I J C T A, 10(9), 2017, pp. 327-351 ISSN: 0974-5572

Exploring Cloud Computing Technological Test Debt

Manu A.R.[1] *, Vinod Kumar Agrawal[2], K.N. Bala Subramanya Murthy[3] and Suma V.[4]

ABSTRACT

Technical debt (TD) arises in web-based cloud computing ecosystems when the stakeholders both intentionally or inadvertently formulate and execute their technical choices and decisions in return for the instantaneous gains or profit in the Cloud service project. Among the various dimensions of the Tech debt, important dimension is quality control and tech test debt or test debt [1]. This work gives a general idea of test debt, in the cloud ecosystem, with its causes and issues that are responsible for the test debt. Also, this paper presents some planned approaches for repaying the test debt in cloud business. This work also offers methods to find the "test smells" which cause decay of the computing system and steps to overcome the test debt. In addition, this work presents various case studies to show how re-factoring reduces the test debt in engineering systems. This is done at various stages such as service level agreements, designs, architecture, code and test script for repaying the accrued technical debt. Test debt is a growing topic and is in its infant stage. This work will be of broader importance for IT industries and academic research in the area of cloud computing service security.

Keywords: Cloud computing test debt, Cloud service debt, Security debt, Architectural debt, Cloud service debt, Cloud technical debt, Test smells, Code testing, white box testing.

1. INTRODUCTION

Tech debt (TD) is a metaphorical and allegorical notion coined by Ward Cunningham in the 1990's. The technical debt heaps up due to diverse stakeholders possibly getting into debt troubles due to execution of their decisions for short term profits in the cloud market which result in additional rework. The debt piles up due to the technical decisions executed by stakeholders for their interim gains in their service offerings due to cloud market competency, time pressure to meet the customer demands and rising cost in the market. The test dimension of tech debt [1] in the cloud computing ecosystem or market is recognized as "Cloud computing technical Test Debt" or "cloud service test debt" (CSTD). TD has various scopes (associated factors) in practice it is progressively significant to the organization [1] in cloud market that offers the large range of complex service components and maintains the expanding computing systems.

In this work, we present an outline of CSTD and the manuscript is planned as follow: In the next section, we present the concise introduction to TD with the reference cloud computing ecosystem. In section 2, we discuss the causes contributing to the CSTD. In section 3, we present the key approaches for administering of CSTD. In section 4, we present a few case studies for managing CSTD in real time cloud production environment, and finally, in section 5, we wrap up with the note on future directions of research on CSTD for Cloud computing system resources.

[1] Cori Lab, Dept of ISE, PESIT, Visvesvaraya Technological University, *Emails: manu.a.ravi@gmail.com; manu_ar@nitk.ac.in*

[2] Senior IEEE Member, Director and professor, CORI Lab**,** ISE Dept, PES-University, *Email: vk.agrawal@pes.edu*

[3] Vice Chancellor, PES University, BSK 3rd Stage, 100 Feet ring road, Bangalore, Karnataka, India-560085, *Email: vice.chancellor@pes.edu*

[4] Dean, Research and Industry Incubation Centre, Professor, Department of Information Science and Engineering, Dayananda Sagar College of Engineering, Bangalore Karnataka, India, *Email: sumavdsce@gmail.com*

1.1. Cloud computing technological debt [CCTD]

Cloud computing technical debt [CCTD] causes accumulation of arrears and this, results in paying of unbalanced interest due to past evasion of loan repayment in terms of technical work. In the short-term, CCTD will be profitable, but in the long run, it results in irreparable damage like bankruptcy due to the accumulation of huge debt. The CCTD accrued due to the stakeholders by consciously or accidentally formulating their incorrect or non-favorable technological choices. These choices made without any knowledge or assessments due to market pressure, vendor's competition, time schedule slippage etc. result in TD [1]. This incurring CCTD in short run business is profitable, but damages in the long run, if not paid back regularly over time. The interest on the arrears piles up over an epoch of time and ultimately resulting in the circumstances where the stakeholder may not be able to pay back the accrued monetary debt. This results in fiscal insolvency/bankruptcy, leading to business ruin.

CCTD is similar to monetary debt, which the stakeholders get trapped into fiscal debt whence they borrow/take a loan. The arrears don't cause any issues till they repay the loan. If they fail to repay the loan, the sum-interest piles up over time duration and they may not be in a position to pay back the debt. This results in fiscal insolvency. If the cloud stakeholders do not address and tackle CCTD using standard benchmarked methodologies, processes, guidelines, and measures. It will lead to "Cloud computing service tech bankruptcy and ruin their cloud business under the arrears stack [1]".

In literature we find lots of research on TD and its impact of the computing business in various ways. In the original paper [2] author coined TD and discussions are made on its impact on the computing business corporations. If neglected, TD it leads to complexities [3], inability to meet the demand of end users by way of poor production, increasing cost [4], affects the quality and security, and other non-functional requirements (NFR) with a debt load. With restricted time period and market constraints, with increase in TD has a negative influence on the coding team's confidence and enthusiasm [5]. Based on our interaction with industry experts, our literature survey and our own industry experience, we present well-known root causes for TD, in a cloud computing system they are:

- Schedule time pressures, service delivery time pressures. Budget constraints, cost to company constraints [3-4].
- Rise of unfinished work within time precedent products/services and version releases/sprints. Lack of understanding on what to be delivered as a service to meet customer expectations.
- Inability to follow the standard cloud computing service architecture, design and development cycle.
- Unskilled engineers/resources experience to handle the cloud service architecture, design, and coding the services.
- Lack of proper documentation, derisory testing and bug fixes.
- Inability to follow standard Quality assurance, Quality control procedures, Processes and Guidance on testing activities.
- Lack of proper test suite, test strategies, test plan, test actions, test cases, test repositories, and test scripts due to unexpected deadlines in service offerings. Testing executed on-fly and thrown out of gear due to quick fix solutions in the service offerings.
- Over usage legacy/proprietary code/platform in the cloud. Lack of proper interface between the open source and legacy systems of cloud computing services and overdue and late refactoring.
- Failing to follow or absence of dev-ops continuous integration methods and automation rules, compliances and other out of control factors.

- Lack of co-ordination between teams, and other cloud service stakeholders. Cloud service developers, sys-admins and testers face difficulty in the last-minute due to unexpected, delivery time of the cloud service on fly deployment. This is due to market competition, management decision to roll out cloud service product at the last moment, lack of communication in advance to testers with tasks without delay.
- Missed tests or "solve it later" surface with deficit test coverage, oversized user stories, scenarios, and short sprints. A Service delivery pressure plays the vast role behind large accumulation of technical debt in QA practice.
- With the Sudden surge in accessing the cloud service using several smart thick and thin clients platforms, the complexity and difficulty for the developers and testers to design and test the cloud services code to support multiple languages, platforms and hardware devices and in more networking sites it has to be leveraged and tested.
- Unwarranted time consumption in penetration testing, system testing, service virtual configuration, testing, web testing, security testing, release testing, app service compatibility testing, service and compliance testing.
- Lack of service UI testing in some browsers types and versions, platforms, devices and scripts' growing with each test sprint, and causes delay in the release cycle leading to loss of time-to-market.
- The escalating expenses of hiring - testers and testing resources required to test the service are doubled due to the growing complexity and increased demand of cloud service. Testing effort many a times is wasted in chasing false positives, with respect to functional and non-functional requirements. Delayed refactoring results in CSTD.
- Amplified effort in the cloud service development and testing efforts. It goes with the territory, and following quick and improper approach. Insufficient automation and testing tools and methods lead to the CSTD.
- Mounting complexity of cloud service offering and implementation, results in tracking the test cases and bugs is a challenge. Failure in reacting to the incident occurs, and results in non-proactive approach to identify and measure defect results in cloud computing system failures.
- Cloud computing system (CCS) development organizations and their stakeholders fails to comply and upgrade their skills on QA and competencies to the last levels desired as per industry benchmarks, standards, processes, rules, regulations and guidelines.

1.2. Engineering process vulnerabilities leading to CCTD

The work on test debt was originally carried out in [1][6-7] but still more research is needed to explore test debt dimension of TD. The testing plays a vital role in SDLC and cloud service life cycle in terms of Cloud service functional and NFR's. Test debt impact should have a more focal point in fabricating the cloud services with the industry benchmark milieu. CSTD arises due to using the shortcuts with incorrect and immature decisions during testing activities. Conventionally, both academia and industry research experts are concentrating more on technical debt arising due to code debt, service design debt, and even cloud service architecture debt dimensions. Pragmatic and realistic supervision of CCTD is vital to govern other dimensions of debt arrears with obligation such as "cloud computing system (CCS) infra service debt". "CSTD" accrued due to testing done by test engineers by using shortcuts and not complying with the standards, along with live industry bench marked best practices and processes for testing, validation, tracking, reporting for the defects and bug fixes [1]. Not executing the unit test cases, or test scripts from test oracle

is a structure of shortcut, to gain the short-term profit to speed up the coding and releasing the sprint or product. Missing the unit test either intentionally or ignoring to execute the test results leads to CCTD.

Figure 1 shows the scope and dimensions of CCTD. Traditionally, code debt or program debt or implementation debt is researched in [6 - 9]. Object oriented design debt is presented in [3] [9], architecture debt is found in [5-10]. In addition Technical Debt (TD) management and pragmatic control of TD are given in [10]. [10] Lists scope and degree of TD, the same methodologies that apply to dimensions of CSTD. With cloud service specifications and design transformed to service implementation code, service testing, building, documentation, infra versioning etc.

2. CLOUD SOFTWARE TESTING DEBT

Cloud computing service testing is a process of finding, assessing, auditing, and root-finding of bugs, defects, vulnerabilities, issues, errors, and failures in cloud computing service software. It is a verification done to bring out the internal and external (behavior), performance, code security of the service and application software (s/w) against functional and nonfunctional requirements. This is the implementation and execution of the test cases against the cloud service s/w requirements to find the defects. Neglecting, the testing process completely or partially, doing under testing, and over testing, without the optimized testing leads to the CSTD. There is no clear-cut adoption of test debt and its dimensions have not been studied and well researched in the literature [1][5][7]. CSTD generally occurs when testers execute the tests using shortcuts, incorrect or non optimal decisions for short-interim benefits to speed up the testing process as part of the software service development process. Short-term decisions in the long run have an adverse impact on the business due to execution of faster and hurried wrong shortcut decisions. This leads to the test dimension of TD [1]. Figure 2 shows the prominent scope of cloud computing technical debt with examples.

