Future Generation Computer Systems 95 (2019) 502–510

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

On using collaborative economy for test cost reduction in high fragmented environments

Kenyo Abadio Crosara Faria [a,*], Raphael de Aquino Gomes [b], Eduardo Noronha de Andrade Freitas [b], Auri Marcelo Rizzo Vincenzi [c]

[a] *Instituto Federal de Goiás, Inhumas, Goiás, Brazil*
[b] *Instituto Federal de Goiás, Goiânia, Goiás, Brazil*
[c] *Universidade Federal de São Carlos, São Carlos, São Paulo, Brazil*

HIGHLIGHTS

- Mobile cloud test environment are expensive and not scalable.
- The collaborative economy may minimize total cost of mobile software testing.
- A new market may be created to device owners.

ARTICLE INFO

Article history:
Received 1 May 2018
Received in revised form 11 October 2018
Accepted 15 January 2019
Available online 19 January 2019

Keywords:
Collaborative economy
Android ecosystem
Software testing

ABSTRACT

The grown adoption of mobile devices makes the development of applications a very attractive market. On top of it, run tests is a crucial activity and a big challenge due to the high fragmentation on Android ecosystem. In this paper, we discuss how a new platform based on Collaborative Economy could be used to create a new alternative to software testing. We present an analysis of using this platform and we confirm its advantages over existing cloud solutions, from a scalability and cost viewpoints. Our solution can provide an average cost reduction upper to 85% and a potential increasing in scalability.

1. Introduction

The worldwide smart-phone market is significantly growing annually. According to data from the Statista [1], 2017 finished with 4.77 billion of mobile phone users worldwide, which were up 3.4% over 2016, and a most recent projection showing 2018 shipments of 4.93 billion corresponding to 3.3% growth over 2017.

Actually, Android has around 85.9% of worldwide market share [2], and the number of related apps grows to 3.5 million 2017 [3]. Also, to illustrate the complexity of that kind of market, Android owns a proliferation of brands, screen sizes and screen densities, resulting in more than 24 thousands of different devices [4]. These devices have four generalized screen sizes (small, normal, large, and extra-large), six generalized densities (ldpi, mdpi, tvdpi, hdpi, xhdpi, and xxhdpi), and 27 operating system versions [5]. Such characteristics present a significant challenge for developers and testers because delivering a fault application in this context, in general, has a profoundly negative impact [6].

Even if the report provided by OpenSignal [4] points out that ten manufacturers dominate 80% of the market, the variation of the features on each model should be addressed in the validation of the apps, as several situations may occur, e.g., some users do not keep the operating system updated.

In addition, given the huge number of different device-setup on Android ecosystem, another practical challenge Android developers face is the limited access to a reasonable number of those real devices to verify the app behavior and compatibility. Basically, there are two options to address it; buy a set of real devices or rent devices in cloud services. In practice, both have been an expensive process.

We realized that Android testing can be modeled and formulated as a business based on Collaborative Economy. In this sense, device owners freely make their devices available under a financial reward, and Android developers and testers demand those devices in an attractive business model.

In this paper, we discuss how a new platform based on the Collaborative Economy named Distributed Bug Buster (DBB) can be used to both create a new market for mobile-device owners and to provide a less expensive and effective way to increase Android app quality cross-device. Our analysis shows the effectiveness of this

* Corresponding author.
E-mail address: kenyo.faria@ifg.edu.br (K.A.C. Faria).

https://doi.org/10.1016/j.future.2019.01.023
0167-739X/© 2019 Published by Elsevier B.V.

business in terms of cost and some existing weakness presented by current mobile cloud test players. We confirm this hypotheses through two research questions: RQ1: How the platform compares with the other proposals in terms of scalability and variety? and RQ2: What is the economic impact of adopting our platform instead of alternative solutions? The results pointed that currently cloud testing are expensive and does not provide substantial devices in their catalogs, beyond scalability limitations. On the other hand, the use of our solution may lead to a considerable cost reduction.

The remain of this paper is organized as follows: In Section 2 we discuss the needed background to problem understanding. In Section 3 we present an overview of proposed platform. In Section 4 we present our results towards the raised research questions. In Section 5 we analyze some related work. We also discuss some threats of validity in Section 6 and we conclude with final remarks.

2. Background

Typically, for that an Android app to be free of defects, mainly these arising from incompatibility issues, developers make use of devices and emulators to validate their apps (target-app). Considering that emulators does not provide a real environment to app execution, tests using real devices is mandatory.

Due the high cost to acquiring real devices, some companies had released services able to provide real devices to be used in the test execution [7–11]. However there are problems about scalability, variety of available devices and cost, arising from architecture adopted.

For that mobile tests can work, some frameworks assist developers in both the writing [12–15] and execution tests [16,17] processes. In this work we focus on UI Tests because it impacts on the app compatibility cross-device, so we will provide an overview about Android instrumented tests with Google Espresso (the main framework to perform UI Tests to Android apps).

2.1. Instrumented tests on android

Instrumentation controls the app under test an permits the injection of mock components required by the app to run. It implies that when a developer compiles its app and related tests, the test framework manages both app and tests.

Typically two apks are generated by Android SDK during the compilation, one about app and one about tests, so the existing runner in the framework is responsible to execute the tests in indeed devices. The issue is that the runner is dependent on Android Debug Bridge (ADB) for accessing device. This characteristic represents a serious limitation regards scalability, once that the devices under test is limited to a number of available USB connections. Our platform eliminate this limitation through a new runner that we built, which is presented in Section 3.

Following, we discuss how the collaborative economy can help developers community in the cost reduction as well as create a new market using idle devices spread around the world.

2.2. Collaborative economy applied on software testing context

Several disruptive solutions make use of collaborative economy concepts to make feasible their views. For instance, Uber [18] and Turo [19] have revolutionized the way people transport, while Airbnb [20] has significantly changed the hosting market. These solutions have in common the goods sharing through a profitable relationship.

As pointed out before, there is a high number of Android devices around the world, potentially idle some hours per day. As devices are valuable resources to validate apps, developers could use them through a rental contract. Thus, a device owner could register its device to be accessible to developers run their tests. This way, a new market involving mobile devices, would be created in order to reduce the cost of testing related to the execution process.

In the next section, we provide an overview of the proposed platform to implement this partnership establishing.

3. Proposed platform

This section describes the proposed platform from a component perspective considering native Android apps. We omit details about platform implementation since our focus on this paper is the analysis of advantages on using it.

