest test to execute. It is the parameter cost of the set cover algorithm. This parameter enables to select the most time-efficient subset of the test methods. If there are cases where more than one subset of the test methods calls all the support methods, the subset with the lower execution time is selected.
- Coverage is the minimum coverage (in %) of the support methods to achieve. This parameter is not present in the original set cover algorithm. It is introduced to give users an option to select test methods based on the percentage of the support methods to be considered. For example, in Figure 3, in Module 3, there are three support methods {S6, S7, S8}. For a 100% coverage, the subsets {U7, U8} or {U7, U9} have to be selected in order to cover all the support methods. For a 30% coverage, only {U7} is sufficient as this requires only one out of the three support methods to be called.

### Cross Module Deduplication

The support method deduplication eliminates the redundant test methods within modules using the set cover algorithm. However, through code analysis, we discovered that some test methods are redundant also across modules. Hence, the reduction can be improved without losing the coverage.

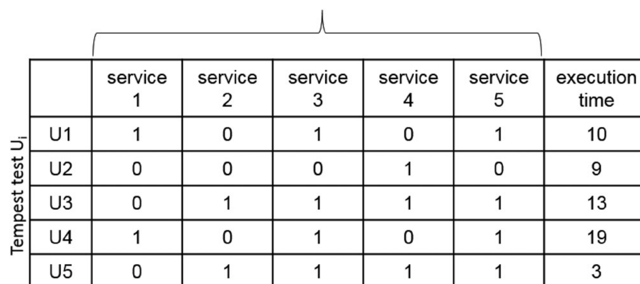
Let us consider Module 2 and Module 4 from Figure 4. The test method U5 calls the support methods {S4, S5}. Similarly, Module 4 consists of test methods U10, U11, and U12, which call the set of support methods {S4}, {S10, S4}, and {S12, S5}, respectively. The support method deduplication subsystem guarantees that {S4, S5, S10, S12} are called by the subset of the test methods in Module 4. This means that 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.

This approach is applied to all the test methods to eliminate the redundant test methods across the modules. It is important to note that the test deduplication strategy developed was based on how Tempest works. Before executing the tests, Tempest downloads and creates all the required resources (e.g., OS resources, users, networks, etc.). Thus, users do not need to test the service with specific parameters as Tempest handles them. Moreover, if two or more Tempest methods that have a common support method, they always use the same parameter (e.g., VM image). Hence, eliminating one of them has no effect on the coverage but reduces the number of test.

Our experiment with OpenStack enabled us to automatically reduce 1392 Tempest tests to 518 tests with 100% coverage.

## Phase 2: Relationship Establishment and Isomorphic Test Elimination

One of the major challenges of using Tempest tests for failure diagnosis is the lack of relationships between Tempest tests and OpenStack services. The execution of Tempest tests outputs a list of passed, failed, and skipped tests. However, it is difficult to determine failed services based on this list. The



| Tempest test $U_i$ | service 1 | service 2 | service 3 | service 4 | service 5 | execution time |
|---|---|---|---|---|---|---|
| U1 | 1 | 0 | 1 | 0 | 1 | 10 |
| U2 | 0 | 0 | 0 | 1 | 0 | 9 |
| U3 | 0 | 1 | 1 | 1 | 1 | 13 |
| U4 | 1 | 0 | 1 | 0 | 1 | 19 |
| U5 | 0 | 1 | 1 | 1 | 1 | 3 |

Figure 4. Relationship between Tempest tests and services.

second phase establishes relationships between Tempest tests and services. It further eliminates Tempest tests by removing tests that establish the same relationships. These tests are called isomorphic tests.

### Tempest Test and Service Mapping

The relationships between Tempest tests and OpenStack services are established based on the ability of the tests to determine if a particular service is working properly or not. The following steps are performed.

For s in services:

1. Disable service s to simulate a failure of a service in OpenStack.
2. Run the reduced set of Tempest tests. Tests that depend on s will fail. Tests that do not depend on it will pass.
3. Restart service s.

Services are disabled one at a time. This is because if more than one service is disabled and a test fails, it is not possible to determine which service caused the test to fail.

The procedure is repeated for all services critical to the functioning of OpenStack. Based on the results, relationships between Tempest tests and services are established and are represented in a matrix, as shown in Figure 4.

Each cell represents a relationship between a Tempest test and a service. Let us represent this relationship with the expression $\mathbf{R(U_i, service_j)}$ where $\mathbf{U_i}$ is a Tempest test and $\mathbf{service_j}$ is a service.

$\mathbf{R(U_i, service_j)} = 1$ means that $\mathbf{U_i}$ depends on $\mathbf{service_j}$.

$\mathbf{R(U_i, service_j)} = 0$ means that $\mathbf{U_i}$ does not depends on $\mathbf{service_j}$.

