Proactive Software Engineering Approach to Ensure Rapid Software Development and Scalable Production with Limited Resources

A. B. Farid
Information Systems Department
Faculty of Computers & Information, Helwan Univ.
Cairo - Egypt

***Abstract*—Nowadays, the need for building scalable systems in narrow time window is needed. While the efforts and accuracy usually required for building high scale systems is not simple, the agile nature of system requirements spawn a need for enhancing some software engineering practices. These practices should be integrated together in order to help software (SW) development teams to build, and test scalable systems rapidly with a high confidence level in their scalability.**

This research explains the proposed Proactive Approach, which presents a set of software engineering practices that could help in producing scalable system while minimizing the wasted time within the production cycle. This set of practices have been validated, verified and tested through building 46 releases of one of the most important, mission critical and scalable systems. Applying these practices succeeded to enhance average response time of web pages by %1921.5, test code churn by more than % 5000, time to release by % 300, and succeeded to produce a system that could stand against 95375 users with % 99.921 scalability ratio.

Keywords—***Software Engineering; Load Testing; Test Analysis; ISO 29119; Continuous Integration; Static Analysis; Stress Analysis; Application Scalability; Building High Scalability System; Build Verification Test; Software Configuration Management; Version Control; Source Control***

I. Introduction

Working in a project that targets building scalable system with a limited man-power is a common request those days. Using Agile practices in an organized way may lead to eliminating a notable portion of wasted time [1], [2], [3]. Many previous research work have spoken about how to develop a high quality scalable system. Unfortunately, all previous work efforts assumed the availability of enough resources to accomplish the mission conveniently [4],[5],[6],[7],[8]. Going through a software (SW) development project that has limited amount of different types of resources including; budgets, time, and man-power resources, needs to implement an enhanced approach that presents a set validated set of enhanced engineering practices. This set of practices should assure, not only, building the system with the limited set of resources, but also it should help in assuring the scalability of the system when this is needed. This paper is targeting bridging this gap through designing a proposed approach of engineering practices that target achieving three objectives:

1) Achieving the development tasks rapidly without wasting time in fixing code through:

a) Minimizing the Inability of development teams to comply with the code writing best practices standards (e.g. writing secure code best practices, writing a reliable multi-threaded code, code naming conventions,...etc.)

b) Designing the version control change- sets taxonomy in a way that helps in extending the code with newer releases, batches, and fixes in a way that minimizes code churn, and code rewriting.

c) Establishing a proactive Quality controls that make sure that the code units are performance-friendly units.

2) Automating many SW engineering tasks that may need some technical staff. This could help in minimize the dependency on man-power thus, minimizing the amount of needed man-power.

3) Assuring acceptable scalability levels of systems through passing reliable set of load tests.

In order to verify the proposed practices, a 4.5 year research study has been conducted on one of the highly scalable systems that has load of 95,375 simultaneous users with a 55 million potential users. At the early stages of this study, Microsoft has published a case study about the engineering practices that have been developed overtime, and marked those practices as successful [9]. It is important to mention that this research results could help under the following assumptions:

1) Team size doesn't exceed 9 members including all roles

2) Time to release is limited relative to the amount of required work items.

3) Computation resources are limited.

4) The required system should be scalable to huge amount of users.

This paper will directly goes through the proposed SW engineering approach's practices. First, it will introduce the recommended design of the version control workspaces for the code stored during the development phase. Second, the paper will show how these practices could assure proactively the code quality during the development and before moving to the quality control. Third, the paper will explain the recommended steps towards conducting a reliable load testing for assuring system scalability. This part will talk also about analysing the load test result and applying corrections. Then, the paper will explain more details about the study that has been conducted to verify these practices and the results that has been shown out of it. The paper ends with showing the future work and conclusion.

II. Lifecycle Configuration Management's Controls

The name of the game for building a reliable solution rapidly is to design the development process in a way that helps in accelerating the development lifecycle while maintaining high quality levels for the written code. The proposed set of practices assumes a group of pre-set controls. These controls are:

- Version Control (VC) Server has to be used[1].
- Before Uploading code to the Version Control Server the code has to automatically pass through static analysis code reviews.
- Before being uploaded to the server some unit tests has to be passed by the uploaded code.
- To assure high level of continuous integration, the version control has to test the written code before accepting it.

III. Configuring the Version Control Branches

The VC server such as Microsoft Team Foundation Server (TFS), or IBM Rationale [10] has to be configured in a way that helps in automating the code uploading process that is usually called 'Checkout'. It is important to organize source codebase in a way that simplify any development, and maintenance of the application's source code.[11] Fig. 1 and Fig. 2 depicts how the source code could be organized in a way that simplifies applying fixes, and releases.

The *Main* codebase workspace branch could be created over version control as baseless branch from the older version [12]. Fig.2 depicts the taxonomy of branches for version 2. This simple model provides an easy and consistent VC taxonomy for utilizing Forward Integration (FI) and Reverse Integration (RI) models between the *Main* and *Dev* branches, yet allows for increasing complexity with the addition of future development branches when needed. A development branch has been taken from the Main branch of code to drop all required components and write new replacing ones.

Multiple development areas are supported by creating additional development branches from *Main*. These are peers to each other and children of *Main*.

Any additional releases are supported by creating additional release branches for each product release. Each release branch is a child of *Main* and a peer to each other (e.g. release2.0 branch is peer to release3.0 and both are children of *Main*). Once the release branch is created *Main* and the Dev branches can start taking changes approved for the next product release. After the first Reverse Integration of the code (RI1) the first final release build takes place and this generates Version 1.0. After this point a new branch is being created which is the *Release* child branch. While running the release in production, the cycle of load testing begins (for more information concerning what are the proposed steps for conducting a reliable load test , please check section V). According to the issues that will be discovered during load testing, some fixes are expected to be applied over the release branch that has updated the *Main* branch through Reverse Integration RI2 that in turn updates the *D*ev branch through the Forward Integration FI2.

