Speeding up Mutation Testing via the Cloud: Lessons Learned for Further Optimisations

Sten Vercammen
Universiteit Antwerpen
Antwerpen, België

Serge Demeyer
Universiteit Antwerpen, België
Flanders Make, België

Markus Borg
RISE SICS AB
Lund, Sweden

Sigrid Eldh
Ericsson AB
Stockholm, Sweden

ABSTRACT

Background: Mutation testing is the state-of-the-art technique for assessing the fault detection capacity of a test suite. Unfortunately, it is seldom applied in practice because it is computationally expensive. We witnessed 48 hours of mutation testing time on a test suite comprising 272 unit tests and 5,258 lines of test code for testing a project with 48,873 lines of production code. **Aims:** Therefore, researchers are currently investigating cloud solutions, hoping to achieve sufficient speed-up to allow for a complete mutation test run during the nightly build. **Method:** In this paper we evaluate mutation testing in the cloud against two industrial projects. **Results:** With our proof-of-concept, we achieved a speed-up between 12x and 12.7x on a cloud infrastructure with 16 nodes. This allowed to reduce the aforementioned 48 hours of mutation testing time to 3.7 hours. **Conclusions:** We make a detailed analysis of the delays induced by the distributed architecture, point out avenues for further optimisation and elaborate on the lessons learned for the mutation testing community. Most importantly, we learned that for optimal deployment in a cloud infrastructure, tasks should remain completely independent. Mutant optimisation techniques that violate this principle will benefit less from deploying in the cloud.

CCS CONCEPTS

• **Hardware → Testing with distributed and parallel systems**; • **Software and its engineering → Software testing and debugging**; *Software performance*; *Empirical software validation*;

KEYWORDS

Mutation testing; Cloud infrastructure; Experience report; Industrial validation

ACM Reference Format:
Sten Vercammen, Serge Demeyer, Markus Borg, and Sigrid Eldh. 2018. Speeding up Mutation Testing via the Cloud:, Lessons Learned for Further Optimisations. In *ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) (ESEM '18), October 11–12, 2018, Oulu, Finland.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3239235.3240506

ESEM '18, October 11–12, 2018, Oulu, Finland

ACM ISBN 978-1-4503-5823-1/18/10…$15.00
https://doi.org/10.1145/3239235.3240506

1 INTRODUCTION

Software testing is the dominant method for quality assurance and quality control in software development organisations [8, 21]. Software testing was established as a disciplined approach in the late 70's when it was defined as *"executing a program with the intent of finding an error"* [20]. In the last decade, this intent shifted dramatically with the advent of continuous integration [1]. Many software tests are now fully automated, and serve as quality gates, safeguarding against programming faults. The scale at which automated software tests are adopted in modern software organisations is mind-boggling. Microsoft for instance reported that approximately 11 months of development on Windows comprised more than 30 million test executions. Google on the other hand reported that *"In an average day, TAP integrates and tests […] more than 13K code projects, requiring 800K builds and 150 Million test runs."* [19]. As a result of this continuous integration approach, software organisations are capable of releasing faster. Tesla, for example uploads new software in its cars once every month [18]. Amazon pushes new updates to production every 11.6 seconds [13].

The growing reliance on automated software tests raises a fundamental question: How trustworthy are these automated tests? Today, mutation testing is the state-of-the-art technique for assessing the *fault-detection capacity* of a test suite [14]. The technique deliberately injects faults into the system under test and counts how many of them are caught by the test suite. Mutation testing is acknowledged within academic circles as the most promising technique for a fully automated assessment of the strength of a test suite [24]. One of the reasons mutation testing is seldom adopted in industrial settings is because the technique is computationally expensive: each mutant must be deployed and tested separately [14].

Mutation testing has shown to be able to run in parallel on a distributed architecture [3, 22, 30]. Researchers are currently investigating cloud solutions to share the computational load across a series of hardware nodes. Most notably among them are Hadoop-mutator [28] and Eminent [5]. These tool prototypes demonstrate that cloud infrastructure indeed allows to speed up the mutation testing. Yet, today it is unclear how to optimally distribute the load across the available hardware nodes.

In this paper we evaluate an alternative cloud solution (named DiMuTesTas) against two industrial projects, one small and one

large. We achieve a speed-up between 12x and 12.7x on a cloud infrastructure with 16 workers, illustrating that substantial speed-up is possible yet that the overhead is significant. We collect detailed measurements on the cloud infrastructure (setup, scheduling, file transfer), analyse how the overhead occurs, suggest avenues for further improvements and elaborate on the lessons learned for the mutation testing community.

The rest of the paper is structured as follows. In Section 2, we elaborate on the concept of mutation testing and list related work. In Section 3, we describe the cloud architecture of our proof-of-concept, identifying where to measure overhead. In Section 4, we explain our case study setup, which naturally leads to Section 5 where we discuss the results. In Section 6 we derive the lessons learned. As with any empirical research, we list the threats to validity in Section 7 to arrive at a conclusion in Section 8.

2 BACKGROUND AND RELATED WORK

In this section, we elaborate on the concept of mutation testing and contrast our proof-of-concept against related work.

2.1 Mutation Testing

For effective testing, software teams need strong tests which maximise the likelihood of exposing faults [20]. Traditionally, the strength of a test suite is assessed using code coverage, revealing which statements are poorly tested. However, code coverage has been shown to be a poor indicator of test effectiveness [4, 11]. Worse, even a 100% MC/DC coverage (Modified Condition/Decision Coverage, the coverage criterion adopted for safety critical systems) still does not guarantee the absence of faults [9, 15].