As mentioned in Section 2.1 the problem with the traditional test running approach is that target devices must be connected to the developer host (through USB connection). This need brings a serious limitation when facing a high fragmented environment such as Android. To solve this problem we built a customized runner based on AndroidJUnitRunner [16], able to execute espresso tests in order to allow an execution of tests without the ADB. This way, the developers must include our runner as a dependency of their apps.

In addition, we built a platform named Distributed Bug Buster (DBB) able to receive tests and distribute it around available devices over Internet. Thus, making use of existing idle processing capability and making it feasible through rental of devices. Fig. 1 illustrates how a tester can benefit from devices spread around the world to run tests. Each device signed in DBB is cataloged in the device farm and is shown as available to test according to its owner convenience. At the end of the test run, a report is sent to its respective testers, who had submitted tests to be executed on the farm.

In order for developers/testers to submit their tests to run on the devices registered on the platform, related to an app and related to tests, they need only send the apks files to the platform. Through the fantastic interface, they can choose on which devices they want to validate the app. The catalog of devices is organized by brand, model, Android version and language. After submitting the test, the developers/testers can track which devices have been accepted and which ones have already returned the test execution report.

We built two components, an agent (DBB-agent) which is responsible to receive, execute and report accepted tests, and a web app (DBB-web) which provides a way to allow testers to send their apps and related tests to be executed on signed up devices. For that developers can access their test reports, they must to pay device owners through a service billing provided by platform.

We intend to take advantage of collaborative economy to solve the problem of monetary cost on testing. The proposed approach requires the engagement on the part of the device owner once the operating system security policies do not allow that the installation of apps occurs silently. Thus, people who want to rent their devices in this platform need to install the DBB-agent, which provides authentication and turns device available for running tests.

As soon as the developers/testers send the test set, the DBB-agent notifies related device owners to accept or to deny test set execution. For that acceptance occur, the device owners need to install both, the app under testing and the UI-tests. For that, it is necessary that the device owners accept installation from unknown sources.

The simple acceptance of test execution does not guarantee reward for device owners, because the value generated for testers is in the test report, sent only after test execution. Multiple devices of the same model receive the test and, even if test acceptance is taken in all of it, technical issues (connectivity, charging, etc.) or even lack of device idleness may break reporting test results. In this

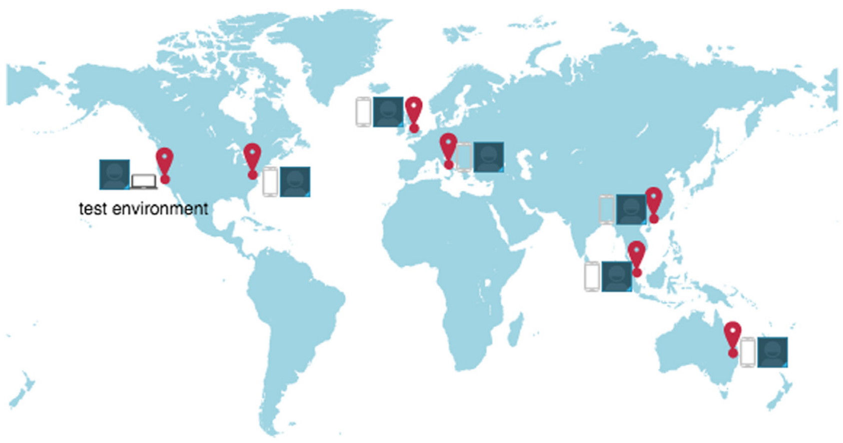


Fig. 1. Tester site and physical location of devices available on the DBB farm.

scenario, billing is performed for the first device that returns the report. This action is carried out to prevent that developers/testers pay for runs of a test suite on devices of the same model, which for UI testing purposes does not make sense.

Finally, once tests execution finished on the devices, their owners can track performed tests as well as the associated billing through both environments DBB-web and DBB-agent.

In the next section we provide an evaluation looking for important issues about existing mobile cloud testing players. In Section 6, we address some issues about remote tests execution and the inherently limited control.

4. Evaluation

In this section, we conducted an analysis which aims to evaluate our proposed platform against existing alternative solutions. We did it through answering the research questions introduced in Section 1. We first analyze the scalability capability, followed by a cost comparison.

4.1. Scalability comparison

In our analysis we consider the major clouds of mobile devices made available for testing: Google Firebase Test Cloud [9], AWS Device Farm [7], Xamarin Test Cloud [8], Kobiton [11], and Perfecto [10]. By taking information about device availability on these solutions we could conclude that test cloud players do not provide a scalable platform and do not provide a big coverage what concern diversity of devices. These players try to provide devices which compose the main market share.

Typically there are two ways to contract cloud testing infrastructure, based on-demand, in which the users pay according to their usage, and by reservation, in which users pay for a static resource to be used in a determined time period. In the demand model, developers are at risk of being overwhelmed by an expensive pricing due to high resource utilization, on the other hand, in the reservation model the inherent risk is to oversize the device allocation.

Our analysis was carried out considering four variables which impact on the scalability of platform and coverage of market share: number of devices, diversity of device models, contract model, and concurrent devices. The device's availability is an important concern because it impacts on the scalability. We analyzed device diversity because as many devices models are provided, the wider the market share coverage. Concurrent devices usage is used by some players to avoid devices unavailability due to the low diversity and an incipient number of physical devices. Contract model is considered by its direct impact on the cost. The results are summarized in Table 1. The data were retrieved from the devices catalog provided by each one of players.

The fact that current mobile test cloud players are based on the acquisition of devices, is due the mobile devices cannot be used on a shared basis due to technical limitations. This way, for that these players, can support for example 10 testers running tests simultaneously, it is necessary to buy 10 different devices. It is clearly not scalable. Taking as example Kobiton cloud [11], it could not support 500 testers running tests, because it provides only 300 devices in its cloud.

Even when the mobile cloud player has a large number of devices, it may suffer from its low device diversity. For example, taking Perfecto test cloud [10] which has the largest number of devices, we can see 10,000 devices available to be used but just 62 different models. It can clearly be a bottleneck on very crowded device models. This limitation is worked around through concurrent devices allowed by test running so that a tester can run their tests in a limited number of different models at the same time. It can be viewed in AWS Device Farm [7] and Xamarin [8], Perfecto [10] and Kobiton [11] which make use of this limitation although they offer plans that allow testers to use all device model in the catalog at the same test running request.

