ABSTRACT

Title of Dissertation:	DYNAMIC PROGNOSTIC HEALTH MANAGEMENT FOR RESPONSE TIME BASED REMAINING USEFUL LIFE PREDICTION OF SOFTWARE SYSTEMS
	Mohammad Rubyet Islam, Doctor of Philosophy in Mechanical Engineering, 2022
Dissertation directed by:	Dr. Peter A Sandborn Professor Department of Mechanical Engineering

Prognostics and Health Management (PHM) is an engineering discipline focused on predicting the future point at which systems or components will no longer perform as intended. The prediction is often articulated as a Remaining Useful Life (RUL). PHM has been widely applied to hardware systems in the electronics and non-electronics domains but has not been explored for software applications. While software does not decay over time, it can degrade over release cycles. Software

degradation is a common problem faced by legacy systems. Today, software health management is confined to diagnostic assessments that identify problems. In contrast, prognostic assessment potentially indicates what problems will become detrimental to the operation of the system in the future. Relevant research areas such as software defect prediction, software reliability prediction, predictive maintenance of software, software degradation, and software performance prediction, exist, but all of these represent diagnostic models built upon historical data – none of which can predict an RUL for software.

This dissertation addresses the application of PHM concepts to software systems for fault predictions and RUL estimation. Specifically, this dissertation addresses how PHM can be used to make decisions for software systems such as version update/upgrade, module changes, rejuvenation, maintenance schedules, and abandonment. This dissertation presents a method to prognostically and continuously predict the RUL of a software system based on usage parameters (e.g., the numbers and categories of releases) and performance parameters (e.g., response time). The model developed in this dissertation has been validated by comparing actual data generated using test beds. Statistical validation (regression validation) has also been carried out. A case study is presented based on publicly available data for the Bugzilla application. Controlled test beds for multiple Bugzilla releases are prepared to formulate standard staging environments to populate relevant data. This case study demonstrates that PHM concepts can be applied to software systems, and RUL can be calculated to make decisions on software management.

DYNAMIC PROGNOSTIC HEALTH MANAGEMENT FOR RESPONSE TIME BASED REMAINING USEFUL LIFE PREDICTION OF SOFTWARE SYSTEMS

by
Mohammad Rubyet Islam

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2022

Advisory Committee:

Dr. Peter A Sandborn, Advisor and Chair
Dr. William Regli, Dean’s Representative
Dr. Mohammad Modarres
Dr. F. Patrick McCluskey
Dr. Abhijit Dasgupta

Dedication

To dear God

To my dear son, Adam, who is the world for me

To my parents, who gave me everything they could have

To my advisor, Dr. Peter A. Sandborn, who guided me with patience

Acknowledgment

This Ph.D. has been an incredible journey for me. Though it took me a while to reach this day, it was a great learning curve. While studying and working full-time has been challenging, there have always been opportunities to learn. Such opportunities exposed me to a massive amount of real-world data. At the same time, helped me to understand how this research could be useful for real-world problems. On top of that, this journey made me realize how little I know and that I never should stop learning.

Special thanks to my academic advisor, Professor Dr. Peter Sandborn, for all his support, without which this journey would have been more challenging.

Thanks to all the government agencies that I have worked for, such as the US Department of Defense (DOD), General Services Administration (GSA), US Department of Homeland Security (DHS), US Depart of Justice (DOJ), US Department of Commerce, The National Aeronautics and Space Administration (NASA), US Air Force and others. Also, thanks to all the private entities that I worked for, such as Deloitte, Lockheed Martin, Raytheon, and others. Thanks for their support including but not limited to access to their massive real world data pools, tuition assistance and flexible work hours.

Of the above, I am very much thankful to my creator.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ANN	Artificial Neural Networks
AUC	Area Under Curve
CBM	Condition Based Maintenance
CPSQ	Cost of Poor Software Quality
CPV	Cumulative Predictive Variable
IF	Impact Factor
ISWHM	Integrated Software Health Management
LR	Linear Regression
MTTF	Mean Time to Failure
MTTR	Mean Time to Repair
MTBF	Mean Time Between Failure
NLP	Natural Language Processing
OS	Operating System
POF	Physics of Failure
POFOD	Probability of Failure on Demand
PV	Predictive Variable
PHM	Prognostics and Health Management
QTP	Quick Test Professional
ROC	Receiver Operating Characteristic
RT	Response Time

RUL	Remaining Useful Life
SDLC	Software Development Life Cycle
SFMEA	Software Failure Modes Affect Analysis
SMLC	Software Maintenance Life Cycle
SP	Story Points
SRLC	Software Release Life Cycle Management
SVM	Support Vector Machine
WF	Weight Factor

Chapter 1: Introduction

Systems failure results in the need for expensive and time-consuming maintenance actions. Systems fail due to a myriad of causes that are linked to more than just their hardware components. System failures due to software result in significant economic impacts impacting many people, Table 1.1.

Table 1.1: Typical Cost of Software Failures in 2018 (Krasner, 2018)

Failures Studied	606 from 314 companies
Financial Loss	US $1.7 trillion
People Affected	3.6 billion
Lost to Downtime	268 years

Again, the total cost of poor software quality (CPSQ) in 2020 (Figure 1.1) in the USA is $2.08 trillion among which legacy system problems contributed to $520 billion

Figure 1.1 : Total CPSQ in the US is $2.08 trillion (Krasner,2021)

(Krasner, 2021). To be noted here, the CPSQ is the cost associated with poor quality software products. It is the total financial losses due to reasons such as software failure (failure to operate in an effective way, i.e., ineffectiveness), maintenance, re-work. The primary cause of CPSQ was software faults. The second largest reason was failures (ineffectiveness) in the legacy system.

In prognostics, the future state or remaining useful life (RUL) is predicted based on current or historical conditions (Ly et al., 2009). A PHM system utilizes the integration of advanced sensors and prognoses, supervises, and manages the state of products with various algorithms and intelligent models. The four key attributes of a Prognostic and Health Management (PHM) system are the abilities to do fault detection, fault isolation, fault prognosis, and prognosis of the remaining useful life. PHM systems can conspicuously decrease the expenses for maintenance, usage, and support, and improve the safety and availability of products/systems (Xu et al., 2010).

PHM uses real-time and historical data on systems, subsystems, and components and provides actionable information that enables intelligent decision-making for improved performance, safety, reliability, and maintainability. PHM's goal is to provide decision support, and actionable information to assist in system-support decision making. While diagnostics detect and isolate faults or failures, prognostics predict the future state or remaining useful life (RUL) based on current and/or historical conditions (Ly et al., 2009). Prognostics is based upon the understanding that systems/products fail after a period of degradation, which, if measured (via an RUL), can be used to prevent system breakdowns, avoid collateral damage, and minimize operation costs (Tian et al., 2012). Data on degradation parameters is used to regress selected mathematical model(s) and to extrapolate the model(s) to a pre-determined effectiveness (failure) threshold. The estimated RUL is the difference between the time (or other appropriate usage parameters) at the current point and the point at which the extrapolated behavior exceeds the effectiveness (failure) threshold (McCarter et al., 2015). Prognostics is focused on predicting the time (or other appropriate usage parameters) at which a system or component no longer performs its intended function. The inability of the system to perform its intended function is often a failure of effectiveness beyond which the system can no longer be used. The predicted time-to-failure (effectiveness) is referred to as the remaining useful life (RUL) (Jaiswal & Malhotra, 2018). The remaining useful life (RUL) represents the useful life left in an asset at a particular point in its operation. The estimation of RUL is central to condition-based maintenance, prognostics, and health management.

A significant body of research exists on software defect prediction, reliability prediction, predictive maintenance, usability, and other similar topics. However, no research exists on the application of Prognostic and Health Management (PHM) or Remaining Useful Life (RUL) to software systems. Also, all the relevant predictive models developed in the past for software are diagnostic in nature (not prognostic). PHM is still an emerging field, and much of the published work has been exploratory or very limited in scope. Diagnostics are essential for successful prognostics because an acceptable prognostic method starts with robust diagnostics since the uncertainties of the estimated system condition affect any future prediction (Hess et al., 2005; Patrick et al., 2009), while diagnostics detects and isolates faults or failures. However, there are many challenges with diagnostics due to problems with verification and validation. Fault diagnosis requires system (or component) usage information and diagnostic search strategies that can match the observed symptoms with a known set of possible failures. Furthermore, robust diagnostics are needed for incipient fault detection to avoid breakdowns. The ability to diagnose component faults in their infancy is currently limited, partly due to sensitivity to noise in data, dependence on environmental and operating conditions, lack of fault detection (Patrick et al., 2009), and uncertainties in maintenance schedules.

PHM reduces time and cost for the maintenance of products or processes through efficient and cost-effective diagnostic and prognostic activities. For example, PHM can indicate when allowing an element to break down may be the best option (Takata et al., 2004). The only way to minimize the probability of failure, downtime, and maintenance cost is with Condition-based Maintenance (CBM), motivating the use of prognostics (Kothamasu et al., 2006). PHM incorporates logistics, safety, reliability, mission criticality, and economic viability, among others (Saxena et al., 2010). Prognostics is essentially a condition-based estimation of RUL to make better-informed maintenance decisions. PHM can produce a continuous real-time estimation of the RUL (Patrick et al., 2009). It decreases the cost of maintenance, usage, and support, reduces the probability of accidents, and greatly improves the broad application of condition-based maintenance (Xu et al., 2010).

The relevant aspects of PHM include assessing the effects of the operational readiness, life-cycle cost, manpower, and training caused by the PHM system scheme, assessment of demands and constrained conditions of performance supervision, fault prognosis, life tracking, and maintenance subsidiary information. These are to ensure that they are completed and coordinated, evaluate the inherent rules of the PHM system design and selection principles of weight coefficient, assess the selected PHM system technique by design, and used design guidance or programs. However, there are limitations with prognostics as well. Some failures (ineffectiveness) are intermittent and hence difficult to predict (Sun et al., 2010). Therefore, there is still no universal prognostic methodology (Lee et al., 2011). Despite being a very challenging part of PHM, prognostics is also one of its most beneficial aspects (Hess et al., 2005). Current prognostics technology is considered immature due to the lack of uncertainty calculations, validation and verification methods, and risk assessment for PHM system development (Saxena et al., 2010). Perhaps the main challenge for prognostics is that there will always be a limit to the accuracy and precision of condition-based estimation of RUL due to the inherent uncertainty of predicting the future. The PHM customer is mainly concerned with the return on investment (ROI) for instituting a PHM system (Banks & Merenich, 2007). Perhaps the most difficult aspect of PHM development is determining what system components should be monitored for prognostics, a step that significantly affects the system design. Based on current capabilities, other critical challenges include real-time diagnostics and prognostics, PHM system evaluation standards, and data integration (from sensors, experts, etc.) within user-friendly PHM systems.

1.1 Software Degradation and Failure

Although no known research exists on the application of PHM to software systems, relevant research is available on the diagnostic health management of software systems. Chapter 2 discusses how the application of PHM to software systems lacks the ability to calculate (or even define) remaining useful life (RUL). The application of PHM in software systems requires identification of appropriate target and predictive parameters and usage parameters.

Technically, software does not decay over time but can degrade over release cycles. Software degradation, also known as software decay, aging, rotting, smell, degradation, code rot, bit rot, software entropy, and software erosion, represents a slow deterioration of software performance over time or its diminishing responsiveness that eventually leads to software becoming faulty, unusable, or legacy, and in need of update/upgrade. Software degradation is a common problem faced by legacy systems (a legacy system is a system that is based on methodologies, processes, architectures, technologies, parts, and/or software that is out-of-date, i.e., old). In addition, software degradation creates the need for updates to accommodate the changing environment in which the software resides.

Software decay can even happen because of unused or leftover code that may contain faults that were not resolved during the changes made in the code base, environments, and software components. Other reasons include the lack of software updates/upgrades, improper maintenance, incompatible changes in architecture, inappropriate integrations, memory leaks, data corruption, unused code, and others. Since PHM can prognostically measure decay/degradation over time and software decays/degrades due to changes over releases, with the use of appropriate parameters and definitions, PHM should apply to software systems. Software decay is the gradual deterioration of performance and functionality over time. Many of the previously discussed methods, such as reliability, maintenance, defect prediction, predictive maintenance, and degradation predict software failure and failure rate. They all serve the purpose of predictive modeling for software degradations. They do not compute RUL. However, unlike hardware, software faults do not develop over time. Instead, they are introduced over releases at every stage of the software life cycle. Hence, the *release cycle* is the most appropriate *usage* parameter. The parameters that govern both software releases and software performance are the faults and enhancements.

Software failure corresponds to unexpected runtime behavior (ineffectiveness) observed by the users. At the same time, a fault is a static software characteristic that causes a failure resulting from incorrect design choices, simple human blunders, or the forces of nature (Trivedi, 2019). The terms bug and defect are often used to describe fault, error, or even failure but are not precise enough (Harris, 2018). In this dissertation, the term ‘fault’ and ‘failure’ are used instead of bug/defect to avoid confusion. Prediction of the RUL of software can save money by reducing unexpected failures and assisting in planning preventive maintenances and releases. None of the relevant research areas estimate the RUL of software. Most of the predictive models are based on diagnostic approaches and are built based on historical data. RUL can assist in many other decision-making processes, including but not limited to, the decision on version upgrades, addition of new modules or requirements, continuance in using the software or re-engineering or replacing them with new ones, planning maintenance, assessment of longevity of the software with projected updates/upgrades. Existing predictive models in the associated disciplines assume that the fault rate remains the same or increases. But in reality, the fault rate can also go down.

1.2 Dissertation Objective

The application of PHM to software systems is a new research area, and no existing work has been found that addresses the application of PHM to software to measure RUL. This dissertation aims to introduce methods to dynamically estimate the RUL of a software system using PHM models. The goal is to predict a software system's health condition and to identify the point (before the software fails) at which a decision needs to be made to update/upgrade, rejuvenate, re-engineer or abandon the software system. The tasks that are necessary to accomplish such objectives are:

Task 1 – Review relevant previous technical work and create definitions

- Investigate other associated research areas potentially relevant to the prediction of RUL for software, such as software reliability, software defect prediction, software maintenance, and integrated software health management (ISWHM). Explore the differences between these relevant research areas and their application to performing PHM for software.
- Define PHM for software and identify appropriate usage parameters and input and output variables.

Task 2 – Develop an RUL prediction methodology for software

- Develop a data quality assessment matrix to ensure that data is quality verified, and cleaned, as needed.
- Develop a categorization matrix to categorize input data into classes and sub-classes for better understanding for their numeric estimation and weighted impacts on performance parameters.
- The estimated values of the input variable(s) must be factored with weighted impact factors based on an impact scale to correctly adjust the estimated values of input variables.
- Architect future releases
- Perform clustering to group analogous releases.
- Select and train prognostic model(s) based on clustered data and predict the target variable(s) for future releases.
- Plot predicted target variables against usage parameters to estimate the RUL for software system.

- Identify the impacts of environmental variables, such as hardware configurations, operating systems, on the target variable(s) and estimated RUL of software

Task 3 – Case study

- Demonstrate the methodology with a case study.
- Collect historical data on both the input and output variables for the selected application (e.g., reported by user group and development team). Such data can be generated based on expert judgement and/or populated using automated/semi-automated tools.
- Follow the steps in Task1 and Task 2 to estimate the RUL.

This dissertation is organized as follows: Chapter 2 presents background on existing research carried out in related software domains. This section also discusses the differences between PHM in hardware and software. Chapter 3 proposes a methodology to prognostically estimate the RUL of software using a fusion approach. Chapter 4 presents a demonstration of the method using an open-source software system. Conclusions and future research directions are summarized in Chapter 5.

Chapter 2: Literature Review

This chapter reviews the literature associated with software reliability prediction, software predictive maintenance, software decay, software performance prediction, and software defect prediction methods. These existing research areas are all potentially relevant to predicting RUL for software.

2.1. Software Reliability

According to the IEEE Standard Glossary of software engineering terminology, software reliability is defined as the software's ability to perform its required function under stated conditions for a stated period of time (ANSI/IEEE, 1991). Software reliability is the probability of failure-free (effective) operation of a computer program for a specified period in a specified environment (Lannino & Musa, 1990). And according to ISO9126, software reliability is the capability of a software product to maintain a specified level of performance when used under specified conditions for a stated period (Malaiya, 2010; Hu, 2004; Brooksby, 2000; ISO/IEC, 1995)**.** Software Reliability Growth Models (SRGM) have been used to predict and estimate the number of faults in the software. Software reliability is an integral part of software quality assurance. Software reliability models help determine the probability of a system failure in a given time interval or the expected time span between successive failures. The presence of faults makes software unreliable (Jaiswal & Malhotra, 2018). Software reliability can have different meanings for different people. For example, a developer, who views code as instructions to hardware, may consider the software to be reliable if all the requirements for that software are executed properly. Whereas a user sees the same software as a set of functions and deems it reliable if nothing malfunctions (Pfleeger, 1992).

Prediction of reliability helps in avoiding failures and large disturbances (Patrick et al., 2009; Sun et al., 2010) and, thereby, increasing the reliability of systems. Reliability is focused on the probability of failure-free operation, and the primary goal is to identify the probability of a software system failing in a given time interval (mean time between failure) due to faults (Jaiswal & Malhotra 2018). However, software does not fail only due to faults but can also fail due to enhancements. Even though not all enhancements necessarily generate faults. Software does not decay over time, but performance fluctuates as enhancement requests are deployed, and faults are generated over releases. As a source of software failure, future enhancement requests are not accounted for in traditional software reliability analysis. The essential attributes for measuring software reliability include Mean Time to Failure (MTTF), Mean Time to Repair (MTTR), Mean Time Between Failure (MTBF), Probability of Failure on Demand (POFOD), etc. (Diwaker & Tomar, 2017). Software reliability is mainly calculated using static methods with historical (diagnostic) data that is collected in designated time frame. Alternatively, PHM estimates RUL based on both diagnostic and prognostic data.