Today, mutation testing is the state-of-the-art technique for assessing the *fault-detection capacity* of a test suite [14, 24]. The technique deliberately injects faults (called mutants) into the production code and counts how many of them are caught by the test suite. Case studies with safety critical systems demonstrate that mutation testing could be effective where traditional structural coverage analysis and manual peer review have failed [2, 26]. Google on the other hand reports that mutation testing provides insight into poorly tested parts of the system, but –more importantly– also reveals design problems with components that are difficult to test, hence must be refactored [12].

Unfortunately, mutation testing is seldom adopted in practice [10]. One of the reasons is that the technique —in its basic form– requires a tremendous amount of computing power. For each injected mutant, the code base must be compiled and tested separately [14]. Algorithm 1 shows the essential steps without any optimisations, in order to understand the time-consuming nature of the mutation testing process. The software system needs to build without errors and all software tests should succeed before mutation testing can even begin; this is called the *pre*-phase. Then, the two main phases are executed: (*A*) the mutant generation phase and (*B*) the mutant execution phase. In phase *A*, mutants are generated for all source files. In phase *B*, each mutant is executed and its result (whether or not it was killed) is saved. Finally, all the results are gathered and the final report is created in the *post*-phase.

Algorithm 1 Pseudocode Mutation Testing

```
1:  function mutationTesting(srcFolder src)
2:      ▷ Pre: verify build and if all tests succeed
3:      if initialBuildAndTests() ≠ True then
4:          return
5:
6:      ▷ A: generate mutants
7:      mutants ← []
8:      for all srcFile f ∈ src do
9:          fMutants ← generateMutants(srcFile f)
10:         mutants ← mutants + fMutants
11:
12:     ▷ B: execute mutants
13:     for all mutant m ∈ mutants do
14:         result ← executeMutant(mutant m)
15:         storeResult(result, mutant m)
16:
17:     ▷ Post: process results
18:     processResults()
```

2.2 Mutation Testing Optimisations

A lot of research is devoted to optimising the mutation testing process, summarised under the vision - *do fewer*, *do smarter*, and *do faster* [23]. The *do fewer* approaches minimise the execution time by reducing the total amount of mutants to execute. Such an optimisation can be implemented by generating fewer mutants on line 9 in Algorithm 1 or by selecting a subset of all mutants on line 13. The fewer mutants that are executed, the more information will be lost. Balancing time reduction versus information loss is key. There are different ways to choose which mutants will be executed, varying in their effectiveness compared to the full set of mutants [14]. *Do smarter* approaches attempt to minimise the execution time by retaining state information between runs, e.g. split-stream mutation testing [16]. Others *prioritise test*, giving priority to the tests with the highest likelihood of failure. These optimisations would be implemented on line 14 in Algorithm 1. Lastly, *do faster* approaches try to minimise the execution time of each individual mutant. One example is using a compiler integrated technique, where the project is compiled only once instead of for each mutant [7]. These optimisations would also be implemented on line 14 in Algorithm 1.

2.3 Mutation Testing in the Cloud

Mutation testing has shown to be able to run in parallel on a distributed architecture [3, 22, 30]. Consequently, researchers are currently investigating cloud solutions to share the computational load across a series of hardware nodes. Hadoopmutator [28] and Eminent [5] are the ones we have found in the literature.

2.3.1 Hadoopmutator. Hadoopmutator is a cloud-based Mutation Testing framework that is implemented using Hadoop's MapReduce[1]. During the mapping phase, each mutation operator is assigned to a separate node, creating a single mutant and executing the test suite. The subsequent reduce phase aggregates the results from all the test executions and calculates the mutation score.

[1] https://hadoop.apache.org

Data transfer between the nodes is handled using Hadoop's MapReduce and the Hadoop Distributed File System (HDFS™). Hadoopmutator is applied on two open source projects, and the authors report a speed-up of 9.57x and 12.83x using 13 nodes. For small projects like Apache Wicket, the authors state that "*the overhead of running Hadoop on the compute nodes becomes significant relative to the time needed to generate and executes the tests and hence the optimal performance gain is not attained*". The influence of speed-up with large projects that affect the I/O and network traffic was not investigated.

For a MapReduce solution to reach an optimal load balance, there should be small variance between the execution times of the respective mapper functions. This minimises the inherent idle time between the termination of the mapper phase and the start of the reduce phase. However, for mutation testing this principle does not hold. First, because the analysis can terminate as soon as one test fails, thus some test executions will terminate earlier than others. Second, because some mutants lead to infinite loops, which can only be detected via time-outs. Therefore there is a large variance between the time to execute the tests.

2.3.2 Eminent. Eminent (EMbarrassINgly parallEl mutatioN Testing) is a distributed Mutation Testing tool that relies on the Message Passing Interface standard for portable message-passing in parallel computing architectures [5]. Eminent uses *test-level grain*, the test suite is split up and each test/mutant combination is run separately. If a single test (of a mutant) fails, the mutant is killed and all other remaining tests (related to that mutant) are canceled.

Eminent handles data transfer between the master and the nodes by means of a shared database. The test cases are sent to the worker processes which will execute them against the mutants and send the results back to the master, which compares them to the original. Eminent is applied on three projects, and the authors report a speed-up between 8x and 22x using 32 nodes. Large projects can have an impact on the scalability "*because of the high volume of network and I/O traffic generated by this application, which acts as a system bottleneck in the database node*".