Assuming that presented mobile test cloud players support the existing demand generated by testers who use clouds to test their apps regarding the availability of devices, another problem to be addressed by RQ1 is about coverage. Considering a report provided by Open Signal in 2016 [4], there were more than 24 thousands of different Android devices in the world, that is, the catalog provided by players in Table 1 cannot help testers to ensure a good compatibility checking, even if they have devices with the highest market share in their catalogs, because there are apps that must be validated against devices with low market share.

According to IDC [21], Chinese devices represent more than 20% of the global market share in the fourth quarter of 2017, this data is presented in Table 2. Chinese vendors are represented by Huwaei [22], Xiaomi [23] and OPPO [24]. The Fig. 2 shows the market share of main Chinese device models. Despite this important market share, only XIOMI RedMI Note 4X is provided by one of mobile test cloud players presented in Table 1. This lack of models is an important problem of coverage, leading to testers the need of buy devices or validate their apps using emulators.

DBB has total conditions to provide a scalable cloud platform with a bigger catalog composed of all active models. Its strategy can take advantage of idle devices around the world in order to provide a large number and wide variety of devices since it makes use of devices already acquired. In the next section, we compare, from a cost perspective, the players discussed in this section.

Table 1
Mobile cloud players scalability.

Platform	Available devices	Device diversity	Contract model	Concurrent devices
Google firebase test cloud	nd	69	On-demand only	Devices on catalog
AWS device farm	1 000	191	On-demand and reservation	5 devices
Xamarin test cloud	2 000	250	Reservation	30 devices
Perfecto	10 000	62	Reservation	3 device
Kobiton	300	95	Reservation	1 device

Table 2
Worldwide vendor market share.

	Samsung	Huwaei	Xiaomi	OPPO
2016Q4	18%	10.6%	3.3%	7.3%
2017Q1	23.3%	10%	4.3%	7.5%
2017Q2	22.9%	11.1%	6.2%	8%
2017Q3	22.1%	10.4%	7.5%	8.1%
2017Q4	18.9%	10.7%	7.2%	6.9%

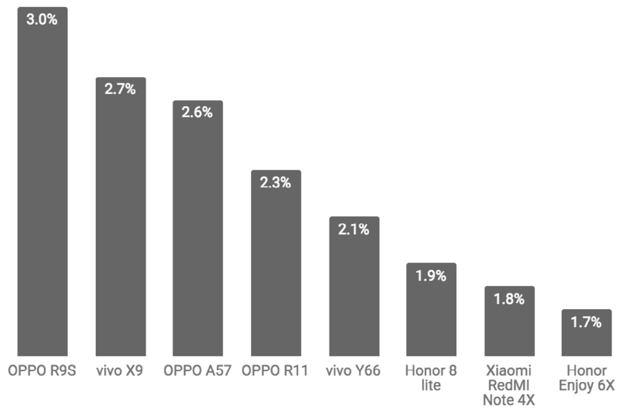


Fig. 2. Chinese models global market share [25].

Table 3
Pricing on top cloud providers and limit of devices–hour included in the price.

Device provider	Category	Billing model	Pricing	DH
AWS device farm	On-demand	$0.17 minutely	$10.2	1
	Reservation	$250 monthly	$250	730
Firebase test lab	On-demand	$5 hourly	$5	1
Xamarin test cloud	Small startup	$99 monthly	$99	30
Kobiton	Unique	$0.1 minutely	$6	1
Perfecto	Monthly basic	$129 monthly	$129	5

4.2. Cost comparison

In the previous section, we analyzed the competitive advantage generated by the potentially vast number of device types accessible via DBB. However, is even more important to evaluate this advantage from an economic viewpoint, as highlighted by RQ2. With this in mind, we have conducted an economic feasibility study on using DBB compared with the alternative solutions.

Our analysis is carried out by assuming that a tester user wants to perform UI testing in a number of different devices. S/he wants to do it for one hour long. The expected pricing on doing this task on the identified cloud platforms, as well as the limit of devices–hour (DH) included in such price, are described in Table 3. For simplicity, we use only the category of billing model more suitable for small testers (except AWS due to its high adoption). On this table we have set the pricing as the minimum expense the user must pay to use 1 DH, although s/he can use more devices without price increases in some cases (as shown in the table).

As can be seen, there is no pattern on billing model and the user has to face a significant difference in pricing. Moreover, even some providers adopting a monthly charging model there is a limit of hours to be used, meaning that actually, the contract is not for a full month (except for AWS[1]). Another problem is that the billing models adopted by these providers make it difficult to perform testing on a small scale. For instance, the use of providers with monthly charging is subject to payment of the full price. This politic makes them attractive alternatives only when the number of DH is close enough to a multiple of the allowed limit.

In face of the highlighted limitations, we have estimated the pricing of using one device through our solution. We use an hourly billing model, which is calculated by the equation:

$$\text{DBB pricing} = \frac{I}{\sum r} + d + p \qquad (1)$$

where I is the platform deployment and execution cost for each hour, r is the number of concurrent requests, d is the device reward cost, and p is revenue gotten by us. To set I, we have estimated the expenses for a 3-year life-cycle using the Total Cost of Ownership (TCO) approach [26], since it can provide reliable decision support [27]. On top of it, we have included the following cost components:

- **Capital Expenses (CapEx)**: New purchases of infrastructure and allocation of new datacenter build-outs.
- **Operating Expenses (OpEx)**: Activities required to install, setup, and keep the platform running.
- **Indirect Business Costs (Ind)**: The potential impact of downtime on productivity plus the time-to-market benefits of increased agility.

The analysis of these cost components were carried out taking five environment alternatives: *(1)* on-premises (ONP), our equipment is at a location that we own; *(2)* colocation (COL), our equipment is at a location that we rent; *(3)* outsource to Amazon cloud (AWS) [28]; *(4)* outsource to Azure cloud (AZU) [29]; and *(5)* outsource to Google cloud platform (GCP) [30]. We set these scenarios targeting analysis of private infrastructure models and outsourcing models using the market-leading cloud vendors, according to Gartner's latest report on cloud Infrastructure as a Service [31]. Table 4 depicts the cost components, and presents the resulting values.

For calculating the capital expenses, we first set the hardware requirements relying on capacity planning recommendations [32]. We estimated for 1000 concurrent requests the need for 4 machines CPU dual core, 2 GB RAM in the application layer and 1 machine CPU 16 cores, 16 GB RAM, 236 GB of storage, for the database layer. For the ONP and COL solutions, we adopt the assumptions used in the AWS TCO calculator [33]. Pricing is based on the estimated costs of equipment from global infrastructure vendors using São Paulo, Brazil as location. Cloud options require no capital expenditures, as no equipment needs to be purchased.