2.1.1 Why Software Reliability is not PHM?

Software reliability is calculated as a function of time, whereas the PHM-based RUL estimation proposed in this dissertation is done as a function of release cycles. Software failure is not simply a failure to operate, but also a failure to meet required performance targets (ineffectiveness). For example, if software is required to respond within 5 seconds but takes more than 5 seconds, it is a failure - even if software responds with correct output. The priority of target performance parameters and their thresholds vary depending on application types, user requirements, and many other factors.

Even though multi-stage reliability models exist in the software domain, reliability does not necessarily estimate RUL by itself. Software reliability growth models assume that reliability growth is due to bugs being removed. In contrast, the RUL of software is impacted by future enhancement requests or the addition of new features that may not generate any faults. Reliability is measured using failures, whereas PHM addresses both the failures and enhancements. Reliability prediction is generally carried out before or during the software development or test phase, most commonly in the design stage. PHM and RUL estimation is carried out continuously at the production and maintenance stages by utilizing system health monitoring tools (Bellcore, 1990).

Software reliability estimation is carried out under stated conditions and within a specified time. Alternatively, PHM can define the time range in which a software system performs according to its performance requirements. Reliability engineering, traditionally, relies upon historical evidence of failures (ineffectiveness) that provide a limited and often inaccurate perspective of true reliability. PHM relies upon historical and prognostic data. PHM has become a leading approach in preventing failures (ineffectiveness) and improving reliability during operations (Modarres, 2017). Reliability improves when software faults are removed. In contrast, PHM-based RUL can improve without removing faults and by adding/editing software logic or hardware components. For example, just adding a high-capacity load balancer or removing a pdf converter can improve the RUL of software without removing any faults. Measuring the number of faults allows the reliability of the software to be predicted (Antoniol, 2012), and PHM is measured in combinations of faults and enhancements. Furthermore, although it may be possible to measure the entire system's reliability at any stage, making changes within the system may not be possible at all stages. On the other hand, the RUL of any software system can indicate future challenges. Based on which, planning and decision can be made to troubleshoot or make changes even before the problem occurs and extend sustainability of software system. Unlike reliability, PHM does not estimate failure-free operation over time. Instead, the goal is to identify how sustainable a software system is, assuming that there will be failures/faults over time. PHM assists in identification of stages at which a decision needs to be made for software to be re-engineered, updated/upgraded, or even abandoned.

2.2. Software Defect Prediction

Software Defect Prediction (SDP) predicts defects in modules that are defect prone and such prediction requires extensive testing (Arora et al., 2015; Shukla & Verma, 2015). Defect prediction is mainly a diagnostic approach based on historical data and is commonly performed during the software development phases and less common in the production environment. A critical objective for software defect prediction is better allocation of resources.

2.2.1. Why Software Defect Prediction is not PHM?

Defect prediction does not account for future enhancement requests and does not predict remaining useful life (RUL). PHM dynamically takes data inputs generated by sensors in the health monitoring tools and is reported by the users and the development team from the production environment. Therefore, defect prediction methods can potentially be used as one of the elements for RUL estimations. However, software defect prediction is based on historic data and does not compute RUL.

PHM estimates the current health state, predicts the future state along with point of failure (ineffectiveness) and determines the failure's impact on the system's performance. Defect prediction predicts future faults only. Whereas prognostics is a process of predicting the point at which a component will no longer perform a particular function. Defect prediction does not calculate RUL (Atamuradon et al., 2017) and considers defects/faults only, whereas PHM considers many other rationales, such as, asset and system degradation, environmental effects, failure behaviors, failure interactions, and related uncertainties (Qiao & Weiss, 2016). Defect prediction models need to run automatically or manually with pre-determined schedules, but PHM is a continuous monitoring and computation system. A defect prediction method includes fault checkpoints, whereas PHM uses sensors and thresholds (Yousef, 2014). PHM can be used for many decision-making processes that defect prediction does not accommodate, such as, if a version should be upgraded, a module should be added, when maintenance should be planned, how reliability could be impacted based on upgrades, and other activities.

2.3. Software Maintenance

Software maintenance is modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changing environment (Shafiabady et al., 2016). Software maintainability measures the effort necessary to keep a system operational.

2.3.1. Why Software Maintenance is not PHM?

Software maintenance is a different set of activities compared to PHM and is planned and predicted based only on historical data and software metrics. Software maintenance incorporates changes, where PHM monitors the system based on which predictive maintenance can be planned. RUL can positively impact the planning of maintenance activities, operational performances, and the profitability of the owner of an asset. Software maintenance does not calculate RUL. RUL estimation also impacts the management of product reuse and recycling. This puts the importance of estimating RUL beyond Condition Based Maintenance (CBM) and prognostics, and health management (Krasner, 2018). PHM can assist in making decisions on the re-usability and re-cyclability of software components.

2.4. Integrated Software Health Management (ISWHM)

Integrated Software Health Management (ISWHM) is a health monitoring system that monitors software using sensors to detect failures with probabilistic modeling like Bayesian networks (Schumann & Mengshoel, 2011). ISWHM is a diagnostic approach that does not address future enhancement requests. However, methods used in ISWHM to collect faulty data can be utilized in data collection for PHM.

To understand software degradation mechanisms, models are used to evaluate software decay (Alonso et al., 2011; Riaz et al., 2009). Albu and Popentiu-Vladicescu (2013) attempted to predict response time for web services and web applications. They measure response time/execution time at a specified point based on the existing system; however, they did not consider the influences of faults and future enhancement requests on performance parameters.

While there is work done on finding latent faults that have been resident in software since its original development, much less work exists on finding problems introduced after the software is deployed to the production environment. Once the software is fielded, faults are generally reported only when a problem arises instead of being continuously assessed. As of today, all the suggested and practiced software solutions are diagnostic in nature, not prognostic. (Vlog et al., 2016).

2.5. What are PHM and RUL for Software?

Like in other application domains, in software, PHM can detect and isolate faults, perform prognosis and predict the remaining useful life (RUL). RUL is the period during which an asset or property is expected to be usable for the purpose it was acquired (Krasner, 2018). RUL is defined as the length from the current time to the end of the useful life. (Si et al., 2011). The RUL for software is the remaining period in which the software system performs according to the standard requirements. To be noted here, there are no set performance standards for software in the industry, but there are best practices, and every software application has its own customer-defined requirements/product documents (de-facto standards). This research plans to utilize de-facto standards set by the users of software on which historical data is collected. However, user expectations on threshold can change over time based on user requirements.

Hardware products decay over time, while software code never decays. However, software products show performance decay and are affected by faults due to version upgrades, integration of new features and update of existing features, and changes in the associated hardware system. A prognosis on hardware can be carried out using decay rate, condition monitoring data, and failure mechanisms. Similarly, the prognosis of software can be carried out by finding the relationships between changes, e.g., version, environment, lines of code; fault rate; and performance decay. Research has been carried out on software failure mechanism, software failure mode affects analysis (SFMEA), and their impacts on software health, which has been referenced in this current research.

To start with, an ineffectiveness/failure threshold for software is decided by the user. The RUL for software is no different from that of hardware. However, how software "ages" is fundamentally different. Software ages by update/upgrade and not by calendar time or usage. Like hardware, software also uses sensors to monitor system health. Using embedded sensors, continuous diagnosis of software decay and prospective degradation rate (future state) can be identified. Based on the above uncertainties and based on the historical data, prognostic models can be built, and the RUL of software can be determined.

Chapter 3: Software RUL Prediction Methodology

This chapter describes the general methodology developed in this dissertation for predicting the response time based remaining useful life (RUL) of a software system. The methodological implementation details are provided in the case study in Chapter 4 and in Appendix A. This chapter begins with a review of the strengths and weaknesses of multiple PHM methods. It identifies the type(s) of methods that could be most appropriate for estimating RUL. Finally, an overall end-to-end solution to determine the RUL of software is described. This methodology addresses the collection of input and output variables, data quality assessment, data categorization, clustering (to identify analogous releases), prognosis, and RUL estimation. Also, methods are discussed to convert input variables that are in text/float format into numbers for the use in the prediction algorithms. A list of other performance parameters is discussed. A hybrid approach is introduced to identify algorithms to categorize analogous releases and predict target variables. The releases and the numerical relationships (impact factors) between the input and target variables are discussed.

3.1. Review of PHM Methodologies

The PHM process begins with dependability analyses to determine the components to monitor. PHM research has focused on analyzing data for fault diagnosis and failure prognosis, establishing condition metrics, seeded fault testing, and incipient failure detection (Ly et al., 2009). Diagnosis and prognosis are integral to the PHM process. Figure 3.1 shows a simplified taxonomy of PHM methods.

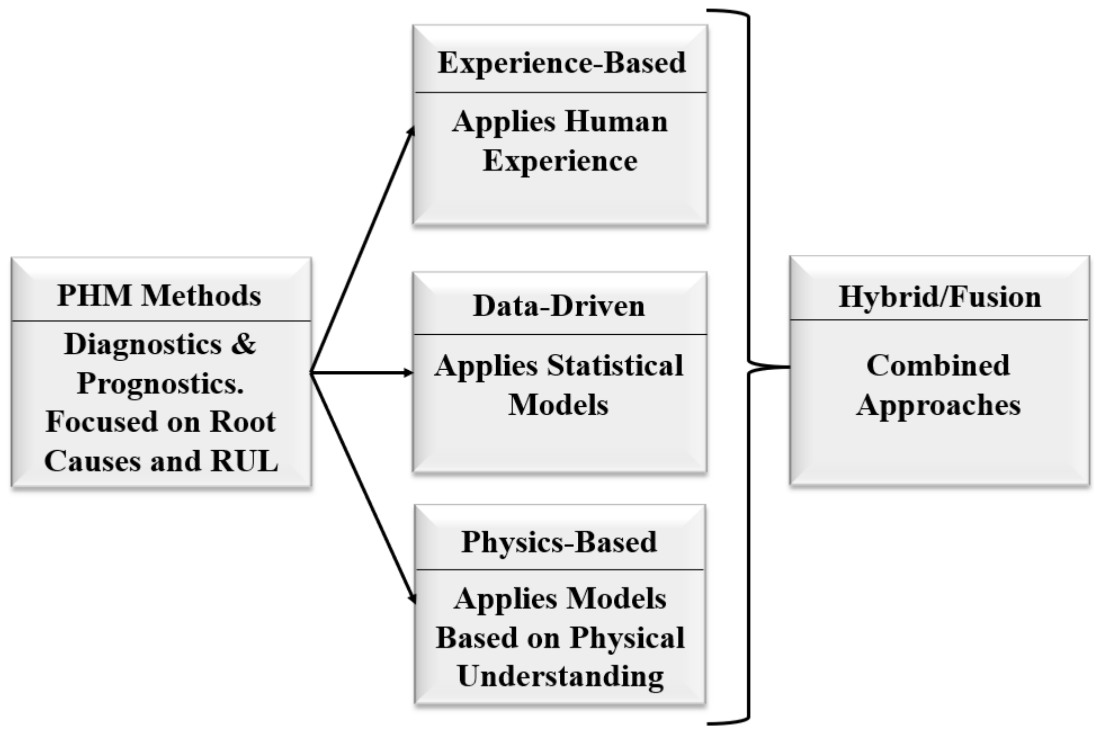

Figure 3.1: Simplified general PHM methods

As shown in Figure 3.1, diagnosis and prognosis can be modeled based on experience-based, data-driven, and physics-based methods. However, multiple methods are often combined depending on the targeted problems. PHM approaches based on experience, statistics or data, physics, or models all have pros and cons. Experience-based PHM uses human expertise for analysis and is the least complex but remains highly labor-intensive and expensive (Barajas & Srinivasa, 2008). Data-driven approaches could create non-linear relationships between inputs and outputs without physical models but are not necessarily convergent. One advantage of data-driven methods is that they can be applied at any level: system, subsystem, or component (Sun et al., 2010). Prognostics algorithms that use a data-driven approach learn directly from the data rather than using pre-determined models based on human expertise (Schwabacher, 2005).

Data-driven methods are based on statistical models, including machine learning (Sun et al., 2010). Machine-learning techniques include artificial neural networks (ANNs), fuzzy logic, support vector machine (SVM), and hidden Markov models (Sun et al., 2010), while statistical techniques are based on parametric or non-parametric methods. However, statistics-based approaches (reliability-centered, Bayesian, etc.) generally ignore correlations among various data (Barajas & Srinivasa, 2008). As a result, data-driven methods often yield a fault model that must be trained with data representing anticipated faults, which may be difficult to validate. Similarly, physics-of-failure-based (PoF-based) methods combine actual operational conditions with PoF models to calculate the accumulated damage as well as predict the RUL of the product (Sun et al., 2010). Physics-based (or model-based) prognostics is the most comprehensive modeling approach, utilizing various inputs but it can yield an RUL distribution as a function of component or usage uncertainties (Roemer et al., 2001). Also, physics-based approaches usually do not account for the combination of analog and discrete processes (Barajas & Srinivasa, 2008). However, the knowledge about a product (e.g., operational conditions) and its failure mechanisms required by skilled personnel to apply such methods is not always available. Another disadvantage is that

PoF-based methods are unsuitable for the system and subsystem levels (Sun et al., 2010).

3.2. PHM Analysis Applied to Software Systems

Venkatasubramanian (2005) concluded that no single method is adequate to handle all the requirements for a desirable diagnostic system. Ly et al. (2009) utilized a hybrid method that combines both physics-based and data-driven techniques for prognosis (Vogl et al., 2019). A hybrid/fusion method has been adapted for software RUL estimation. In this process, multiple methods are fused for diagnostics or prognostics based on their effectiveness. This may be effective in many cases to yield reliable just-in-time RUL predictions (Engel et al., 2000). Other factors must be kept in mind, such as feature analysis and condition metrics selection and extraction is essential for accurate and reliable fault diagnosis and failure prognosis (Ly et al., 2009). PHM works well when appropriate data is available, performance metrics are well defined and consistent, and predicted results are validated (Barajas & Srinivasa, 2008). Another critical best practice is incorporating the "human factor" in PHM system. Human-driven predictive diagnostics by the right people with the right information is required (Barajas & Srinivasa, 2008). A prognostic model must be trained and then tested based on data. Verification and validation is also crucial for PHM.

The research described in this dissertation dynamically utilizes faults, enhancements, and performance data associated with software systems. The proposed architecture shown in Figure 3.2 includes three modules of the PHM process: Data Acquisition and Pre-processing, Optimization, and RUL Prognosis.

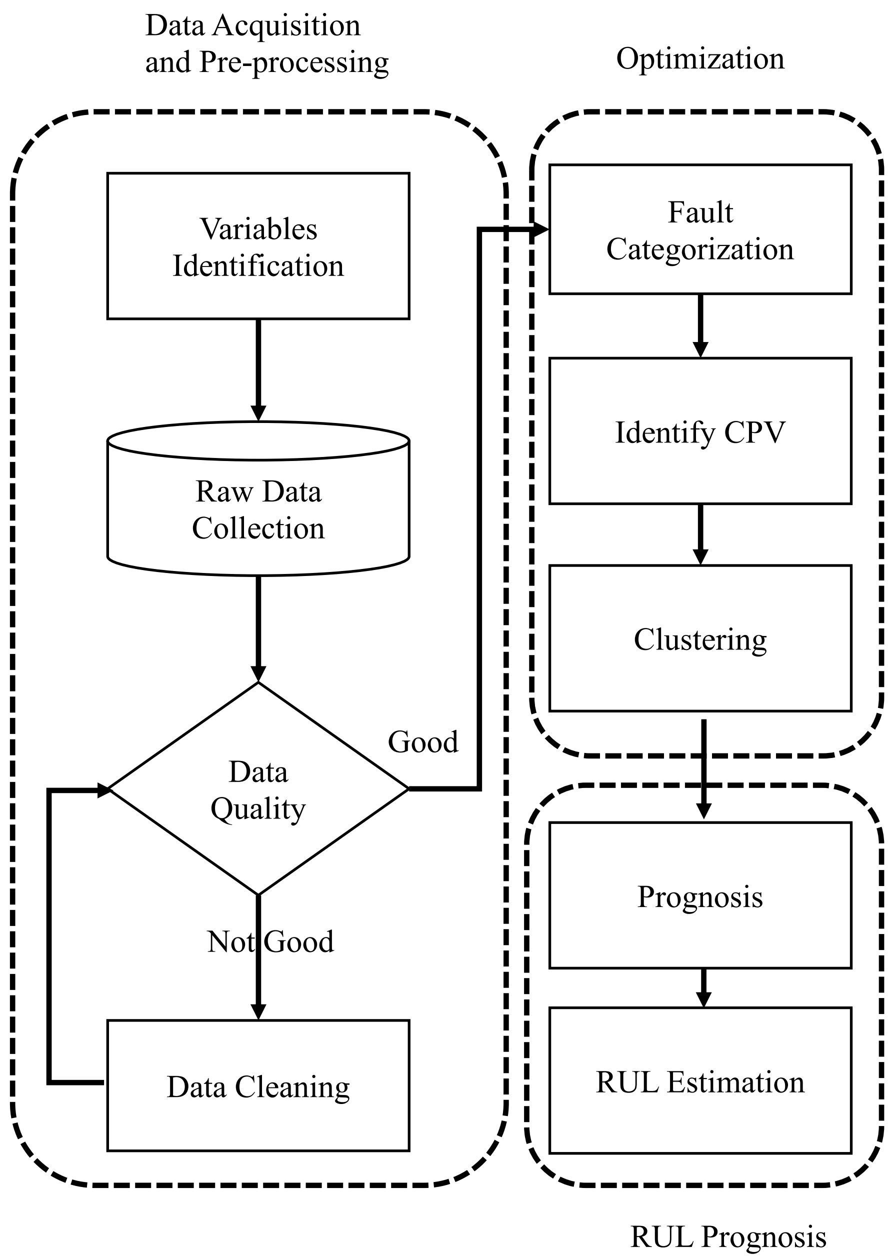

Figure 3.2: Pognostic model

In the first step, target variables and predictive variables are identified. Target variables are the performance parameters (see Section 3.3), and predictive variables are the faults and enhancements related to the corresponding target variables. Data on the variables are collected using software health monitoring tools and from reports populated by the user groups and the development teams. Collected data are pre-processed (cleansing) using a data quality assessment matrix to ensure quality data. In the optimization phase, for better understanding and successful estimation of complexities, and impacts on software performance, pre-processed data are categorized using a categorization matrix. This matrix classifies faults/enhancement requests into categories based on attributes such as server, network, database, configuration, and into sub-categories based on overloaded server, sluggish client, slow network services and others, that brings better scalability and ease in the mapping of predictive variables with corresponding target variables.