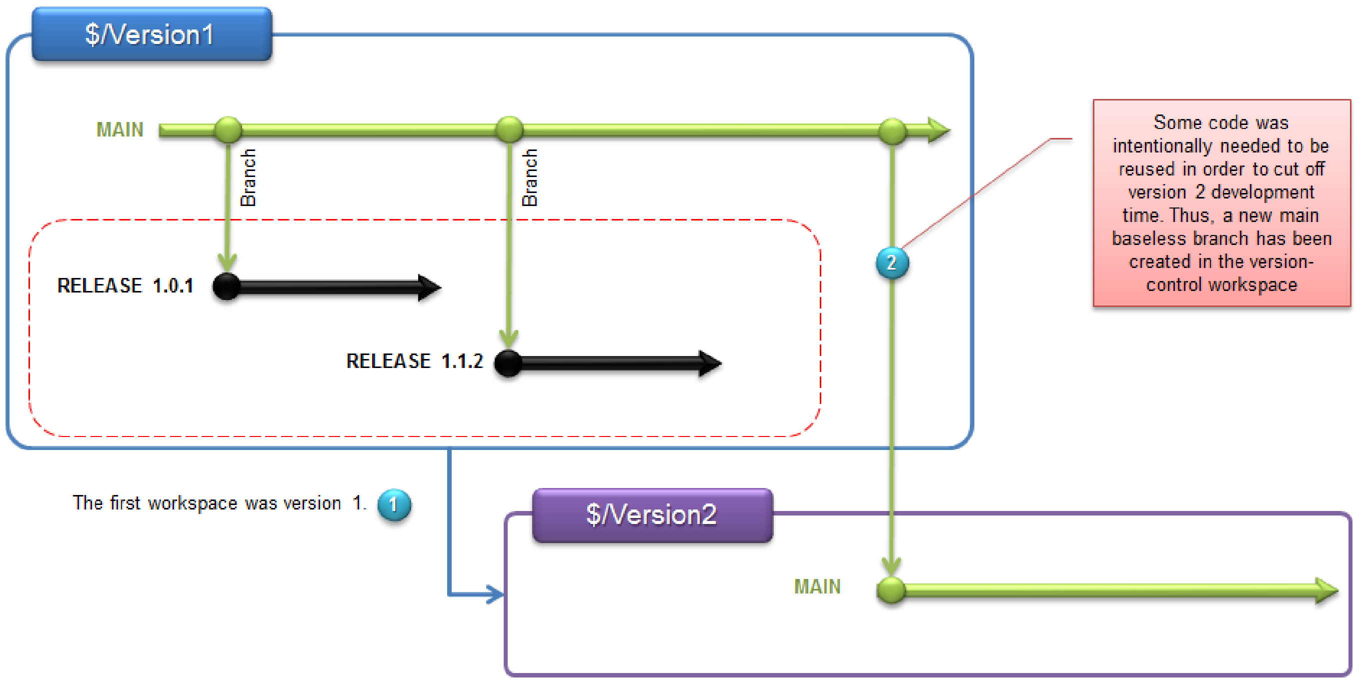

$/Version1
MAIN
Branch
Branch
RELEASE 1.0.1
RELEASE 1.1.2
2
Some code was intentionally needed to be reused in order to cut off version 2 development time. Thus, a new main baseless branch has been created in the version-control workspace
The first workspace was version 1.
1
$/Version2
MAIN

Fig. 1. The Design of the Version Control Workspace based on a previous available version

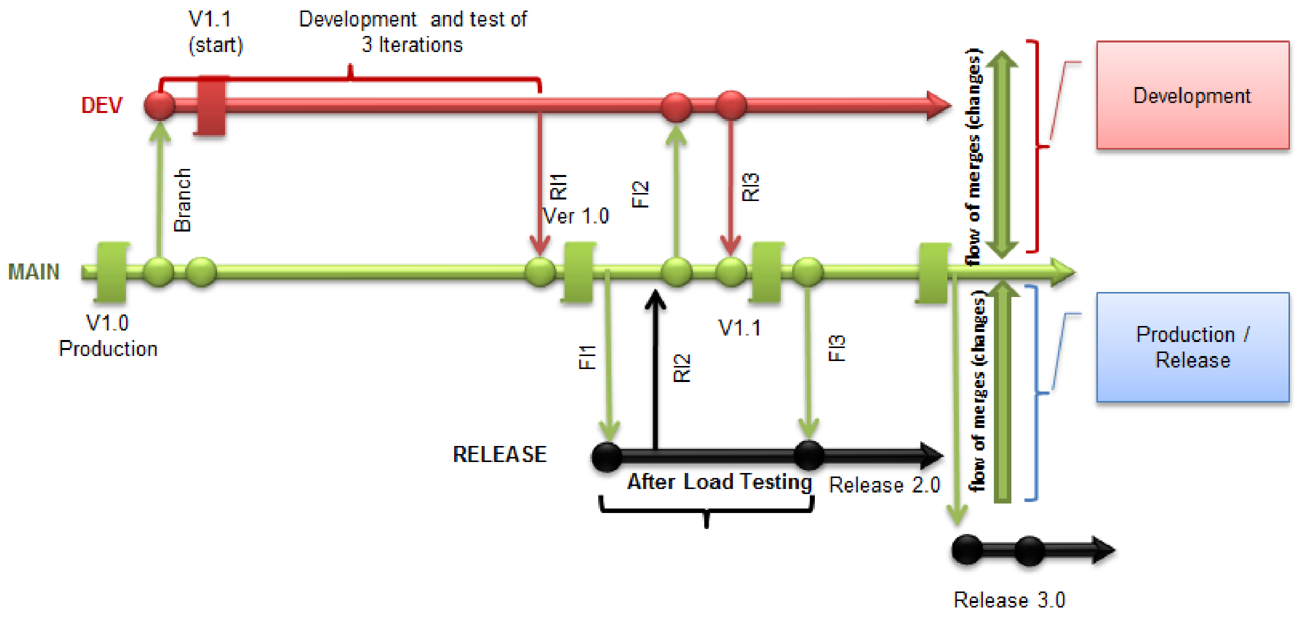


Fig. 2. A Proposed Branching Taxonomy of Version Control Workspace

After fixing some minor final bugs in the *D*ev branch, the main branch should be updated back through Reverse Integration RI3. This version of *M*ain is built to generate version 1.1 that updates *R*elease 2 branch through the Forward Integration 3 (FI3). A new branch could be created at this point that is release 3. Release 3 could be managed as a major final release. This release is assured to be scalable enough and complying with the code writing conventions and standards!

IV. Proactive Quality Control Check-Up

Configuring Version Control (VC) code uploads (check-ins) in a way that helps in controlling the quality in a proactive mood is, of a great importance for the team. This could save the time and staff that are needed to review the code in a manual mood with humanitarian effort. This applied specially when the available time is too tight, and no room is available for discovering the bugs even after regular nightly builds are accompanied with Build Verification Tests (BVTs).