The main advantage of Eminent's test-level grain is that the sooner a mutant is killed the faster the mutation testing tool becomes. Therefore, the test execution framework should be configured such as to stop the execution of the test suite when the first one fails. The second advantages of test-level grain is that the load can in principle be evenly distributed over multiple nodes. Nevertheless, the worst case scenario for test-level grain occurs when the last test is executed on one node while all the other nodes are finished. The additional overhead of running the test-level grain may be more than the execution time of the complete test suite.

At the time of writing there was no implementation available for HadoopMutater nor for Eminent. Replicating their results on other projects and measuring where overhead occurred was impossible. Therefore we resorted to our own proof-of-concept named DiMuTesTas.

2.3.3 DiMuTesTas. We developed DiMuTesTas to minimise the idle-time of the nodes and to minimise the network load of the distributed application itself. The first is tackled by removing dependancies between executions on the workers, allowing to distribute the generation of the mutants and the execution of the mutants independently of each other. We keep the network load low by only sending references to files over the network. We use *mutant-level grain*, thus apply a mutant and execute the complete test suite. Mutant-level grain allows to further reduce the amount of messages that need to be exchanged.

In our current system, we use a file system to distribute the project, mutants, and store their executed results. The data that needs to be transferred for the mutant is limited to a single file. This can even further be reduced by only sending the delta of the file. Writing to the file server is only done by each worker which has generated the mutants (single files) or by each worker which executed a mutant and needs to store its build output (multiple files).

Summary. The current state-of-the-art demonstrates that cloud infrastructure indeed speeds up the mutation testing. Yet, the optimal way of distributing the load across the available hardware nodes is currently unknown. First of all, there is the potential for idle-time when nodes are waiting for others to finish their tasks before they can proceed. Secondly, there is the data-transfer bottleneck, the consequence of copying files and exchanges messages across the nodes. Today, detailed measurements on the impact of both the idle-time and the data-transfer are lacking.

3 PROOF OF CONCEPT

In this section we first describe the cloud architecture of our proof-of-concept (loosely inspired by the 4+1 model [17]) and then identify where delays may have a significant impact, thus where we should measure overhead.

3.1 DiMuTesTas Architecture

Logical View — Single Task queue. To execute Algorithm 1 in the cloud, we adopt a single task queue model for the main phases *A* and *B*, as depicted in Figure 1. For phase *A*, this means creating a first kind of task, i.e. to *generate the mutants from a source (src) file* (represented by *A* in Figure 1), for each source file and push it onto the task queue. When a worker processes such a task (e.g. *1*), a new, second kind of task is created for each of the generated mutants, i.e. to *execute the mutant* (corresponding to phase *B*). These newly created tasks are then pushed back onto the task queue (e.g. *1a*, *1b* and *1c*).

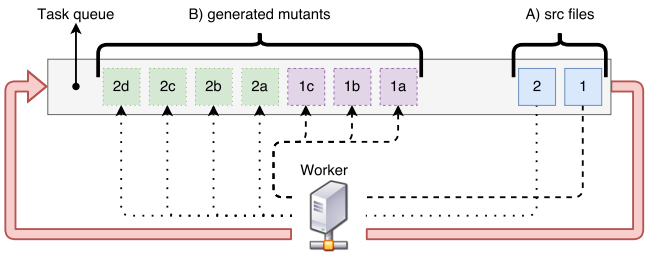

Figure 1: DiMuTesTas Architecture – Single Task Queue
©Sten Vercammen

Process View — RabbitMQ. The single task queue is handled by RabbitMQ[2], a broker which handles the message passing between multiple computers. To map the logical view onto RabbitMQ we follow a series of steps depicted in Figure 2. First, the master performs the initial build and verifies if all tests succeed (*1*, a.k.a. *pre*-phase); if not, the process is canceled. Afterwards, all source files are gathered and (their file names) are sent to the task queue (*2*). From here on, the master waits until he received all results. Once tasks are in the task queue, workers will pull and process them. If a worker pulls a task containing the name of a source file (*3a*), it will generate mutants for the corresponding file, store them on the file server (*3b*) and send a reference for each mutant back to the queue. If the worker pulls a task containing a reference to a mutant (*4a*), it will fetch the mutant from the file server (*4b*) and execute it before storing its result on the file server (*4c*) and sending a "done" message to the result queue (*4d*). Finally, when the master received all results, he creates the final report for the mutation testing (*5*, a.k.a *post*-phase).

Note that the mutants can be generated and executed by different workers and are needed for the final report as well. Therefore, they are not stored in the task queue as this would increase its memory usage and would require sending the mutants over the network one additional time. Instead, the mutants, results and the final report are fetched stored on a separate file server (*3b*, *3c* and *4c*) .

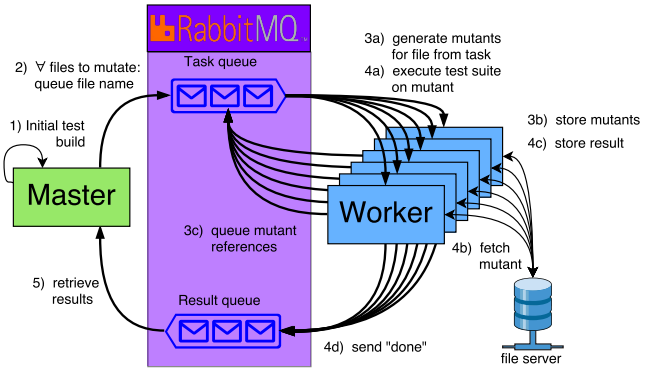

Figure 2: DiMuTesTas Architecture – Process View
©Sten Vercammen

Physical View – Docker. To deploy the architecture on a physical system, we rely on Docker[3]. Figure 3 provides an overview of the different components and their interactions. As setting up each node individually is impractical, the master and the worker are encapsulated into Docker images, i.e. an executable package that includes everything needed to run an application: the code, the runtime environment, the libraries, the environment variables, and the configuration files. As Docker images contain the installed software, they will startup very quickly because no further installation is required. An image for the RabbitMQ server already existed. The master and worker images do not include the project itself, but will copy it from a file server once they startup. We used NFS for the file server, but iSCSI, FC, and others are possible as well. The file server is also needed as the local storage of each Docker container is removed together with the container after execution. To distribute the tool over multiple PC's and enable the different containers to talk to each other, we make use of Docker Swarm.