2.1. Significance of Test Debt in cloud computing

Let us consider the case in the cloud computing ecosystem, where the Small and Medium Enterprises (SME's) and many of start-up corporations pilot to test debt due to executing the crosscut decisions and their negligence's on trivial testing issues. Consequently, SME's strive for continued existence owing to more CSTD, in order to be first to market and release the service product in market early. This is due to

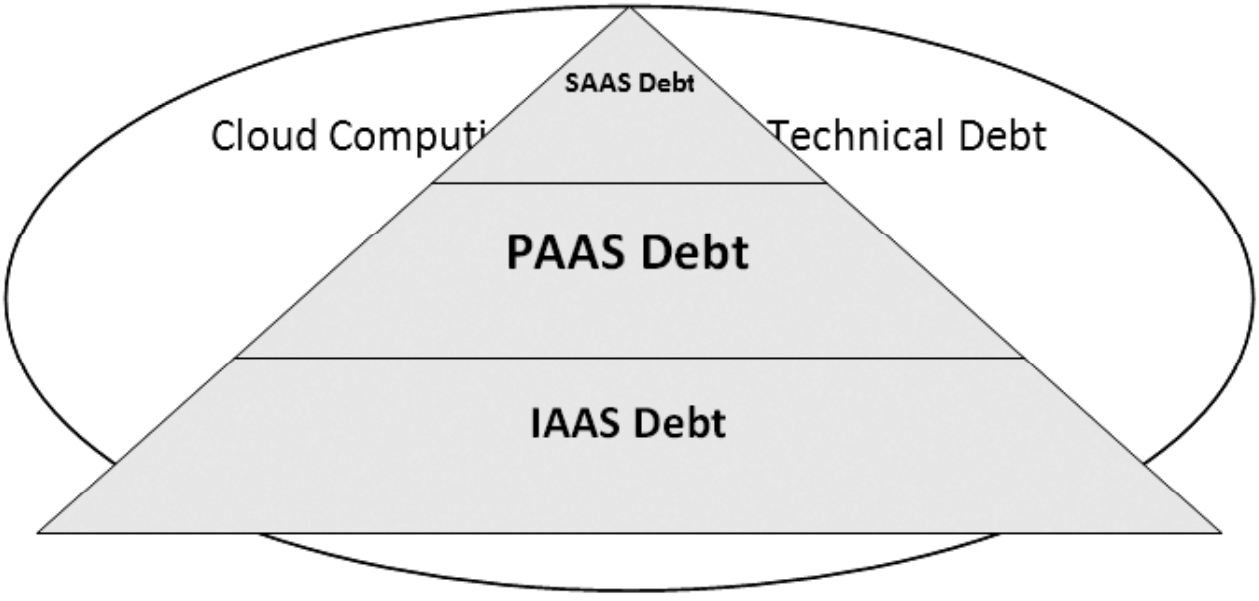

Figure 1: Showing the various debts of CCTD

increased market competition, to show their existence among the competitors, and to get the credit of "foremost carter gain" [1]. The SME start up's concentrate completely on developing new attractive, customized and creative features due to time pressures and schedule to market early. When it comes to testing with verification, validation, auditing and accounting the products, the programmers and testers use shortcuts and resort to the least common features/ scenarios/ stories/ epics, perform drastic and ad-hoc testing and do not do wide and ample testing the service or product. However, the SME's stakeholders to prove their existence with larger corporations with the creative development team, design the service in short comparative period, finally, develop and design the working service or product, and ends up naively gaining the real share in the market. But their success is limited to, a short period, but not sustainable due to incurring of huge TD, in the first release version. Explicitly over the precise dimension of Test Debt of cloud services for first version SME team fails to design scripts and execute the automated tests. The testing is executed with reduced test coverage leads to CSTD.

Below are various well-known dimensions of the CCTD with illustrations in the table:

With the intention to win and being the first to capture the market and to ship the product first in the cloud marketplace by neglecting the coverage of test scenarios results is high-test debt. In future releases the SME's are ought to focus on repaying the incurred CSTD by devoting profoundly on testing the unreliable product service offered over the cloud. More delays and lacking to focus on paying back the debt results in business ruin of SME's [1] due to the heavy debt incurred by the development and progression of the product will become standstill and become unreliable, leading to Cloud computing service test debt. In order to avoid inefficient service or supplying unreliable product to the end users SME's should focus on repaying technical test debt and avoid technical bankruptcy and business ruin.

Technical Debt Dimensions	*Causes and types with illustrations*
Cloud Service Level Agreement Debt	Over trusting, under trusting the vendor, SLA Smells, Misconceptions and logical fallacies, applications of randomness due to a cloud environment.
Cloud computing service requirement debt	CRS, SRS, FS guidelines, rules, policy violations, SRS smells, FS Smells, inconsistent styles, violations of SRS, CRS, FS constraints, Randomness coming from the initial conditions. SRS smells, CRS smells, user experience smells- simple work flows violations, help cues unavailability, non consistent design, usability testing, delayed re factoring performance smells - no quick response, non high availability, no continuity in services, non clustering, violation of stress/load/stability test suite and its allied principles, last minute specification changes percolating through the time and budget without documentation. Requirement decay, requirement smells, requirement entropy.
Cloud computing service architectural Debt	Architecture Smells [duplicate design artifacts, unclear role of involved stakeholders and interfacing entities, inexpressive or complex architecture, the entire thing in architecture is centralized, over generic design, asymmetric architectural structure and behavior of entities, dependency cycles, redundant dependencies, implicit dependencies], lacking modularity, architectural patterns and anti-patterns violations, Architecture Functional spec guidelines, rules, policy violations, principles violations, modularity violations, tight and loose coupling violations, interface violations, non portability and uncertainty in architecture. Failure to build the loosely coupled components, architectural flaws, architectural weakness, danger signals of architectural defects, faults, error, mistakes, wrong assumptions, fault patterns, change smells, architectural mismatch, architectural bad smells, contraindicated patterns, structural anomalies and problems, spoiled patterns, non functional requirements, technology and architectural constraints [11].
Cloud computing service Design Debt	Design rules violations, OOD principles and interface violations, modularity violations, Design smells, violations of design constraints, uncertainty in design, and forms, Design critics, structural problems, design problem

(*contd...Table*)

Technical Debt Dimensions	*Causes and types with illustrations*
	symptoms, design defects, problem patterns, malignant patterns, indicators of design problems, violations of design heuristics, distorted design patterns, design pattern defects, design dis-harmonies, structural flaws, quality defects, design flaws, design inconsistencies, spoiled patterns, design principles violations [5].
Cloud computing service Code Debt	Code smells, violations of coding standards and styles, guidelines, and principles. Failing to follow web standards, glue or duplicate code, architecturally relevant code smells, platform – OS inconsistency, Browser inconsistency, language inconsistency to support the feature's implementation, Infrastructure Code smells, PAAS code smells, SAAS code smells, Storage as a service code smells, Infrastructure, Configuration code smells, lacking of tools and services, Software System, Production code, Apply traditional software engineering practices, static analysis tools violations, inconsistent coding styles, code decay and code entropy. Common code smells- Class-level smells: Cyclomatic complexity [2], Programming: (unplanned intricacy, exploit at a space, shade faith, Boat newscaster, hectic waiting: overriding CPU, Caching crash, Cargo cult programming, Coding by exclusion, Comment Inversion, Design pattern, Error hiding Stack trace, Hard code, Magic-numbers, Lava flow, Lasagna code, Magic strings, repetitive patterns and sub strings over again, Shotgun surgery, Loop-switch succession, Soft code: Spaghetti code) [3 - 10].
Cloud computing Service test debt	Comprehensive coverage, brittle or missing test coverage violation, lacking tool and automation support, lack of relevant test suites, duality defects, quality flaws, quality issues - [testing error, testing defects, testing issues, testing failures, testing mistakes, aging symptoms, lack of tests, insufficient and inadequate coverage] testing decay, test entropy. Improper test design Measures and tests. Practical measures of randomness discrete transforms, and complexity, A TEST is "cursed" or "blessed", Boy or girl paradox. Software rot, Data degradation [10], rarely used code. Rarely updated code-Dependency hell, dormant rot or active rot, [14-15] removing dead code, reinventing the wheel, software components, creeping futurism, Facing a dead-end, Stuck in a rut, entropy rate, nary entropy. Redundancy (information theory).
Cloud computing service configuration debt [11]	Configuration management tools: Ansible, Chef, Engine, Puppet, GIT tool, Docker container flaws, configuration smells, no tools to detect configuration smells, leading to configuration debt execution configuration smells include: Deprecated statement usage, incomplete task, long statements, missing default case, missing conditional and improper alignment, misplaced attributes, invalid property value, unguarded values, improper quote usage, duplicate entity. Design configuration smells – [Multifaceted Abstraction are not cohesive, Unnecessary Abstraction or module, Imperative Abstraction, Missing Abstraction- resources and language elements are not encapsulated, Duplicate Block, Deficient Encapsulation, Insufficient Popularization, Unstructured Module- repository structure, Dense Structure, dense dependencies, Weakened Modularity- a module with high coupling and low cohesion, Broken Hierarchy] ,configuration decay, configuration entropy. Abstraction inversion, Ambiguous viewpoint, Object-oriented analysis and design (OOAD), IPC-inter-process communication, Gold plating, Inner-platform effect, [11-15] Configuration management: [Dependency hell:, DLL hell: derisory supervision of (DLLs) dynamic-link libraries, precise Microsoft Windows OS, expansion conflict -Mac OS X, versions, JAR hell: JAR files, Java class loading files, etc.] [16].