When a developer checks-in a new code that breaks the build, the result could be a significant hassle for the teams. The cost to larger teams can be even more expensive when measured by lost productivity and schedule delays. To guard the code base against these problems, a fake build Server could be configured. This fake build server could be an auxiliary build server. Before checking the code into the VC server, the VC server sends first the code to the fake build server where specific build definition along with its Build Verification Tests (BVTs) are being applied. The build definition doesn't really build the code. Instead, it helps in determining whether the source code that is required to be checked-in, will most probably pass the BVTs, and the build when they take place, or not? This minimizes the number of build failures thus, minimizing the lost time, while preserving the flexibility of Continuous Delivery (CD) [13]. To achieve this, the VC server (e.g. TFS) could be configured through writing a program that acts as a coded Check-In rule. Each time a piece of source code is requested to be checked-in to the VC server, this Check-in rule triggers that verification build definition in order to run over the fake build server before checking-In the source code. The build definition in turn, triggers the associated Build Verifications Tests (BVTs) that has to be applied automatically over the source code. Part of these tests and checks was about checking the architecture rule and constraints. These collections of constraints have put some mandates on the architecture of the system. For instance, a rule that prohibits any developer from checking in a code that directly access the System's database (DB) from presentation tier, or application tier components. If the code passes this set of BVTs and checks; the fake build server notifies the VC server that the source code is acceptable to be checked-in otherwise, it notifies the VC server to reject checking-in the source code. Fig. 3 explains these steps in a graphical way.

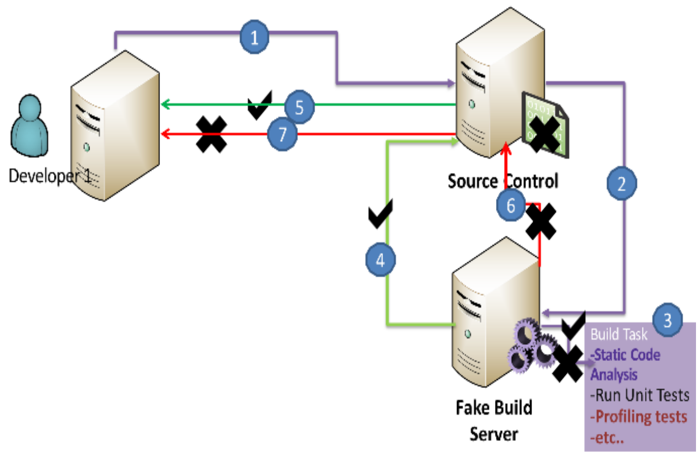


Fig. 3. Proactive quality control check-up steps

V. Load Testing Scalability of the System

Building a scalable system that is intended to serve for instance millions of users is not an easy task. Unless the system is scalable enough the system will be useless and all investments that have been spent into building it will vanish. When the system has a strategic importance (e.g. mission critical systems, financial institutions systems, or national-level), the financial loss is nothing compared to the loss it might be caused to the national security. That is why, it is crucial to assure that the system of this effort will scale to the required amount of users [14]. That is why it is important to have a clear load test plan for assuring the scalability of the system throughout the application lifecycle [15],[16]. The plan is highly recommended to be based on ISO 29119 Part 2 that describes the test process, 29119 part 3 that describes test documentation, and 29119 part 4 that describes the test techniques [17], [18], [19], [20]. This roadmap should include the following ordered tasks:

- Envisioning, and planning for the system scalability.
- Conducting an assurance Strategy.
- Planning for a load test.
- Conducting the load test.
- Analysing results
- Taking corrective actions.
- Conducting Isolation retesting.
- Reviewing lessons learned.

A. *Envisioning, and planning for the system scalability*

During envisioning, it is important to understand the vision, and the required scale of the system. During the planning of the system, it is important to calculate the maximum expected number of users. Sometimes, it could be a fixed expected number (e.g. the total number of citizens of a country for national systems, or total number of employees for enterprise systems,..etc.). This is the planned number that the team has to put in consideration while planning for system's capacity. Understanding the potential load of a given system is crucial towards understanding the required appropriate system architecture to be designed.

B. *Conducting an Assurance Strategy*

At first place, it is important for the team to assure that the code performance on a single user mood will be acceptable enough, and no design or code writing performance unti-patterns will be made. Afterwards, it is important to make sure that the whole application will be scalable enough against the expected users load, and no architectural mistakes will be committed. This should be done through configuring the proactive check-ups during checking the code into the VC server (for more information about proactive check-ups, please check The Proactive Quality Control Check-ups). These check-ups assures in certain way that the written code complies with the code writing performance friendly best practices that are requested by the team's leadership. It is phenomenal fact that the overall system scalability couldn't be assured for huge amount of users unless performance suitability level could be assured initially in a single user level. That is why part of the proactive pre-Check during conducting tests should be; the Performance tests. If the page or unit of code will not pass the predesigned QC proactive checks through achieving certain performance threshold levels, the VC server refuses to accept checking in this source code. This assures the quality of the code during the code writing phase, and before transferring it to any Quality Control (QC) team.

C. *Planning for a Load Test*

After building the first release of the system, it is important to conduct a comprehensive load test over the system. Conducting this load test is one of the most crucial tasks during the development lifecycle in order to check the system resources' behaviour against the expected users load. This should include the following steps:

1) Preparing the Testing Environment: In order to conduct a realistic load test, it is important to have a test environment that typically imitates the actual production environment from the resources capacity point of view (i.e. network bandwidth, storage, memory, Input-Output (IO) speed, and processing power) capacity. That is why deploying this environment over physical machines is highly recommended and preferred than using a collection for virtual machines.

2) Defining the duration of the load test: Defining a suitable duration period of the load test is crucial towards receiving accurate results. According to this study, the suitable load test duration should be defined according to the amount of time that you expect to have a peak load in. For instance, if it is expected to have a three day special offer on an e-commerce system, it expected to have a peak load during these days. In this case the suitable load test duration length should be 72 hours. Choosing a shorter period of the load test may result in experiencing a crash of the system when being put in production for a longer period. According to the experiences that have been gained through this study, part of the load is coming not only from the number of users, but also from the amount of this number that is pressing on the system's resource for certain amount of time.

3) Calculating the actual load size:

Calculating the expected users load means calculating the simultaneous users that use the system at any point of time. Three factors always affect calculating the expected user load; the total number of users, duration of load, and the major usage scenario that mostly unveils the peak time of load. If the expected users have specific maximum number then, you have to have an assumption on how much time it is expected to find all users visiting the system (e.g. one day, one week, one months,…etc.). This means it is needed to calculate the peak load based on that. Additionally, you need to calculate the average time that each user will consume while working on the system based on the main user story. For instance; let us assume that the total number of users is 54 million users over 48 hours with an average usage time of five minutes per user. Based on that, it is possible to calculate the maximum expected simultaneous users as follows:

Total Number of targeted users (TNV) = 54 Million.