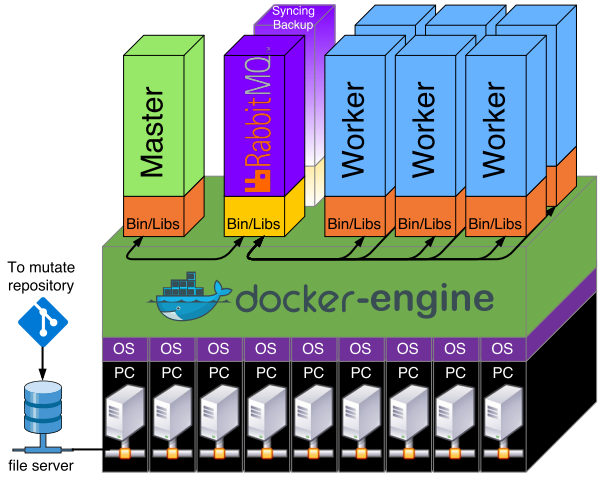

Figure 3: DiMuTesTas Architecture – Physical View
©Sten Vercammen

3.2 DiMuTesTas Potential Delays

Now that we laid out the components and how they interact (see Figure 3) we can identify where potential delays might occur compared to a mutation test run on a local PC.

- *Setup delay.* After the initial build on the master, DiMuTesTas copies the build dependencies from the file server to the own local storage in each worker. This is done to prevent the network connection from becoming a bottleneck, as otherwise each worker has to download the build dependencies separately.
- *Initial build.* The *pre*- and *post*-phase are similar for a mutation test run on a local PC and one that runs in the cloud, differing only in the place they store and gather information.
- *Mutant generation.* The total time all workers needs to generate the mutants, excluding the time to read from the file server/disk and writing the build output to the file server/disk. For a mutation test run on a local machine, the total time to generate the mutants, excluding the time to read or write the results.
- *Mutant execution.* Similar to the *mutant generation* phase, it measures the total time for the mutant execution phase excluding the time to read or write.
- *RabbitMQ (scheduling) delay.* The time to gather the source files and push them to the task queue on the RabbitMQ server. The time needed by the workers to pull tasks from the task queue is also part of this delay.
- *File server/disk delay.* The most likely cause for delays in a cloud solution is copying data files back and forth between the different nodes. This delay occurs when a container copies the project from the file server to its own local

[2]https://www.rabbitmq.com/
[3]https://docs.docker.com/

storage. However, it also occurs when transforming Docker images (static, unchangeable) into Docker containers (dynamic, changeable) at startup.

4 CASE STUDY SET UP

This section describes how we evaluate the impact of running mutation testing in the cloud. Essentially we apply DiMuTesTas on two industrial projects, comparing a cloud solution against a version running on a local PC.

4.1 Cases

We collected two cases via our network of industrial partners, one small (Intris), and one large (HealthConnect) project.

- *Case 1: Intris [https://www.intris.be].* The project at Intris relies heavily upon the visualisation and manipulation of database data. This manifests itself in the way the tests are written, as most of them are scenario tests. We choose a (core) subproject which does not rely on the database, but has few unit tests. This resulted in a small task execution time, as building the project and running the test suite only takes 7 seconds. The Intris project makes an interesting case for investigating mutation testing in the cloud as the execution times of the tasks are quite small thus we expect more overhead from scheduling delays and file server/disk delays.
- *Case 2: HealthConnect [https://www.healthconnect.be].* The project at HealthConnect is 1.8GB large, and contains 48kLOPC (Lines of Production Code), and 5kLOTC (Lines of Test Code). As each hardware node needs to copy the entire code base, the fileserver hosting the source files from the project may become a bottleneck. The project from HealthConnect makes an interesting case to examine the behaviour on large (especially in data transfer) projects.

The descriptive statistics of the project are listed in Table 1, while the details regarding basic mutation testing are shown in Table 2. Note that the given start date and number of commits from HealthConnect is counted from the switch to the new version control system, the actual start date is earlier. Note as well that the time to execute the test suite assumes that all dependencies are loaded and stored locally.

4.2 Research Questions

The case study is driven by the following two research questions.

RQ1: Speed-up. How much speed-up can be achieved by running a mutation testing on cloud infrastructure?
Motivation. Assuming that we distribute the mutation test run over a system with N hardware nodes, the ideal speed-up is a factor Nx. Here we investigate whether DiMuTesTas approaches this ideal.
Approach. We deploy DiMuTesTas on a cloud system with a maximum of eight hardware nodes, where each hardware node is configured with 2 workers. We measure the total execution time with a set-up of 1, 2, 4, 8, and 16 workers and average the execution time across 3 runs.