After categorization, a Natural Language Processing (NLP)-based classification (Naïve Bayes) is carried out to classify and size faults/enhancement in terms of continuous values. Here, sizing is the process of normalizing and estimating faults and enhancements only. Note, this normalization is applied to estimate faults and enhancements only, not to estimate software code. Estimation is carried out based on story points (see Section 3.6). Upon estimation, story points are individually multiplied with impact factors and summed up to estimate Predictive Variable (PV) and Cumulated Predictive Variable (CPV). The equations to calculate PV and CPV for each software releases are:

$$PV_1 = (\pm SP_1)(IF_1) + (\pm SP_2)(IF_2) + \ldots + (\pm SP_n)(IF_n) \quad (1)$$

$$CPV_m = PV_1 + PV_2 + \ldots + PV_m \quad (2)$$

Where, CPV is the cumulated predictive variable, *PV* is the total story point for individual releases (that can either be positive or negative, see Section 3.5), *SP* is the estimated story for individual fault/enhancement, *IF* is the impact factor, *n* is the number of faults and enhancements within a release, and *m* is the number of releases after cumulation.

Equation (1) estimates the total story points, *PV*, of faults and enhancements by taking account of the impacts of individual faults/enhancements on performance parameters by factoring the story points with impact factors. Impact factors represent a different level of estimated impacts of faults/enhancements on software performance parameters. In this way, *PV* represents the complexity and efforts required to fix the faults/enhancement requests, *IF* represents impacts of faults/enhancements on software performance. *PV*s are cumulated to get *CPV* (Equation (2)) for individual releases, since any deployment of fault fixes/enhancements in previous release remain impactful in the following release. *CPV* is used to predict the target variable(s). Clusters of analogous releases are formed using k-means clustering for better performance of the prognostic model(s). An elbow diagram is drawn to identify the number of clusters.

During the formulation of the prognostic model(s) and the estimation of the RUL, the goal is to combine several features into an optimal prognostic parameter that can easily be modeled with a chosen function (Barbieri et al., 2015). The rationale behind the fusion of the features is to combine several information sources into one predictive parameter that has improved robustness and an underlying trend related to the overall condition of the software. If the trend is not clear and is not defined over the system’s lifetime, the difficulties in analysis can increase, and the accuracy decrease. Therefore, reducing dimensions by selecting the best features is necessary to remove the irrelevant and erroneous features.

To be noted here, *enhancements (only)* of other environmental variables, such as clock speed, operating system (see section 4.4) are excluded while estimating PV and CPV. The reason behind this is that environmental variables are not changed (or upgraded) frequently (often with intervals of years) and their direct impacts on performance parameters can be estimated easily and separately. Including them in PV and CPV would be redundant. Though, the *faults* due to the above changes are to be included in estimation of PV and CPV since faults, even related to hardware and other environmental parameters, can happen frequently over releases.

To select an appropriate prognostic model(s), factors such as problem type (supervised/unsupervised), data type (structured/unstructured/semi-structured), data volume, the relationship between variables, number of variables, and other aspects are considered. The target variable's prognosis is carried out based on the *CPV* from analogous releases. Prognostic values are used to estimate RUL by plotting them across usage parameter(s). Data on predictive variables are continuously collected and used in the proposed model that estimates RUL dynamically.

3.3. Performance and Usage Parameters

3.3.1. Performance Parameters

The following is a list of software performance parameters that are widely used in the industry (Shafiabady et al., 2016; Arora et al., 2015; Porru et al., 2016; Ting et al., 2011; Mancoridis, 2015):

- **Execution time:** Time spent by the software system executing the task, including the time spent executing run-time or system services on its behalf (Mancoridis, 2015).
- **Timeliness:** Timeliness is measured in terms of software response time and throughput (Arora et al., 2015)
  - **Throughput:** The rate at which the system can process inputs. As an example: the number of hits on a web page within a given period is referred to as throughput (Porru at al., 2016; Ting et al., 2011; Mancoridis, 2015; Hammer, 2000).
  - **Response time (RT):** The time between the user requesting a response from the application and a complete reply (Mancoridis, 2015).
- **Availability:** The measure of availability is the probability that a system is available for use, considering repairs and other downtimes. The amount of

time an application is available to the user. Users would not be able to use an application effectively if the application is simply not responding or if the response time has degraded to an unacceptable level (Mancoridis, 2015).

- **Utilization:** This is the percentage of the capacity of a resource being used. An example is the amount of network bandwidth consumed by users or the amount of memory used on a web server farm when visitors are active (Mancoridis, 2015).

For the case study in Chapter 4, Response Time (RT) has been chosen as the performance parameter (target/output variables). RT is one of the most important performance parameters and the cost of slow RT is very high. Even catastrophic for safety critical software, such as, software for nuclear reactor, airplane navigator, roller coasters, elevators, pacemaker, rail, space shuttle and others. In the Web 2.0 conference in 2006, Google Vice President, Marissa Mayer, mentioned that users really respond to speed and a delay of an extra half a second results in web traffic drop by 20% (Mayer, 2006). In 2006, Amazon reported that 100 milliseconds of added page load time costs 1% of their revenue, which is $107 million in 2006 and $4.70 billion in 2021 (Olenski, 2016). Akamai, a leading content delivery network (CDN) services provider for media and software delivery, and cloud security solutions, found that slow performance results in users abandoning a software application by 28% (Everts, 2014). On the positive note it has been reported that 1 millisecond faster RT could result in $100 million gain in trading (Moallemi & Saglam, 2013). Both Yahoo and Mozilla reported an increase in traffic and number of downloads because of faster RT.

- Yahoo: 400 ms of improvement ≈ Web traffic increases by 9% (Everts, 2014)
- Mozilla: 2.2 second faster ≈ 60 million more Firefox download (Everts, 2014)

Table 3.1 shows more impacts of slow RT.

Table 3. 1 : Typical Cost of Slow RT (Everts, 2014)

1-second delay	2.1% decrease in shopping 3.5-7% decrease in conversions 9-11% decrease in page views 8% increase in bounce rate 16% decrease in customer satisfaction

Google shared its research on how different mobile page load benchmarks affect the bounce increment rate (Figure3.3). Bounce rate is the percentage of visitors to a

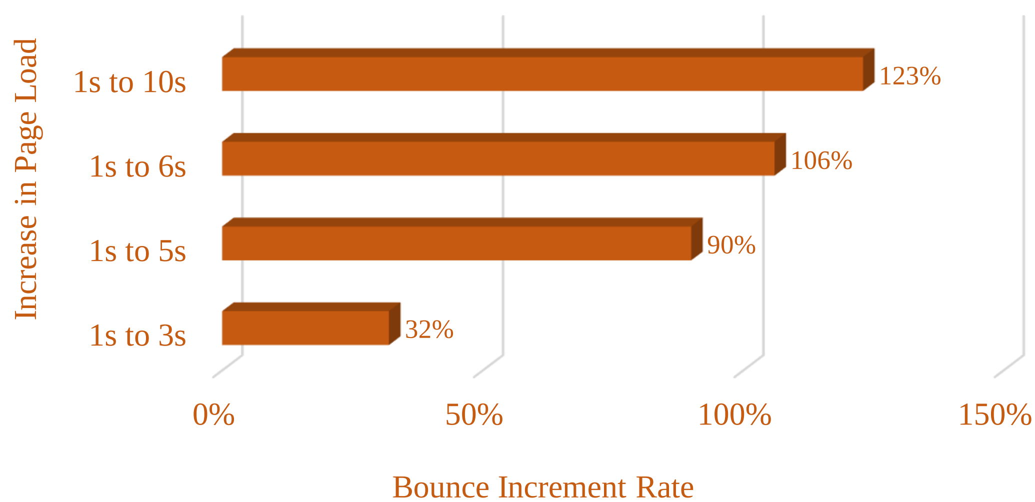

Figure 3.3: Google chart on mobile page load industry benchmarks (An., Da., 2018)

website who navigates away from the site after viewing one page only.

Input variables are every reported and deployed fault fixes and enhancement that influences the target variable. This methodology is applicable for all other performance parameters with appropriate modifications.

3.3.2. Usage Parameters Software PHM

Unlike hardware, software does not decay over time. Hence time is not an appropriate usage parameter. Software rots/degrades/decays as changes are made, and these changes happen over releases. Hence, *the release cycle* is a better selection of usage parameters. As a result, in this methodology, RUL is measured over releases rather than over time. Input variables would be faults or enhancement requests that influence corresponding usage parameters.

3.4. Software Release Life Cycle (SRLC) Management

The Software Release Life Cycle (SRLC) encompasses everything that needs to happen before the software is usable. It is a set of milestones that are achieved through various release types. This section defines different releases and other elements of the SRLC, like software upgrade and update during its life cycle. Understanding the SRLC helps in grouping faults/enhancements according to different types of releases. Grouping assists in the consistent estimation of story points (SP).

3.4.1. Software Release

A software release is a set of authorized changes to a software system. That means a release can include changes to hardware and/or software, documentation, processes, or other components essential to deploy an approved change successfully. A release can be carried out by adding new features or extending existing functions, and by deploying fault fixes. In this research, software RUL is estimated by estimating software performance. All enhancements and fault fixes that are related to the selected performance parameters are to be considered during this estimation. In commercial practice, releases are classified, such as major release, minor release, emergency release, maintenance release, service packs, patches, and hotfixes. Such classifications are formulated based on individual user need.

Major Release: A Major release contains software system upgrades, generally including a large number of new functionalities and fault fixes. A major upgrade or release supersedes all minor upgrades, releases, and emergency fixes. Often, a major release equates to introducing completely new functionality. A major release has the following contents:

- Major new features, Functional enhancements, architecture changes, product components
- As many fixes for outstanding issues as is feasible within the development schedule
- Full, standalone product build
- Will normally be localized

Minor Release: Minor releases are upgrades/updates in the software system, usually containing small enhancements and fixes, some of which may have already been issued as emergency fixes. A minor release usually supersedes all preceding emergency fixes and significantly improves existing functionality, often combining several fixes. Minor releases may have the following contents:

- May include functional enhancements and significant new features beyond previous minor/major version
- As many fixes for outstanding issues as is feasible within the development schedule
- All minor fixes from previous releases
- Will normally be localized

Emergency Release: Emergency Releases contain software system upgrades/updates, usually containing the corrections to a small number of known (but critical) problems. For example, an emergency release is carried out if something needs to be fixed or deployed as soon as possible or if a temporary fix is needed.

Maintenance release: According to Teradata, maintenance release fixes discrepancies discovered since the current major, minor, or previous maintenance releases were issued and may include high-priority enhancements (Teradata, 2018). A need for a maintenance release is identified when a fault or a set of faults are severely affecting a significant number of customers, and the next major or minor product release will not be released soon enough. These updates do not affect the life cycle of the product. A maintenance release is provided on an as-needed basis and typically includes the following:

- Previously released service packs and other fixes
- Full, standalone product build
- All fixes from previous releases
- As many fixes for outstanding issues as is feasible within the development schedule

Hot Fixes: Hotfixes are a type of release where one or two specific issues that are affecting a large number of customers are fixed. These updates do not affect the life cycle of the product.

All the different types of releases described above can be grouped into two main categories: Major releases and Minor releases. For the sake of robustness, in this research, releases are classified into these two categories. However, dividing them into more groups and sub-groups does not affect the methodology's performance.

3.4.2. Software Updates and Upgrades

Software update is downloading an application, operating system, or software suite that provides fixes to features that are not functioning as intended. Updates also include additional minor software enhancements and compatibilities. Update offers small, frequent improvements rather than major changes. They are the software equivalent of keeping a bicycle oiled for a smoother run. Software updates are released to resolve security issues, address minor fault fixes, and improve functionality and operation of hardware or peripherals, improve software operability, fine-tune a product and even add support for drivers and new hardware. Software updates can modify, fix, and alter existing software programs regularly. Updates sometimes run automatically in the background.

Whereas a software upgrade is generally the newest version of the software. It usually comes with major improvements that dramatically change or alter applications, operating systems, or software. A software upgrade adds whole new components or new features. Unlike software upgrades, updates need the existing software system to work. Update tweaks and improve the program's operation, but do not fundamentally alter it.

3.5. Data Collection and Pre-processing

After the selection of appropriate target variables and predictive (input) variables, data collection is carried out. For software systems, such data can be generated based on end-users and developers' enhancement requirements, reported faults, and software test results. Software can be tested manually or by using automated health monitoring tools and performance and functional testing tools. Such testing is carried out by designing checkpoints that are comparisons or verifications of operations performed during testing. Checkpoints are designed to validate that a software system is performing according to its requirements. Example checkpoints include, but are not limited to, property checkpoints, object checkpoints, file checkpoints, PDF checkpoints, XML checkpoints, webpage checkpoints, web service checkpoints, web accessibility checkpoints, and others. Examples of software health monitoring tools are

SolarWinds Server and Application Monitor for server performance monitoring; Atera to monitor unlimited endpoints both for software and hardware like availability, memory leaks, CPU utilization, and others; eG Innovations for end-to-end performance visibility; Nagios; Redis; Apache Tomcat; PRTG Network Monitor; Zabbix for problem detection with defined smart thresholds with high flexibility options. Many performance and functional testing tools are available in the industry, such as JMeter, Load runner, Selenium, and Quick Test Professional (QTP).

For prognostic model(s) training, the predictive variables, also known as input variables, are those that impact the corresponding target variable(s). Such variables can be *newly emerged* and *resolved* faults and deployed enhancements. This is because only the impactful and newly emerged and/or resolved faults and deployed enhancements are responsible for changes in the corresponding performance parameters in the relevant releases. For prognosis, unresolved faults and future enhancement requests are used to architect future release contents. Historical data on target variables are needed to train the prognostic model(s). To collect such data, individual test beds are built for each software release, and the performance of target variable(s) is measured using appropriate testing tool(s).

3.5.1. Data Quality Assessment Matrix

Data is pre-processed through cleaning activities, following assessments by a data quality assessment matrix (Table 3.1). Data quality is a measure of how well-suited a data set is to serve a specific purpose. To properly train a predictive model, historical data must meet exceptionally broad and high-quality standards. Data must be right, properly labeled, de-deduped, and so forth. (Redman, 2018). The objective of data quality assessment matrix is to identify data errors and fill gaps, remove duplicate values, fix anomalies, identify if data is fit for modeling and identify scopes for data cleaning, and mapping. Assurance of quality data is important as *garbage in, garbage out.* A minor error at one step can cascade, causing more errors and growing ever larger across an entire process (Redman, 2018).

Example impacts of bad quality data: Google Flu Trend (GFT) is a prediction application that claimed remarkable accuracy on flu predictions based on search data that was carried out by people who thought they had flu. Google assumed that anyone who searches for flu is affected by flu. But not all of them had flu. Such inaccurate data lead to wrong prediction for 100 out of 108 weeks as shown in Figure 3.4 (Lazer et al., 2014; Salzberg, 2014).

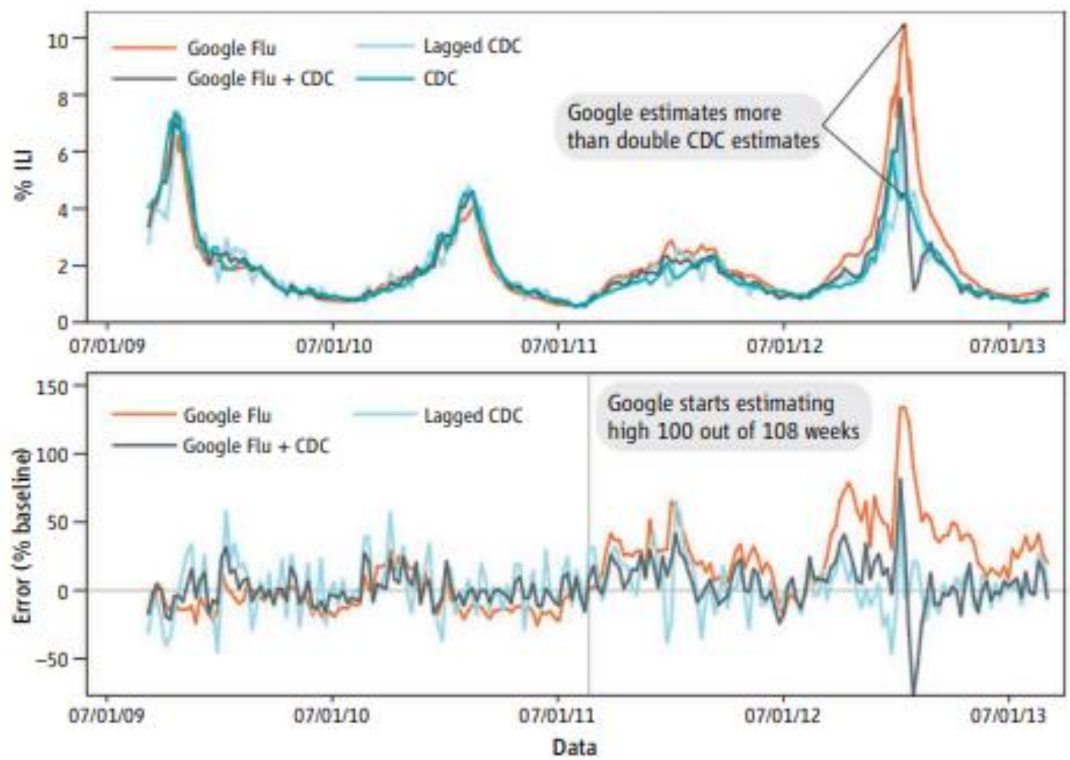

Figure 3.4: GFT overestimated flu in 2012–2013 and overshot actual level in 2011–2012 by >50%. (Lazer et al., 2014)

As shown in Table 3.2, the data quality assessment matrix incorporates four quality dimensions: accuracy, completeness, timeliness, and usability, many sub-dimensions, such as depth and breadth of the data record, missing records, transformation, lag time, data concurrency, spelling errors, data refresh frequency, accessibility, and duplicate values, followed by appropriate criteria questions to assess data quality. Such assessment assures quality data that are mapped with corresponding target variable(s).