Peak Duration (*PDM*) in Min. =48 hrs. * 60

Average Time/Usage Scenario (*AVU*) = 5Min.

Number of Simultaneous Users Load (*SUL*).

$$SUL = AVU(\frac{TVN}{PDM}) \qquad (1)$$

SUL = 5 (54,000,000/2880)

SUL = 93750 Simultaneous Users Load.

After adding a safe Margin of 5%

The Final User Load (FSUL) = 98500 Users

4) Setting an architecture for the load test Architecture

Fig. 4 depicts a the proposed load test servers' architecture. Based on this architecture, there is only one Test Controller (TC) server that executes and controls the test, and finally sends back to the client machine the load test counters' values. In another hand this TC server manages and calls one or more Test Agent (TA) servers that imitate the actual users load.

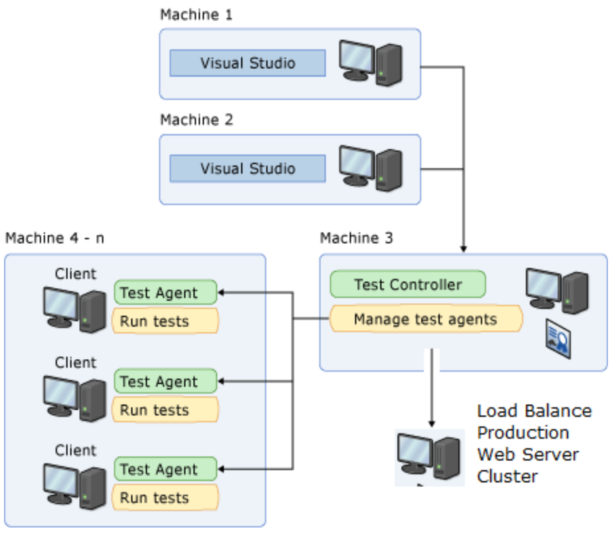


Fig. 4. Load Test Architecture

Each Test agent is responsible of generating number of certain number of users. The main challenge is to define the suitable number of agents that can generate the desired number of users. One good technique to define this number is the Goal Based Load Test. This gives the number of generated users when the TA resources reach certain threshold level. Assuming that the amount of users load that could be smoothly generated by the TA is η. Then the number of required TAs NTA to conducted a load test that could give reliable could be calculated as follows:

$$\text{NTA} = \frac{SUL}{\eta} \qquad (2)$$

5) Risk Mitigation is an important part of any test. Based on the previous assumption of resources limitations, the whole test load test process is in a risk of experiencing any hardware failure. If hardware crashed, there will not be enough resources available including money, and time to replace these resources. According to ISO 29119, it is important to have a risk mitigation plan. This mitigation plan should be based on using the cloud infrastructure instead of using the on premise deployment. This may provide a more appropriate economical, and fast to gain solution especially when the amount of servers needed is not attainable due to resources limitation. Working over the cloud is another effort that will be extended in the near future.

D. Conducting a Load Test

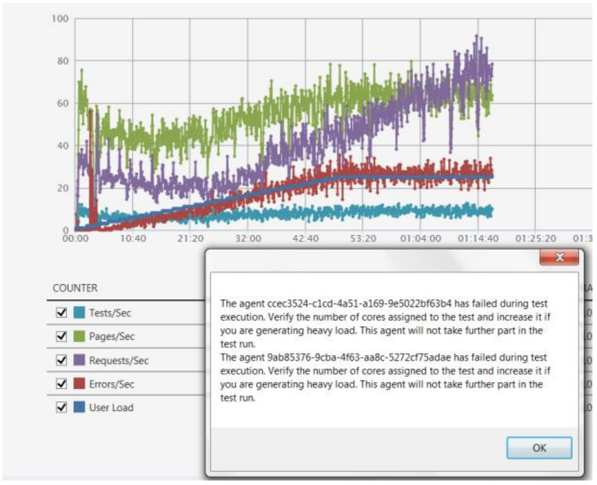
In order to have a good analysis to what is going on; the load test should be conducted through a gradual step by step process. According to the study, Scalable load tests should go gradual since they need to run for long times. Since it needs a serious amount of resource to be available, it is not practical to start a 48 hours intensive load test all of a sudden. This should take place as follows:

Step 0: Running a constant load of the designated potential users load (previously referred to as SUL) for 10 minutes.

It is important to begin with a short period of test to make sure that the deployed architecture could stand against the potential load or not. Sometimes, the resources shows inability to stand against the load at all, and the load crashes at the early beginning or malfunctions so, running the test for this short amount of time could be a good start as just try.

Step 1 Run the test for an hour.

Running the test for one full hour gives more confidence in the available resources and their ability to cope with the load test itself. While the remaining steps are directed towards testing the system itself, steps 0 and 1 are directed towards testing the load test itself. Fig. 5 shows an example of the intense of errors that could arise when the resources are not enough. The message alert says that the agent has failed.



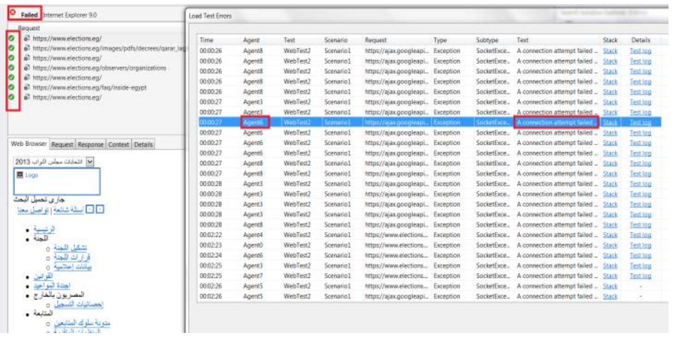
The agent ccec3524-c1cd-4a51-a169-9e5022bf63b4 has failed during test execution. Verify the number of cores assigned to the test and increase it if you are generating heavy load. This agent will not take further part in the test run.
The agent 9ab85376-9cba-4f63-aa8c-5272cf75adae has failed during test execution. Verify the number of cores assigned to the test and increase it if you are generating heavy load. This agent will not take further part in the test run.

Fig. 5. shows an excerpt of erros after 1 hour

Step 3: Run same load test ¼ the total load test duration with full load size:

For instance, if the system is expected to work with the maximum load for 48 hours, accomplish this load test step for 12 hours. This gradual load test duration is important to have when there is a large scale of users that is expected for a constant relatively long period. In relatively long tests that exceeds 24 hours, it is important to accomplish the load test gradually so, if your test spawns number of errors that exceeds the predefined threshold you don't need to wait till the end of the long load test. This saves resources, time, and increase the speed of reaching a more stable system.