Consider the matrix from Figure 4. If U5 fails, then based on the relationships $service_2$, $service_3$, $service_4$, and $service_5$ could be down. However, if U5 passes, it means that $service_2$, $service_3$, $service_4$, and $service_5$ are available.

### Isomorphic Unit Test Elimination

As shown in Figure 4, each Tempest test can be represented with a binary string. For example, U1 is represented as 10101, U2 as 00010, etc. Tempest tests that have the same string are called isomorphic tests. They establish the same relationships with services and detect failures of same services. Therefore, they are redundant. In order to eliminate them, all isomorphic tests are grouped together and the one with the lower execution time is selected.

In our experiment, there were 18 services running and 518 Tempest tests after reduction from Phase 1. At the end of Phase 2, the isomorphic test elimination procedure reduced the number of tests to 19.

## Phase 3: Failure Detection

This phase analyzes the relationships established in Phase 2 by constructing a decision tree and using the tree to detect failed service(s). Each internal node represents a Tempest test selected in Phase 2. Each branch represents the result of the test and each leaf represents a service which is in failed state or in some cases, it represents a set of failed services.

In our experiment, we used the sequential fault diagnosis[14] to construct a decision tree. In this approach, a test can split the state of a system into two subsets. Let A denote the set of candidate system states when a test fails and B the set of candidate system states when the test passes. For a symmetrical test: $\mathbf{A \cap B = \emptyset}$.

The test-sequencing problem has been defined by Pattipati and Alexandridis[15] by tuple (S, p, T, c), where

> $S = \{s_0, s_1, \ldots, s_m\}$ finite set of system states with $s_0$ denoting the fault free state of the system and $s_i$ $(1 <= i <= m)$ specifying the different faulty states of the system;

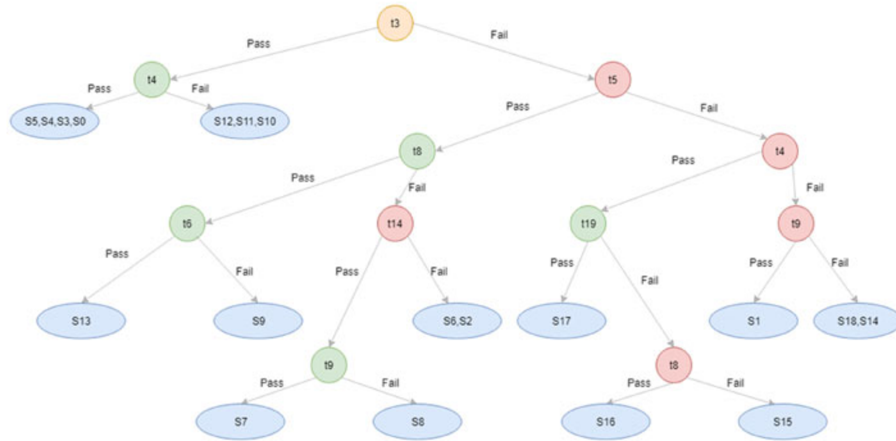



Figure 5. Decision tree. The figure shows the decision tree generated from the relationship derived in Phase 2.

p = $[p(s_0), p(s_1),..., p(s_m)]$ *a priori* probability vector of the system state;
T = $\{t_1,t_2,..., t_n\}$ finite set of $n$ available tests;
c = $[c_1, c_2,..., c_n]$ vector of test costs measured in terms of time, manpower or other economic factors;

test matrix D describing diagnostic capabilities of tests T. This is the relationship established in Phase 2 (see Figure 4).

The test-sequencing problem falls under the category of a Markov decision problem. The Markov state $x$ denotes the suspect set of system state, which is a subset of S. Each state has a corresponding test to further divide this state into two subsets of states. This is repeated till the state cannot be further divided. Figure 5 shows the decision tree generated in our experiment.

A test corresponding to a Markov state is executed. The result of the test splits the Markov state (also known as ambiguity set) into two subsets, thereby reducing the ambiguity. As per the convention, the left branch corresponds to the passed test and the right branch corresponds to the failed test. The root node represents the state of complete ambiguity and the leaf node represents one of the failed states.

In Figure 5, the root node containing test $t_3$ corresponds to the Markov state $x$ containing all the states of S. The execution of $t_3$ divides the system into two Markov states $x_{jp}$ and $x_{jf}$, thereby reducing the ambiguity based on the result of $t_3$. The Markov state $x_{jp}$ denotes the states {S0, S3, S5, S10, S11, S12} and $x_{jf}$ denotes {S1, S2, S4, S6, S7, S8, S9, S13, S14, S15, S16, S17, S18}. These Markov states are further divided till no more divisions are possible.