Table 3.2: Data Quality Assessment Matrix[1]

Dimension	Sub-dimension	Criteria Question
1. Completeness	1.1 Depth	…………………………….
	1.2 Breadth	…………………………….
	1.3 Missing Records	…………………………….
2. Accuracy	2.1 Edits (Transforms)	…………………………….
	2.3 Data Delivery	…………………………….
3. Timeliness	3.1 Refresh	…………………………….
	3.2 Lag Time	…………………………….
4. Usability	4.1 Data Relevancy	…………………………….
	4.2 Data Currency for Use	…………………………….

Completeness: Asses how comprehensive the data is and if all data is available. Missing fault and enhancement data will adversely affect the prediction of associated target variable.

Accuracy: Accuracy ensures if the data is correct. As an example, accuracy can assess if the fault is really a fault based on which its direction of impact can be estimated.

Timeliness: Asses how up to date data is. Timeliness can detect if the input variable(s) are aligned to the correct Release.

Usability: Ensures that data is usable and is relevant. Example: if a fault/enhancement does have no impact on the target variable (RT), using such value to feed predictive model(s) may lead to wrong predictions.

[1] The data quality assessment matrix is application specific and may need to be updated to assess data for different applications.

Following the quality assessment, data cleaning is carried out by changing, updating, and/or removing data that are incorrect, incomplete, duplicate, erroneous and irrelevant. Data cleaning is a standard practice for predictive modeling.

3.6. Optimization

A categorization matrix is introduced for better granularity and understanding of variables through scaling and grouping. Such categorization also provides notion of how faults and enhancements interact, and impact RT. Features of entire groups can be learned by studying few faults/enhancements. As shown in Figure 3.5, a fault/enhancement categorization matrix is introduced for better classification and alignment of the input variable(s) with the target variable(s):

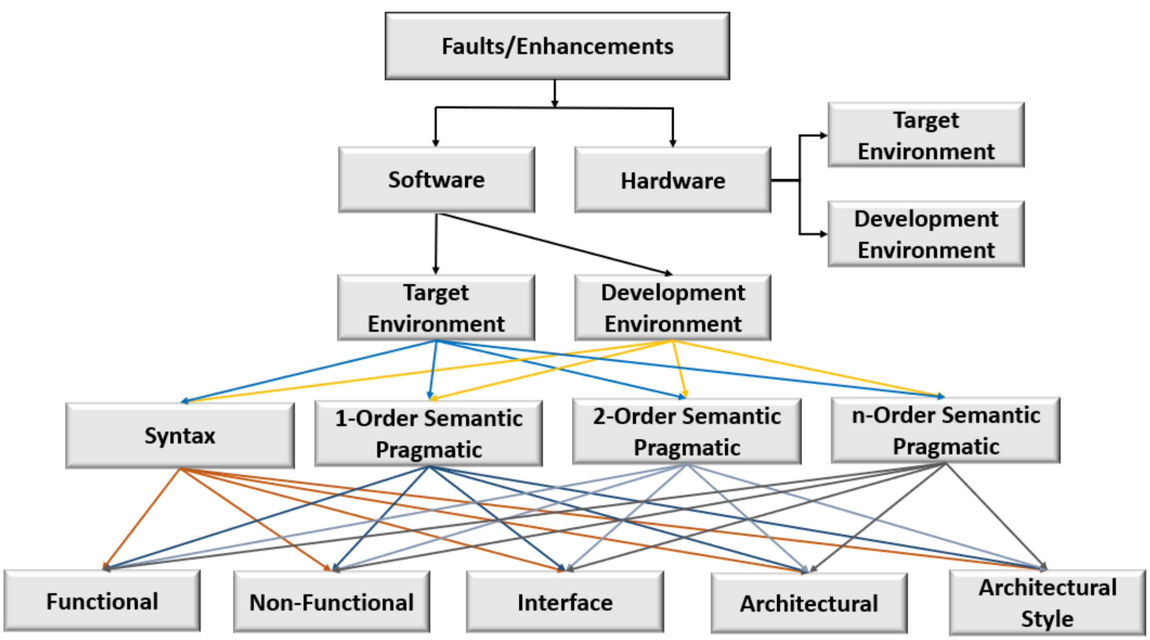

Figure 3.5: Fault/enhancement categorization matrix

In this matrix, faults and enhancement requests are classified into multiple categories and sub-categories. This classification is carried out into four different steps. In the first two steps, all the faults and enhancement requests are classified according to source environments, e.g., hardware or software and target environment (also known as production environment) or development environment. Then they are sub-categorized into four different groups based on the degree of changes required: Syntax, 1-Order Semantic Pragmatic, 2-Order Semantic Pragmatic, and n-Order Semantic Pragmatic. Such changes involve modifications in syntaxes and in the semantic-pragmatic layer, which defines the semantic and pragmatic specification of the interaction, i.e., what is the functionality and how the component performs it. 1st-order semantic-pragmatic incompatibility is when components can be put together, but a component does not function according to the requirements. 2nd-order semantic-pragmatic incompatibility is when components can be put together and satisfy requirements, but ambiguities in their interactions cause problems. Finally, nth-order semantic-pragmatic incompatibility is when components can be put together, and each component and each pair integrate and function, but a larger group of components have any problems (Yakimovich et al., 2001). The last layer of sub-division is based on impact areas: Functional, Non-Functional, Interface, Architectural, and Architectural style. This layer classifies according to the application's feasible solution, e.g., changes in architecture, architectural style, functionality, and interface. Categorization helps in better estimation of story points and subsequent calculation of impact factors and PV by providing scaling and grouping.

All the faults/enhancements are drawn through this matrix to categorize into groups. The following diagram (Figure 3.6) shows examples of such groups. As shown in the figure, categorization matrix grouped 10 of the faults/enhancements into 6 classes.

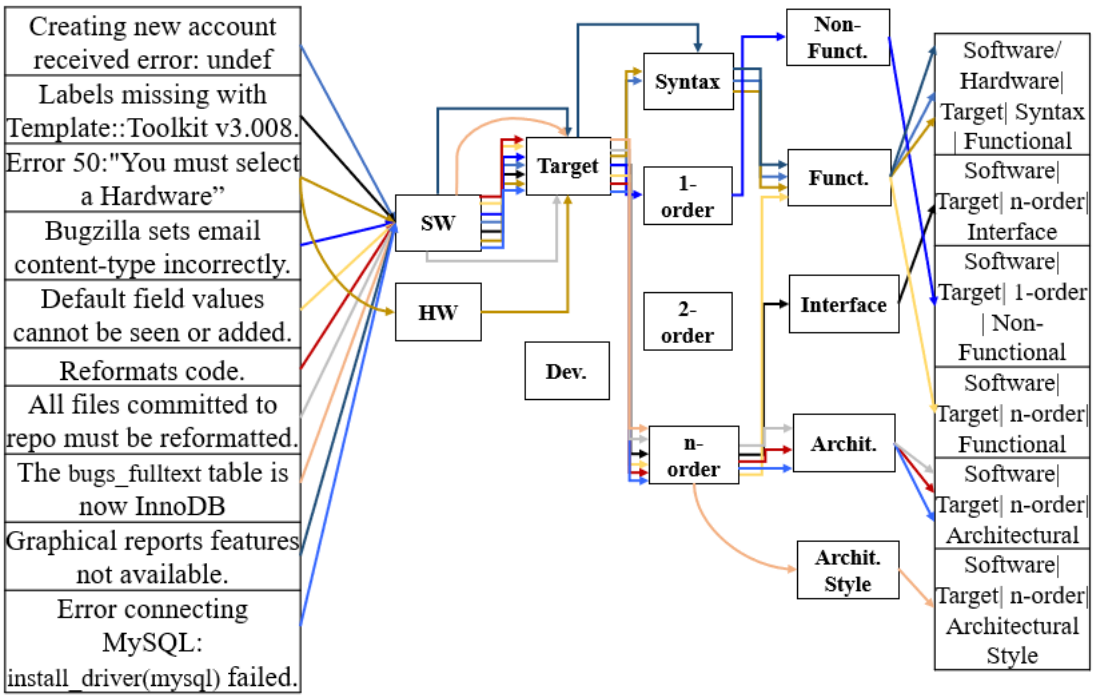

Figure 3.6: Example usage of categorization matrix

Each fault and enhancement requests are estimated numerically in the following step. Though there are many available estimation techniques, story pointing is the most widely adopted numeric estimation technique (Cohelo and Basu, 2012). Story points are a measure for expressing the overall size of faults/enhancement requests and consider attributes like the volume of work, inherent risks, complexities, and the effort required to complete faults and enhancement requests. There are a few different scales to measure story points, among which the Fibonacci sequence is the most popular. Each number in the Fibonacci scale is exponentially larger (by about 60%) than the previous number. This helps users and developers to understand and easily capture the differences between faults and enhancement requests and define each story point's complexity. For example, for this dissertation, the Fibonacci sequence (Porru et al., 2011) used is “1,2,3,5, and 8”. Where 1 represents the smallest and 8 the largest size of story points.

A Natural Language Processing (NLP) is introduced to learn and estimate story points automatically to reduce the impacts of human errors. NLP is a machine learning approach to reveal structure, relationship and meaning of text and to extract relationships between words, identify entities in sentence and much more. Such algorithm teaches machines how to read and understand corpus (texts) like humans, weighs relevant keywords and automatically analyzes text and assigns in pre-defined tags or categories. NLP is utilized to automate story points to reduce impacts of human errors. However, manual estimation of story pointing is still needed to prepare the training and testing data set for NLP. Periodical checks and balances need to be carried out manually to verify that the estimation is correct. Story points are mainly text/float data, and NLP works based on weighing relevant keywords. Therefore, users/developers must ensure that correct and consistent key words are used to write story points. This is why human factors are important in this research.

A Naïve Bayes Classifier is used for NLP. Naïve Bayes is a popular algorithm for text classification as it is highly efficient, easy to use, requires a small amount of training data, and requires less pre-processing. It performs well in multi-class prediction (Rish, 2001). Naïve Bayes considers each Fibonacci number (1,2,3,5, and 8) as an individual class and classifies the faults and enhancements accordingly.

To prepare data for natural language processing (NLP), punctuation and stop words are removed, sentence segmentation, word tokenization, parts of speech prediction, text lemmatization, dependency parsing, noun phrases identification and name entity recognitions are carried out. The performance of Naïve Bayes is verified and validated by computing a confusion matrix and comparing actual (test data) results with classified results. Naïve Bayes uses the probability of the events for its purpose and is based up on the Bayes theorem that assumes no interdependence among the variables. For example, in Equation 3, the posterior probability of class (c, target) given predictor (x, attributes) is:

$$P(c|x) = \frac{P(x|c)P(c)}{P(x)} \tag{3}$$

Where $P(c)$ is the prior probability of class, $P(x|c)$ is the likelihood, which is the probability of predictor given class, $P(x)$ is the prior probability of predictor.

Naive Bayes helps in computing conditional probabilities of the occurrences of two events based on probabilities of occurrence of each event; encoding those probabilities is extremely useful. It also assumes that the features are conditionally independent and easy to implement. It handles continuous and discrete data and is highly scalable with a number of predictors and data points. It is fast and can be used to make real-time predictions. Naïve Bayes is not sensitive to irrelevant features and has a higher bias but lower variance. As it considers the feature set to be Naive, the Naive Bayes algorithm can be trained with less training and even with mislabeled data which is an advantage in this case.

To be noted here, larger story points do not necessarily mean higher impacts on the target performance parameter. Hence, an impact scale with appropriate impact factors has been introduced. It is a common practice that impact factors are scaled as critical, major, medium, and minor, and it is ideal for converting them with an increment of 0.25. The higher the factors, the higher the impacts. Table 3.3 shows the conversion between impact scales and impact factors:

Table 3. 3: Conversion Between Impact Scale and Numeric Factors

Impact Scale	Impact Factor
Critical	1
Major	0.75
Medium	0.5
Minor	0.25

However, the scale shown in Table 3.3 can be modified as needed. Any conversion scale is fine if the same scale is used throughout the life cycle of an individual software system. To factor in the impact of individual fault/enhancement on software performance, the individual story points (SP) are multiplied with corresponding estimated impact factors before they are summed to estimate the Predictive Variable (PV) for individual releases. This impact estimation is carried out based on expert judgments by the development team. It has also been realized that the effects of degradations or improvements are accumulated over releases that create the need for a cumulative damage model (Ramakrishnan and Pecht, 2001). This methodology assumes that all the degradations incurred in each release accumulate until some external source actively repairs the system. Hence, CPV is the cumulative value of PV over releases. The CPV is the fusion of predictive variables that prognostically estimates the target variable(s).

Another important note is that some fixes/enhancements may improve software performance, such as deploying or increasing capacity of load balancer, or removal of certain features that demand lots of processing powers causing the application to run slow. To factor such improvements, SPs of the corresponding faults/enhancements of relevant target variable(s) are labeled as negative. Negative story points reduce the PV value and can even result in negative PV, which subsequently can cause lower CPV and improvement in software performance.

In the next step, clusters of analogous releases are formed by k-means clustering. If clustering is not accounted for in the prediction model development, model calibration becomes suboptimal when applied in new subjects (Bouwmeester et al., 2013). Adding cluster effect in prediction model increases amount of predictive information, resulting in improved overall discriminative ability and calibration within clusters (Bouwmeester et al., 2013). Prediction models incorporating cluster level information showed better performance (Ni, 2018). Besides, clustering helps in understanding new releases and their characteristics and internal structure of data.

Analogous estimation is especially useful when there is limited data available. CPV is used to identify the number of clusters. Cleaned data (faults/enhancement requests) is transformed into Unicode before plugging it into k-means to form clusters. Every time a new data point(s) is available on releases, clustering is carried out to identify appropriate cluster(s) for the new data and to detect if any new cluster(s) has emerged. The objective function to identify the clusters (k) of analogous releases is given by in Equation 4:

$$J = \sum_{j=1}^{k} \sum_{i=1}^{n} \left\| x_i^{(j)} - c_j \right\|^2 \tag{4}$$

where, k is the number of clusters, n is the number of cases, x_i is case i and c_j is the centroid of cluster j.

The clustering process does not contain a ground truth label, which makes its validation tricky. However, the number of clusters is identified and validated based on an elbow diagram (within-cluster Sum of Square). In the elbow method the explained variation is plotted as a function of number of clusters and the elbow of the curve is chosen as the number of clusters to use.

3.7. Prognostic Model

Several paradigms for solving PHM problems exist, including data-driven and model-based approaches. Model-driven approaches start with an understanding on how a physical system works and how it can fail. Whereas data-driven approaches are enabled by algorithms determined from machine learning without focusing on the underlying processes (Medjaher & Zerhouni, 2013). Implementation of model-driven approach requires a deep understanding of the system's complexity. In software systems, complexity includes the specific set of characteristics of the code and its interactions with other pieces of code, device drivers, and operating systems. An accurate understanding of the complexity may not be available, especially for a new software system. Software systems can also be very versatile and can change frequently.

The good news is that controllable test beds (staging environments) can be generated for software systems, and appropriate data can be populated. The availability of data, even for a new software system, is not a big challenge, but still, models are needed to predict and interpret data sets. Therefore, a fusion approach, which combines a data-driven approach with a model-driven approach aided by expert judgments, has been developed for this methodology. The earlier versions of the software system can be configured in controlled test beds if the corresponding artifacts are available or historical data is available. Control over the environmental variables is possible in test beds (e.g., see Appendix B). Due to advancements in sensor technology, computational capabilities, and dedicated software/hardware interfaces, the fusion approach can detect faults accurately when the system monitoring data for nominal and degraded condition is available (Tidriri et al., 2016).

Based on the estimated CPV and historical performance data, the performances of future releases can be predicted using appropriate prognostic model(s). Given the problem type (supervised) and the properties of the predictive and target variables (structured), prognostic model(s) selection is motivated by data volume, the relationship between variables (linear/non-linear/correlated/non-correlated), and the number of variables. The RUL is identified by plotting the historical and predicted performance parameters against the usage parameter and by defining a threshold (Figure 3.7).

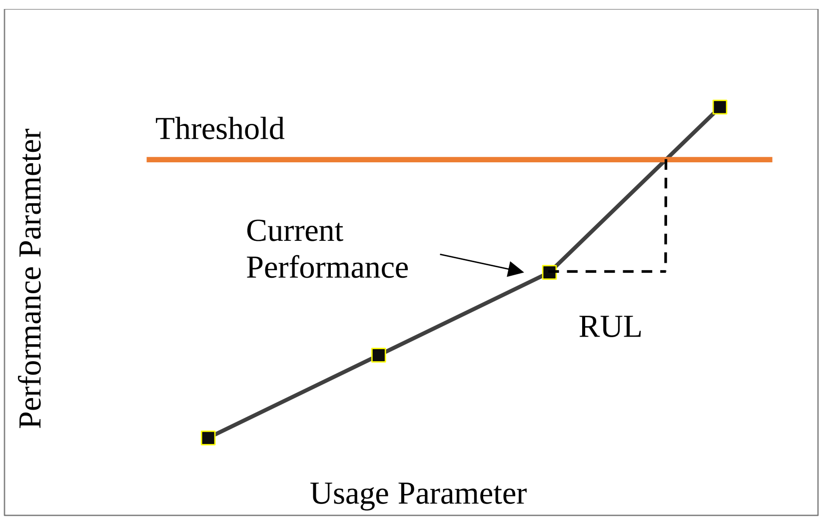

Figure 3.7: Remaining Useful Life (Illustrative)

As discussed earlier, the usage parameter is *software release*. Furthermore, when it comes to thresholds, there are no industry standards for performance thresholds. Instead, users define and agree upon thresholds that vary from software system to system and from user to user. RUL changes continuously as the number of faults and future enhancement requests change, making this methodology dynamic. In this methodology, the contents of future releases can be controlled, architected, and re-architected in any desired manner. Estimated RUL can be used for deciding at what stage and to what extent a software system should be updated/upgraded, re-engineered, components should be integrated or removed or disabled or even abandoned.