Fig. 6 shows one of these experienced cases during the study, more than 60 errors have been discovered during the first 2 hours. This is enough to stop the first run to analyse and trace logs. When such experience happens, it is highly recommended to trace the log files to check the root cause of the resulting errors. One of the most common root causes is the capacity of the Input-Output storage capacity. Due to the huge load, load agents fail to continue applying the load test due to problems with Input-Output storage unit that writes the test log. Whatever the problem is, it should be solved and the test should be restarted smoothly gain with no issues. Running the test successfully for ¼ of the total test length period (12 hours if more than 24 hours) could be the first positive sign that the architecture of the system is scalable enough to stand against such huge load for certain amount of time. In another hand, it couldn't assure that the system will perform well for the full required duration.


Failed Internet Explorer 9.0

Request
- https://www.elections.eg/
- https://www.elections.eg/images/pdfs/decrees/qarar_la
- https://www.elections.eg/
- https://www.elections.eg/observers/organizations
- https://www.elections.eg/
- https://www.elections.eg/faq/inside-egypt
- https://www.elections.eg/

Web Browser | Request | Response | Context | Details

انتخابات مجلس النواب 2013

Logo

جاري تحميل البحث

أسئلة شائعة | تواصل معنا

- الرئيسية
- اللجنة
  - تشكيل اللجنة
  - قرارات اللجنة
  - بيانات إعلامية
- القوانين
- اجندة المواعيد
- المصريون بالخارج
  - إحصائيات التسجيل
- المتابعة
  - مدونة سلوك المتابعين

Load Test Errors

Time	Agent	Test	Scenario	Request	Type	Subtype	Text	Stack	Details
00:00:26	Agent8	WebTest2	Scenario1	https://ajax.googleapi...	Exception	SocketExce...	A connection attempt failed ...	Stack	Test log
00:00:26	Agent8	WebTest2	Scenario1	https://ajax.googleapi...	Exception	SocketExce...	A connection attempt failed ...	Stack	Test log
00:00:26	Agent8	WebTest2	Scenario1	https://ajax.googleapi...	Exception	SocketExce...	A connection attempt failed ...	Stack	Test log
00:00:26	Agent8	WebTest2	Scenario1	https://ajax.googleapi...	Exception	SocketExce...	A connection attempt failed ...	Stack	Test log
00:00:26	Agent8	WebTest2	Scenario1	https://ajax.googleapi...	Exception	SocketExce...	A connection attempt failed ...	Stack	Test log
00:00:27	Agent3	WebTest2	Scenario1	https://ajax.googleapi...	Exception	SocketExce...	A connection attempt failed ...	Stack	Test log
00:00:27	Agent3	WebTest2	Scenario1	https://ajax.googleapi...	Exception	SocketExce...	A connection attempt failed ...	Stack	Test log
00:00:27	Agent6	WebTest2	Scenario1	https://ajax.googleapi...	Exception	SocketExce...	A connection attempt failed ...	Stack	Test log
00:00:27	Agent6	WebTest2	Scenario1	https://ajax.googleapi...	Exception	SocketExce...	A connection attempt failed ...	Stack	Test log
00:00:27	Agent6	WebTest2	Scenario1	https://ajax.googleapi...	Exception	SocketExce...	A connection attempt failed ...	Stack	Test log
00:00:27	Agent6	WebTest2	Scenario1	https://ajax.googleapi...	Exception	SocketExce...	A connection attempt failed ...	Stack	Test log
00:00:27	Agent8	WebTest2	Scenario1	https://ajax.googleapi...	Exception	SocketExce...	A connection attempt failed ...	Stack	Test log
00:00:27	Agent8	WebTest2	Scenario1	https://ajax.googleapi...	Exception	SocketExce...	A connection attempt failed ...	Stack	Test log
00:00:28	Agent3	WebTest2	Scenario1	https://ajax.googleapi...	Exception	SocketExce...	A connection attempt failed ...	Stack	Test log
00:00:28	Agent3	WebTest2	Scenario1	https://ajax.googleapi...	Exception	SocketExce...	A connection attempt failed ...	Stack	Test log
00:00:28	Agent3	WebTest2	Scenario1	https://ajax.googleapi...	Exception	SocketExce...	A connection attempt failed ...	Stack	Test log
00:00:28	Agent3	WebTest2	Scenario1	https://ajax.googleapi...	Exception	SocketExce...	A connection attempt failed ...	Stack	Test log
00:00:28	Agent8	WebTest2	Scenario1	https://ajax.googleapi...	Exception	SocketExce...	A connection attempt failed ...	Stack	Test log
00:02:22	Agent4	WebTest2	Scenario1	https://www.elections...	Exception	SocketExce...	A connection attempt failed ...	Stack	Test log
00:02:23	Agent0	WebTest2	Scenario1	https://www.elections...	Exception	SocketExce...	A connection attempt failed ...	Stack	Test log
00:02:24	Agent6	WebTest2	Scenario1	https://www.elections...	Exception	SocketExce...	A connection attempt failed ...	Stack	Test log
00:02:25	Agent3	WebTest2	Scenario1	https://www.elections...	Exception	SocketExce...	A connection attempt failed ...	Stack	Test log
00:02:25	Agent7	WebTest2	Scenario1	https://www.elections...	Exception	SocketExce...	A connection attempt failed ...	Stack	Test log
00:02:26	Agent5	WebTest2	Scenario1	https://www.elections...	Exception	SocketExce...	A connection attempt failed ...	Stack	-
00:02:26	Agent5	WebTest2	Scenario1	https://ajax.googleapi...	Exception	SocketExce...	A connection attempt failed ...	Stack	-

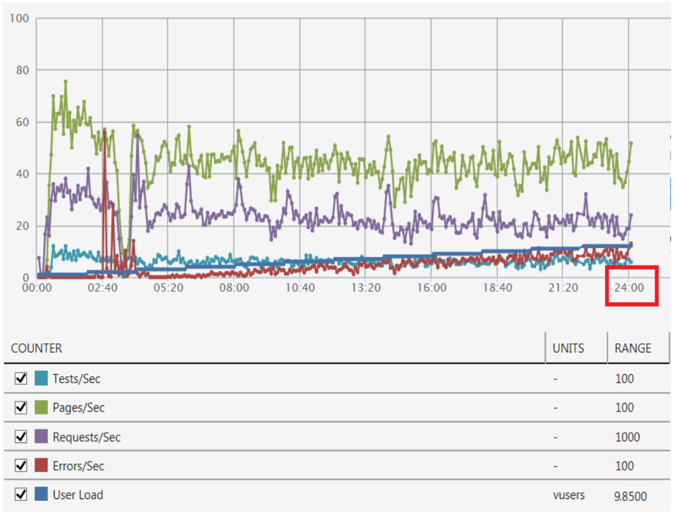
Fig. 6. Reviewing load Agents' status and log files

Step 4: Run Load Test of a half duration length.