(*contd...Table*)

Technical Debt Dimensions	*Causes and types with illustrations*
Cloud computing service security debt	Security, safety, defense protection– merged with confidentiality, privacy, secrecy, authenticity, legitimacy, integrity, veracity, reliability, continue-ability, availability governance and technical, locality jurisdictional, legal and operational domains. Legally authorized and Electronic Discovery, Compliance and Auditing, accounting, privacy of information Life-cycle supervision, and Portability and Interoperability. Testing smells, test script entropy, test script rotting.
Cloud service business debt	Comprehensive in all business use cases, scenarios, browser/ language/ platform support, super user/admin rights, business pressures, release soon without necessary changes and testing comprising the uncompleted changes and lack of process understanding, lack of collaboration, lack of understanding the business process, lack of the knowledge, lack of ownership, lack of alignment to standards, poor technological leadership, scope doping, risk of losing market, lacking the state-of-the-art technology.
Documentation debt:	No records for significant concerns, poor citations, archaic documentation, no support and help documentation, no standards in documentation [1].

2.2. Common Sources of CSTD

In order to tackle the test debt, it is vital to distinguish what the test debt encompasses of, how to spot it, and, how to tackle its existence in a computing system. CSTD mainly occurs due to the lack of tests, scripts/test cases in test oracle, lack of test coverage, or improper and wrong tests epics/ stories/ sprints/ releases/ versions of the test code base oracle/suite. In general, the factors contributing to the tech debt also in turn add to test-debt as well [1]. IT teams owing to rising costs, cloud market pressure, time schedule limit to deliver faster value based services to their stakeholders, and increase their customers wow factor. Typically, in such circumstances, lacking business domain skill, lacking knowledge about the CSTD, the testing teams sacrifice the industry benchmarked testing procedures and best practices contributing to accruing of the test debt. Shortage of trained and experienced testing expert engineers, due to cloning of tests (test scripts and test cases), redundant service with increased complexity, sins of development incomprehensible test scenarios, over sized test cases adds to CSTD. In addition, long methods in the test scripts, short and incomprehensible naming conventions of the test cases leads to CSTD. Deeply nested logic with tight coupling features. Also, shotgun surgery, data clumps, lazy class features, feature envy, missing or inadequate exception handling and test scenarios, missing security checks and testing features, insecure constructs, lack of fallback procedures, missing input and output validation, lacking to follow standards and procedures contribute to the CSTD.

Amplified quality degradation occurs, due to failure to execute quality control actions [17] comprises of - Reviewing the CCS agile user epics, stories, incidents, use-cases, test scenarios, into test cases, and test scripts. Failure to do architectural review, design-inspections and walk-throughs of code and test case results in test debt. Failure to execute acceptance test case scenarios from the end user perspective at user site results in CSTD. Unskilled test engineers, untrained developers on unit testing during their academics who become skilled on job at writing unit or automated test scripts, with inexperienced resources adds to mounting CSTD. Agile process in CCS includes the process of refactoring, rewriting if necessary, walk-throughs, inspections, reviews, dev-ops continuous integration, improvement and software, automated decay testing as alleviation strategies. Derisory or undocumented CRS, SRS, FS, and other specs, test cases, test scenarios, test scripts, etc. With deprived test coverage, deprived test scenarios, deprived test configuration, deprived test practices, deprived requirement trace ability matrices, deprived code quality and execution practices as noted from our experiences lead to mounting CSTD.

Inadequate CCS service management, deficient metrics, deprived adherence to industry bench marked processes, practices, guidelines, standards, policies, rules and regulations adds to CSTD. Poor engineering elicits take account of poor CCS requirement specs, poor CCS High level design [HLD], and poor architectural design [17-19] CCS service complexity, incomplete requirement specs and lacking field skill-set and knowledge mounts to CSTD. CSTD brings out cascading consequences of shortcut choices executed in CCS services or in the broad spectrum due to deadline schedules and/or market pressures in resourcing the services. Testing entities lacking skills or testing infrastructure lacuna, lack in poor test automation tools, scripts. And besides inaccessible knowledge, poor test script reporting experience, resulting in huge bugs, and defect counts. This is exploited by attackers and end-users and business competitors. Measuring CCS service QOS metrics, or dimensions or indices comprise: test code re-usability, platform or test code/ script portability, maintainability, affordability, serviceability, security, continuity, integrity. And all the "ilities", includes effectiveness, unpredictability, changeability, measure-ability, dependability and test ability etc. Each instance breaching any one of the CCS-QOS metrics or indices, incurs more CAPEX and OPEX to fix the CCTD in later stages. Lacking in value leads to multi-dimensional CSTD, a lacking in Business Intelligence (BI) Testing, domain intelligence testing, lacking in out-of-box thinking, lacking to establish multiple test scenarios. Testers lack understanding and creative thinking and fail to analyze all the possible scenarios for all new features, enhancements, integration, UI updates etc. contributes to the accruing test debt in a real time production environment in the cloud ecosystem.

The tester creates test cases, sanity check lists and required test information so that when the service build comes to testing, they are equipped with their testing parameters and environment configured for testing. Testers apply all of their creativity to test the functionality, explore each and every possible scenario and conflicts. Many times, it turns into extremely tough to locate the origin of the bug i.e. it may be a coding bug, documentation bug, architectural design bug or may not be a bug even. But it's the job of the tester to report every bug. A good developer is one who takes feedback in a positive and constructive manner, diagnoses the problem, and debugs it. But developers often avoid conflicts and that causes hindrance. The 'like' factor may be different, but it surely helps in understanding and mapping uncovered areas of testing. By choice or by chance of known for quality of bug or for quantity of bugs, non-emphasizing on manual testing or automation tests leads to the CSTD.

2.3. Methods to repay CSTD

The tester needs to test the system under test (SUT) or service under test from diverse allied viewpoints with end user expectations, their perspectives, and other stakeholder business perceptions, write the test script with proper testing mind set. Adequate testing the CCS services and finding bugs, reporting it and finally following the defects with domain thoughts, needs to be done. Tester need to work towards betterment with QOS of CCS by taking into account special aspects like utility aspects, security concert, GUI etc, request and response time, stability, thick and thin client complexity. In addition, CCS testing starts with the writing, preparing and writing the test plan, preparing the manual and automation test strategy processes, identifying the test scenarios, user stories, use cases. It also consists of testing feasibility, epics/stories identification writing the test cases, coding automation testing script, execution of test cases, reporting and tracking of bugs and document preparation, writing the test reports. Preparing the test summary reports over the test life cycle is followed. The overall testing activity involves productive work to make CCS more effective and of better quality and contribute to the continuous improvement of the product by accepting the customer needs to QOS.

A good tester is one who understands patron needs, learns and knows the marketplace, hottest trends, has in-depth pertinent knowledge about the product and client's business. He can put himself in customers' shoes and test the product to make sure good quality control and quality assurance of the product. CCS service testing is concerned learning with faster testing and execution of innovative exploratory testing,

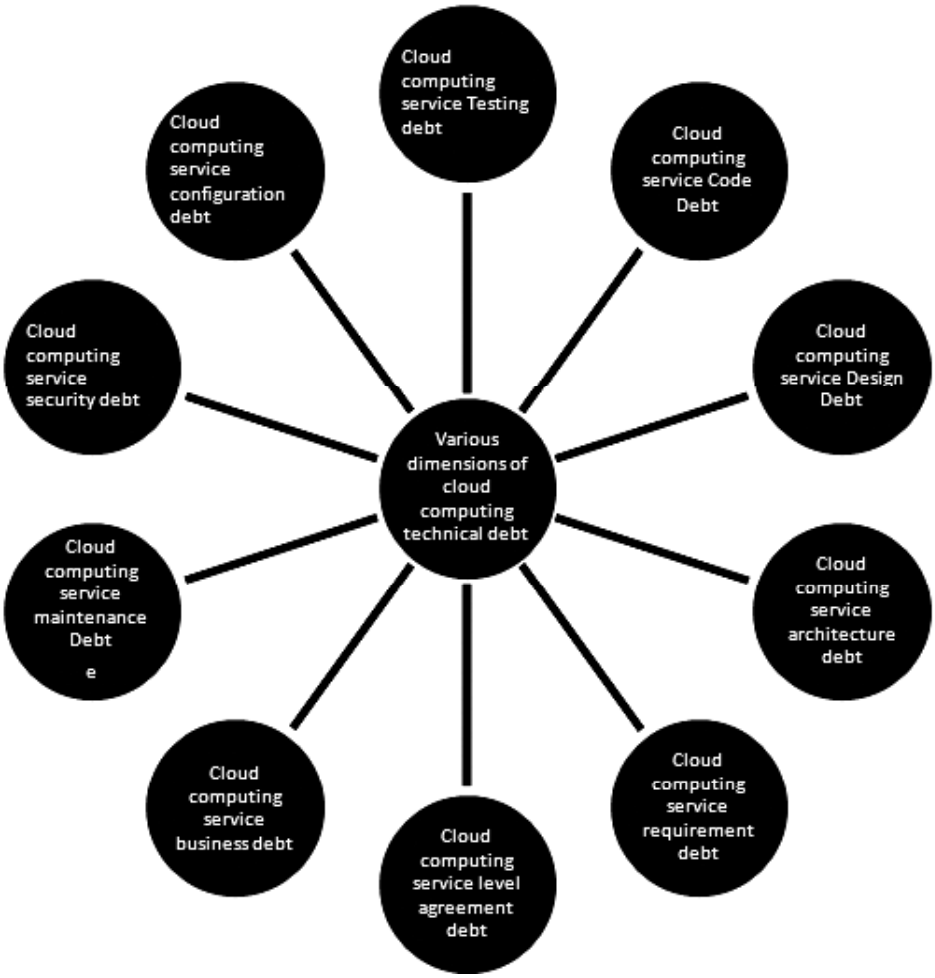

Figure 2: shows prominent scope of cloud computing technical debt with examples.

using new ideas to interpret broad-spectrum behavior using power of analysis and critical thinking. Also, he should write the Test case or Test script using new tools and techniques in real life to test the product by fixing priority and investigate the test data and analyze the results. The tester should not have number mania by increasing the test scenario counts and scripting count without effective coverage. This is the cause for accruing test debt. The test case number mania is only to satisfy the customer and or the organization managed to impress them that they have good test coverage and huge set of test cases maintained without proper test quality.