The non-personnel operating expenses for the ONP and COL scenarios were also estimated using the AWS TCO calculator, using a "as a service" model for software licensing. For the cloud scenarios, we use the VM types c3.large and m4.4xlarge, for AWS; and Standard_A2 and Standard_F16, for AZU since they closely

[1] The number of DH is estimated setting a month with 730 h. The same assumption is used in the remaining of the paper.

Table 4
TCO components and resulting values for a 3-year period.

Expense	3-year expense				
	ONP	COL	AWS	AZU	GCP
Capital expenses					
Server Infra.	\$48,623.62	\$48,623.62	–	–	–
Storage	\$1814.13	\$1814.13	–	–	–
Backup	\$38.16	\$38.16	–	–	–
Networking/Security	\$13,127.81	\$28,629.04	–	–	–
Total: 3 years	\$63,603.72	\$79,104.95	\$0	\$0	\$0
Operating expenses					
Personnel	\$403,491	\$403,491	\$100,872.75	\$100,872.75	\$100,872.75
Server maintenance	\$58,113.91	\$58,113.91			
Software licensing	\$10,255.39	\$10,255.39	\$55,143.67	\$46,183.56	\$52,059.91
Space/Power	\$102,553.96	\$203,323.75			
Total: 3 years	\$574,417.28	\$675,184.05	\$156,016.42	\$147,056.31	\$152,932.66
Indirect costs					
Estimated loss due to productivity	\$16,293.6	\$16,293.6	\$162.94	\$814.68	\$162.94
Estimated revenue lost due to delays	\$287,784	\$287,784	\$82,224	\$82,224	\$82,224
Total: 3 years	\$304,077.6	\$304,077.6	\$82,836.94	\$83,038.68	\$82,836.94
TCO	\$942,098.6	\$1,058,366.6	\$238,853.36	\$230,095.29	\$235,769.6

match our needs. For GCP we use customized VM types with the aforementioned configurations. In order not to favor our approach we use the on-demand pricing for these VM resources, which is more expensive than the reservation.

People costs are the largest operational expense. The personnel costs calculated in this model for the ONP and COL scenarios include salaries and benefits of one full-time infrastructure manager. For estimating this cost we use the average salary plus labor overhead on Brazil [34]. One of the primary benefits of cloud scenarios is the lower internal personnel staff required for the platform. For this reason, we adopt that the employee is allocated only 25% of the time in DBB's infrastructure management.

Capital expenses and operating expenses are not the only things to consider when evaluating TCO. A number of indirect costs affect the business when infrastructure experiences downtime or takes additional time to bring the resources up to support a new revenue opportunity. While these costs are difficult to measure, we have included a set of assumptions here to demonstrate the potential impact. Our model assumes 99% uptime for the ONP and COL scenarios (87.6 h annual unplanned downtime). For AWS and GCP, system availability service levels are guaranteed to 99.99% (0.876 h annual unplanned downtime) [35,36]. For AZU, system availability service levels are guaranteed to 99.95% (4.38 h annual unplanned downtime) [37]. For modeling lost labor productivity in this paper, we estimated that 20 employees, with an average salary of \$62 per hour each one, could be impacted by application downtime either directly as IT staff or indirectly as other functions for the platform. We adopt that there is no significant impact on labor productivity from unplanned downtime, setting a weighting factor of 5%.

One of our goals is improving time-to-market with additional infrastructure for business growth or traffic spikes. Our model adopts a \$1M gross annual revenue, and conservatively a revenue implication of 5% annual capacity growth against the time to deploy additional capacity. For the ONP and COL scenarios, we adopt an average time to procure additional infrastructure of 45 days per year and an average time to building and deploying of 25 days per year. For the cloud scenarios, there is no overhead of infrastructure searching, but we set an average time to building and deployment of 20 days per year.

As can be seen, using the Azure cloud is the less expensive alternative. We use this option in the estimative of our platform pricing. The second component in cost estimation (as depicted on Eq. (1)) is the device reward cost. On our analysis, we set the device reward proportionally the timing it is provided on DBB. More precisely, we initially set the reward as the cost of acquiring a new device divided by the number of hours in one year. This way, if the device owner offers it to be used on DBB on a 24/7 basis, after one year s/he is able to buy another device with the same pricing. However, although it seems to be promising enough to persuade device owners, we analyze an even more attractive option, which is to pay three times this reward. In doing it, the device owner can buy another of the same price by providing the original device 8 h per day on DBB. It is a very attractive strategy since the device owner can choose to join DBB only in idle periods, such as during the night. In our analysis, we set the device price as \$800, which is expensive enough to not invalidate our analysis. This way, the device reward cost is calculated using $\frac{\$800}{365\times h}$, where h is the number of hours joined DBB each day: 24 and 8, respectively.

Finally, for the revenue (third component in Eq. (1)), we analyze scenarios with a Return of Investment (ROI) of 20%, 30%, and 40% since it represent strong relation with market-share [38]. Using the three components, we can estimate our platform pricing on each scenario. Fig. 3 describes the DBB cost breakdown.

To compare our approach with the alternative solutions, we vary the number of requested devices per hour from 1 to 6000, which is enough to cover $\tilde{2}$5% of device models currently available in the world. Fig. 4 presents the cost for each case, taking the four scenarios of DBB pricing estimation. The vertical axis is in logarithmic scale for better visibility.

In the four scenarios of DBB pricing estimation, the final cost defeats the ones from other solutions in practically all cases. By using the scenarios with revenue of 20% our solution is always the best choice. On the other hand, when revenue is set as 30% or 40%, the DBB cost is more expensive than AWS by reservation, on some individual cases (3.24% of all cases). However, we advocate that this difference can be tempered by the other benefits of using our solution. As we have discussed, even using high numbers of devices as parameters in the comparison, AWS does not allow such a number of simultaneous devices without previous request. Another disadvantage is the number of different models this provider (as well as the others) offers, which is significantly smaller than most of the given scenarios. To strengthen our arguments, we also have compared the cost reduction on using our solution over the others. Fig. 5 presents the results of comparison. Negative values indicate an increasing on cost.