Figure 3.8 provides an overview of the methodology described above. After collection, data goes through quality assessment. Quality verified data are categorized and clustered to identify CPV based on which software performance for future releases is estimated prognostically. Finally, by plotting release-wise prognostic performance values against usage parameters, RUL is estimated.

In this proposed dynamic method, every time sensors/users/developers report new faults or enhancements their rate of deterioration and intensity changes. RUL, also, gets updated dynamically.

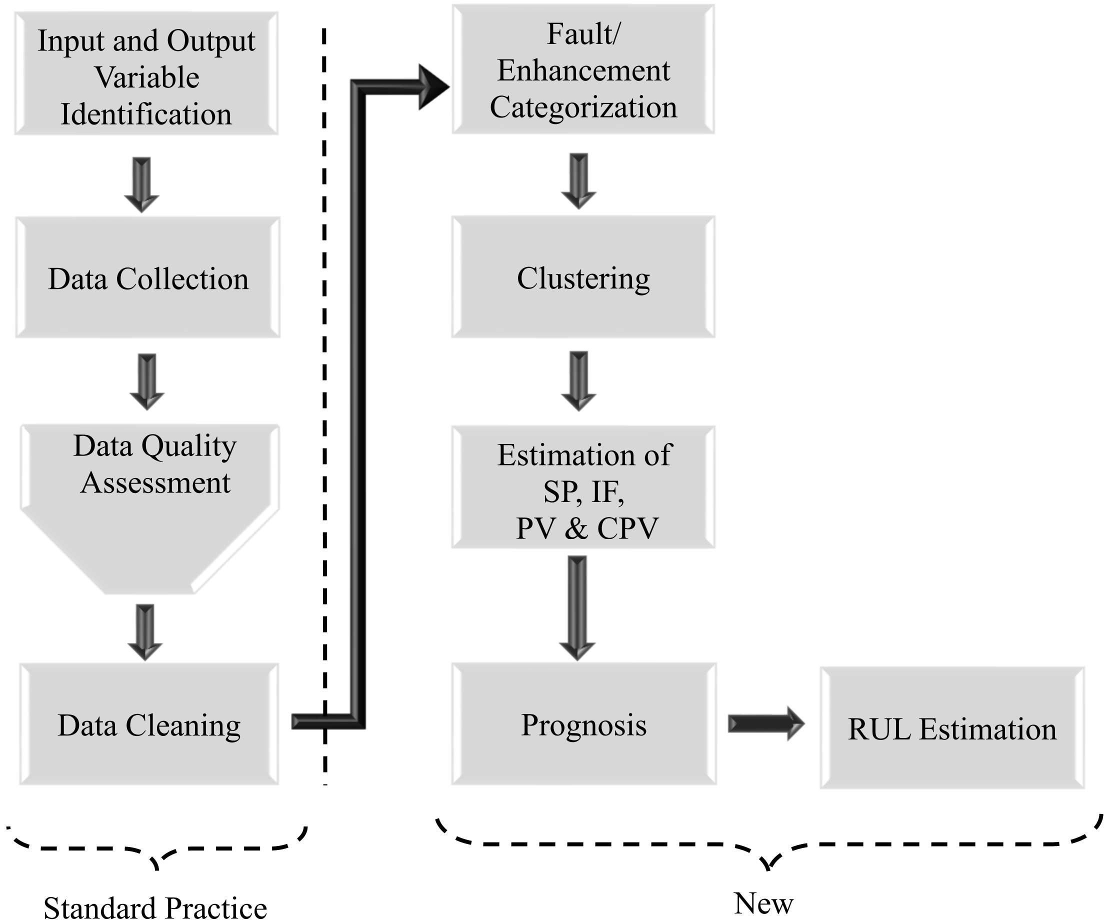

Figure 3.8: Proposed research

3.8. Model(s) Verifications & Validation (V&V)

Model validation determines whether the model accurately represents the system's behavior (Aumann, 2007). Metrics commonly used to estimate the predictive validity of a model are: R^2, mean squared error, sensitivity, specificity, receiver operating characteristic, concordance index), and methods to estimate the predictive validity include cross-validation, bootstrap, adjusted and shrunken R^2, and others (Ivanescu et al., 2016).

Models can be validated by comparing the output to independent fields or experimental data sets that align with the simulated scenario (Kerr & Goethel, 2014). Model validation can be grouped into internal and external. Internal validation includes techniques like re-sampling methods, such as bootstrap and cross-validation. However, internal validation does not contemplate the differences in the distribution of predictors included in the model observed in the new; therefore, quantifying the predictive performance in an independent dataset known as external validation can be done.

There are many validation methods available to measure performance of model(s) such as:

Split sample: The most straightforward approach to address the optimism of apparent performance when estimating the predictive performance in the same sample used to develop the predictive model is randomly splitting the sample into training and testing dataset. Especially when the predictive model does not have hyperparameters to be tuned. Another type of split can be carried out into three partitions: training, validation, and testing in a way that predictive model(s) can be developed and tested on independent samples.

AUC: Area Under the Curve (AUC) is a measure of the ability of a classifier to differentiate between classes and is used as a summary of the Receiver Operating Characteristic (ROC) curve. The higher the AUC, the better the model's performance at differentiating between the positive and negative classes. The ROC curve was developed in the signal detection theory and consisted of the plot for the probability of detecting a true signal (sensitivity) against the probability of detecting a false signal (specificity). Once a cut-off point is defined for the probability of the event of interest, the probability of true positive, known as sensitivity, and the probability of true negative, known as specificity, can be calculated.

Accuracy: In the same context defined for sensitivity and specificity, accuracy can be calculated as the probability of correct classification independently to the presence or absence of the event of interest. It is very easily calculated from a confusion matrix; therefore, it is a very common measure, used in the predictive modeling literature, even though it is not recommended as a performance measure because it could be very misleading in scenarios with unbalanced data. Furthermore, it is an improper scoring rule (Diniz et al., 2022).

The other methods available for validation of model performance include:

- k-Fold Cross-Validation
- Leave-one-out Cross-Validation
- Leave-one-group-out Cross-Validation
- Nested Cross-Validation
- Time-series Cross-Validation
- Wilcoxon signed-rank test
- McNemar’s test

- 5x2CV paired t-test
- 5x2CV combined F test
- The bias
- The mean absolute error: MAE
- The root means square error: RMSE
- Bias-variance trade-off

Applying expert judgment and evaluating model(s) based on experience is another method for model evaluation. However, expert judgment commonly requires expertise in the application area that may not always be available.

The evaluation of the performance of a clustering algorithm is not as crucial as counting the number of errors or precision and recall of a supervised classification algorithm. The clustering process does not contain a ground truth label, which makes validation tricky. However, the number of clusters is identified and validated based on an elbow diagram (within-cluster Sum of Square). In the elbow method, the explained variation is plotted as a function of the number of clusters. The elbow of the curve is chosen as the number of clusters to use. Clustering defines separations of the data like ground truth set of classes or satisfying assumption in a way that members belonging to the same class are more similar than members of different classes according to some similarity metric.

Chapter 4: Software RUL Prediction Demonstration

This chapter demonstrates the methodology described in Chapter 3 using an open-source software system called Bugzilla. Test beds are built to configure multiple Bugzilla releases in a controlled and standard environment. Test beds are composite abstractions of systems to study system components and interactions to gain insight into essence of real system. They are built of prototypes and pieces of real system components (Fortier & Michel, 2003). Test bed include both software and hardware components and be able to configure historical versions imitating user environment(s). Test bed facilitates measurement of RT. Test bed needs to be isolated from unexpected biases and is controllable and standardized environmental attributes.

In this chapter the high-level architecture of Bugzilla is discussed. This chapter also discusses how historical data is collected, quality verified, cleaned, categorized, clustered, and how the SP, PV, and CPV are estimated. SP is estimated both manually and automatically based on NLP. PV is estimated based on SP and IF. Once PV is cumulated CPV is identified. Upon estimation of CPV, an appropriate prognostic model is selected, and the performance parameter (RT for this demonstration) is estimated. The predicted RT is plotted against the release cycle to estimate RUL. RUL is also estimated for historical releases by varying operating system (32-bit, 64-bit) and clock speed to understand the impact of computing environmental changes on performance parameter(s).

Why open-source software?

For this demonstration the software system needs to have publicly available data on:

- All historic releases, including but not limited to, faults/enhancements, changes, version updates/upgrades, enhancements, and fault fixes.
- All future faults and enhancements requests.
- All functional and performance test reports
- Historical functional configuration file for:
  - Software and
  - Associated environmental components

In addition to those, the software needs to be configurable on a local machine with local URL to avoid unnecessary bias and to be able to standardize and control the environmental variables. Software needs to have integrated components (both software and hardware components). Finally, RT needs to be an important performance parameter for such software.

4.1. Bugzilla Test Beds

Bugzilla is an open-source tracking tool that is used to track faults/enhancement requests in software applications.[2] Bugzilla has publicly available historical release

[2] The demonstration in this dissertation is on the Bugzilla application itself, not to be confused with the function of Bugzilla, which is to capture and track defects in other software applications. In this demonstration faults and enhancements in the Bugzilla application itself are captured and tracked.

notes, faults and enhancement data and test data. It is configurable on local machine (local URL) in controlled and standardized test beds has multiple integrated components and RT is one of its important performance requirements. Another advantage of using Bugzilla for this demonstration is that it has been in existence for over 22 years, and many of its previous versions are still available and configurable. Users and the development team report faults and enhancement requests on Bugzilla in an open-source repository (https://bugzilla.mozilla.org/). Some of these requests have been addressed, and the rest are to be addressed. This demonstration is focused on response time (RT) as performance parameter (target variable), and faults and enhancements as predictors that are only impactful to RT. However, the methodology described in Chapter 3 applies to other performance parameters as well, of course with appropriate modifications. Test beds are built in standardized, scaled, and controlled staging environments on Microsoft Virtual Machines. Response time (RT) is populated by building multiple test beds for individual versions of Bugzilla. Open-source data are gathered on faults and enhancements reported by the users and the developers (Serrano & Ciordia, 2005).

Bugzilla is a web-based system, but for this demonstration, multiple versions are installed on local server (laptop computer) to avoid bias and to control other environmental variables. Configuring versions on a local server facilitates Bugzilla's performance to be observed as a function of relevant faults/enhancements only. This application is written in Perl that can run on cross-platform operating systems and is compatible with multiple databases, such as SQL and Oracle. Integrated modules of

Bugzilla include Perl, Apache, SQL, PostgreSQL, Oracle, and SQLite. Figure 4.1 shows the architecture of Bugzilla. Faults and future enhancement requests are reported based on functional, regression, and performance test results, user experiences, and requirements.

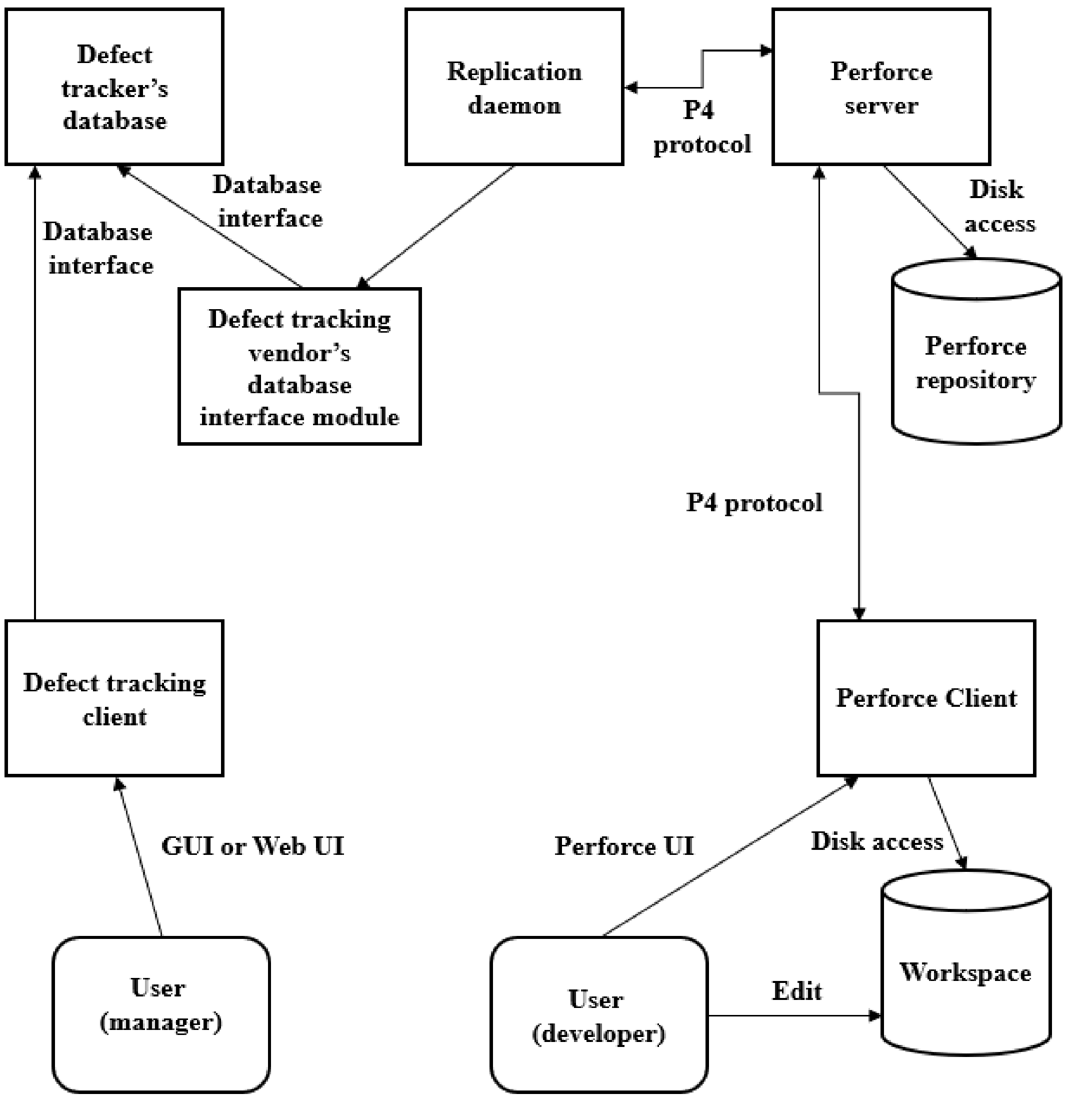

Figure 4.1: Bugzilla's architecture (Brooksby, 2000)

Much of this data is collected automatically using software health monitoring tools. Some of the common health monitoring tools include Nagios, Zabbix, Grafana, Netdata, and others. However, Bugzilla’s actual applications and functionality are irrelevant to this demonstration.

Test beds produced historical response time (RT) over multiple releases in controlled and standard test (staging) environments in a Virtual Machine (VM) on a dedicated computer (laptop). All the test beds are built with following specifications:

Host Machine Specifications

- HP Spectre x360 Convertible 15-bl1XX
- Processor: IntelI Core I i7-8550U CPU @ 1.80GHz
- RAM: 16.0 GB (15.9 GB usable)
- System type: 64-bit operating system, x64-based processor
- Edition: Windows 10 Home

Oracle Virtual Machine (VM) Specifications

- Microsoft VM ware (Windows 10)
- Version: 6.1.16
- Operating System (OS): 64-bit & 32-bit
- Processor: IntelI Core I i7-8550U CPU @ 1.80GHz
- Chipset: PIIX3
- RAM: 4 GB
- HDD: 50 GB

- Clock Speed: 1.88 GHz and multiple decrements of 10%
- Available space on HDD after installing all the required software and tools 23 GB.

Bugzilla Environmental Specifications

- Perl 5.32.0
- Java 15.0.1
- SQL 5.0.15
- Apache 2.4

A room temperature of 75° F (approximately) and humidity of 65% (approximately) are maintained. Each version of Bugzilla is configured on individual machines. During testing an equal amount of process loads are applied across all the test beds. 25 readings are taken for each test to identify mean and standard deviation-based error bars.

All the existing fault/enhancement data for Bugzilla is collected. The installed versions (also known as releases) in the test bed are stated in Table 4.1.

Table 4.1: Bugzilla Release Version and Dates

Bugzilla Release #	**Release Data**
Release 3.6.3	02-Nov-10
Release 4.4.0	22-May-13
Release 4.4.9	15-Apr-15
Release 5.0.0	07-Jul-15
Release 5.0.1	10-Sep-15
Release 5.0.2	22-Dec-15
Release 5.0.3	16-May-16
Release 5.0.4	16-Feb-18
Release 5.0.5	30-Jan-19
Release 5.0.6	09-Feb-19

The selection of versions is motivated by the availability of configuration files. Selected releases are as consecutive as possible to quickly capture all the faults and enhancements in between releases. Though there are other releases between releases 3.6.3 and 4.4, and 4.4 and 4.4.9, all the associated functional files are not available. Therefore, no releases in between those releases are configured. However, data on all the faults (both fixed and not fixed) and deployed enhancements in between those releases are collected. Unfixed faults still may have impacts on performance parameters. Faults and enhancement requests that are populated but never were fixed beyond Release 5.0.6, are used to architect future releases.

Generally, the software is divided into two main categories: system and application. System software, such as operating systems, game engines, software as a service, APIs, and others, keep everything working, and application software allows users to complete tasks, such as creating text documents, web browsing, playing games, and others. Most of the time both categories are integrated and works hand in hand for a software system. In this demonstration, RT is measured by varying versions of both the application and system of Bugzilla. Initially, RT is measured by changing Bugzilla versions (application software) but keeping all the other variables the same across all the test beds. The idea is to quantify the influence of changing application software on the performance parameter (RT). RT is then measured by changing the operating system (system software) from 32-bit to 64-bit and keeping the remaining variables the same. At the end RT is measured by varying clock speed and keeping both Bugzilla versions and the operating system to understand the effect of hardware.