Running a test for 1/2 duration gives more confidence to the test analyst through showing a clear image about the resources required to accomplish the actual full duration test. Usually, running the test with this period will give some lessons that could be helpful towards conducting the full-length test. Fig. 7 shows an excerpt of a sample run of 24 hours (1/2 duration length in the study) that shows a smooth load run. The figure shows that the number of errors has been dropped significantly after 24 hours (less than 10 during 24 hours).

Step 5: Run the predesigned full test duration period.

This is the actual planned load test. After solving the load test issues gradually over different durations, through the previously conducted load test steps, the test has to pass the test duration successfully. This doesn't always mean running the test with no errors; it means that the number of errors shouldn't exceed a threshold that is defined during the test planning phase.



COUNTER	UNITS	RANGE
✓ 🟦 Tests/Sec	-	100
✓ 🟩 Pages/Sec	-	100
✓ 🟪 Requests/Sec	-	1000
✓ 🟥 Errors/Sec	-	100
✓ 🟦 User Load	vusers	9,8500

Fig. 7. An excerpt of the Load test analysis for the 24 hours load test run

E. Analysing Test Results and Applying Corrections

Some errors/warnings may appear at the end of the load test that was related to the storage speed, processing power, and a minor warning for the network bandwidth.

According to the conducted study, after investigating the possible causes for the load tests for 46 releases under the verification study, the root cause of the insufficient average response-time may be the processor clock speed and storage IO speed. Enhancing the response time with a significant enhancement, while consuming the least possible resources could be achieved through two logical options:

1) Option -1 Increasing the Processing Clock Speed

Firstly it is important to decide whether it is more feasible to enhance the performance through increasing the number of processors, or to enhance the performance through enhancing the IO storage speed? If the recourses availability is infinite then, there will be no issues however; the truth is that always in software development, it is important to trade-off decisions with the available resources. This enforces tackling the available budget for availing resources. In order to answer this question it is important to find out the percentage enhancement based on the dollars spent in that. If the dedicated budget would increase the number of CPUs 4 times from 8 to 32, and giving in considerations that 60% of the system's code is written as a parallel code. Based on that Amdahl's law [21] could be used to calculate the expected performance enhancement if CPUs increased from 8 to 16, from 16 to 32, and from 32 to 64. Table 1 shows Amdahl's Law calculation for 10% parallel code [21]. This law calculates the performance enhancements due to CPU increase knowing the percentage of existing parallel code as follows:

Amdahl's Law: (2)

$$x = \frac{1}{\propto \ + \frac{1-\propto}{p}}$$

Where:

x : The maximum speedup that could be achieved.

$\propto$: The percentage of the sequential code in the system.
p: Number of processors used

TABLE I. PERFORMANCE INCREASE DUE TO PROCESSOR COUNT INCREASE IN CASE OF 10% PARALLEL CODE

Trial #	p	x	Performance Increase	Cumulative Increase
1	8	1.09589	N/A	N/A
2	16	1.10345	0.69%	0.69
3	32	1.10727	0.35%	1.04%

Fig. 8 shows the achieved speedup for a system with 50%, and 60% parallel code. These numbers shows that the maximum possible speedup that could be achieved with a reasonable amount of cores (510 Core) is less than 1.5. This means that if it is needed to increase the speed of a given system. Adding additional processing power will not result into a huge difference. This leads to searching for another option for making an evolutionary speedup enhancement.

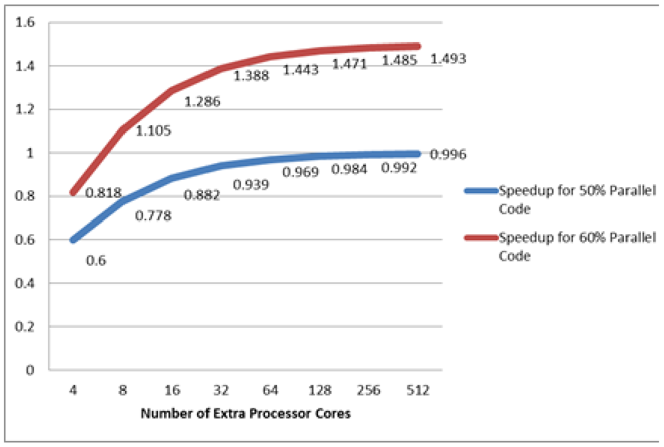


Fig. 8. System Speedup according to No. of processing Cores when parallel code percentages are 50%, and 60%

2) *Option -2 increasing the Storage speed*

Usually, in a limited resources environments system are being deployed on storage regular storage Hard disk drives (HDDs), or HDDs enrolled in RAID array. Increasing the speed of the system may need increasing the IO speed. According to many studies and benchmarks that has been applied and tested during this research moving all database files to a solid stage device (SSD) storage media could enhance the speed of the overall system up to 2030.34%. SSD technology has been compared to Flash SSD technology. It is very well known for everybody that SSD storage technology is faster than regular SAS HDD storage devices. According to DELL [22], and IBM [23] using Flash SSD is highly recommended for READ/Random-Access intensive systems specially when the read ratio is not less than 85%

According to a study that has been conducted by DELL [22] on random access, Read intensive systems, Flash SSD storage can perform up to 59.46 times better than SAS HD, with a price(USD)/IOPS (IO Operations Per Second) ratio of only 33% [22]. This means that the performance gain is strongly justifying the price difference! This shows that enhancing the storage speed (Maximum enhancement could be 5946%) will give better results than enhancing the processing power (maximum performance enhancement could be 1.04%).

According to some studies that has been conducted previously, and to the research that has been for done this study, moving the DB indexes to SSD drives enhance the system performance with less than 10% extra cost [23].