RQ2: Delays. Where does a cloud solution like DiMuTesTas suffer from delays? Do these delays correspond to what may be expected?
Motivation. Deploying mutation testing in the cloud induces delays, in particular with respect to data-transfer between the nodes. This research question compares a mutation test run executed on a local PC against a mutation test run deployed in the cloud. By making a thorough analysis of where delays occur we can suggest avenues for further improvement.
Approach. Based on the set-up described in RQ1 (1, 2, 4, 8, and 16 workers) we measure the points in the architecture where delays might occur as described in Section 3.2: Setup delay, Initial build, Mutant generation, Mutant execution, RabbitMQ (scheduling) delay and the File server/disk delay. We compare the actual measurements against what we expect.

4.3 Hardware Set-up

The infrastructure used for the analysis of both projects is the same. We used 8 Intel(R) Core(TM)2 Quad Q9650 CPU's, each with two 4GB (Samsung M378B5273DH0-CH9) DDR3 RAM modules and a 250GB Western Digital (WDC WD2500AAKX-7) hard drive. All PC's were connected to the same subnet using a 3Com Baseline Switch 2016 (100Mbps, full duplex). The in- and outgoing internet connections/inter-PC communications were limited by the 100Mb links. We used a dedicated switch to remove any external influences on the network load. All PC's where running Ubuntu 16.04.2 LTS with kernel 4.10.0-27.
The workers from the DiMuTesTas approach where divided equally over the nodes: when using N nodes and $2N$ workers, each node will run two workers. When executing LittleDarwin, only one of the PC's was used.

4.4 LittleDarwin

In principle, DiMuTesTas can be set-up with any mutation tool, as long as it can be configured to apply a single mutator on a given file. For this particular case study we relied on LittleDarwin, a tool distributed within our lab thus conveniently accessible [25].

5 RESULTS

5.1 RQ1 - Speed-up

How much speed-up can be achieved by running a mutation testing on cloud infrastructure?

Table 3 compares the execution times of LittleDarwin against DiMuTesTas using 1, 2, 4, 8, and 16 workers. Each result is an average of 3 runs; the variance between each run is less than 2%. For easy interpretation of the scaling, we took the execution time of DiMuTesTas running on a single worker (therefore running sequentially) as our baseline, indicating it as 100% of the execution time. As the number of workers double, we see that the execution time almost halves thus the speed-up increases linearly. Nevertheless, the relative execution time of LittleDarwin is below 100%, because —as expected— executing mutation testing in the cloud adds overhead.

For the Intris Case (8 kLOC production code, 300 LOC test code) we could reduce the full mutation testing cycle from 2.7 hours to 13.5 minutes. For the HealthConnect case (48 kLOC production code, 5k LOC test code) we could reduce from 48 hours to 3.7 hours. As such, our proof-of-concept achieved a speed-up between 12x and 12.7x on a cloud infrastructure with 16 nodes.

Table 1: Industrial Cases: Descriptive Statistics

Company	Project Start Date	Nr Commits	Nr Developers	Java Files	LOPC	LOTC	Test Cases	Branch Coverage
Intris	26 May 2014	27,034	14	85/4,070	8,389	343	45	1.27%
HealthConnect	18 Jun 2014	22,956	10	601/273,722	48,873	5,258	272	28.34%

LPOC = Lines of Production Code; LOTC = Lines of Test Code

Case	Project Size	Test Suite Run Time	buildOutput-FileSize	avg-FileSize	interNode-LinkSpeed	readWriteSpeed-LocalDisk
Intris	116MB	7 s	37kB	15kB	100Mbps	95MBps
HealthConnect	1.8GB	47 s	840kB	8.52kB	100Mbps	95MBps

Table 2: Industrial Cases: Mutation Testing Data

Company	Project Size	Executed Mutants	Invalid Mutants	Actual Mutants	Killed Mutants	Mutation Coverage
Intris	116MB	1,364	312	1,052	33	3.14%
HealthConnect	1.8GB	4,104	50	4,054	360	8.88%

Table 3: Results (Distributed) Mutation Testing Experiment

Mutation Testing Tool / Case	LittleDarwin	DiMuTesTas				
		1 worker (1 node)	2 workers (2 nodes)	4 workers (4 nodes)	8 workers (8 nodes)	16 workers (8 nodes)
Intris	9,718.15 s	10,360.64 s	5,237.88 s	2,676.80 s	1,404.76 s	810.49 s
	93.80%	100.00%	50.56%	25.84%	13.56%	7.82%
HealthConnect	173,061.51 s	190,313.06 s	96,218.86 s	48,805.47 s	25,301.98 s	13,618.04 s
	90.94%	100.00%	50.56%	25.64%	13.29%	7.16%

5.2 RQ2 - Delay

Where does a cloud solution like DiMuTesTas suffer from delays? Do these delays correspond to what may be expected?

The delays of the different phases are listed in Table 4. Table 5 summarises these results, comparing the delay we expected against the delays observed.

5.2.1 Setup delay. In Table 4, we see that the *setup delay* grows linearly with the number of workers. This is as expected, as the source files from the project are copied to each worker individually. While we would expect the initial setup delay of the 16 workers to be twice as long as the one of the 8 workers, we see that they are alike. When running the 16 workers, each node has two workers. Because we used NFS as our file server, the kernel caches the data from the requests, allowing the second worker on each node to use the cached data instead of copying it from the file server.

The linear growth of the *setup delay* mainly impacts the execution time of the project from HealthConnect. The larger the project, the more time the copying of the files will take. In our case, the limited network connection of 100Mbps is making the delay extra apparent. The delay can be decreased by using a gigabit network and by minimising the amount of data that needs to be sent over the network. The latter can be achieved by keeping the project in a Docker volume between consecutive runs of the distributed mutation testing.