Inadequate testing infra, improper configuration of the testing environment, lack of proper knowledge about the cloud testing ecosystem, non availability of automation tools, progression, modus operandi add to test debt. Outdated test automation tool version, test framework, language version, platform version with older setup. This results in postponing of tests, failure to replace with the new framework, the newer version, updates, and patches. And, also, inadequate test planning, insufficient test strategy, design, testing effort estimation errors, test cycle planning etc. contribute to accruing of test debt. We inscribe a test script for the corresponding service code program. The Test engineers write these test scripts and store in a suite in the same language as the source code. The testers or developers execute these test scripts as unit test programs which connect to the main programs and sprint as the programs. If there is any CRS/BS change or bug fix in the main source code, then, the changes are implemented in the source code. Further a modification is done in the test script, and test script to rerun from the beginning to run the test suite. If bug is detected at an early stage the investment cost of bug fix will be less. If testing process is delayed, the price of bug fixes increases exponentially.

2.3.1. Various SDLC cycle consideration of Test debt

1. The degree of gathering cloud service business requirements, cloud service specifications (CRS are transformed to service implementation business language software requirement specification, (SRS) and functional specifications etc. 2. Failure to make Feasibility Study by the team consisting of cloud service stakeholders, marketing executives, managers, business and market analysts, system architects, finance executives, HR, developers and testers. They have to verify for: - technical achive-ability, - economical viability, - resource feasibility platform, with all 4 wares feasibility etc. Ignoring to test each document in each phase leads to accruing of CCTD. 3. Design and architecture: Failure to follow the design and architecture principles [12], violation of standards [20], violation to document proper interfaces between modules. Usage of shortcuts to craft service design and architecture leads to smells resulting in design debt and architecture debts. Metrics for architecture debt are established in [8]. Patterns and Anti-patterns are closely associated with the design and architecture debt resulting in Tech debt [5]. 4. Coding / Programming: — done by cloud service developers, including - senior coders, junior coders, etc. Failure to understand and non compliance to the coding standards lead to coding debt [18] [20]. 5. Testing: negligence for testing activity and violation to follow the standards and guidelines in manual testing, agile testing and automation testing [6] leads to test debt [7]. 6. Cloud service/application/resource configuration debt [11]. Installation debt-accrue due to poor action by installation engineer's and failure to follow dev-ops process and method leads to configuration debt. 7. Failure to offer proper service and maintenance results in maintenance debt: - if the patron locates the errors, failures, defects and bugs and are not fixed and reported back to offer the assured QOS assigned in SLA's, it results in Tech debt. As customer requests the vendor back for service error changes and bug fixing, in accumulation to trifling business changes like addition, removal or transforms of foremost features in the CCS which lead to tech debt if not properly maintained.

In cloud service backtracking for service requirement is not feasible, i.e., customer flexibility may not be realizable as he cannot reverse and make the changes in the service spec and Service Level Agreements (SLA) once the design stage is reached. Transforming SLA's and service requirements leads to the amendments in service architecture, and design of the service. Thus, bugs, vulnerabilities, defects creep up. And finally, it results in the service failure of the cloud service design and cloud service architecture. This leads to transforming and re factoring the cloud service code. Moreover, it results in more bugs and vulnerabilities leading to CCTD. Thus, the requirements are generally free-zed once the design of the service is started.

Major drawbacks include testing is a small phase, and it is done after service coding, so the SRS, Service design, and service architecture are not tested. If there is a bug/error/fault in the requirement spec, it creeps and crawls till the end and leads to lots of re-work, and re factoring [15]. If neither the service - requirements do change, nor do SLA, nor does CCS service architecture and CCS service design and CCS code do not change, we get steady secured well-tested and validated with good QOS CCS services accessible to the cloud patrons.

Testing is the process of validating the correctness precision, completeness, and QOS, - because, the way test engineers check/verify and validate the service or product assorted from the way customers use the service or product. And testing the SLA, service requirements, design, and architecture is very important to avoid the vulnerabilities, defects, errors, bugs, failures at the early stage of service configuration. Testing activity calls for the tester to have good patience, be creative, open-minded and committed to test the product. They need to think and act from an end-user perspective. Testing lessens the unraveled bugs, if bugs are not reported it's not the proof of precision and accuracy of the product, rather than it's the marking of the testing activity as not proper. Non-appearance of fault is a myth and does not fulfill the end customers requirements and necessities.

2.3.2. Verification/constructive action and Validation/destruction model [verification and validation]

As all the SDLC activities take place, the testers test the CRS in parallel to developers by using the method of reviewing, inspection, testing, agile scrum meetings and walk-through on CRS. Testing is done a) for

inconsistency in the service prerequisites/specs. b) for misplaced service prerequisites /conditions and or requirements. c) for erroneous vigilance of service specifications. The testing team assesses the CRS, locates, classifies and catalogues and categorizes faults, and defects and reports to the customer seeking the corrected requirement spec. Then, it updates the CRS, SRS and Functional Spec (FS). In the next stage, the testing team reviews the SRS.

The testers continue testing each document simultaneously and verify from the end-user acceptance perspective and business perspective and tests SRS. The tester checks assesses, evaluates SRS aligned with CRS - All CRS are transformed into SRS, checked for interface issues, checked for security implementation properties and tested whether SRS is defined properly or not. Also, tester evaluates SRS for software, hardware, firmware, virtual ware compatibility issues in SRS. The testing team evaluates every factor of the SRS, and whether the CRS is transformed accurately to SRS or not. It tests and confirms each end-to-end system testing scenarios in the SRS. Later, in next stage they review HLD- architecture where, it is tested for any violation of the architectural principles, guidelines, compliance's, rules, regulations, and violations which may lead to architecture debt. Testers also check for the integration of data between the modules, and interface issues in HLD. In the next stage, testers evaluate LLD (Low level design), test it from the functionality testing perspective, and check for the design principle violation which may lead to Design debt. The coding team starts coding the service product. The testing team experts cart tests, the code from black box testing tasks perspective. All the above processes are verification activities to avoid the bug creeping to into next stages in downward direction. After coding, the coders do code unit testing, also named as WBT or glass box testing or structural testing. It is also called transparent testing (path testing, loop testing, conditional testing, memory leakage testing, penetration testing, code security testing etc). Also, gray-box testing all this is verification activities to deliver the right artifact in precise time and accurately consign, with the right quality. Here, the coders verify each and all LOC (line of code) for correctness, syntax, grammar etc. after WBT. The CCS software (s/w) is launched to the testing team to do black box testing [BBT].

Advantages of V and V model includes: Verification is construction type activities and validation are destruction type activities performed to ensure product safety, and service safety. Here, testing starts at premature stages of cloud service development. This evades descending or surging or creeping of bugs, and viruses which in turn lessen a lot of rewriting, reworking and refactoring activities. Testing is performed at each stage of service/ product development, Deliverables take place in parallel. This model incurs huge investment since right from beginning, the testing activities are performed. It invites more documentation work. We go for V&V models in the following situations: for long-term service development, complex computing service offerings, and when a customer expects a very high quality of service within stipulated time frame, under marketing schedule pressures, and time schedules with service delivery on the fly.

3. CSTD CATALOGING

There are many facets that adds to CCTD. We catalogue CCTD based on the depth and brand of testing executed in a cloud environment.

3.1. Unit testing or White box testing (WBT)

In WBT various short cut methods are followed to execute the unit testing which gets added to CCTD debt. The unit testing for software coded service is done throughout the programming (coding) of an application or service. By isolating a sector of code and validating it for its correctness [21]. Normally, the developers/ coders who code their own module in a particular language are thoroughly aware of their own modules or components they have developed. But they are not aware of the other modules developed by other coders; also they are not aware of the same module developed for multiple platforms using polyglot programming

languages developed by other coders for other platforms. If the coder debugs and does WBT on the other modules or complete modules other than the ones he developed, then, the coder with negligent or does not have in-depth knowledge of the complete modules. If he executes the WBT script, it leads to CSTD. It is preferable that scripting and execution of WBT is done by test engineer, since he is well aware of coding standards, including the language platform in which the WBT scripts are scripted in multiple languages. Also he is well versed in debugging the source code and locating bugs using his own knowledge on business domain and technical scenarios and find the bugs without any bias and offer the justice to the customer. If the developer, does the testing there are chances of missing the scenarios and making the testing procedure using short cuts and make the residual defects to creep in the service leading to the CSTD. So, the test engineer as a separate entity with multiple programming, scripting and domain skill and end-to-end testing knowledge helps to lower the bugs from scooping with shallow and wide in-depth tests of the services [4]. Appropriate unit testing made throughout the programming stage saves both time and money in the end. If neglected with thrifty on WBT, it leads to more bug fixing costs during the Black box testing. Neglecting the WBT using shortcuts leads to accruing of Test debt.

- It leads to insufficient, pathetic and poor unit tests scenarios, missing scenarios, wrong scenarios, conflicts in the scenarios and test script source code, change in the scenarios, test cases and test scripts.
- Lacking to use automation tools support in cloud environment leads to test debt. In such a case, the concealed bugs linger for a long time. If the missing bugs are caught in black box system testing or user acceptance testing, it takes longer time to fix the defect incurring more cost for fixing the bugs.
- Ambiguous and vague unit test cases/scripts lead to difficulty in documentation and create difficulty in exploring the functionality of the CCS code under test.
- Cloud service production code under unit testing with exterior dependencies using public or hybrid cloud, should execute independently, by isolating external dependencies, failing which it results in slow execution of unit test scripts.
- Unrelated business need in SaaS app code trying to connect to, access the PaaS code and IaaS code in a virtual environment and the DSaaS code, etc. unit test as a result of external rationale network failure slow down the unit test, leads to accruing test debt. Inappropriate claim [1] in unit test scenario, wrong assumptions results in test debt.
- Just to create and impress the end user and management on the number mania the dev-ops team adopts a bad practice of writing the unit test cases without the obligatory claims to boost up and increase the unit test case coverage without adding any value to the test suite, and test framework. Some wrongly asserted testing, scenarios consume an extra time to debug and repair the source code.
- Failure to execute the unit test code in isolation from source code by using copy and paste, the function/ procedure/object of source code to its own unit testing code without scripting the code in testing environment to erstwhile in its natural environment. By separating the code facilitates to discover needless dependencies linking the code being tested and other units.