Since faults and enhancement requests are mostly written in a descriptive format, input data are mainly in text format (unstructured). Collected data types:

- Unstructured data (texts): Faults and future enhancement requests
- Categorical variables: Fault/enhancement classes
- Continuous variables: Story points, impact factors, Cumulative Predictive Variable (CPV)

There are no industry standards for thresholds of software performance parameters, not even for response time (RT). The threshold is determined by de facto standards that vary from application to application and governed by user requirements. However, there are three primary time frames for RT when optimizing web performances (Neilsen, 1993):

- 0.1 second: The time for the user to feel that the system is reacting instantaneously.
- 1.0 second: The limit for the user's flow of thought
- 10 seconds: The limit for keeping the user's attention.

For this demonstration, an RT threshold of *9 seconds* is considered. However, the threshold can vary based on user and application requirements and evolve over time.

4.2. Data Collection

4.2.1. Performance Data (Output Variables)

Historical performance data (RT) on Bugzilla is populated and collected using a load/stress testing tool in a controlled test environment. There are many load/stress testing tools out there, such as Apache JMeter, Load Runner, Neo Load, Locust, and others. JMeter (Halili, 2008) is one of the most widely used open-source load testing tools that are used for a wide array of system components, e.g., web apps, databases, web services, and more. It is 100% java based and can test performance of both static and dynamic resources. Using JMeter, user loads can be simulated on individual servers, groups of servers, networks, or objects to test strength and analyze performance under different load types.

JMeter is used to measure RTs across all the test beds in this demonstration both on 64-bit and 32-bit test environments. RT is also measured for Bugzilla versions

3.6.3, 4.4, 5.0.4, 5.0.5, and 5.0.6 to identify the effect of change in other environmental attributes such as 64-bit operating system vs. 32-bit operating system. At the end RT is measured for Bugzilla version 5.0.4 by varying clock speed. Here are the test parameters in JMeter (version 5.4) to create virtual loads:

- Number of threads (users): 5
- Ramp-up period (seconds): 5
- Loop Count: 5
- Readings taken: 25 each

The number of threads represents the number of virtual users that will connect to the web service at once, and the loop count is the time individual users connect to the web service. The ramp-up period represents how long it takes to “ramp up” to the total number of threads chosen. For each page of Bugzilla, 5 x 5 (the number of threads multiplied by the loop count) = 25 readings are taken for each page, and averages are calculated. An average of 5 such averages is measured. This is done for all the pages of Bugzilla, including Home, New Bug, Browse, Advanced Search, Reports, Preferences, and Administration. Figure 4.2 shows the graphical user interface of Bugzilla, which contains tabs that lead to the above pages. Finally, the mean value of RT is estimated. The advantage of using this type of mean RT is that it minimally affects fluctuations in sampling and considers all the values in the series. However, the averaging approach used here is specific to this demonstration; other approaches may be best suited for other performance parameters and other types of software systems.

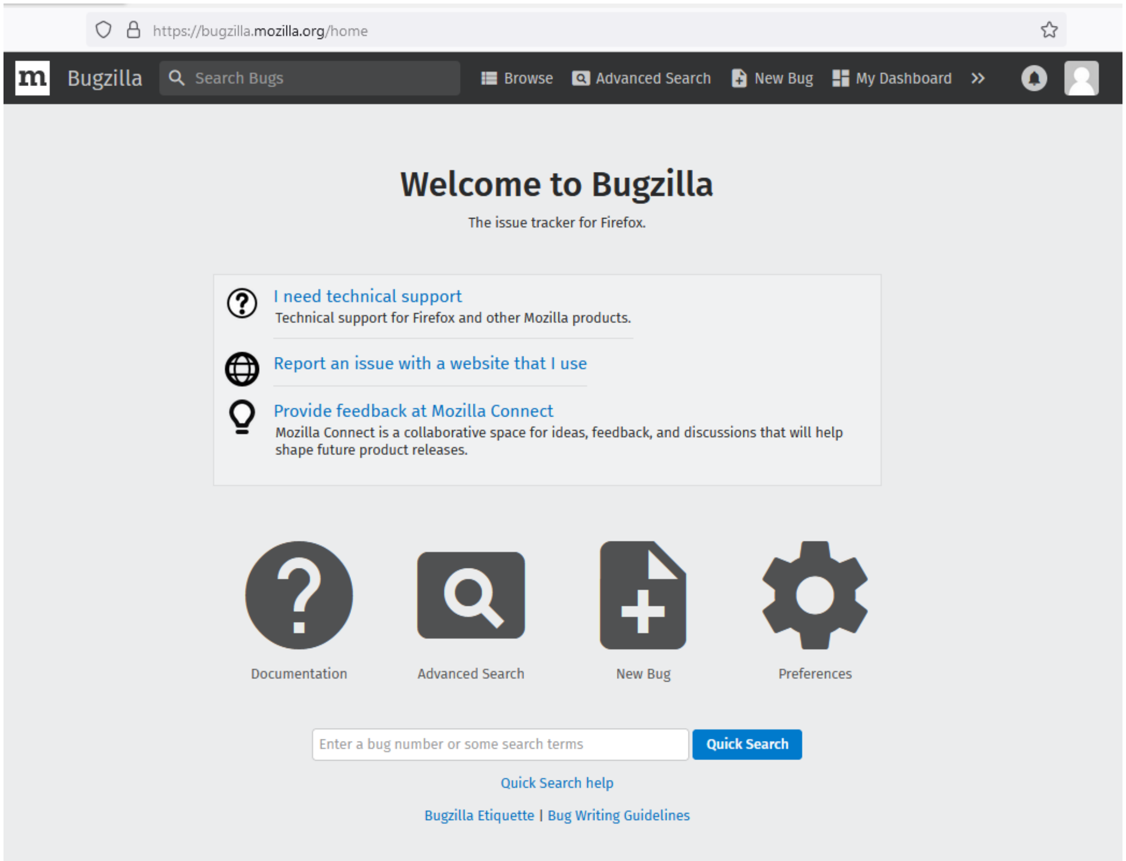

Figure 4.2: Bugzilla's graphical user interface

As discussed in Chapter 3 (section 3.3.2) it is assumed that software performance can only be impacted when there are changes within the software system, such as, generation of new faults or fault fixes or deployed enhancements. Hence, the impacts of faults are estimated only in releases where such changes are made. However, not all the changes impact software performance. Expert judgement is made to understand which changes have impacts to what extents. Figure 4.3 and Table 4.2 show the collected data with a lognormal trend line. This trend line does not represent any relationship between the bars, as there is none. This figure demonstrates that response time increases as CPV increases, and reduced response time (RT) means increased performance. To calculate error bars, standard deviations are calculated for RT for each release. This exercise is carried out for all releases. Standard deviations are used both for the positive and negative sides of the error bars.

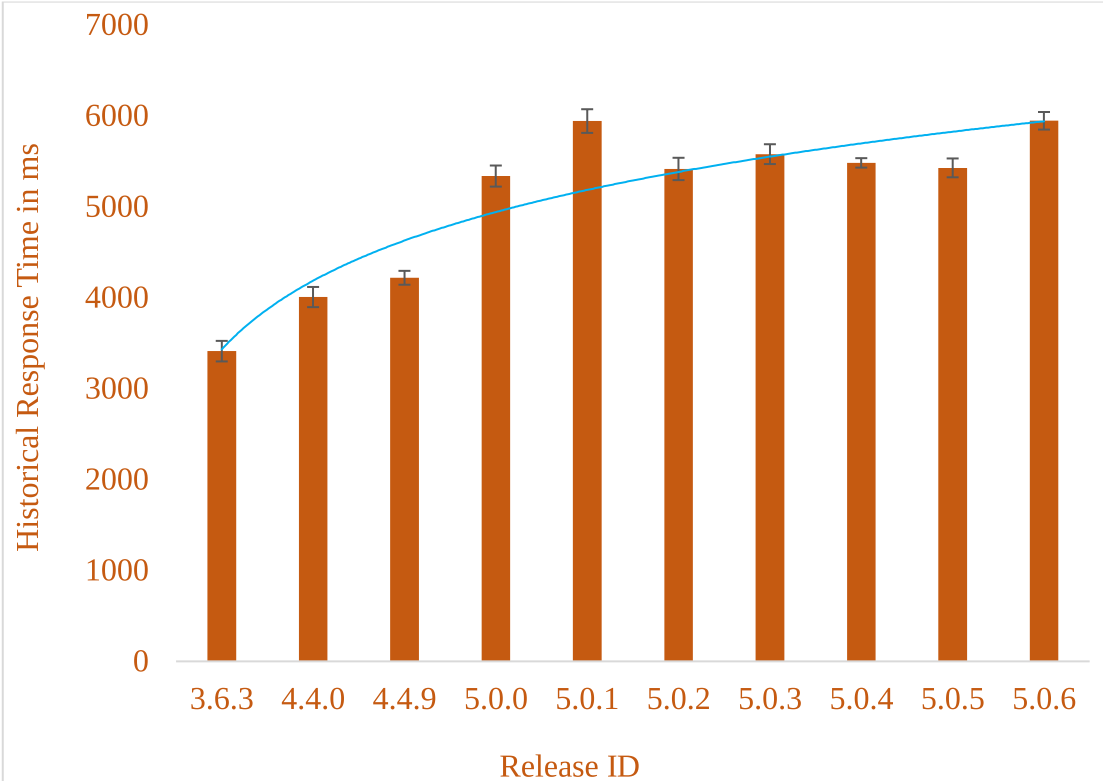

Figure 4.3: Response Time (RT) over multiple Bugzilla releases

Table 4.2: Response Time (RT) Over Multiple Bugzilla Releases

Bugzilla Release #	**Mean Response Time (ms)**
Release 3.6.3	3412
Release 4.4.0	4007
Release 4.4.9	4219
Release 5.0.0	5338
Release 5.0.1	5943
Release 5.0.2	5416
Release 5.0.3	5579
Release 5.0.4	5483
Release 5.0.5	5428
Release 5.0.6	5946

4.2.2. Faults/Enhancement Data (Input Variables)

As mentioned earlier, all the faults and enhancements (input variable) are reported by the users and developers from across the world. They are reported in the descriptive format in an open-source Bugzilla repository. Since this demonstration is based on response time (RT), only faults/enhancements, that impact RT, are collected. Such selection of faults/enhancements is carried out based on expert judgments. At first, collected data are quality verified and modified according to the data quality (Table 3.1) assessment matrix. Then data is categorized using a fault/enhancement categorization matrix as described in Figure 3.2 (Section 3.6). Such categorization helps better scaling of the input variables (faults/enhancement requests) while estimating story points and their impacts on RT.

NLP is carried out to automatically estimate faults/enhancements in story points. Figure 4.4 shows the high-level flow chart of Natural Language Processing (NLP). Since faults/enhancement requests are written in text format, a word dictionary with a pool of standard keywords is created based on historical data (faults/enhancements). This pool evolves as more key words become available. Faults/enhancements are written using as many keywords as possible. NLP estimates story points based on use of key words in the fault and enhancements descriptions. Data is then 73hroughh a naïve bayes algorithm to estimate story points (SP), also known as sizing. In sizing, faults/enhancement requests are converted into numbers based on the previously discussed Fibonacci sequence (1, 2, 3, 4, and 8). For the NLP to function properly, Naïve Bayes needs to be trained; therefore, a training data set is required. In addition, a set of testing data is also required to verify the performance of Naïve Bayes. Training and testing data sets are built by estimating a portion or all the historical faults/enhancements based on expert judgements. A random split of historical data is done using R-code. In the next step, input data (faults/enhancements) are sized in terms of story points on a scale of Fibonacci sequence (1,2,3,5,8).

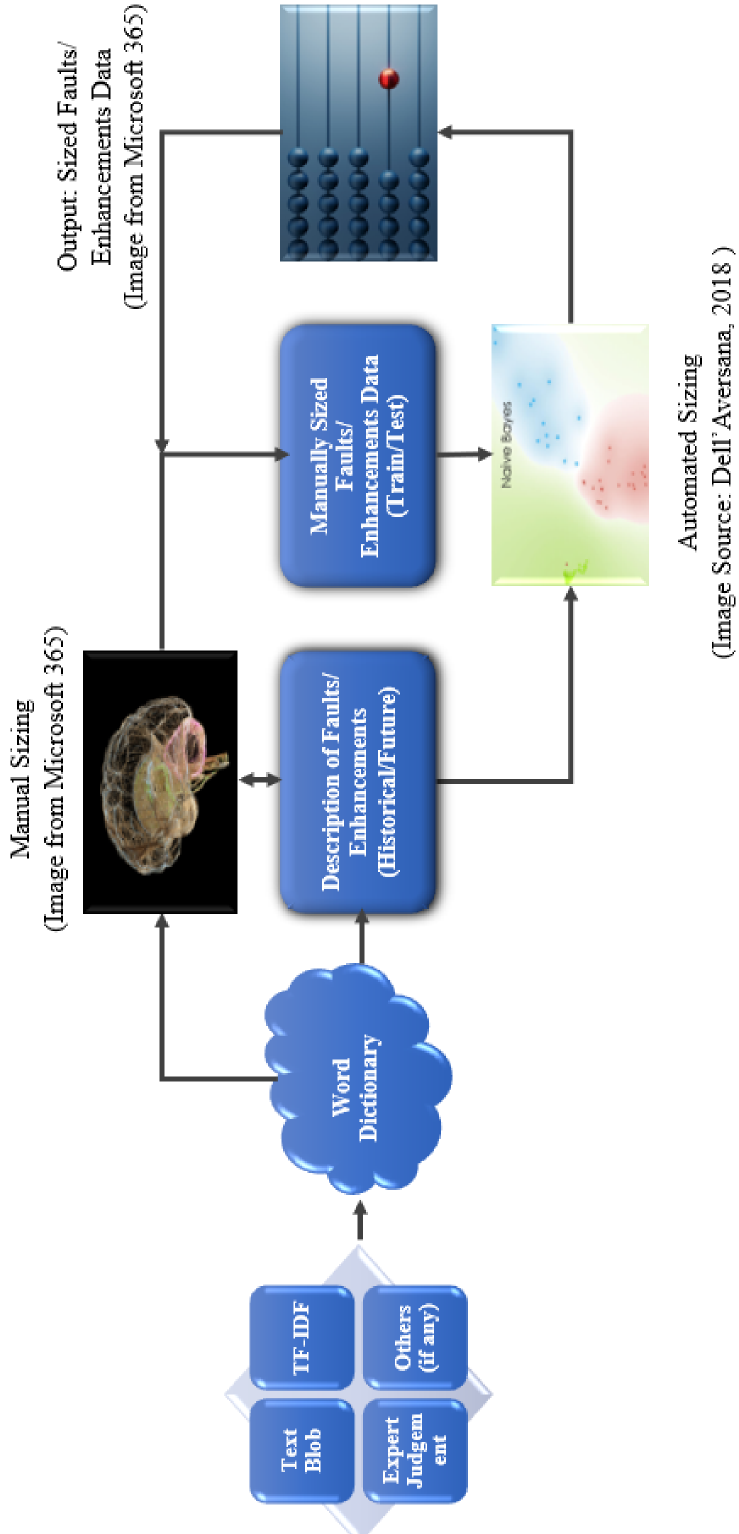

Figure 4.4: Flow chart for NLP-based estimation

The following steps are involved in this NLP process:

Segmentation and Tokenization: In this step, sentences are split into smaller chunks of words.

Part of Speech (POS) Tagging: POS is tagged with precision to categorize words in a text/float (corpus) in correspondence with a particular part of speech (POS).

Stemming and Lemmatization: Stemming removes the last few characters of words that may lead to incorrect meanings and spelling errors. Lemmatization normalizes and converts the words into their base form.

Stop words Removal: Stop words are considered as noise in the text. Text may contain stop words such as is, am, are, this, a, an, the, etc. In this step, stop words are removed.

Dependency Parsing: Dependencies between the phrases of a sentence are examined to determine its grammatical structure.

Classifications: Naïve Bayes is used to classify texts following a pre-defined look-up table of categories of SP based on the Fibonacci sequence (1, 2, 3, 5, 8).

Story points take accounts of complexities and efforts required to address and deploy faults/enhancement requests but do not consider the impacts of this deployment on performance parameters. Hence the estimated SPs are multiplied by corresponding impact factors as stated in Table 3.2. This estimation, again, is done based on expert judgment. Then they are summed up to estimate PV and are cumulated to get CPV for individual releases. Figure 4.5 and Table 4.3 show the estimated CPV for all the releases. Since CPV is estimated mainly based on expert judgments, such estimations do not have any reported error value. That is why Figure 4.5 does not carry any error bars.

There are releases where RT decreases as specific improvements are made in those releases. To realize such benefits, negative SPs are accounted for. Positive SP means an increase in the RT (i.e., performance reduction), and a negative SP contributes to the reduction in the RT (i.e., performance improvement). The improvements are often so significant that the resultant PV for the entire release is identified to be negative. For example, in Bugzilla Version 5.0.4, a graphical report feature was removed. Eliminating this feature positively impacted the RT since graphical features make applications run slower. The resultant PV, specific to that release, is negative (-0.75), and CPV is 37, a decrease of 0.75 from the earlier Release (5.0.3). A decrease in response time (RT) has been observed in this release, which is an improvement.

Similarly, in Release 5.0.5, the code was reformatted according to the same conventions (Sebastian, R., & Mojolicious contributors, 2021), and the bugs_fulltext table was then located inside the database InnoDB instead of MyISAM. This resulted in improved performance. As a result, the cumulative CPV in this Release is 36.5, a reduction (-0.5) from the previous Release, 5.0.4, where cumulative CPV was 37.