The use of our platform can provide an average saving of 85.67% on total cost when compared with other solutions. Moreover, for

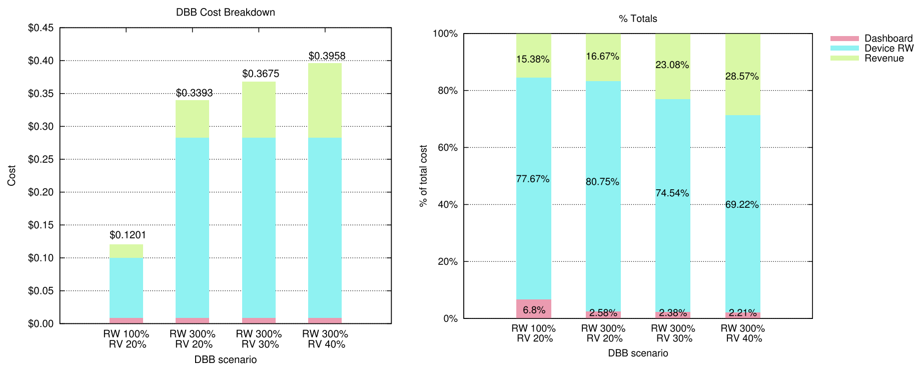


Fig. 3. DBB cost breakdown for one device–hour.

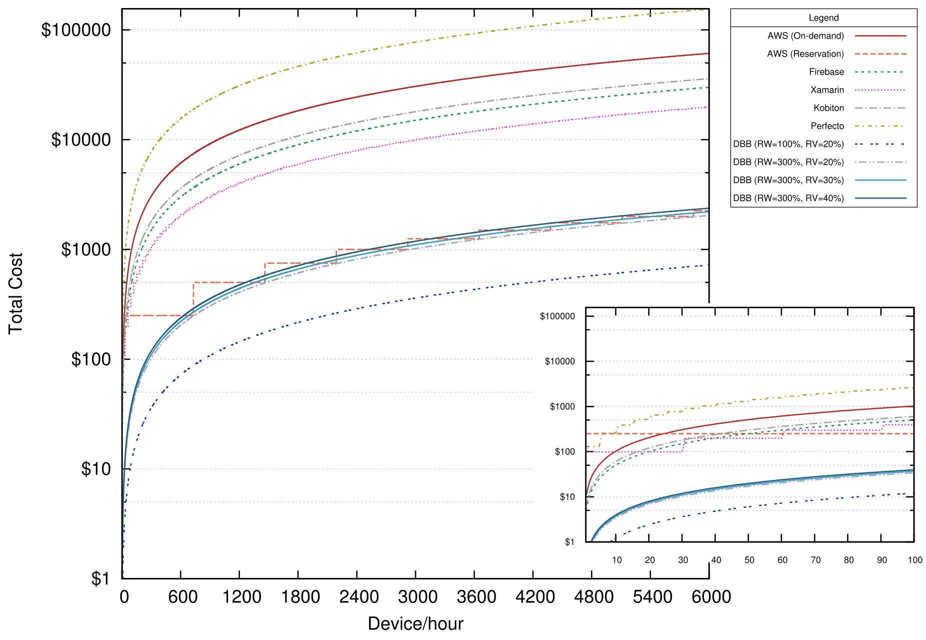


Fig. 4. Cost comparison of DBB with other solutions.

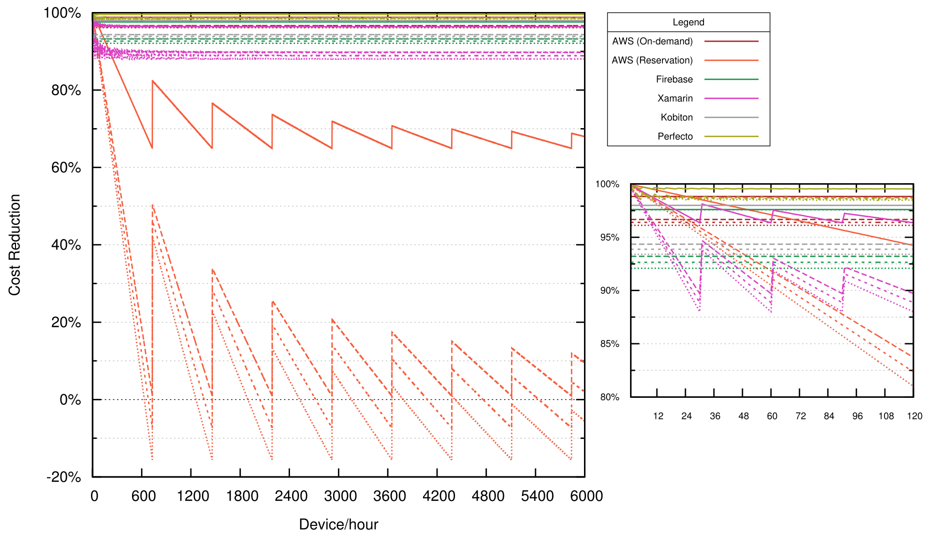


Fig. 5. Cost reduction of using DBB compared with other solutions.

75.67% of analyzed cases, the cost reduction is higher than 90%. To reinforce the insignificance of cases where our platform cost is less attractive, the cost difference is only of 6.41% in average, with a maximum of 15.58%.

By relying on the presented results, we can advocate that our approach is a very cost-effective solution. This affirmation can be strengthen by the potential better results if we change the premises we have taken in on this analysis. For instance, a more attractive pricing can be achieved if we hold a less expensive reward for device provision or if we adopt the reservation model on cloud resource outsourcing. Having said that, we can affirm our solution is a very attractive option not only from the device availability viewpoint but also from an economical view. It is possible because our proposal is based on the Collaborative Economy model, which make feasible the utilization of existing third-part resources to provide cheap and scalable services.

5. Related work

We focus our discussion related to tools which addressing test cost reduction on mobile devices. For that we divided the works into three categories: (A) Tools based on record and replay tests; (B) Tools which provide input data generation; and (C) Tools based on static analysis.

(A) Tools based on record and replay tests

In 2014 Gao et al. published a work [39] which aims to present a new version of the MobileTest [40], which would be provided through a service. Originally MobileTest was developed to assist testers in high-level script generation about app usage and replying it to different devices.

Others similar tools were proposed such as AMT in [41] and Testdroid in [42]. AMT is composed of two components named amt-capture, able to generate a JSON file regards app GUI usage, and amt-replay, able to reply the execution recorded by amt-capture in others Android devices. Testdroid is an online platform for assist user interface testing on a variety of physical Android devices. It allows developers to record test scripts, which along with their application are automatically executed on physical devices in parallel.