VI. PRACTICES VERIFICATION

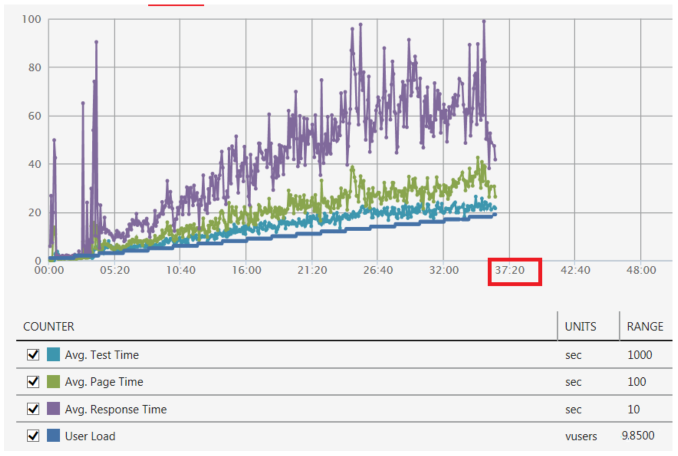
In order to measure the ability of the previous software engineering practices in enhancing the speed of building a scalable system, a four years study has been conducted as part of the process of building two of the national information systems with a potential total load size of 55 million users and 95375 simultaneous users [9]. The study began on March 2011, and finished on September 2015. During this study, six full development cycles/Major versions have been conducted with 46 different releases [24][25], [26].

The first Major version was managed without applying any part of the above proposed practices to be the reference sample version for any changes that could happen after applying the above proposed engineering practices. In 2012 the system has been fully rebuilt with a new version 2.0 while developing, applying and verifying the proposed engineering practices. Then, another four releases of both systems has been produced (Version 3.0, Version 4.0, Version 5.0, and version 6) through separate four development major versions. During each version out of the five (Ver. 2 to Ver. 6), some lessons have been learned and the practices have been enhanced to help in enhancing the next release production. Table 2 shows a comparison between Version 1.0 practices situation and version 6.0 including; the group of proposed practices that have been applied in Version1 and version 6.0 Fig. 9 shows the final load test result of the final release (release 46) of this study with no load errors.

Fig 10 shows the enhancements that happened in the Code Churn metrics due to applying the proposed practices. It is clear that the effect of applying the proposed practices has led to a significant improvement in the No. of lines of codes that are deleted, and modified.

TABLE II. SUPPORTED PRACTICES IN VER.1.0 AND VER. 6.0

Practice	Ver. 1.0	Ver. 6.0
Using Proposed Branching Taxonomy	No	Yes
Using Proposed Proactive architecture check-up	No	Yes
Using Proposed Proactive Static code analysis	No	Yes
Using Proposed Proactive Code Performance Test	No	Yes
Using Proposed Proactive build checks using fake build server	No	Yes
Using Proactive BVT	No	Yes
Using Proposed Goal Based Load Test to define required number of test agents	No	Yes
Applying Gradual Load Test.	No	Yes
Storing Database Indexes of the system data on a SSD drive	No	Yes
Update proactive check-ups based on lessons learned	No	Yes



COUNTER	UNITS	RANGE
✓ Avg. Test Time	sec	1000
✓ Avg. Page Time	sec	100
✓ Avg. Response Time	sec	10
✓ User Load	vusers	9.8500

Fig. 9. the results of the load test with no errors after 37 hours of the Load test run

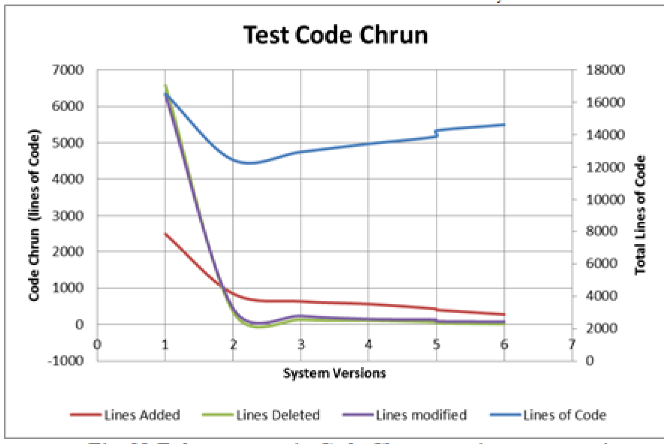


Fig. 10. Enhancements in Code Churn metrics across versions

The numbers of lines that have been added are basically depending on the new set of requirements that have been requested over the six major versions of the system under development during the study. The difference between the Ver.1 where no practices have been applied and other practices is clear to be noticed.

According to the Fig.10, applying the proposed SW engineering practices has enhanced the test code churn more than % 5000. The Average response time of the system pages has been enhanced from 73 seconds to 3.8 seconds by % 1921.5 enhancement ratio. The time to release has been cut down from 15 weeks though 5 sprints to 5 weeks through two sprints with % 300 enhancements. Additionally, it succeeded to reach the required availability percentile of 99.92 after being 17.886% only. At the same time the system scaled up from 1843 simultaneous users to 95375 with 5170% scalability enhancement rate. Table 3 concludes the key enhancements that could be measured between version 1.0 that used regular agile practices, and version 6.0 that applied the Proactive Quality Approach. Table 3 summarises these results with the enhancement achieved in each results between Ver.1.0 that has used none of the proactive approach's practices, and Ver. 6.0 that has used all practices of the approach. Fig.11 summarises how are these practices distributed over the different cycle activities. According to the figure, it clear that the lessons learned during the different phase over the cycles should lead to updating the proactive test rules that are being applied on the uploaded code to the version control server in the next releases.

TABLE III. COMPARISON BETWEEN USING REGULAR AGILE PRACTICES IN VER.1.0 AND USING THE PROACTIVE QUALITY APPROACH IN VER. 6.0

Comparison Aspect	**Ver. 1.0**	**Ver. 6.0**	**Enhancement %**
Code churn(Lines modified)	6321	75	842.8
Average page response time	73 Sec.	3.8 Sec.	1921.5
Time to release (Per sprint)	15 Weeks	5 Weeks	300
Availability	17.886	99.92	558.6
User Scalability	1843	95375	517.5

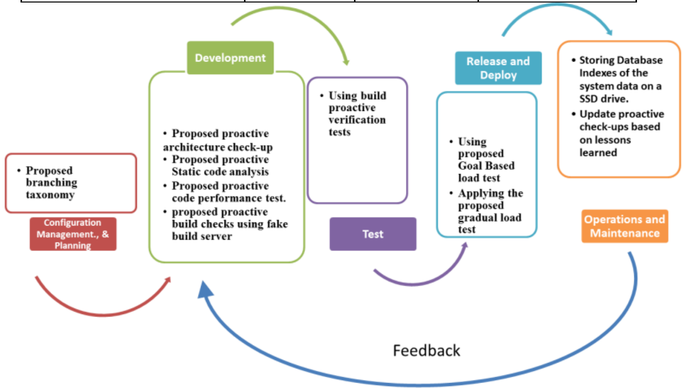

Development
Proposed proactive architecture check-up
Proposed proactive Static code analysis
Proposed proactive code performance test.
proposed proactive build checks using fake build server
Using build proactive verification tests
Release and Deploy
Storing Database Indexes of the system data on a SSD drive.
Update proactive check-ups based on lessons learned
Proposed branching taxonomy
Configuration Management., & Planning
Test
Using proposed Goal Based load test
Applying the proposed gradual load test
Operations and Maintenance
Feedback

Fig. 11. Major practices of the Proposed Proactive approach across the different cycle activities

VII. FUTURE WORK

In the future this research study will be repeated over cloud-based platform in order to test the effect of using the cloud vs. using the on premise deployment within a limited resources based project. Some extra Application Lifecycle Management (ALM) maybe used to enhance the overall quality of the system.