3.2. Exploratory or probing or groping or expert Testing

The expert testing (or innovative testing) and its consequences to test debt is reported in [23]. It is informal, and non-documented testing. It does not rely on a formal testing methodology. It is a random method of testing the application using the tester's domain knowledge and intuition, using creative, explored innovative scenarios, instead of formal test design strategies. The exploratory testing (ET) involves creative, unconstrained, unbounded, innovative, and cost-effective characteristics. It uses both negative and positive scenarios. And expertise testing is done by testers in security and penetration testing, vulnerability assessment

testing, to dig and mine the vulnerabilities and unearth the bug risks. Nevertheless, over relying on exploratory and ad-hoc testing can destabilize the documented and formal value and efficacy of the procedure and object testing strategy. ET is the process of detection, innovation, and research and learning. This underlines the individual tester freedom and accountability. Test cases/scripts are not documented well in advance, and are written on the fly over testing with its spontaneous creative thinking with system adaptability and learning. It fails to set up Bug Taxonomy, root cause analysis, test charter, and time box thinking.

The factors of exploratory testing or expert testing leads to the CSTD, Exploratory testing is informal and non-documented, which increases extra effort and resources in terms of CAPEX, and OPEX. As the complexity in cloud computing environment increases the coverage is difficult with the complex and distributed environment using the testing benchmarks. Testers use shortcuts to make a decision on progressive steps of action or pattern. In exploration testing, process of test plan, investigation, and design test suite execution, are completed as one and immediately. It is usually based on testers thinking progression, and creativity. In the ET there are no prescribed standards or planned test approaches to execute the test strategy. It is an adhoc testing helps in detecting hidden bugs. ET involves only a defect report / defect log [1]. ET is a concurrent and instantaneous progression of system test design and test implementation and executions are all performed in one short time. It is suitable for short-term testing, but not right for the longer execution time, and replication of failure scenario is hard, it is hard to decide when to stop testing the product, it incurs a vicious exploratory work and it adds to cstd some of them are listed below.

- Expert investigative testing involves huge testing effort, with increased CCS complexity. It is harder to follow the complete system coverage, and is hard to track the CCS under test.
- Testers cannot adapt well structured testing approach with inaccurate assessments. It is difficult to understand, track, manage, and watch the testing progression.
- Factors which affect exploratory testing are: testing strategy, existing tools and services resources to carry testing, tester's role, responsibility. Knowledge and skills (includes good listening, guiding, tracking, interpretation, judgment, opinion, documenting and coverage, rapid feedback, creative, critical thinking and assorted ideas).
- Re-execution or retesting or of tests suite oracle [1] or strategy or scenarios or scripts or frameworks or cases is complicated and pricey. Expensive results in undocumented and uncovered features and functionalities with high possibility of missing components or scenarios, along with schedule slippage and increasingly contribute to the CSTD. Exploratory testing is beneficial at tight schedule and non availability of the requirement specs, compared to procedural manual and automated testing methodologies.
- Some factors contributing to CSTD are: Inaccurately explored or wrong understanding of the application, CCTD is due to result of the execution is based on missing, wrong, and conflicting test oracle in exploratory testing suites, leading to extraneous rework and more residual defects affecting the longevity of the product or cloud service maintenance expenditures and leads to more repayment of the accrued debt due to missing features.
- Inexperienced and untrained test engineers contribute more to accruing CSTD, due to lack of domain knowledge or understanding the applications. They fail to test the application or service with effective approach and documented methodology. Due to lower test set suite and inconsistent test oracle inaccuracy, it results to addition of residual bugs [1].
- Non-existence of proper documentation work of SRS, CRS, BS or FS leads to CSTD, while, executing the exploratory testing based on available code. It jeopardizes the knowledge management with low implementation and maintenance costs and affects the speed and agility, and performance of the business.

- Due to poor documentation and proof reading past logs of test suite are not maintained, it's an unplanned testing strategy, causing CSTD. In order to avoid scheduling slippages the testers take shortcuts to meet the cut-off date leading to test debt.

3.3. Manual Testing in cloud computing system

It is a laborious complex, and monotonous process of executing the test cases. It is a course progression of finding, assessing, auditing, root finding and manually spotting bugs, defects, vulnerabilities, issues, errors, failures and security threats in cloud computing service software. In general, a tester manually executes the test cases as specified in the test specification based on requirement spec in the form of CRS, FS, BS, and SRS. The common cause which contributes to accumulation of of test debt from manual testing is: Limited testing due to the time schedule and resource constraints, marketing pressure, and management pressure to release the product early to the market. Due to this the tester may have to carry out his testing job without doing complete and ample tests execution. Hence, they use shortcuts and execute only selected subset of the tests to release the service with the possibility of not detecting the residual defects. Due to usage of inappropriate test case design techniques with out proper test planning and strategies leads to CSTD. The test cases is designed using various test cases design techniques with lot of discrepancies via a diverse mixture of data combinations to execute the test suite. Testers use the shortest route or path to execute the important test scenarios, identify and document in test set oracle with residual bugs flourishing in the CCS under test.

The qualities of the test cases depend on the re-assessment in quality control and finding the issues in test case for missing test cases, misplaced, misunderstood test cases. Due to time and market pressure, test engineers skip the tests review. Missing tests review and halt processing of finding vulnerabilities and increase the effort and cost expenditure of maintaining the test cases. The course of action of testing the behavior and system functionality against the requirement spec is known as BBT or functional testing: Over testing for all the junk values of test cases may lead to the CSTD, Under testing the application may lead to the CSTD, whereas optimized testing, and positive testing and negative testing are desirable.

The test case and test requirement should be 1) Unitary (solid, interconnected, cohesive) - address one thing at a time uniquely, 2) Complete - the test case should be fully stated without any missing information and test data. 3) Consistent -Test cases should be reliable, consistent and should not contradict with one another, maintainable. 4) Non-Conjugated and infinitesimal - the test requirement and test case should be atomic with only assured conjunctions. 5) Traceable - the test requirement should meet all stakeholders needs and is convincingly documented. 6) It should be concurrent, and up to date. 7) Unambiguous it should be concise and does not give a choice to technical jargon's, acronyms, etc. with facts, not subjective and objective opinions. 8) Mandatory - defined characteristic be ameliorated. 9) Verifiable, through one of 4 possible methods - inspection, demonstration, walk-through, audits, review's, sprint meetings, evaluations, test or analysis. Violation of the above characteristics leads to the CSTD.

3.4. Integration testing [IT]

Testing the data flow using the connection linking and bridging two or more features is known as integration testing. It requires exploring the requirements and knows the application, systematically about how each component and its feature works, and discovers all the potential scenarios. After prioritizing the scenarios, the tester should execute the scenarios bridging the user interface between the applications for testing the data flow. If the bugs are detected, they are reported to the dev-ops team to fix the bugs or defects. Here, we have to do both positive and negative values and type of integration testing. In IT, we do two types of integration testing 1. Incremental IT {IIT} IIT is of two types, namely bottom-up IT and top down IT. 2. Non - incremental combination Testing.

In these types of testing, testers, incrementally affix and conduit the components and test for the interface and information flow between the CCS application components called IIT. Whereas in top-down IT, testing starts from one component regarded as parent module bridging the interface with another module regarded as child modules. It is incrementally added to check the data flow between the bridged interface using the relationship between parents and child module and is called as top-down IT. In Bottom-up IT, tests initiated from earlier last child module to its earlier parent module by incrementally adding the modules and tests with the data flow between modules. In Non - incremental IT, whence there is a complex data flow and whence it is difficult to set up the parent and child relationship between the modules, we test all the possible ways of data flow between modules. It is known as Non-Incremental IT or Big-Bang testing method. Here, we coalesce complete modules and do IT. In this method whence there is a chance of missing to test the connection and difficulty to do the root-cause analysis of the bug occurrences and its origin. If any one parent or child module is under development then we simulate the non-resistant module to do IT we use simulator called as Stub which is a dummy module to send or receive the data and the connection created to check the data flow is known as driver. This creates the test environment and helps to communicate data flow and prepares and performs the analysis of the report and sends the report, this contributes to CSTD.

3.5. System testing (ST)

ST is an end-to-end testing by navigating through all possible scenarios in all vertical and horizontal directions by simulating the test environment like real-time production environment. Once the build arrives to the testing team after WBT, testing team tests all the basic features using smoke testing and then test new features done as functional or component testing, then does integration testing. And, retests all the fixed defects, tests the unchanged old existing features to make sure that it is not broken, in the new build and retests only fixed defects. Once System testing is started the least features are ready and all the basic functionalities are working. We test the product in the environment similar to production environment, when there is no critical show stopper, blocker bugs, and before the release deadline date.

Once the build is handed over to testing team the build is compiled, compressed and installed in testing server in cloud ecosystem and testers start doing the system test cases. When the new builds are given we should uninstall the old build and do installation of new build and execute the system test cases. After installing the build software and testing the application , the testing team finds blocker bugs and then it is reported back to developers, they immediately fix the bug within middle of the cycle and another modified fixed build comes within that cycle which is known as re-spin. Without fixing the blocker bug it is not possible for testers to start testing the software further. If there is more number of re-spins in a cycle it is a clear sign of the more test debts being created. The testers locate bugs and report it to the coders. Then coders fix the bug in source code, compile it and compress it to executable format and send the build to the testing team as a patch file. Test engineers install the patch file replace the code with a defect, with new patch file [which is a modified program]. The build in compressed format include: .zip file, .tar file, .rar file, .jar file, .war files. We execute test cases for a standalone application, web application, and client-server application. More number of patches is a clear sign of the accrued test debt.

3.6. Acceptance testing

End-to-End is testing done by end users before accepting the application when it’s delivered to the customer the testing executed from the client side. This ensures the software in real-time business scenario or situations. In this testing, we check for the test debts test for some features, complexity of the features etc. If the customer finds the test debt, and critical bug, developers fix the bug and testers test it and re-install the build, and patron performs the re-tests in terms of acceptance testing process and reuse the new fixed or changed the CCS software. This process is known as hot fix. It happens with in few hours or 1 day, and accrued test leads to more number of hot fixes and cost consumption will be more.