Approaches search by state-machine generation based on available properties in widgets used by app. Amalfitano et al. published two works which use this approach. In [43] was presented AndroidRipper, an automated technique that tests Android apps by exploring apps' GUI with aim of exercising the application in a structured manner. AndroidRipper dynamically analyzes the application's GUI with the aim of obtaining sequences of events fireable through the GUI widgets. Based on retrieved sequence, test cases are created and a state machine model of each GUI is built. Later activities can be exercised. In [44] was developed MobiGUITAR, a tool able to generate test cases automatically, basing on a model which performs a ripping that dynamically traverses an app's GUI and creates its state-machine model and record it to be run later.

In [45] was developed a very interesting system for generating and replying Android tests. A tool named DroidMate was presented as an automated GUI execution generator for Android apps. It explores an app, monitoring its device GUI and calls to Android APIs methods for later reproduction it physically using a robot.

(B) Tools which provide input data generation

Another type of problem which exists in any kind of application is about the input to tests. To address this type of problem we cite two important works [46,47]. Machiry et al. [46] presented a system called Dynodroid which generates inputs to Android apps. It performs a monitoring of app events and monitors the reaction of the app upon each event, using it to guide the generation of the next event to the app. Mao et al. [47] presented a tool able to reduce search space of input but without loss coverage of scenarios. The authors used a multi-objective search-based testing model to do it.

(C) Tools based on static analysis

Some work try to point defects prone without the necessity of written test, i.e. through static analysis. Wei et al. published a technique [48] and proposed FicFinder tool, which is able to indicate defects prone based on the static code analyses. The technique proves effective to 27 large-scale open-source Android apps. In [49] was designed a mobile application compatibility test system for Android fragmentation. Based on comparison code analysis result and API pre-testing to detect android fragmentation, this tool is able to statically check the compatibility of an app against desired devices. Thus, through comparing the fragmentation in the code level and the API level, the time and cost of mobile application test could be reduced.

Since Google Espresso was released as the official framework to build and run UI tests to Android apps, some authors had addressed their questions using this framework. In [50] the authors proposed a tool called Barista, able to support testers on Espresso tests generation. This tool generates espresso test code like a test recorder based on app usage, making ease the written of Espresso tests. Taken advantage of Barista we discussed the collaborative economy paradigms usage in the testing process of Android apps and we first introduced DBB in [51]. The proposed platform is based on distributing of Espresso test cases in idle devices spread around the world. This work had as inspiration a paper published in 2015 [52], which posed the possibility of use of idle mobile devices through the Femtocloud system, which is able to share devices computing capability out of the clusters. The authors based on the aggregation of devices in places such as public transportations, coffee places, schools, etc.

Despite the fact of several works had been published which aim to reduce inherent costs involved in mobile software testing, we could not find any work focused in minimize the tests execution costs.

6. Threats of validity

Although DBB contributes to reducing the cost of the UI test execution process, through use of the distributed computing and collaborative economy, there are some aspects that must be taken into account.

It is not possible to guarantee that there is enough developers' demand to maintain the necessary infrastructure, described in Section 4, to the operation of the platform. However, several published reports give us the possibility to believe that there is significant demand for this type of service. Although the report published by Heavy Reading [53] contains optimistic predictions, a newer report presented that this market may achieve $46.90 Billion in 2019 [54]. Others similar estimates, made by Statista [55], predicts that the number of cloud computing subscribers worldwide around 3.6 Billion in 2018.

The proposed model is based on third-party resources, so its popularization is a risk and is subject to people acceptance i.e, device owners must allow their devices to be used by an unknown person/company, who wants to run a test. We intend to validate this issue through extensive experimentation, involving real cases of acceptance by device owners and their proper reward. Convincing a person to rent their device for testing is undoubtedly the most challenging part. To do this, initially a large number of invitation emails would be sent, as soon as the first users register and then download and configure the DBB-agent, fake tests would be sent to these users and run for at least 1 h. DBB also sends a fake report as a confirmation of the completion of the execution, and the device's owner would receive a credit, monitored through the dashboard (DBB-agent and DBB-web).

At the other end, developers could be encouraged to propagate the existence of the platform through discounts held by possible indications.

Some technical issues still need to be mitigated, in order to deliver a controlled environment to testers, who would remunerate the platform. Moreover, despite the fact that DBB is prepared to run instrumented tests written in Espresso, there are several others technologies regards mobile software testing which could compose this solution, Robotium [56], UiAutomator [15], Calabash [14] and Appium [13].

Another critical issue is regarding the device owner's data privacy and other security functionalities. Currently, the DBB platform does not provide any security service. Thus, once submitted for execution, apps and related test cases can access data available in the file system only under the app behavior, so, in this case, the user is relying on the app, and not in DBB. Therefore, we advocate that this not be a problem because we notify the device owner in advance about all permissions required by app under test. The user can accept or deny the test execution according to their convenience. This task is possible by analyzing the manifest file, which is mandatory in any Android app.

In addition to the limitations above, another problem is the compromising of test running due to app notifications triggered by background processes in the remote device. Most of the time this does not affect test execution because app usually displays notifications on the notification bar. However, when an exception is thrown due app notification our first solution is to attach to the test report the caught exception and the corresponding screenshot. Also, billing does not occur in this case.

Even that the DBB is a worldwide platform, there is a risk that, for a substantial number of device models, there is none equipment registered on the platform, compromising the diversity of devices available for testing. However, because DBB is a profitable platform for device owners and an economical alternative for developers, we believe that this minimizes the problem. Our argument on stating it is that each region may have different predominant models, and these models can be included in the platform by marketing actions, achieving device diversity. Moreover, as mentioned in Section 4, one limitation in the top cloud players is devices' availability since currently some of them enforce limits on concurrent usage of devices to avoid monopolization.

7. Conclusion

In this paper, we presented a feasibility analysis about a disruptive platform called DBB, designed for supporting testers/developers run test cases in multiples real devices in an effective and viable business.

One distinctive feature of our platform is that it creates a new market for device-owners to rent their mobile devices while both the cost for executing test cases in real devices decrease and the compatibility tests are more effective. We implemented our approach relying on CE principles. Our preliminary evaluation of DBB shows that it can be useful and effective in practice in both technical and business terms.

We need to address some issues in future works involving CE applied on mobile software testing, such as security aspects about private existing user data on remote devices, and devices used for purposes other than the execution of tests.