At each node, the test is selected based on the distinguishability heuristic defined by the following expression [Eq. 1], which is given by Pattipati and Alexandridis[15]:

$$d_c(x, t_j) = p(x_{jp}).p(x_{jf}) \qquad (1)$$

where $x_{jp}$ and $x_{jf}$ are the Markov states of the pass and fail outcomes of test $t_j$ such that $x_{jp} \cup x_{jf} = x$ and $p(x_{jp})$, $p(x_{jf})$ are the conditional probabilities of the pass and fail outcomes of test $t_j$, respectively. Furthermore, $p(x_{jp})$ [Eq. 2] and $p(x_{jf})$ [Eq. 3] are expressed as follows, which are given by Cui *et al.*[11]:

$$p(x_{jp}) = \left[\sum_{s_i \in x} (1 - d_{ij}).p(s_i)\right].[\hat{p}(x)]^{-1} \qquad (2)$$

$$p(x_{if}) = 1 - p(x_{jp}), \hat{p}(x) = \sum_{s_i \in x} p(s_i). \qquad (3)$$

The test $t_j$ that maximizes $d_c(x,t_j)$/cost, where cost is the execution time of the test $t_j$ is selected.

Figure 5 shows the constructed decision tree. Let us diagnose a system with a defective service $s_{13}$. The diagnosis starts with the root node containing the test $t_3$. If the test $t_3$ fails, the right branch of the tree is followed and the test $t_5$ is executed. If the test $t_5$ passes, the left branch is traversed and the test $t_8$ is executed. Similarly, if $t_8$ passes, the test $t_6$ is executed. The execution of the test $t_6$ would determine the faulty service. If the test $t_6$ passes, it would lead us to the leaf node containing the faulty service $s_{13}$.

We had 19 Tempest tests after Phase 2 and 18 services running in our OpenStack cloud. In the worst case, the solution was able to detect the faulty service after executing five Tempest tests.

# EVALUATION

The evaluation of the method was carried out in OpenStack Mitaka with a multinode setup. In more recent experiments, we rely on OpenStack Rally to schedule, orchestrate, and replicate failure diagnosis experiments for evaluation in a controlled and deterministic manner. The OpenStack configuration was the following.

- 1 Controller node (8 GB, Intel Xeon E5-2600, 4 cores).
- 1 Network node (4 GB, Intel Xeon E5-2600, 4 cores).
- 2 Compute nodes (8 GB, Intel Xeon E5-2600, 4 cores).

To evaluate Phase 1, eight modules from different components of the Tempest framework were randomly chosen and the reduction of the tests was performed manually. The results were then compared with the ones produced by the solution. The reduction performed automatically matched the reduction performed manually.

To evaluate Phase 2, we ran the algorithm to establish relationships between Tempest tests and services. We established the relationship between 518 tests and 18 services running. Isomorphic test elimination reduced the number of tests to 19. The relationships were evaluated and were correct. An important aspect to consider is that the establishment of relationships must be carried out in a valid OpenStack setup. Tests can fail for various reasons, such as timeouts, network congestion, insufficient memory, storage issues, etc. These problems can result in the establishment of incorrect relationships. Hence, to eliminate false positives, tests were run multiple times to make sure that the relationships were accurate. Therefore, the tests were run four times to establish accurate results.

To evaluate Phase 3, the decision tree was validated by conducting failure diagnosis in different scenarios. Services were manually disabled to simulate failures in the production environment. The method was able to predict all service failures.

In case a cloud platform is heavily loaded, displays insufficient capacity, and drops service requests that cannot be handled, our approach will signal these requests as service failures. This design decision reflects the fact that customers perceive a service to be unavailable as a failure.

There is another interesting practical evaluation worth discussing that highlights the usefulness of the method proposed. One of our OpenStack cloud was deployed in a network shared with the other deployments resulting in IP conflicts. The virtual machines created by OpenStack were configured with an IP used by another deployment. The tools that are made available by the OpenStack community were not helpful to diagnose the root cause of the problem as they showed that the Neutron service was running without problems and there were enough IPs available. Moreover, Tempest tests also did not prove to be helpful as many tests belonging to different components failed because of the underlying problem. However, our method correctly diagnosed that the component responsible for the problem was Neutron.

The cost of each of the phases is also very reasonable. As an example, for the models that we have reconstructed for OpenStack releases, the total time taken to run all the 1392 Tempest tests was approx. 175 min. After the first phase, the number of tests was reduced to 518, which took approx. 75 min to execute. The application of the model needs only to execute five to six tests to detect a service failure, which typically takes less than 2 min to complete.

While our approach is intrusive, our experiments show that at most six tests out of the 19 tests left after eliminating the isomorphic tests are needed to detect a service failure. The average latency of the 19 tests is 16.55 s (SD 7.59). Furthermore, if we pay attention to all the paths from the root to the leaves of the decision tree to diagnose failures, the most expensive path takes approx. 80 s to run all the tests. Thus, the cost is acceptable as well as the impact on the cloud platform.