3.7. Cloud computing system software Automation Testing

Cloud computing system software programmed script based automation testing involves scripted testing frameworks stored in test suite based on manual test cases or test scenarios identified earlier. The automated test scripts are executed using automation tool and results are documented using the tools without manual effort, it documents, analyses the results and gives the output in required format. If the testing processes are lacking to use automation tool, it chips in to the CSTD. The cloud computing service needs the retesting and regression testing process whence the service code is modified, altered or changed for bug fixes, or for implementing the new customer requirements. Then, it requires automating the product or service to re-execute and re-test the service code/build with new changes, impact area, bug fixes, using the automated tool and test scripts. If the test scenario for automation testing is missing, it makes the automated regression testing, performance testing is difficult and complex, and contributes to CSTD. As there will be a probability of residual bugs remaining in the service it affects the QOS if not retested properly and thoroughly. This affects reputation of the service provider later if reported by the customer. The main contributors of automation test debt are:

1. Even though the cloud is suitable for the test environment, if the application in not configured properly it could lead to configuration anomaly. Test suite executed in such an environment would lead to erratic results of the application. Choosing proper language for multi-tenant computing systems with complex multi-behaviour of a computing system in multiple environment using diverse software, hardware devices, firmware, virtual-ware devices computing systems is called for the real computing systems due to multiple platforms, environments. In order to simplify the testers use the shortcut by skipping to execute the automation script in all real test environments of virtual systems, using emulators, simulators, real devices etc. which contributes to accretion of CSTD.
2. Failure to follow and non-adherence to testing, documentation, tracking and reporting standards, increases the CSTD.
3. Failure to update test cases, retest and re-execute, as per modified/changed, re-factored patches, updates, hot-fixes, re-spins and not fixing the broken test scenarios, decreases the quality of automated test cases contributes to CSTD.
4. If the test scripts are not coded, and scripted manually to write the test scripts, and instead usage of shortcut method of record and playback tools for testing the GUI has enormous drawbacks. If the environment or platform or the browser is changed the recorded script may not work. It may create the havoc of testing the application. It contributes for the accruing of CSTD via increased maintenance effort and re-factors the test script.
5. Control and administration of automation CSTD. Overly dependency of automation tools also contributes to the CSTD.
6. Automation testing is started once the system under test is stable and sound which is the prerequisite. The vital principle of automation testing is to lessen outlay over an elongated period to make sure that no fresh bugs are launched as the focused action executing the existing test cases.
7. In an agile ecosystem a regular sprint time taken in designing, coding, validation of designed script with existing information is captured after typically 1-2 weeks to complete its execution and but affording such period for an automation script is cumbersome and leads to test debt. Agile methodology responds quickly to customer induced changes in business spec and is prone to frequent changes, but automation does not lend itself to frequent changes in CRS which leads to test debt.
8. Selection of the automation tool for automation script execution is very crucial in terms of finance and tool support which includes legacy and open source tools, failure to select a suitable tool leads to test debt.

9. As Agile methodology highlights for regular open teamwork and united team interfaces with less limiting policies it affects cohesion and collision within the team and non-conducive to the automated test script execution.

4. MANAGING THE CLOUD TEST DEBT

Strategic debt is acquired intentionally for tactical and competition reasons (such as first to promote in market release versions). Planned debt accrues on purpose for rapid gains and is-intended to pay back in the interim. Unintended debt occurs accidentally due to lack of coder's proficiency in coding, or lack of alertness of TD. Incremental Debt: occurs due to Business pressures, time schedule, cost pressures, due to lack of testing process understanding, lacking to build loosely coupled architecture components. In addition, lacking paired collaboration, lacking in documentation, parallel development, delayed refactoring - to support future requirements, as a project or service evolves. Lacking in alignment to testing standards, features, frameworks, technology testing, lack of knowledge, lacking of ownership, poorer technical leadership, scope doping, un-managed scope dimensionality reduction and shredding the dimensions leads to CCTD. The CSTD is unavoidable in reality inevitable in cloud services that executes on fly schedule load, scalability and security constraints in the public cloud and hybrid cloud. Knowing the symptoms of debt cause helps to prevent the accruing debt. Last minute specification changes leads to CCTD. Interest payments, paying off the debt back is essential for future maintenance for in progress development of the upstream computing project. The first action is heightening alertness and understanding the test debt; by holding team meetings, trainings relevant processes to discuss and focus on repaying existing test debt. Recognizing, cataloguing, discovering, documenting, and tracking the test debt is important with prioritized effort to deal with it at first iteratively formulate and re-factor to reduce the test debt. Incentivize team and create the culture to control the CSTD.

Accrued Debt is the main reason for missing the deadlines in completion of the testing schedules. Cloud computing testing is executed on a web-based platform for various computing services like hardware, firmware, application software, virtual-wares and other computing services remotely accessed using the Internet. To resume control and cut the load of debt: craft a budget- assign and reserve funds for 'requirements'. Assess test debt- collect and congregate all debt statements measured and find owing debt measure, and interest rates to be paid back through refactoring this information will help to plan for payoff. Adjust time, cost, and effort spending to pay back properly in the correct quantity. Communicate with all the stakeholders to understand test debt and the goals to repay it back and take help to plan for the changes to be made, so they can fully support those changes. Accounting for a debt helps in changing the principle of debt accrual. Test Debt will be accounted for change impacts with the principal cost needed for concurrent repayment of the loan. Unamortized fees credits for accrued debt, repaying the debt is important as part of refactoring.

The entire cloud is visualized as an entire entity, based on its characteristics and user app service needs testing to be carried out. Testing within the cloud: cloud service suppliers/producers/vendors can execute testing within the own private cloud. Testing crosswise cloud: Testing cases are executed over all types of cloud models. SAAS testing in the cloud: it includes performing both functional and non-functional tests. The main focus of cloud testing is: Core component testing. The Apps include: functional testing, end-to-end business system and workflows testing, Information security testing, Platform compatibility testing include browser, OS, compatibility, etc.

Network testing includes: validating different communication network bandwidths, communication network protocols and data transfer rate over network validation, network connectivity, and data integrity testing etc.

Infrastructure testing includes: Backup and Disaster recovery testing, testing the secure connection systems and testing the storage policies, regulatory compliances testing etc. Further cloud testing includes,

performance testing, continuous availability testing, compliance testing, security testing [VPN testing, firewall testing, load and stress testing, scalability and elasticity testing, multi-tenancy support testing, testing for hot-fixes, updates, patches and upgrades, migration testing, Interoperability testing, portability testing, legacy system testing, open source compatibility testing etc.]

4.1. Challenges of cloud testing

It include: a) Security and privacy of data: Due to multi-tenancy nature, data theft risk persists. b) Updating of patches and up gradation of application software, platform and server maintenance and changes on short time notice to the customers. C) Difficulty in handling the interface version and compatibility. D) Difficulty in integrating and migrating of the inbound and out-bound data from the customer's network to SaaS point and vice versa for enterprise application.

Factors affecting Cloud testing debt are interoperable, Performance quality functions, Compatibility, and Usability. Difficulty in testing the end-to-end testing in a SaaS environment, simulating real-time and virtual online test data, over diverse computing resources, difficulty in creating test harness, and test library simulation, etc.

The main aspect in administering the CSTD subset is to track the CSTD and cyclically pay back the incurred debt in a time bound manner to keep it under control from its impact. The primary solution is to carry out attentive and realistic approach en route to assess and pay back TD. In some situations, it's beneficial to incur the debt as it allows the stakeholders to assemble their goals. For illustration it perhaps may not be possible to completely execute the test cases to avoid the schedule slippage as a shortcut to release the service to market. The testers may not address the finding and may not be able to track the detected bugs which incur the debt. It requires proper planning to tackle such issues in future releases and by monitoring and executing to pay back to settle the debt in a timely manner [1].

Another vital facet in the direction of controlling and supervising the CSTD is to avert and stopping the accrual of CSTD. Preclusion is achieved by mounting alertness through the Developers, Dev-ops and testing teams on CSTD and its impacts, pioneering pertinent ways, guidance, policies, bench marking the industry practices, training. Creating awareness helps to prevent the accrual of CSTD. For illustration, the clean room coding [9] [1], optimized coding practices, guidance and benchmarks are shared and allowed to follow for test coding practices as well. If it's currently not followed or pursued it may be fortified with regular reviews, walk-through, and inspections with looping feedback to improve the QOS in testing practices.

5. Comprehensive practices to pay back CSTD

We can control the accrued CSTD by repaying the CSTD cyclically over a period, by estimating the degree of CSTD and chart to repay it. But in practice it's not feasible and it's cumbersome and hard to stop performing the testing activities and concentrating only on paying back the CSTD. But we need the equilibrium with unbiased realistic and feasible way to pay back the CSTD by considering the account of involved cloud stakeholders, using the entities feasible with organizational and technical dynamics. We try to present below the major steps for repaying the accrued CSTD based on industry benchmarks practices from our industry experience, academic literature studies and analysis.

1. Measure and enumerate [22] the CSTD, after getting the approval from the management to execute the repayment of accrued CSTD. When the CSTD accrues over span of time, the technical stakeholders and the team entities may need to spend their committed effort to repay for longer duration with their continued effort, and focus attention to the extent of the depth CSTD sustained. They should make measurement and capability of the CSTD using manual ways to quantify the CSTD using Microsoft-excel sheets. This includes planning the schedule, with resources, and coordination required to repay the CSTD [1].

2. Recompensing CCTD cyclically- In the cloud computing services, when the TD has not attained crucial levels, it would be necessary for the involved stakeholders to repay the accrued debt in regular short instalments. It - is desirable and pragmatic as part of the end release progression and addition to the regular work. Using dev-ops test suite repository testers and development engineers trained and directed to re-factor the test cases before they check in the test script or test case to the central repository maintained to track the test cases.
3. Avoidance of the mounting CSTD: Planned advances for managing CSTD to avoid the rising TD in potential. Test Driven Development (TDD) as a strategic approach to managing the accruing TD involves writing the test cases using a strategic approach to write the test cases, execute test suite, either manually or using automation tools, and re-factoring the test scripts [24].

6. Premeditated Approaches towards Managing CSTD

Premeditated approaches towards managing CSTD for practical supervision of CSTD, strategic methods such as refactoring, test script and test suite, test plan, test framework, testing architecture, test strategy should be followed. Major Industry specific benchmarked engineering practices linking methodologies, in managing CSTD needs to be adopted.