Another optimisation is to replace unicast with multicast [6]. With unicast, the project is transmitted to each node individually, thus we send the same project 8 times over the network when using 8 nodes. Multicast, on the other hand, sends the (same) project only once over the network, making the amount of data sent over the network independent of the amount of nodes.

5.2.2 Initial build. Next in Table 4 is the *initial build* on the master. We see that the execution times of the initial build from the different workers are similar. As this step is the same, independent of the amount of workers, we expected a constant *initial build* delay, hence this result is as expected.

5.2.3 ***Mutant generation.*** The third row in Table 4 shows the total time to *mutant generation*, excluding the time to read (write) the files from (to) the server/disk. The time to generate the mutants using LittleDarwin or DiMuTesTas for all worker configurations should be the same, as the same amount of work should be done. While this is the case for the project from HealthConnect, we see a decrease in execution time for the project from Intris. The shorter execution time using fewer workers is due to a memory limitation when needing to process multiple large files. With more workers, each worker needs to process fewer of these files, allowing for more memory to be used for each file.

The project from Intris has fewer mutants but a higher LOC/test file than the project from HealthConnect (Table 1). However, the mutant generation time of Intris is 3.6 times as long (see Table 4). We assume this behaviour is caused by the complexity of generating mutants. With LittleDarwin this complexity is exponential compared to the amount of lines in a file. In general, if two projects have the same LOC count, but differ in the number of files, then the

Table 4: Results: Delays incurred in DiMuTesTas

(in seconds, bold program section represent cumulative timings, i.e. the sum of time all workers spend in that phase)

Case	Program section	LittleDarwin	1 worker	2 workers	4 workers	8 workers	16 workers
Intris	Setup delay	N.A.	18.39	27.83	46.63	84.34	*81.59*
	Initial build	7.98	7.72	7.92	7.82	8.18	8.35
	Mutant generation	928.46	1,018.49	558.14	332.04	256.38	255.23
	Mutant execution	8,770.34	9,296.29	9,791.96	10,090.61	10,142.68	11,057.54
	RabbitMQ delay	N.A.	9.09	10.69	23.71	37.43	95.08
	File server/disk delay	2.58	5.24	12.94	18.20	19.13	21.56
HealthConnect	Setup delay	N.A.	156.37	305.91	613.83	1,265.15	*1,319.37*
	Initial build	114.78	159.03	163.5	177.63	165.25	180.31
	Mutant generation	253.27	296.11	257.19	270.29	311.63	269.92
	Mutant execution	172,681.67	189,488.88	190,937.38	191,310.10	189,881.46	191,940.89
	RabbitMQ delay	N.A.	22.69	30.73	106.06	203.18	386.73
	File server/disk delay	6.68	144.86	160.63	187.34	190.85	253.95

Table 5: Overview of Expected and Actual Delays

Program section	Expected delay	Actual delay
Setup delay	Linear to nr. of workers	Linear to nr. of **nodes**
Initial build	Constant	Constant
Mutant generation	Constant	**Decreasing**
Mutant execution	Constant	Constant
RabbitMQ delay	max $n-1$ workers * test execution time	max $n-1$ workers * test execution time
File server/disk delay	Linear to nr. of workers	Linear to nr. of workers

one with the most files (less LOC/file) will result in a faster mutant analysis.

5.2.4 ***Mutant execution.*** The fourth row in Table 4 shows the time needed to *mutant execution*, here as well excluding the time to read (write) the files from (to) the server/disk. We observe execution times which are more less similar, which should be the case as the code responsible for this in DiMuTesTas is the exact same code as of LittleDarwin. Not surprisingly, the mutant execution time comprises the largest chunk of the whole mutation testing time, hence that is where optimisations should focus on.

5.2.5 ***RabbitMQ delay.*** The fifth row in Table 4 shows the *RabbitMQ (scheduling) delay*, i.e. the time needed to gather the names of the source files send them to the task queue plus the time needed for the workers to pull tasks from the task queue. This delay is calculated by removing all timed functions (delays, mutant generation and execution) from the local execution time of the worker. The delay incorporates the execution time of some small, untimed functions. We can see that the delay increases, but it is important to note that this is cumulative. If e.g. seven out of the eight workers are done processing, then for every second that passes before the last worker is done, seven seconds are added to this delay. If we divide the delay by the number of workers, then we see that this delay is smaller than the time it takes to process a single task. We conclude that this delay is caused by idle-time when some nodes stop processing earlier than others.

5.2.6 ***File server/disk delay.*** The sixth row in Table 4 shows the *file server/disk delay*, i.e. the time needed to copy data files back and forth between the different nodes. Although this delay is relatively small, it grows linearly with the amount of workers, hence is a point for concern. This delay could be minimised by sending deltas of the files over the network instead of sending the complete file.

Based on detailed measurements concerning delays in the analysis we point out directions for further optimisation. In particular, the use of *multicast* should ensure that the *set-up delay* —the current bottleneck– would take constant time, regardless of the number of nodes in the system. In the same vein, we can minimise the *file server/disk delay* by only sending the deltas of the files.

6 LESSONS LEARNED

In this section we will derive the lessons learned geared towards the mutation testing community.

Nightly Builds. The mutation testing algorithm in Algorithm 1 is inherently parallel, thus a cloud solution allows to share the computational load across a series of hardware nodes. Given sufficient hardware it is possible to off-load a full-scale mutation testing on dedicated hardware.