Our platform is in the validation stage. Therefore, there are numerous of improving aspects to be performed on DBB, such as increasing the number of experiments, extends the scale of the business validation, and explore deeper the integration of DBB with more testing technologies. Finally, we intend to consider those aspects in our next experiments.

Acknowledgment

The authors would like to thank IFG and CNPq for funding part of this work.

References

[1] Statista Inc, Statista, https://www.statista.com/statistics/274774/forecast-of-mobile-phone-users-worldwide/. Accessed on: 04/27/2018, 2018.
[2] Statista Inc, Statista, https://www.statista.com/topics/876/android/. Accessed on: 04/27/2018, 2018.
[3] S. Inc, Statista, Project Web Page, https://goo.gl/g9EyZD. Accessed on: 02/28/2018, 2018.
[4] O. Inc, Opensignal, Project Web Page, https://opensignal.com/reports/2015/08/android-fragmentation/. Accessed on: 02/28/2018, 2018.
[5] Google Inc, Android screen size, https://developer.android.com/about/dashboards/. Accessed on: 04/27/2018, 2018.
[6] E.N. de Andrade Freitas, C.G. Camilo-Junior, A.M.R. Vincenzi, Scout: a multi-objective method to select components in designing unit testing, in: Software Reliability Engineering (ISSRE), 2016 IEEE 27th International Symposium on, IEEE, 2016, pp. 36–46.
[7] A. Inc, AWS device farm, Project Web Page, https://aws.amazon.com/pt/device-farm/. Accessed on: 02/28/2018, 2018.
[8] X. Inc, Xamarin test cloud, Project Web Page, https://www.xamarin.com/test-cloud. Accessed on: 02/28/2018, 2018.
[9] G. Inc, Firebase test lab, Project Web Page, https://goo.gl/7GK9G9. Accessed on: 02/28/2018, 2018.
[10] P.M. Inc, Kobiton, Project Web Page, https://www.perfectomobile.com. Accessed on: 02/28/2018, 2018.
[11] K. Inc, Kobiton, Project Web Page, https://kobiton.com/. Accessed on: 02/28/2018, 2018.
[12] G. Inc, Espresso, Project Web Page, https://goo.gl/pzjgQS. Accessed on: 02/28/2018, 2014.
[13] J. Foundation, Appium, Project Web Page, http://appium.io/. Accessed on: 02/28/2018, 2012.
[14] X. Inc, Xamarin, Project Web Page, http://calaba.sh/. Accessed on: 02/28/2018, 2015.
[15] G. Inc, UI automator, Project Web Page, https://goo.gl/fw4bJV. Accessed on: 02/28/2018, 2018.
[16] Google Inc, Android junitrunner, https://developer.android.com/training/testing/junit-runner. Accessed on: 04/27/2018, 2018.
[17] Square Inc, Spoon, http://square.github.io/spoon/. Accessed on: 04/27/2018, 2018.
[18] Uber Inc, Uber, https://www.uber.com. Accessed on: 04/27/2018, 2018.
[19] Turo Inc, Turo, https://turo.com. Accessed on: 04/27/2018, 2018.
[20] Airbnb Inc, Airbnb, https://www.airbnb.com. Accessed on: 04/27/2018, 2018.
[21] I. Inc, Idc, Project Web Page, https://www.idc.com/promo/smartphone-market-share/vendor. Accessed on: 02/28/2018, 2018.
[22] H. Inc, Huawei, Project Web Page, http://www.huawei.com/. Accessed on: 02/28/2018, 2018.
[23] X. Inc, Xiaomi, Project Web Page, http://www.mi.com/. Accessed on: 02/28/2018, 2018.
[24] O. Inc, Oppo, Project Web Page, https://www.oppo.com. Accessed on: 02/28/2018, 2018.
[25] C. Inc, Counterpointer, Project Web Page, https://goo.gl/XZsbZF. Accessed on: 02/28/2018..
[26] B. Martens, M. Walterbusch, F. Teuteberg, Costing of cloud computing services: a total cost of ownership approach, in: System Science (HICSS), 2012 45th Hawaii International Conference on, IEEE, 2012, pp. 1563–1572.
[27] L.M. Ellram, S.P. Siferd, Total cost of ownership: a key concept in strategic cost management decisions, J. Bus. Logistics 19 (1) (1998) 55.
[28] Amazon, Amazon web services, https://aws.amazon.com. Accessed on: 04/27/2018, 2018.
[29] Microsoft, Microsoft azure, https://azure.microsoft.com. Accessed on: 04/27/2018, 2018.
[30] Google, Google cloud platform, https://cloud.google.com. Accessed on: 04/27/2018, 2018.
[31] L. Lydia, B. Raj, L. Craig, S. Dennis, Magic quadrant for cloud infrastructure as a service, worldwide, https://www.gartner.com/doc/reprints?id=1-2G2O5FC&ct=150519. Accessed on: 03/28/2018, 2017.
[32] Oracle, Capacity planning and deployment guide, Tech. Rep. 9.2.1, https://docs.oracle.com/cd/E11116_04/otn/pdf/install/E11130_01.pdf. Accessed on: 03/28/2018, 2007.
[33] Amazon, AWS total cost of ownership (tco) calculator, http://awstcocalculator.com/. Accessed on: 04/20/2018, 2018.
[34] Robert Half, Salary guide 2018, Tech. rep., https://www.roberthalf.com.br/guia-salarial. Accessed on: 02/28/2018, 2018.
[35] AWS, Amazon compute service level agreement, https://aws.amazon.com/compute/sla/. Accessed on: 04/27/2018, 2018.
[36] Google Cloud, Google compute engine sla, https://cloud.google.com/compute/sla. Accessed on: 04/27/2018, 2018.
[37] Microsoft Azure, SLA for virtual machines, https://azure.microsoft.com/en-us/support/legal/sla/virtual-machines/v1_8/. Accessed on: 04/27/2018, 2018.
[38] R.D. Buzzell, B.T. Gale, R.G. Sultan, Market share-a key to profitability, Harvard Bus. Rev. 53 (1) (1975) 97–106.

[39] J. Gao, W.-T. Tsai, R. Paul, X. Bai, T. Uehara, Mobile testing-as-a-service (mtaas)–infrastructures, issues, solutions and needs, in: High-Assurance Systems Engineering (HASE), 2014 IEEE 15th International Symposium on, IEEE, 2014, pp. 158–167.