6.1. Standard benchmarked efficient programming practices for writing test scripts

Development team codes the product offered as a service, in NIST defined cloud computing offerings as a software coded service such as Infrastructure as a service, network as-s service, database as a service, platform as a service code, aata storage as a service. But, the organization in cloud neglects the standards to follow the same procedure with due importance of the coding practices to the test script automation. For testing the product or service using automation, script involves more coding effort than the real coding service which needs to test the product on all platforms. So, the testing code is written from the end-user perspective. It involves testing both positive and negative conditions, and executed several times over continued time, duration, and it needs to be tested for all functional components, test conditions, test scenarios, cyclically iterated loops, test cases, etc. But, in reality the development team or test engineers or management in the organization never give importance to test script to test the service. It requires urgent need and attention across the industry to follow the standards benchmarked practices in managing coding, and to design and architectural planning to cut the CSTD.

6.2. Pair testing

In an agile way, time-boxed development of test scripts and test cases needs to be initiated. Agility fetches quicker pace of developing scripts, rationalized with varying customer needs. In short, time offered for testing is narrow; do anything accessible with efficiency and effectively execute the tests. It is a technique to be followed by a pair of people involved in testing activities to test the system under test (SUT) on the same environment or computing system by continuous exchange of the ideas, knowledge and experience. The pilot is accountable to do the tangible testing tasks, and actions executed. Whilst the co-pilot scrutinizes, rate, reassesses and directs the pilot. The extent of activity is fixed with established scope, measurable goal, executing actions as pilot and co-pilot by sticking to a goal. There will be no deviation from the extent of the scope without over complicating and the KISS principle should be followed.

Pros of duo Testing: this includes enhanced knowledge, skill sharing, innate test case/script coding and reviews, high accountability, better understanding of the domain with higher creativity with amplified productivity. Use of Superior testing strategy, plans, methodology, time boxed action with real-time activities. It also helps with healthier bug reporting and documentation, efficient knowledge sharing and training technique, better coordination and reproduction of the bug with tracking activity.

In this practice of coding, multiple test engineers who code the scripts to test the product or service are well versed in multiple domains, and platforms are aware and have the knowledge of the complete modules. They have expertise in manual testing by performing end-to-end testing, at different versions and releases of the service with in-depth knowledge about the product and with various programming skills in polyglot programming practices. The multiple testers collaborate using the multilateral collaboration team who all code together in multiple teams with joint team effort to design the test script to test the vast complex coded service in the cloud called as driver team. And other team by working alongside is called as a pilot team in parallel level to test script coding team to develop software. The Pilot team watches, and scrutinizes like a watchdog and assists the driver team by observing the script with mutual understanding and in-depth analyses observes and helps the driver team. Profits by using agile benchmarked practices embrace abridge effective team collaboration, and communication, and reduced bugs. It also increases the test coverage and analyzing the complete end to end scenarios and tests with better technological skills, increases the responsibility of the involved teams. In Pair programming involves war room strategy and follow standards in agile method and extreme programming methods, lean methods, and Kanban methods [25] [26]. Both coders as drivers, testers as navigators share the same room and code the service software. Testers test manually and write the test suite scripts to automate and test the code or service produced. If any issues are detected by test engineers, the coders instantly converse together and fix the issues, by both cross checking each other to suffice the bug, issues and resolve the matter cordially.

6.3. Clean room test script programming [27]

It is a hygienic coding practice in test scripting using good testing craftsmanship [1] [27]. It is very much necessary to follow the clean room programming practices for writing test script and store in test suite with clean, healthier coding practices to store, execute tests and update the defect. Defect reporting, documenting method, defect management using the clean room testing practices helps in managing CSTD. Clean room Coding [26] practices involve dynamic and static analysis, violation identifications, writing cryptic, standards, and following common coding fashion, maintained in common test code base stored in the test suite [37].

6.4. Re factoring

“Performance is defending program revolution and renovation of the transaction over a period as per demand and needs”, for writing the unit test script code, regression test script suite, load/performance testing script suite. Refactoring the test script involves the alteration(s) of test script code appending, adjoining to the existing test scripts, alterations or eliminations, delete conflicting test cases. It formulates and crafts the existing test script code to enhance, improve, with comprehensively have better coverage with better readability and easily maintainable. Refactoring [2] is principally executed as a good component measurement to the test script coding process. Every time the cloud service source code changed due to changes in the requirement, in business scenarios or due to bug fixing or due to enhancements etc. in order to have better test suite management, test code coverage and maintenance. The same stratum should be followed for test script code refactoring. The entire time test suite is pre-eminently executed every time it is handled. More the refactoring is neglected or delayed, the more test code be used in the present structure which will adds to the mounting CSTD.

6.5. Value based efficient Testing methods

It is a well planned, calculated, measured managing method involving value-based efficient, testing process to manage CSTD.

- Ten minute construct code build/product/release: Rapid with swift, naive, and total builds are vital for plummeting the spin time for structural changes to the service code in coders own local computing

machine for shipping service code to real-time production. A Thumb rule is to execute the whole complete engineering process at a time slice duly by consuming 10 minutes to complete all the processes of fabricating, design, assembling, testing and installing, and configuring, the cloud service software. Initially it starts from code compiling into an executable file, compressed to a zipped code with its linked libraries and its system files. Installation and configuration (registry set up, linking the files and unzipping the executable file, platform setup, driver setup, copying, and installing the files) into the test setup in cloud similar to production real-time environment and execute the tests, running tests, boot up and start of the processes is done.

7. CASE STUDY 1

This case study concerns a software coded service offered over cloud computing with complex architecture and design. This service it was executed in a cloud environment and worked for all Linux flavours, UNIX flavours, Windows version, and MAC OS. And also, in multiple browsers and their versions using multiple thick clients and thin clients with support Mobile OS's typically Symbian OS, Android, IOS, Blackberry OS, Microsoft Windows versions, flavours, etc. The application is offered as a service over the cloud having 1000's of API's, assorted functions which have to be invoked in command line argument. This is executed in client as front end and on the back-end servers. The application service was offered as a service over the cloud. The service was released every three to six months with added changes in the code in every release. The application was coded in CPP, Java, Python, Ruby, grails, groovy and Go Lang and other various polyglot programming languages. There were various field experts in testing, leads and testers associated with their roles in the organization. The test lead wrote test strategy, plans and the tests in a test matrix executed, it was created earlier and was reused for current project which has not changed over the time. The Manager checked and measured the progress by test cases executed by comparing with the earlier report.

At that time the tester was assigned to run 10 tests, he executed the scripts in the vicinity. The tester did not execute any of them as he noticed the critical blocker bugs and a few other areas uncovered which are risky. The lead noted it and objected that there was no progress and none of the test cases are executed as the test matrix was approved and he did not have any powers to ask for the change of scope of test to cover the risky areas in the scope. Newly hired fresh graduate test engineers without domain knowledge and testing expertise was deputed to test the product with short time. The tests were executed for a few days without clarity. Later, they realized with experienced and followed an exploratory test approach by letting the tester's freedom, with responsibility and accountability for their work. The testing sessions were scripted using test matrix over time based on new risk threats were noted.

The coding process used waterfall model, with only Black box testing done manually, when the new features were added to the existing product, more bug fixes were incorporated, without WBT in the software. No tools or advance techniques were used to improve the quality of the code/product with obsolete test infrastructure. Delayed new hardware procurement was arduous and time-consuming in clouds due to the start-up nature of the organization.

In the organization where we worked they were using the automated scripts initially using selenium TEST NG, POM Apache, test engine and other hybrid frameworks. They also used GTest and Perl scripts, with exposed API. CloudTest, SOAtest, HP Load Runner, Microsoft Azure, SOASTA, LoadStorm, CloudTestGo, AppPerfect, Cloudslueth, CloudTestGo Jmeter etc. Unit tests were executed using CppUnit, Cucumber and Robot, SONAR ICube, its open source code reporting tool to follow the degree and depth of code roofed by the tests.

Later, management realized that the co-operation between the lead and the test engineers needs to be established, with everyone in the team involved freely with accountability to do the testing tasks in making

the plan and changing it over time. As more bugs were found systematically the team took pride over testing, with the updated test and bug reports and artifact delivered in good shape with clear hierarchy.

Having the traditional waterfall model does not fit well in agile without clarity. It is a bottleneck with a weight for the corporation without clarity is a burden. When the tester stops caring, he starts using the short cuts and ignores areas, even worsens the job of testing towards the negative missing valuable point. It causes friction between end-user and service vendor distrusting each other. Boasting an investigative test approach: Using the chaotic plans and charted terrain helps to adapt the flexible team and build available test and plan. Being quick and agile makes it is easier to start executing the test cases. By decreasing the test debt with team agility, composition and flexibility and team spends time on achieving the progress, gaining momentum in the work. If new bug/risks are detected, they add to the charter missions, and contribute by identifying new risk making the product under test more secured.

The focal crisis that worried team and the administration was that lots of bugs existed in the computing system. Due to complexity, the Coders faced difficulty in making alterations due to no automated regression tests in place. Tests are carried out using manual BBT, incurring considerable amount time and working effort. Lack of automation tool for performing the unit/part serviceable test adds to the existing CSTD. The progression was dispiriting to build up fancy cloud software code service due to lack of automation tool sets, principles, actions, quality gauging measures for quality perfection. After apprehending the need of CSTD, the management decided to use the automation tool. These changes took more time to use and execute. This deferred the next service version release to the cloud market. These transforms were lucrative to the customer service with reduced bugs. In addition, it helped the coding and testing team to gain confidence with security implementation was met. With automation tool in place, the testing team got an opportunity to increase the skill by learning the new automation tool, and helped to find new risky defects. This in turn helped to decrease the test debt and increased the stability of the product, and the team had enough time to focus on newer risky defects using the exploratory method. The usage of the automation tools provided opportunity to use modernized cloud test infrastructure. This case study provides the knowledge of the test debt and its impact by overcoming it using the automated tool driven strategy and successfully repaying the debt using the exploratory testing, maintaining the agility with dev-ops team.