☑ Cloud infrastructure allows to speed up mutation testing depending on the number of hardware nodes available. This speed-up allows to perform mutation testing during the *nightly build.*

Cloud Technology. We observed that it is beneficial to run many workers on the same node to reduce the *setup delay*. However, the set-up delay is still significant and grows with the number of nodes, mainly because we copy the whole project source code to each of

the available nodes. In the future, we intend to use *Multicast* the setup delay should become a constant, independent of the number of nodes in the system. In a similar vein, we will reduce the file server/disk delay by sending deltas of the files over the network.

☑ There is a lot of research and development on cloud infrastructure and the field is making rapid progress. In the near future, we may expect new features that can be exploited to make a cloud based mutation testing even faster.

Side-effects. During *mutant generation* we noticed that with more workers, each worker needs to process fewer of these files, allowing for more memory to be used for each file.

☑ Deploying mutation testing in the cloud sometimes lead to side-effects having a positive impact on the execution time.

Completely independent tasks. DiMuTesTas is designed to minimise idle times between tasks. The single task queue does not add any delay because it only needs to pass the name of the mutated file, and only the mutated file will be copied from the file server to the worker. This can only work when there are no other dependencies between tasks, in particular between the generation of the mutants (Phase A in Algorithm 1) and the execution (Phase B in Algorithm 1). Likewise, executing a single mutant (line 14 in Algorithm 1) should not affect any other executions.

☑ For optimal deployment in a cloud infrastructure, tasks should remain completely independent. Mutant optimisation techniques that violate this principle will benefit less from deploying in the cloud.

Complementary Optimisation. Deploying mutation testing in the cloud reduces the total mutation testing time according to the number of hardware nodes available. Nevertheless, the *mutant generation* and *mutant execution* phases take the largest proportion of the whole analysis.

☑ There is ample room for complementary optimisation techniques that reduce the time needed to generate and execute mutants.

7 THREATS TO VALIDITY

As with all empirical research, we identify those factors that may jeopardise the validity of our results and the actions we took to reduce or alleviate the risk. Consistent with the guidelines for case studies research (see [27, 29]), we organise them into four categories.

Construct validity: do we measure what was intended? In essence, we wanted to know which parts of the distributed mutation testing process were causing extra delays. Therefore, we compared execution times on phases where we suspected delays could occur. One may conceive other choices for measuring these delays. However, the suggestions for further improvement (i.e. multicast and sending of deltas of files) are likely to remain valid.

Internal validity: are there unknown factors which might affect the outcome of the analyses? As the performance of a (distributed) system can be influenced by external factors, a replication experiment could end up with different timings. For example, if the computer cannot sufficiently dissipate the generated heat by the CPU, the CPU could slow down over time. Similarly, the condition of the hard drive in the file server and the load of the network could affect the measurements. However, the results would need to differ significantly before they would invalidate the suggestions for further improvement.

External validity: to what extent is it possible to generalise the findings? We evaluated our proof-of-concept distributed mutation testing tool to industrial cases, both a small and a large one. We assume that similar measurements would apply on other projects, however the proposed solutions will need to be tailored to the projects. Projects that are small in size but have long running test suites are different from projects large in size but with very short running test suites.

Reliability: is the result dependent on the tools? We deliberately choose a mutation testing tool which clearly separates the different steps of the mutation testing process. This allowed us to measure the delay of running the mutation tests in the cloud. If the mutation testing tool contains optimisations which can be distributed without a negative impact on the performance (i.e. test prioritisation), then the results should be similar. However, as mentioned in Section 6 For optimal deployment in a cloud infrastructure, tasks should remain completely independent.

8 CONCLUSION

In this paper, we demonstrated that cloud infrastructure allows to speed up mutation so much that it can be performed during the *nightly build*. For a small scale system (8 kLOC production code, 300 LOC test code) we could reduce the full mutation test run from 2.7 hours to 13.5 minutes. For a large project (48 kLOC production code, 5k LOC test code) we could reduce from 48 hours to 3.7 hours. As such, our proof-of-concept achieved a speed-up between 12x and 12.7x on a cloud infrastructure with 16 nodes.

Despite these improvements, there are still opportunities for further optimisation. Based on detailed measurements concerning delays in the analysis, we point out directions for further optimisation. In particular, the use of *multicast* should ensure that the *set-up delay* —the current bottleneck– would take constant time, regardless of the number of nodes in the system. In the same vein, we can minimise the *file server/disk delay* by only sending the deltas of the files.

Moreover, we also derive a few lessons learned for the mutation testing community. Most important is the principle that for optimal deployment in a cloud infrastructure, tasks should remain completely independent. Mutant optimisation techniques that violate this principle will benefit less from deploying in the cloud. Nevertheless, there is ample room for complementary optimisation techniques that reduce the time needed to generate and execute mutants.

9 ACKNOWLEDGMENTS

This work is supported by (a) the ITEA TESTOMATProject (number 16032), sponsored by VINNOVA – Sweden's innovation agency; (b) Flanders Make vzw, the strategic research centre for the manufacturing industry.

REFERENCES

[1] Bram Adams and Shane McIntosh. 2016. Modern release engineering in a nutshell–why researchers should care. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, Vol. 5. IEEE Press, Piscataway, NJ, USA, 78–90.

[2] Richard Baker and Ibrahim Habli. 2013. An Empirical Evaluation of Mutation Testing for Improving the Test Quality of Safety-Critical Software. *IEEE Transactions on Software Engineering* 39, 6 (June 2013), 787–805. https://doi.org/10.1109/TSE.2012.56

[3] Choi Byoungju and Aditya P Mathur. 1993. High-performance mutation testing. *Journal of Systems and Software* 20, 2 (1993), 135–152.