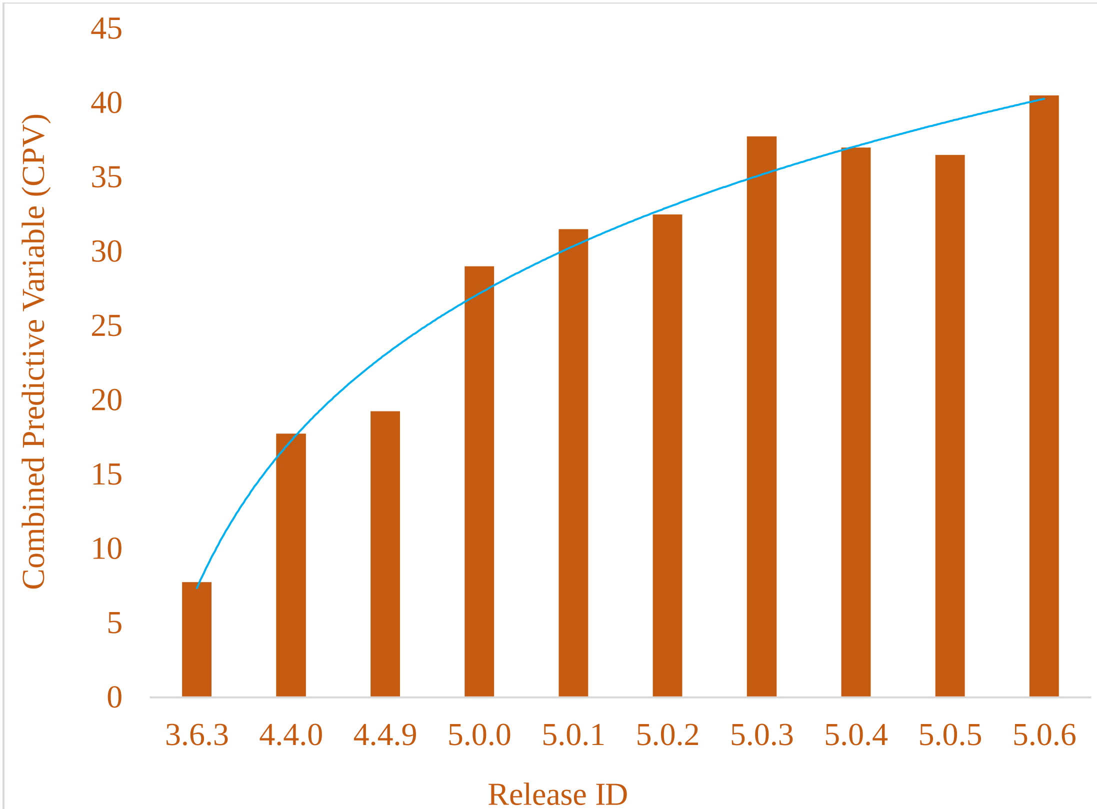

Figure 4.5: CPV over multiple Bugzilla releases

Table 4.3: CPV Over Multiple Bugzilla Releases

Bugzilla Release #	**Cumulated Predictive Variable (CPV)**
Release 3.6.3	7.75
Release 4.4.0	17.75
Release 4.4.9	19.25
Release 5.0.0	29.00
Release 5.0.1	31.50
Release 5.0.2	32.50
Release 5.0.3	37.75
Release 5.0.4	37.00
Release 5.0.5	36.50
Release 5.0.6	40.50

The historical data is randomly split into training (80%) and testing (20%) data sets to train the predictive model. R-Studio is used for this split and for the subsequent analysis.

In the next step, the clustering of releases is carried out to identify analogous releases. An elbow within-cluster Sum of the Square diagram (Figure 4.6) is formed. As shown in Figure 4.6, three would be an appropriate number of clusters for this case, as the elbow to the curve starts at three clusters. However, due to the limited data, no R Square or *p* values could be determined when running a predictive algorithm (regression) on one of the three clusters. Hence, for this demonstration, two instead of three clusters are selected.

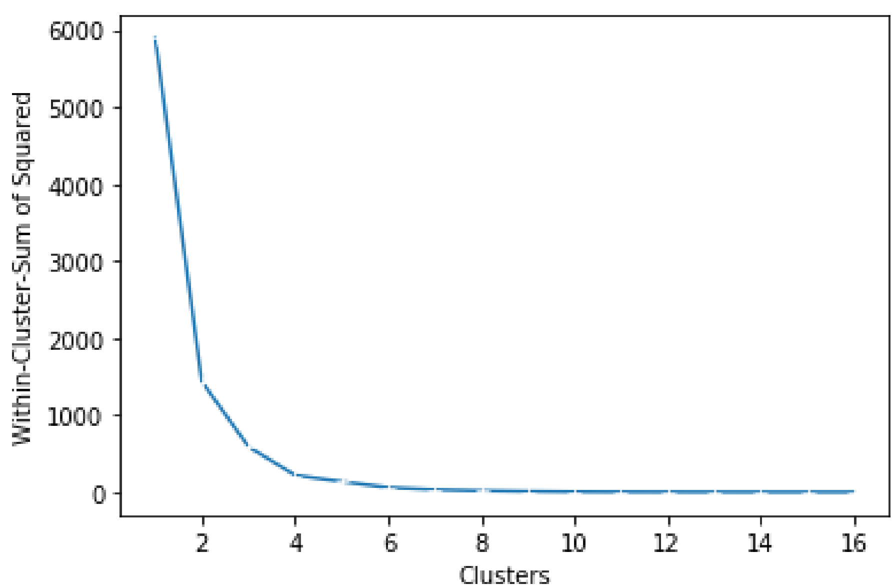

Figure 4.6: Elbow Diagram

Google Colab Notebook is used to write and run the classification and clustering algorithm in Python. Colaboratory, or “Colab” in short, is a product from Google Research. Colab allows anyone to write and execute python code arbitrarily through the browser. This tool is especially well-suited for machine learning, data analysis, and education.

Table 4.4 shows the clustering results. Releases 3.6.3 to 5.0.6 are *historical* releases, and Releases 6.0 to 7.5.1 are *future* releases. Among the future releases, 6.5.0 appears to be in the same cluster (B) as Releases 5.0.2, 5.0.5, and 5.0.6. The remainder of the releases fall into the other cluster (A).

Table 4.4: Analogous Releases Based on k-means Clustering

Bugzilla Release #	**Cluster**
Release 3.6.3	A
Release 4.4.0	A
Release 4.4.9	A
Release 5.0.0	A
Release 5.0.1	A
Release 5.0.2	B
Release 5.0.3	A
Release 5.0.4	A
Release 5.0.5	B
Release 5.0.6	B
Release 6.0.0	A
Release 6.5.0	B
Release 6.5.1	A
Release 7.0.0	A
Release 7.5.0	A
Release 7.5.1	A

With the availability of more data on future releases, such a clustering can be re-evaluated, and newer predictive model(s) or ensemble techniques may be explored. The prognostic model is validated in several different ways:

1. The actual data (on RT) generated by the test beds are compared with the predicted data.
2. The model is statistically validated based on the adjusted R^2 value. A higher adjusted R^2 value, with a low *p-value* (<0.05) indicates that the null hypothesis can be rejected, i.e., the changes in the target variable are associated with the changes in the predictive variable.
3. k-fold cross-validation is carried out to determine the fitness of the model.

A higher cross-validation score can confirm that the model is a fit for this demonstration. For validation purposes, the software under test doesn't need to reach its end-of-life since historical release data can be generated and used as test data/validation data to compare with. Since the RUL is estimated by plotting performance parameters (e.g., RT) against release cycles and since a user-defined performance threshold is used, validation of the RUL is dependent upon the validation of the RT. R-Studio is used as an analytics platform to predict RT based on regression analysis.

4.3. Prognosis and RUL Estimation

To identify the correct prognostic model(s), a correlation analysis (Figure 4.7) is carried out to evaluate the relationship between the target variable (RT) and the independent variable (CPV). Quadrant-I shows a higher degree of association (0.95) between RT (Quadrant-IV) and CPV (Quadrant-II). The direction in Quadrant-III confirms the positive nature of this relationship, i.e., as the CPV increases, RT increases. The horizontal axis in Quadrant-I represents RT while the vertical axis represents CPV and the horizontal axis in Quadrant-III represents CPV while the vertical axis represents RT. Quadrant II and Quadrant IV represent the distribution of corresponding CPV and RT. Given the high (0.95) correlation, linear regression. Equation (5), is applied to estimate the RT based on the CPV:

$$y = mx + b \tag{5}$$

where, *y* is the target (RT) variable, *x* is the predictor (CPV), *m* is the slope and *b* is the y-intercept.

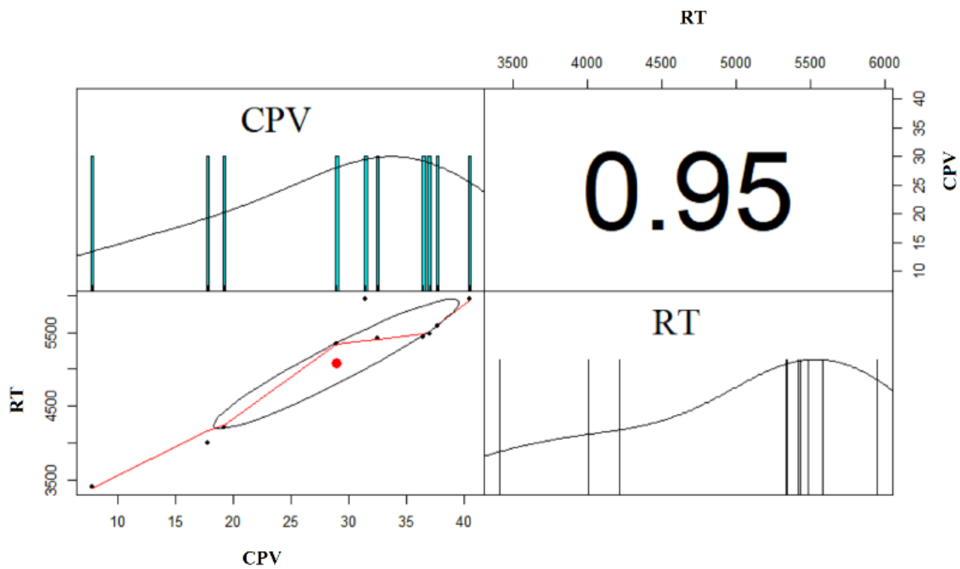

Figure 4.7: Correlation between CPV and RT

Residual analysis is carried out to validate selection of this linear regression model. The residual plot (Figure 4.8) shows that the data points are randomly dispersed across the horizontal axis. This confirms that the linear regression model is appropriate for this data.

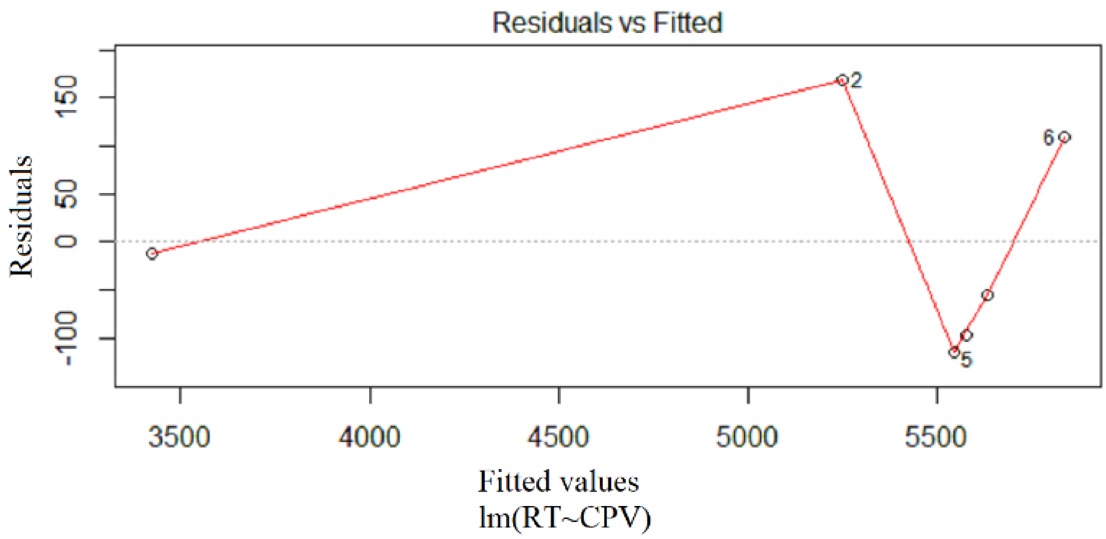

Figure 4.8: Residual analysis plot

After training the univariate linear regression with training data set, the performance of the model is validated based on test data (Table 4.5):

Table 4.5: Actual Versus Predicted RT Based on Test Data

Release	**Cluster**	**Actual RT (ms)**	**Predicted RT (ms)**
5.0.2	B	5416	5248
5.0.3	A	5579	5634

As more data becomes available in future other prognostic algorithms and their validation methods (e.g., cross-validation) could be explored. Table 4.6 shows the results of the statistical validation:

Table 4.6: Model Performance Summary

Model Performance Indicators	Cluster A	Cluster B
Multiple R^2 value	0.9839	0.8968
Adjusted R^2	0.9798	0.8796
p-value	0.00009	0.00036

For both the clusters multiple R^2 and adjusted R^2 are higher with significantly smaller *p-values (*<0.05). This confirms the good performance of the model. K-fold cross-validation is carried out to confirm the fitness of the model. This cross-validation also confirmed that the model is not overestimating or underestimating. K = 2 (2-fold cross-validation) is used due to the limited amount of data and the 70-30 split between training and test data just for the cross validation. The mean validation score is 0.961, which is an indication of reasonable fitness.

After training the prognostic model, unresolved fault data and future enhancement requests are collected and spread over six future releases with four optional combinations (referred to as Combos 1- 4) by arranging and re-arranging faults and enhancement requests in each bucket of releases. The combos are the different combinations of the same volume of planned fault fixes/enhancement deployments, spread over the same number of future releases. Different combinations help users plan and choose which combo to adapt for the longest sustainability. Release IDs for these future releases are 6.0.0, 6.5.0, 6.5.1, 7.0.0, 7.5.0 and 7.5.1 (Table 4.7). To be noted here, contents for these releases are comprised of unresolved faults/expected future enhancements. It is possible to architect as many future releases as the user wishes. The previously discussed steps are followed to estimate CPVs for all these combo releases. Clustering is carried out to identify to which cluster each of these 6 releases belongs.

Table 4.7: Multiple Optional Combination of Future Releases

Release ID	Combo-1 (CPV)	Combo-2 (CPV)	Combo-3 (CPV)	Combo-4 (CPV)
Release 6.0.0	59.50	52.50	49.50	56.50
Release 6.5.0	55.25	54.00	56.50	64.50
Release 6.5.1	59.25	59.00	63.00	67.75
Release 7.0.0	61.25	76.50	73.00	75.50
Release 7.5.0	64.25	80.00	72.00	72.50
Release 7.5.1	81.00	81.00	81.00	81.00

RTs of the future releases are predicted by plugging this data into the trained model, and the RUL is identified (see Figure 4.9) by plotting them against the future release cycle. Based on the RUL plot, it can be determined which combination of releases provides the best RT sustainability. Sustainability refers to the combination of releases that gives the maximum number of releases before the software system reaches its threshold, i.e., the longest RUL. In this demonstration, Combo-1 shows the most extended performance cycle. In Combo-1 in Release 6.0.0, several new faults emerged, e.g., JSON parsing error, query returning more comments than called for, and components popping up twice unexpectedly. These faults increased the RT (reduced performance). Also, a few enhancements were made, e.g., the user interface (UI) was enhanced, system message architectures were changed, and others increased the RT. A few faults were fixed, e.g., index.cgi (home) was made helpful for logged-in users; all

Bugzilla JavaScript was moved into separate files and others, slightly decreasing the RT. The PV for Release 6.0.o in Combo-1 is 19.

Similarly, the PVs for the subsequent releases are: (- 4.25), 4, 2,3, and 16.75. The same exercise is carried out for the three other combos. After prognostically measuring RTs and plotting them in the RUL graph (Figure 4.9) it is realized that if the developers plan releases according to Combo-1, the highest number of releases can be obtained before the software reaches the RT threshold. That means Combo-1 provides the longest RUL of this software. The OS for this test bed is 64-bit and clock speed 1.80 GHz.

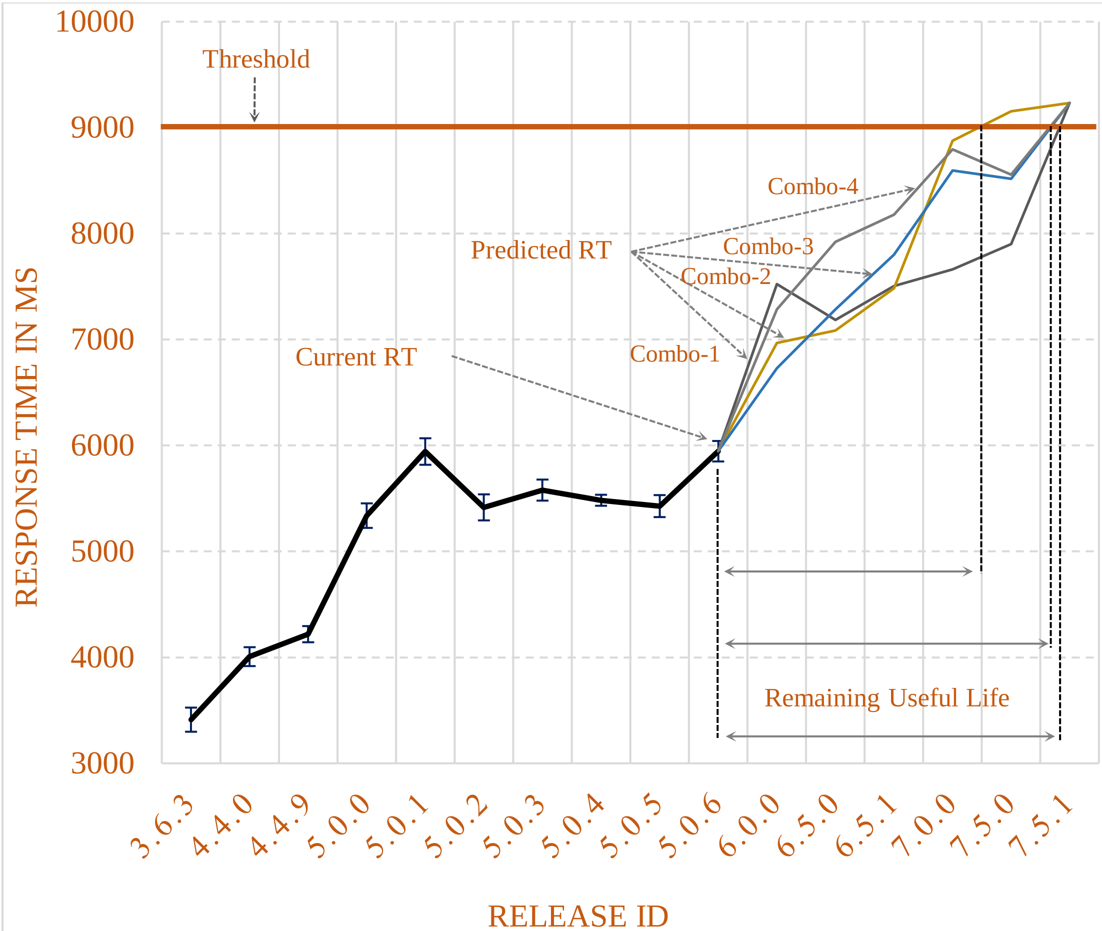

Figure 4.9: Estimation of RUL of Bugzilla on 64-bit OS and 1.8 GHz clock speed

As shown in Table 4.7, in Combo-1, the cumulated PV (CPV) for Release 6.5.0 is lower than the previous release as the PV for Release 6.5.0 is negative (-4.25). This is because more enhancements that improve RT are planned to be deployed in this release, and faults causing lower RTs are fixed in this release. As an example, loading time of query.cgi is improved with changes such as updates in improving compatibility with the Oracle server, improvements in system message architecture, fixes in JSON parsing error and code fixes so that elements do not show up twice anymore and other fault fixes and enhancement deployed.