VIII. CONCLUSION

This research study has shown the planning of the SW engineering process design for developing a resource-limited high-scale, and mission-critical system. The study showed how does version control could be utilized to streamline any changes that may arise in the middle of the project, and how continuous integration could be mixed with some proactive check controls that can assure the compliance of the checked-in code with the predefined quality assurance measures.

The study explained the proposed gradual load testing process that has to be conducted to assure the scalability of the system to the expected amount of transactions and users. Having a clear load testing strategy that complies with the major business requirements is a major success factor for the whole system. This is compliant with ISO standard number 29119 that is concerned with SW testing. Conducting the load test in a right way is important, however, analysing the resulting errors and taking corrective actions could make the load testing more valuable since it helps in enhancing the overall scalability of the system. Some good lessons have been learned and elaborated at the end of this study. Applying the proposed practices has led to enhance many indicators with a notable percentage.

REFERENCES

[1] Otvio Augusto Lazzarini Lemos, Sushil Bajracharya, Joel Ossher, Paulo Cesar Masiero, and Cristina Lopes, "A test-driven approach to code search and its application to the reuse of auxiliary functionality", *Information and Software Technology, No. 53, P. 294-306, 2011, , ElSevier.*

[2] F.G. Wilkie , "An Investigation of coupling, reuse, and maintenance in a commercial C++ application", *Information and Software Technology, No. 53, P. 801-812, 2001: ElSevier,*

[3] Y. M. Helmy , A. B. Farid, and A. S. Abd elghany, "Simplifying CMMI Version 1.3 Implementation By Using Agile Practices: An Empirical Study", *International Journal Computing, and Information Science",Vol. 14, No. 4, P.31-45, Oct. 2104.*

[4] M. Rizwan Jameel Qureshi, and Isaac Sayid,"Scheme of Global Scrum Management Software",, *Information Engineering and Electronic Business, Vol. 7, No. 2, P.1-7, 2015*: MECS

[5] Saja Al Qurashi, M. Rizwan Jameel Qureshi, "Scrum of Scrums Solution for Large Size Teams Using Scrum Methodology", *Life Science Journal.Vol. 11 No. 8, 2014.*

[6] Samireh Jalali, Claes Wohlin, "Agile Practices in Global Software Engineering - A Systematic Map", International Conference on Global Software Engineering (ICGSE),2010. Princeton,NJ – USA.

[7] Del Nuevo, E., Piattini, M., Pino, F.J., "Scrum-based Methodology for Distributed Software Development", Global Software Engineering (ICGSE), 6th IEEE International Conference, 2011.

[8] Jeong Ah Kim, Seung Young Choi, Sun Myung Hwang, "Process Evidence Enable to Automate ALM (Application Lifecycle Management", Parallel and Distributed Processing with Applications Workshops (ISPAW) 9th IEEE International Symposium, 2011.

[9] Microsoft, "Egyptian Ministry Facilitates Transparently Open Elections with New Applications: a Solution Case Study".[Online].Available:https://customers.microsoft.com/Pages/Download.aspx?id=15111

[10] T. E. Murphy et al., "Gartner, magic quadrant for application development life cycle management", 19 November 2013.

[11] Steven St. Jean et al, "Professional Team Foundation Server 2013", Wrox, May 19 2014.

[12] Mickey Gousset et al, "Professional application life cycle management with Visual Studio 2013", Wrox, March 31 2014.

[13] Eric Sink, "Version control by example", Pyrenean Gold Press, 2011.

[14] Larry Brader, and Roberta Leibovitz, Jose Luis Soria Teruel "Building release pipeline using Team Foundation Server 2012 (TFS 2013 Editions)", Microsoft Patterns and Practices. Feb 2014.

[15] Larry Brader, Howie Hilliker, Allan Cameron Wills, "Testing for continuous delivery using Visual Studio 2012", Microsoft Patterns & Practices, March 2, 2013.

[16] Satheesh Kumar N, , and Subashni S, "Software Testing Using Visual Studio 2013", Packt Publishing, July 26 2013.

[17] *Software Testing: Concepts & Definitions, ISO/IEC/IEEE Std. 29119-1:2013*, September 2013.

[18] *Software Testing: Test Processes, ISO/IEC/IEEE Std. 29119-2:2013*, September 2013.

[19] *Software Testing: Test Documentation, ISO/IEC/IEEE Std. 29119-3:2013*, September 2013.

[20] *Software Testing: Test Techniques, ISO/IEC/IEEE Std. 29119-4 FDIS Draft*, March 2014.

[21] Amdahl, Gene M.. "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities". *AFIPS Conference Proceedings* (30) P. 483–485. (1967).

[22] Vamsee Kasavajhala, "Solid State Drive vs. Hard Disk Drive Price and Performance Study", DELL, May 2012.

[23] Aamer Sachedina and, Mathew Huras, "Best Practices Database Storage", IBM, January 2012.

[24] Jiang, Z.; Hassan, A., "A Survey on Load Testing of Large-Scale Software Systems," in *Software Engineering, IEEE Transactions on* , vol.PP, no.99, pp.1-1 doi: 10.1109/TSE.2015.2445340.

[25] Y.M. Helmy , A. B. Farid, and A. S. Abd Elghany, "Simplifying CMMI Version 1.3 Implementation By Using Agile Practices: An Empirical Study", in International Journal of Intelligent Computing and Information Science *IJICIS, Vol.14, No. 4 OCTOBER 2014.*

[26] Ahmed Bahaa Farid, Enas M. Fathy, Mahmoud Abd Ellatif, "Towards Agile Implementation of Test Maturity Model Integration (TMMI) Level 2 Using Scrum Practices", *(IJACSA) International Journal of Advanced Computer Science and Applications, Vol. 6, No. 9,P. 230-238. 2015.*