7.1. Case Study 2

The testers in a cloud agile dev-ops team must attempt to do it in a day or maximal two days, ahead of the developers. Continuous Integration is the key for agile model. The test team ought to keep up an automated and planned testing framework to readily integrate the new component, block, features, and modules as business demands to fire the testing process in one go along with the existing one. The newly added functions, also the bug fixed components, corrected build or release is expected, the test scripts and test data associated and absolutely assorted, using a common test scripting language framework, common defect reporting and documentation with rapid feedback, in acute “face to face” communication. Cloud computing is a new panorama of prospect requires massive, dedicated infrastructure and resources. CCS requires increased business complexity with technological innovation using virtualization, set-up costs, with operational flexibility and scalability with less expensive test environments. Cloud-based testing has an issue with respect to information security, lacking standards, hybrid and public clouds and other technical challenges Client-server and web-based testing the diversity amid client-server testing, CCS are broadly divided into two types: 2 and 3 - tiered application services offered over the cloud.

Considering the specific case to make PaaS as vulnerability testing ground**:** *Construction of a cloud security test bed is difficult, to assess, amalgamate, combine and define various security testing perceptions in conflicting scenarios. PaaS testing ecosystem, is structured as a root for a security testing model to appraise, assimilate, and label any security testing concepts, by linking concepts to PAAS scenarios. In the*

app's development phase tracks the application on the PaaS beginning from requirements through design of cloud service programming and security testing to operation stages. The risk-control phase conducts the trails of the processes of extenuating hazard from discovering infrastructure and assets to execute lucrative maintenance. Risk is the chance of threat cause which leads to the lone or extra vulnerabilities.

The business progression phase allows the programmer/coder to control and protect apps in each PaaS stage. PaaS testers recognize risks in application progression, and then craft risk-based loom to security testing, including vulnerability assessments. Security testing model**:** Identifying the weakest link, in the chain series by conducting pen testing, and relying on principles and frameworks in deficient techniques to realize and name these three types of security issues: a) Security defect/flaws, at the PAAS apps design stage. b) Security creepy-crawly bugs at the PAAS apps execution stage. c) Resource outages at the PAAS platform stage.

Penetration testing**:** The penetration test conjures up the PAAS under attack. It tests the abuse of attacker originated and detected vulnerabilities to expand auxiliary access to the PAAS and IAAS to get admission to secret data access, and causes miss influence information records veracity, harms accessibility of the CCS service, and harms the platform. Penetration testing is performed using automated tools and scripts. It helps in testing the ability to react to the attacks and enact rightly. A security audit reveals threshold levels that are not in place despite the earlier security tool approach to correct the problem.

Security review process: Compliance verification: Security review involves the process of verifying in-house security applications to CCS mechanism using walk-through, scrum meet-ups, group discussion, postmortem meetings, or retrospect- meeting, appraisal of the code or by assessment of CCS design and CCS architecture mapping the reviewer logs testing using tools such as Nessus, Wireshark, N map etc. and other tools.

Future Directions: *The CCTD as an allegory has attained faster and wider recognition both in academic and industry. CSTD is the part of CCTD, where the defects, flaws, errors, issues, failure, mistakes, vulnerability, miscalculations, misbehavior, concerns, malfunctions, stoppage, breakdowns, crashing malware's, problems occur. Security issues in CCS domain are regularly effortlessly subjugated by attackers, hackers, and ill motivated pupils to cause damage and harm the product or service. At present scenarios, various vulnerabilities are increasing and leading to serious security debts, in turn leading to future threats in CCS. Defense testing endeavors to depict those vulnerabilities, violating circumstances reliability; input legitimacy and logic, accuracy close to the angle of attack vectors with the aim to abuse these vulnerabilities to curtail threat of safety infringes and guarantee confidentiality, privacy, discretion, reliability and accessibility of customer transactions. This work helps academics, students, researchers, industry practitioners and security expert's awareness to work on CSTD leading to CCTD, and its dimensions, including code debt, architecture debt. "Test smells", leads to the CSTD, in this vicinity, test readiness frameworks and methods, where cloud testing community would tackle this CCTD. From an end-user perspective - attack resistant product, better quality software, minimizes extra cost. testing - [CCS security requirements, service test plans, architectural testing, design base testing, test scenarios, test defect review], CCS security testing, finally CCS service deployment and maintenance helps in reducing CSTD[15] [16].*

ACKNOWLEDGEMENTS

We thank Ganesh Samarthyam for timely advise and needful suggestion to complete this work in timely manner.

REFERENCES

[1] Ganesh Samarthyam , Mahesh Muralidharan, Raghu Kalyan Anna, "Understanding Test Debt", Trends in Software Testing, Springer, 17 june (2016).

[2] W. Cunningham, The WyCash Portfolio management system, experience report, OOPSLA (1992).

[3] Jim Highsmith, "Zen and the Art of Software Quality", Agile2009 Conference, (2009).

[4] Z. Li, P. Avgeriou, P. Liang, "A systematic mapping study on technical debt and its management". J. Syst. Softw. (2014).

[5] R.C. Martin, Clean code: a handbook of agile software craftsmanship, Prentice Hall, USA, (2009)

[6] K. Pugh, The risks of acceptance test debt. Cutter IT J. (2010).

[7] G. Suryanarayana, G. Samarthyam, T. Sharma, Refactoring for software design smells: managing technical debt (Morgan Kaufmann/Elsevier, (2014)

[8] K. Wiklund, S. Eldh, D. Sundmark, K. Lundqvist, Technical debt in test automation. IEEE Sixth International Conference on Software Testing, Verification and Validation (2013)

[9] A. Qusef, G. Bavota, R. Oliveto, A.D. Lucia, D. Binkley, Scotch: test-to-code traceability using slicing and conceptual coupling. Proceedings of the 27th IEEE International Conference on Software Maintenance (2011).

[10] R.L. Nord, I. Ozkaya, P. Kruchten, M. Gonzalez, "In search of a metric for managing architectural debt", Joint 10th Working IEEE/IFIP Conference on Software Architecture (WICSA) and 6th European Conference on Software Architecture (ECSA), Helsinki, Finland, August (2012).

[11] Tushar Sharma, Marios Fragkoulis and Diomidis Spinellis, "Does Your Configuration Code Smell?", 2016 ACM. ISBN 978-1-4503-4186-8/16/05. MSR'16, May 14-15, (2016), Austin, TX, USA

[12] Grady Booch, Ivar Jacobson & Jim Rumbaugh (2000) OMG Unified Modeling Language Specification, Version 1.3 First Edition: March 2000. Retrieved 12 August (2008).

[13] Jon Holt Institution of Electrical Engineers (2004), UML for Systems Engineering: Watching the Wheels IET, 2004, ISBN 0863413544. p.58, (accessed on 30-11-2016)

[14] Jonas Söderström. "Onceability: The consequence of technology rot", (accessed on 30-11-2016).

[15] Fowler, Martin (October 11, 2007). "What Is Refactoring". (accessed on 30-11-2016).

[16] Amr Elssamadisy, Jean Whitmore, "Functional Testing: A Pattern to Follow and the Smells to Avoid", ACM conference Proceedings, (2006).

[17] A.D. Leon, M.F. Moonen, A. Bergh, G. Kok, "Refactoring test code". The Technical report (CWI, Amsterdam, The Netherlands, (2001).

[18] A. Martini, J. Bosch, M. Chaudron, "Architecture technical debt: understanding causes and a qualitative model", 40th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA) (2014)

[19] A. Qusef, G. Bavota, R. Oliveto, A.D. Lucia, D. Binkley, "Scotch: test-to-code traceability using slicing and conceptual coupling", Proceedings of the 27th IEEE International Conference on Software Maintenance (2011).

[20] Jon Holt Institution of Electrical Engineers (2004), UML for Systems Engineering: Watching the Wheels IET, 2004, ISBN 0863413544. p. 58, (accessed on 30-11-2016)

[21] S.M.A. Shah, M. Torchiano, A. Vetro, M. Morisio, "Exploratory testing as a source of technical debt", IT Prof. 16, (2014).

[22] K. Beck, Test-driven development: by example, Addison-Wesley Professional, USA, (2003).

[23] G. Campbell, Patroklos P. Papapetrou, SonarQube in action, Manning Publications Co., USA, (2013).

[24] L. Williams, R.R. Kessler, "Pair programming illuminated" , Addison-Wesley Professional, USA, (2003).

[25] Alister Cockburn, L. Williams, "The costs and benefits of pair programming. Extreme Programming Examined" (2000)

[26] S. Mancuso, "The software craftsman: professionalism, Pragmatism, Pride", Prentice Hall, USA, (2014).

[27] W.F. Opdyke, Refactoring object-oriented frameworks, Ph.D. thesis, University of Illinois at Urbana-Champaign, Illinois, (1992).

[28] S. Nachiyappan, S. Justus: "Cloud Testing Tools and Its Challenges: A Comparative Study", 2nd International Symposium on Big Data and Cloud Computing (ISBCC'15), Elsevier Science Direct - Procedia Computer Science volume 50, pp. 482 – 489, (2015)

[29] Matias Waterloo, Suzette Person, Sebastian Elbaum, "Test Analysis: Searching for Faults in Tests", 30th IEEE/ACM International Conference on Automated Software Engineering, pp. 149- 154, (2015).

[30] Arvinder Kaur, Kamaldeep Kaur and Shilpi Jain, "Predicting Software Change-Proneness with Code Smells and Class Imbalance Learning", Intl. Conference on Advances in Computing, Communications and Informatics (ICACCI), Sept. 21-24, 2016, Jaipur, India, (2016).

[31] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, "An Empirical Investigation into the Nature of Test Smells", ACM, ASE 2016 Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, pp. 4-15, (2016).

[32] Fabio Palomba, Dario Di Nucci1, Annibale Panichella, Rocco Oliveto, Andrea De Lucia "On the Diffusion of Test Smells in Automatically Generated Test Code: An Empirical Study", SBST'16, ACM, Proceedings of the 9th International Workshop on search-Based software testing pp 5-15, (2016).

[33] Gabriele Bavota, Barbara Russo, "A large-scale empirical study on self-admitted technical debt", ACM, Proceedings of the 13th International Conference on Mining Software Repositories, pp. 315-326, (2016).

[34] Amjed Tahir, July 2015, PhD thesis on: "A Study on Software Testability and the Quality of Testing In Object-Oriented Systems", UNIVERSITY OF OTAGO.

[35] Jerry Gao, Xiaoying Bai, and Wei-Tek Tsai, "Cloud Testing- Issues, Challenges, Needs and Practice", Software Engineering : An International Journal (SEIJ), Vol. 1, No. 1, pp. 9-23, SEPTEMBER (2011).