[40] J. Bo, L. Xiang, G. Xiaopeng, Mobiletest: a tool supporting automatic black box test for software on smart mobile devices, in: Proceedings of the Second International Workshop on Automation of Software Test, IEEE Computer Society, 2007, p. 8.

[41] E.N. Freitas, K.A.C. Faria, C.G. Camilo-Junior, A.M.R. Vincenzi, AMT: an android mirror tool for instant feedback across platform, in: Congresso Brasileiro de Software (CBSoft), 2016 7th Congresso Brasileiro de Software on, SBC, 2016, pp. 429–440.

[42] J. Kaasila, D. Ferreira, V. Kostakos, T. Ojala, Testdroid: automated remote ui testing on android, in: Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia, ACM, 2012, p. 28.

[43] D. Amalfitano, A.R. Fasolino, P. Tramontana, S. De Carmine, A.M. Memon, Using gui ripping for automated testing of android applications, in: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ACM, 2012, pp. 258–261.

[44] D. Amalfitano, A.R. Fasolino, P. Tramontana, B.D. Ta, A.M. Memon, MobiGUITAR: automated model-based testing of mobile apps, IEEE Softw. 32 (5) (2015) 53–59.

[45] K. Jamrozik, A. Zeller, Droidmate: a robust and extensible test generator for android, in: Mobile Software Engineering and Systems (MOBILESoft), 2016 IEEE/ACM International Conference on, IEEE, 2016, pp. 293–294.

[46] A. Machiry, R. Tahiliani, M. Naik, Dynodroid: an input generation system for android apps, in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ACM, 2013, pp. 224–234.

[47] K. Mao, M. Harman, Y. Jia, Sapienz: multi-objective automated testing for android applications, in: Proceedings of the 25th International Symposium on Software Testing and Analysis, ACM, 2016, pp. 94–105.

[48] L. Wei, Y. Liu, S.-C. Cheung, Taming android fragmentation: characterizing and detecting compatibility issues for android apps, in: Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on, IEEE, 2016, pp. 226–237.

[49] H.K. Ham, Y.B. Park, Mobile application compatibility test system design for android fragmentation, in: International Conference on Advanced Software Engineering and Its Applications, Springer, 2011, pp. 314–320.

[50] M. Fazzini, E.N.d.A. Freitas, S.R. Choudhary, A. Orso, Barista: a technique for recording, encoding, and running platform independent android tests, in: Proceedings of the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST 2017), IEEE, 2017, pp. 01–11.

[51] K.A.C. Faria, E.N.d.A. Freitas, A.M.R. Vincenzi, Collaborative economy for testing cost reduction on android ecosystem, in: Proceedings of the 8th ACM SIGSOFT International Workshop on Automated Software Testing, ACM, 2017, pp. 11–18.

[52] K. Habak, M. Ammar, K.A. Harras, E. Zegura, Femto clouds: leveraging mobile devices to provide cloud service at the edge, in: Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on, IEEE, 2015, pp. 9–16.

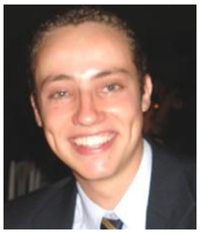
[53] H. Reading, The mobile cloud market outlook to 2017, 2013.

[54] MarketsandMarkets, Mobile cloud market by application, https://goo.gl/AUVMHA. Accessed on: 04/27/2018, 2018.

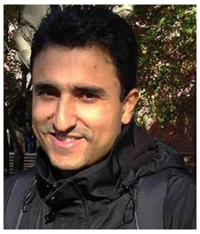
[55] Statista, Number of consumer cloud-based service users worldwide in 2013 and 2018, https://goo.gl/TrUv16. Accessed on: 04/27/2018, 2018.

[56] Robotium, Robotium, Project Web Page, https://github.com/RobotiumTech/robotium/wiki. Accessed on: 02/28/2018, 2018.

Kenyo Abadio Crosara Faria concluded his bachelor in Computer Science at Universidade Católica de Goiás - UCG (2003) and his master (2006) in Computer Engineering at School of Electrical and Computer Engineering of Universidade Federal de Goiás. Currently is PhD candidate in Computer Science at Instituto de Informática of Universidade Federal de Goiás. Since 2008 he is professor at Instituto Federal de Goiás. His current research interests include software testing and software architecture.



Raphael de Aquino Gomes is graduated (2006), master (2009), and PhD (2017) in Computer Science from the Universidade Federal de Goiás with internship period at Institut National de Recherche en Informatique et en Automatique (INRIA), France. Currently, he is professor at Instituto Federal de Educação, Ciência e Tecnologia de Goiás (IFG). His current research interests include cloud computing, service-oriented solutions, Internet of Things, and performance optimization.



Eduardo Noronha de Andrade Freitas received his degree in Computer Science in 2000; his specialization in Software Quality in 2003, his master's degree in Electrical and Computer Engineering in 2006, and his Ph.D. in Computer Science from the Universidade Federal de Goiás in 2016. From 2013 to 2015, during his Ph.D. studies, he collaborated in the Checkdroid startup (www.checkdroid.com) at the Georgia Institute of Technology in Atlanta, GA. He served as Information Technology Manager at the Secretariat of Public Security of the State of Goiás from 2006 to 2010, participating in the development and implementation of strategic processes. He also developed numerous strategic planning projects and data analysis in the public and private sectors in diverse areas: health, education, security, sports, politics, and religion. Since 2010, he has served as a professor at the Instituto Federal de Goiás (IFG). He has extensive experience in computer science with a focus on computer systems, principally in the following areas: systems development, software engineering with an emphasis on search-based software engineering, Android testing, multiagent systems, strategic management of technology, and computational intelligence. He can be reached at eduardonaf@gmail.com.

Auri Marcelo Rizzo Vincenzi concluded his bachelor in Computer Science at Universidade Estadual de Londrina – UEL (1995) and his master (1998) and doctor (2004) degree in Computer Science and Computational Mathematics at Universidade de São Paulo – ICMC/USP. During his doctor he visited the University of Texas at Dallas UT-Dallas – USA. Since 2008 he was professor at Universidade Federal de Goiás – UFG and member of the Sociedade Brasileira de Computação (SBC), Association for Computing Machinery (ACM), and the Institute of Electrical and Electronics Engineers (IEEE). Since 2015, he is professor at Universidade Federal de São Carlos, and his current research interests include software testing, static and dynamic analysis of open-source applications, object-oriented program analysis, and software evolution.