## CONCLUSION

This research demonstrated the use of a new method for using automated tests for service failure diagnosis. The method assists cloud administrators to diagnose failures in OpenStack within 4–5 min. Nevertheless, there is room for future research. The relationships between the tests and services were established by completely disabling services, but it is worth looking into scenarios where services are not completely disable but rather to simulate situations under which services are running with limited bandwidth, memory, or storage capacity. Moreover, to further improve failure diagnosis, log and control flow graph analysis techniques can be incorporated to reconstruct the underlying distributed system topology to narrow the state-space search for services that have failed.

## REFERENCES

1. (2016). [Online]. Available: https://www.openstack.org
2. A. Avižienis, J. C. Laprie, and B. Randell, “Dependability and its threats: A taxonomy,” in *Proc. Building Inf. Soc.*, Springer, vol. 22, 2004, pp. 91–120.
3. (2016). [Online]. Available: https://wiki.openstack.org/wiki/Operations/Tools
4. K. Srinivas and R. B. Michael “Methods for fault detection, diagnostics, and prognostics for building systems—A review, part I,” *HVAC&R Res.*, vol. 11, 2005.
5. T. Jia, P. Chen, L. Yang, Y. Li, F. Meng, and J. Xu, “An approach for anomaly diagnosis based on hybrid graph model with logs for distributed services,” in *Proc. IEEE Int. Conf. Web Services*, Honolulu, HI, USA, 2017, pp. 25–32.
6. M. Peiris, J. H. Hill, J. Thelin, S. Bykov, G. Kliot, and C. Konig, “PAD: Performance anomaly detection in multi-server distributed systems,” in *Proc. 7th Int. Conf. Cloud Computing*, 2014, pp. 769–776.
7. B. Sigelman *et al.*, “Dapper, a large-scale distributed systems tracing infrastructure,” Google, Mountain View, CA, USA, 2010.
8. P. Musavi, B. Adams, and F. Khomh, “Experience report: An empirical study of API failures in OpenStack cloud environments,” in *Proc. IEEE 27th Int. Symp. Softw. Rel. Eng.*, 2016, pp. 424–434.
9. T. Yoshinobu and Y. Shigeru, “Practical reliability and maintainability analysis tool for an open source cloud computing,” *Qual. Rel. Eng. Int.*, vol. 32, pp. 909–920, 2016.
10. P. Cuong *et al.*, “Failure diagnosis for distributed systems using targeted fault injection,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 60, pp. 503–516, Feb. 2017.
11. C. Baojiang *et al.*, “Code comparison system based on abstract syntax tree,” in *Proc. 3rd IEEE Int. Conf. Broadband Netw. Multimedia Technol.*, 2010, pp. 668–673.
12. R. Juan, “Using TF-IDF to determine word relevance in document queries,” in *Proc. 1st Instructional Conf. Mach. Learn.*, section 2.1, p. 2, 2003.
13. P. Slavík, “A tight analysis of the greedy algorithm for set cover,” in *Proc. 28th Annu. ACM Symp. Theory Comput.*, 1996, pp. 435–441.
14. Ž. Alenka, A. Biasizzo, and F. Novak, “Sequential diagnosis tool,” *Microprocessors Microsyst.*, vol. 24, pp. 191–197, 2000.
15. K. R. Pattipati and M. G. Alexandridis, “Application of heuristic search and information theory to sequential fault diagnosis,” *IEEE Trans. Syst., Man, Cybern.*, vol. 20, no. 4, pp. 872–887, Jul./Aug. 1990.

# ABOUT THE AUTHORS

**Ankur Bhatia** is currently working toward the M.Sc. degree in informatics at the Technical University of Munich, Munich, Germany. His research interests include cloud computing and Internet of Things. He received the B.Tech degree in computer science from the Manipal Institute of Technology, Manipal, India. Contact him at ankur.bhatia@tum.de.

**Jorge Cardoso** is a chief architect for Cloud Operations and Analytics at Huawei's German Research Centre, Munich, Germany, and an associate professor at the University of Coimbra, Coimbra, Portugal. His research interests include intelligent cloud operations, cloud computing, distributed systems, and business process management. He received the Ph.D. degree in computer science from the University of Georgia, Athens, GA, USA. Contact him at jorge.cardoso@huawei.com.

**Michael Gerndt** has been a professor for parallel computer architecture at the Technical University of Munich, Munich, Germany, since 2000. He is focusing of performance analysis and tuning tools for HPC systems. Further research focuses on parallel programming languages and Cloud systems. Contact him at gerndt@in.tum.de.