[4] Xia Cai and Michael R Lyu. 2005. The effect of code coverage on fault detection under different testing profiles. *ACM SIGSOFT Software Engineering Notes* 30, 4 (2005), 1–7.

[5] Pablo C Cañizares, Mercedes G Merayo, and Alberto Núñez. 2016. EMINENT: EMbarrassINgly parallEl mutatioN Testing. *Procedia Computer Science* 80 (2016), 63–73.

[6] Stephen E Deering and David R Cheriton. 1990. Multicast routing in datagram internetworks and extended LANs. *ACM Transactions on Computer Systems (TOCS)* 8, 2 (1990), 85–110.

[7] Richard A DeMillo, Edward W Krauser, and Aditya P Mathur. 1991. Compiler-integrated program mutation. In *Computer Software and Applications Conference, 1991. COMPSAC'91., Proceedings of the Fifteenth Annual International*. IEEE Press, Piscataway, NJ, USA, 351–356.

[8] Vahid Garousi and Junji Zhi. 2013. A survey of software testing practices in Canada. *Journal of Systems and Software* 86, 5 (2013), 1354–1376.

[9] G. Gay, M. Staats, M. Whalen, and M. P. E. Heimdahl. 2015. The Risks of Coverage-Directed Test Case Generation. *IEEE Transactions on Software Engineering* 41, 8 (Aug 2015), 803–819. https://doi.org/10.1109/TSE.2015.2421011

[10] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, New York, NY, USA, 72–82.

[11] Laura Inozemtseva and Reid Holmes. 2014. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, New York, NY, USA, 435–445.

[12] Goran Petrović Marko Ivanković, Bob Kurtz, Paul Ammann, and René Just. 2018. An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions. In *Proceedings of the International Workshop on Mutation Analysis (Mutation)*. IEEE Press, Piscataway, NJ, USA, To appear.

[13] Jon Jenkins. 2011. Velocity Culture. (2011). Keynote Address at the Velocity 2011 Conference.

[14] Yue Jia and Mark Harman. 2011. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2011), 649–678.

[15] Susanne Kandl and Sandeep Chandrashekar. 2015. Reasonability of MC/DC for safety-relevant software implemented in programming languages with short-circuit evaluation. *Computing* 97, 3 (March 2015), 61–279. https://doi.org/10.1007/s00607-014-0418-5

[16] Kim N King and A Jefferson Offutt. 1991. A fortran language system for mutation-based software testing. *Software: Practice and Experience* 21, 7 (1991), 685–718.

[17] Philippe Kruchten. 1995. The 4+1 View Model of Architecture. *IEEE Software* 12, 6 (Nov. 1995), 42–50. https://doi.org/10.1109/52.469759

[18] Rob M. 2014. Everything you need to know about Tesla software updates. (2014). [on line] https://www.teslarati.com/everything-need-to-know-tesla-software-updates/ — last accessed In May 2018.

[19] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale Continuous Testing. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP '17)*. IEEE Press, Piscataway, NJ, USA, 233–242. https://doi.org/10.1109/ICSE-SEIP.2017.16

[20] Glenford J Myers. 1979. 77ie Art of Software Testing. (1979).

[21] SP Ng, Tafline Murnane, Karl Reed, D Grant, and TY Chen. 2004. A preliminary survey on software testing practices in Australia. In *Software Engineering Conference, 2004. Proceedings. 2004 Australian*. IEEE Press, Piscataway, NJ, USA, 116–125.

[22] A Jefferson Offutt, Roy P Pargas, Scott V Fichter, and Prashant K Khambekar. 1992. Mutation testing of software using a MIMD computer. In *in 1992 International Conference on Parallel Processing*. CRC Press, Boca Raton, Florida, II–257–266.

[23] A Jefferson Offutt and Roland H Untch. 2001. Mutation 2000: Uniting the orthogonal. In *Mutation testing for the new century*. Springer, Berlin, Germany, 34–44.

[24] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation Testing Advances: An Analysis and Survey. *Advances in Computers* In Press, Corrected Proof (Available online 7 May 2018 2019), —. https://doi.org/10.1016/bs.adcom.2018.03.015

[25] Ali Parsai, Alessandro Murgia, and Serge Demeyer. 2017. LittleDarwin: a Feature-Rich and Extensible Mutation Testing Framework for Large and Complex Java Systems. In *Proceedings FSEN2017 (7th IPM International Conference on Fundamentals of Software Engineering)*, Mehdi Dastani and Marjan Sirjani (Eds.). Springer International Publishing, Berlin, Germany, 148–163. https://doi.org/10.1007/978-3-319-68972-2_10

[26] Rudolf Ramler, Thomas Wetzlmaier, and Claus Klammer. 2017. An Empirical Study on the Application of Mutation Testing for a Safety-critical Industrial Software System. In *Proceedings of the Symposium on Applied Computing (SAC '17)*. ACM, New York, NY, USA, 1401–1408. https://doi.org/10.1145/3019612.3019830

[27] Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical Softw. Engineering* 14, 2 (2009), 131–164.

[28] Iman Saleh and Khaled Nagi. 2015. Hadoopmutator: A cloud-based mutation testing framework. In *International Conference on Software Reuse*. Springer, Berlin, Germany, 172–187.

[29] Robert K. Yin. 2002. *Case Study Research: Design and Methods, 3 edition*. Sage Publications, —.

[30] Christian N Zapf. 1993. *MedusaMothra-A distributed interpreter for the Mothra mutation testing system*. Master's thesis. Clemson University.