This methodology gives the development team better control over future release planning for software updates/upgrades after identifying the critical path for the longest RUL for the software system. The number of releases and the points at which the developers will need to do a software update, upgrade, and others, can be predicted, and controlled.

Using the methodology described in this dissertation, one could find a combined RUL by combining RULs for all the relevant performance parameters. However, different performance parameters are prioritized differently for each software system by each user group. Users always make tradeoffs among performance aspects based on their requirements. For example, for some users, RT is a priority, and for others, availability (even with a lower RT) or something else. Also, there are cases when deploying a single fault fix/enhancement may affect multiple performance parameters. In such cases, the users and the development team will want to visualize how future changes could affect individual performance parameters. For example, updating a version of Bugzilla to make it compatible with a newer version of the SQL database may result in a better (shorter) RT but could reduce the availability if there are coding errors, interphase errors, or compatibility issues. Such visualization is best achieved when RULs, based on individual performance parameters, are estimated, and presented as separately as possible.

4.4. Effects of Environmental Variables on RT

In the real world, RT is not affected only by releases. Though the impacts of environmental variations in the test beds, that could affect RT, have been controlled, RTs can still be affected by other environmental attributes that influence the performance of software. And they include:

- **Operating system (OS):** Operating system (OS) manages memory and processes, as well as software and hardware of a machine(computer). There are attributes of the OS that influence software performance. One of them is *bit*. Bit is an architecture type that represents the capacity of data transfer and is closely related to the processor type (32-bit Versus 64-bit). Like processor type, the two most common bit sizes for OS are 32 and 64. From the software perspective, 64-bit computing means the use of machine code with 64-bit virtual memory addresses. As an example, a 64-bit register can hold any of 2^{64} (over 18 quintillion or 1.8×10^{19}) different values and a 32-bit any of register can hold any of 2^{32} different values. A 32-bit system has a limit of 32-bit Windows 3.2 GB of RAM even though it can address about 4 GB of RAM. The limit in its addressable space doesn't allow to use the entire physical memory of 4GB. On the other hand, A 64-bit system enables its users to store up to 17 billion GB of RAM. Software runs less efficiently on 32-bit compared to 64-bit OS, when it comes to performance.

- **Clock speed/processor speed:** The clock speed, also known as clock rate, or processor speed, indicates how fast the CPU can run. Clock speed measures the number of cycles the CPU can execute per second. A "cycle" is technically a pulse synchronized by an internal oscillator, which is a unit that helps understand CPU speed. Clock speed is measured in Hz (GHz, MHz, KHz). As an example, a 2 GHz CPU can perform 2 billion calculations every second. The higher the close speed, the faster the CPU is and the lower the RT (Sims, 2021).

- **RAM (Random Access Memory):** RAM is a computer's (also referred to as, machine) short-term memory, where data is stored temporarily so that it can access memory much faster than data on a hard disk, SSD, or other long-term storage devices. Therefore, the capacity of RAM is critical for system performance. The higher the RAM, the faster the machine (computer/server) and faster is performance, including RT.

- **Number of machine cores:** The core is a small central processing unit (CPU) or processor inside a big CPU or CPU socket. The core can independently process or perform all computational tasks (Wu & Buyya, 2015). More cores mean a faster running machine.

- **Storage or hard drive space:** A hard drive is the hardware component that stores all the digital contents. A hard disk drive (HDD), hard disk, hard drive, or fixed disk is an electro-mechanical storage device that stores and retrieves digital data using magnetic storage with one or more rigid, rapidly rotating platters coated with magnetic material. Larger storage results in faster performance of the machine (Arpaci-Dusseau & Arpaci-Dusseau, 2014)

In the following subsection, the adjusted RUL is measured based on test beds with 32-bit versus 64-bit OS. Experiments are also carried out by varying clock speed to measure the effects of hardware changes on RT and thereby on the RUL.

4.4.1. Operating System (64-bit versus 32-bit)

Experiements are carried out by varying the operating system but keeping all the other test bed attributes the same. Individual virtual environments are created for multiple versions of Bugzilla and the corresponding RTs are captured in Figure 4.10 and table 4.8. The host machine and virtual machine specifications are the same as described in Section 4.1.

Figure 4.10 shows the impacts of both 32-bit and 64-bit OS on RT . In both the cases the machine clock speed is 1.80 GHz. Experiemental results confirm the assumptions that RT is expected to be higher 32-bit OS comparing to to 64-bit OS for all releases, as the 64-bit OS has a higher processing speed. RT is also predicted for all future releases based on 32-bit OS for all the combos. Predicted RTs are plotted against release cycles to identify RUL both on 32-bit and 64-bit OS, as shown in Figure 4.11. This figure clearly shows that when Bugzilla is configured on a 32-bit machine, it reaches thresholds for all the combos even before Release 6.0.0. Whereas 64-bit OS extends the remaining useful life of Bugzilla for a few more releases.

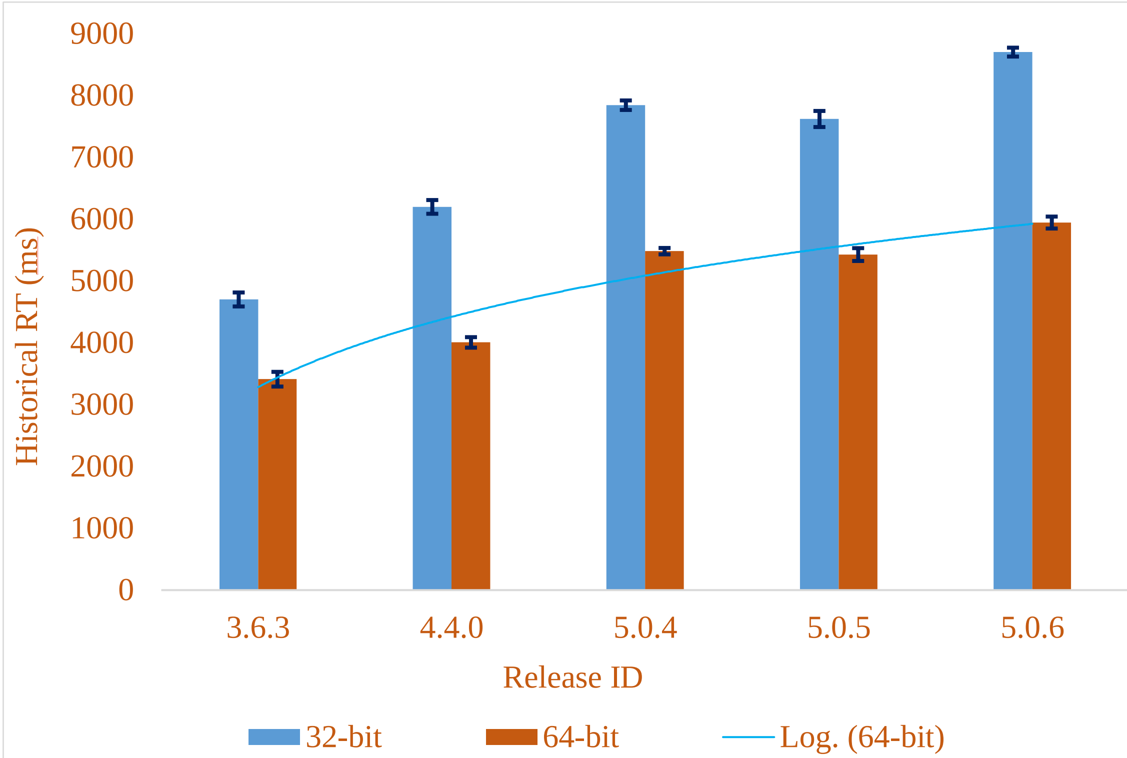

Figure 4.10: RT on 1.80 GHz machine with 32-bit versus 64-bit OS

Table 4.8: RT On 1.80 GHz Machine with 32-bit Versus 64-bit OS

Bugzilla Release	**Mean Response Time (ms) 32-bit**	**Mean Response Time (ms) 64-bit**
Release 3.6.3	4702	3412
Release 4.4.0	6199	4007
Release 5.0.4	7844	5483
Release 5.0.5	7621	5428
Release 5.0.6	8703	5946

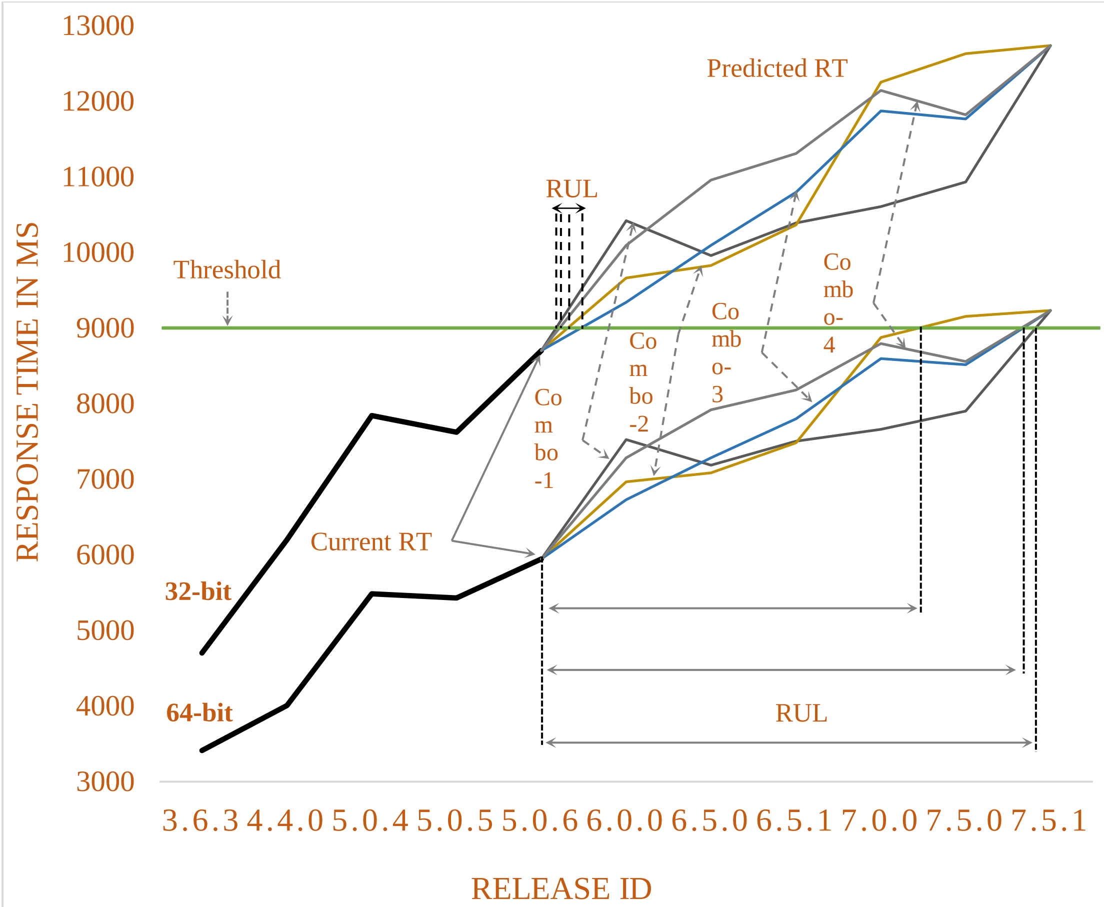

Figure 4.11: Estimated RUL on 1.80 GHz machine with 32-bit versus 64-bit OS

4.4.2. Clock Speed (Processor Speed)

As stated earlier, clock speed measures the number of cycles the CPU executes per second. During each cycle, billions of transistors within the processor open and close. Figure 4.12 shows a graphical representation of cock speed. And Figure 4.13 represents historical upgrades in clock speed over time (Microprocessor Chronology,

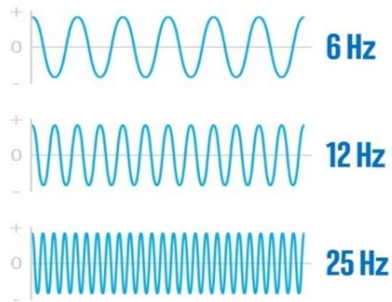

Figure 4.12: Clock speed cycle (Intel, 2022)

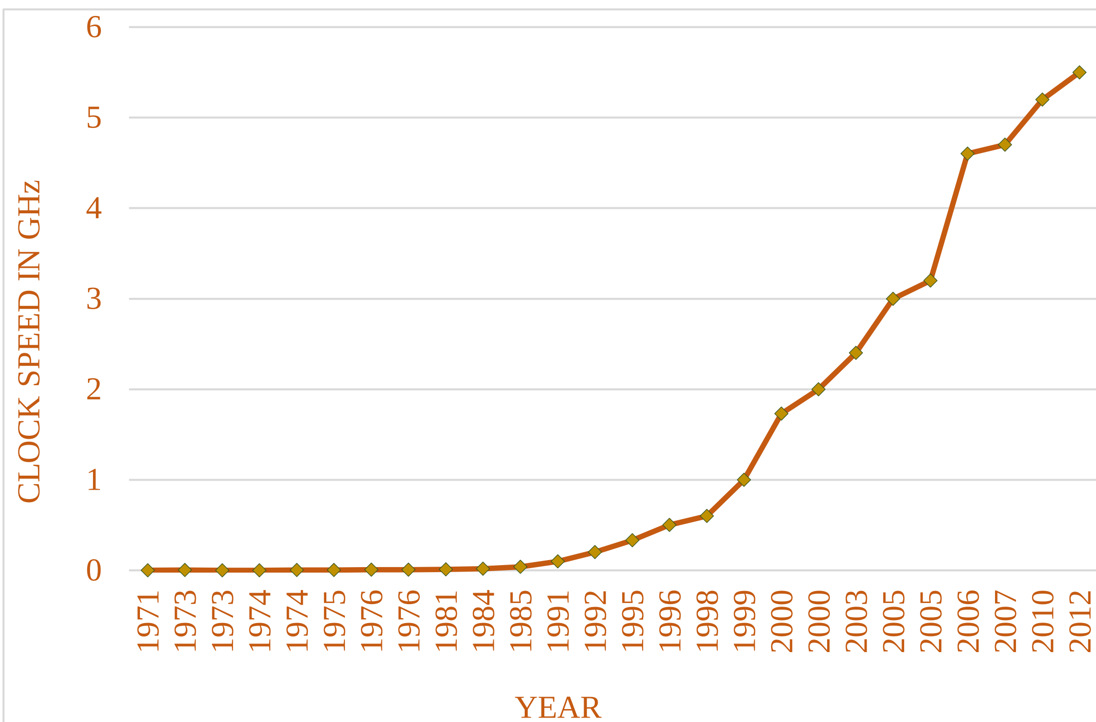

Figure 4.13: Historical clock speed (Microprocessor Chronology, 2022)

2022). According to this diagram, the highest speed of 5.5 GHz existed in the market by 2012. Processors that came out in the market beyond 2012 are either the same or lower than 5.5 GHz. Table 4.9 lists the processor speeds that came to the market after 2012.

Table 4.9: Clock Speed Introduced Beyond 2012 (Microprocessor Chronology, 2022)

Date	Model Name	Developer	Max Processor Speed (GHz)
2013	SPARC T5	Oracle	3.6
2014	POWER8	IBM	5.0
2014	Intel Broadwell	Intel	4.0
2015	z13	IBM	5.0
2015	A8-7670K	AMD	3.6
2017	Zen	AMD	4.1
2017	z14	IBM	5.2
2017	POWER9	IBM	4.0
2017	SPARC M8	Oracle	5.0
2018	Intel Cannon Lake	Intel	3.2
2018	Zen+	AMD	3.7
2019	Zen 2	AMD	4.7
2019	z15	IBM	5.2
2020	Zen 3	AMD	4.9
2020	M1	Apple	3.2
2021	M1 Max	Apple	3.2

The clock speed of the host machine on which the VMs were running, and the test beds were built is 1.80 GHz. This section estimates adjusted RULs based on varying clock speeds (2 GHz and 2.4 GHz, respectively). Experiments are carried out by varying clock speed of the test beds for Bugzilla 5.0.4 and RT is measured (Figure

4.14) for all the different clock speeds. Different "Execution Cap" setting is used to make such variances (see Appendix B for more details).

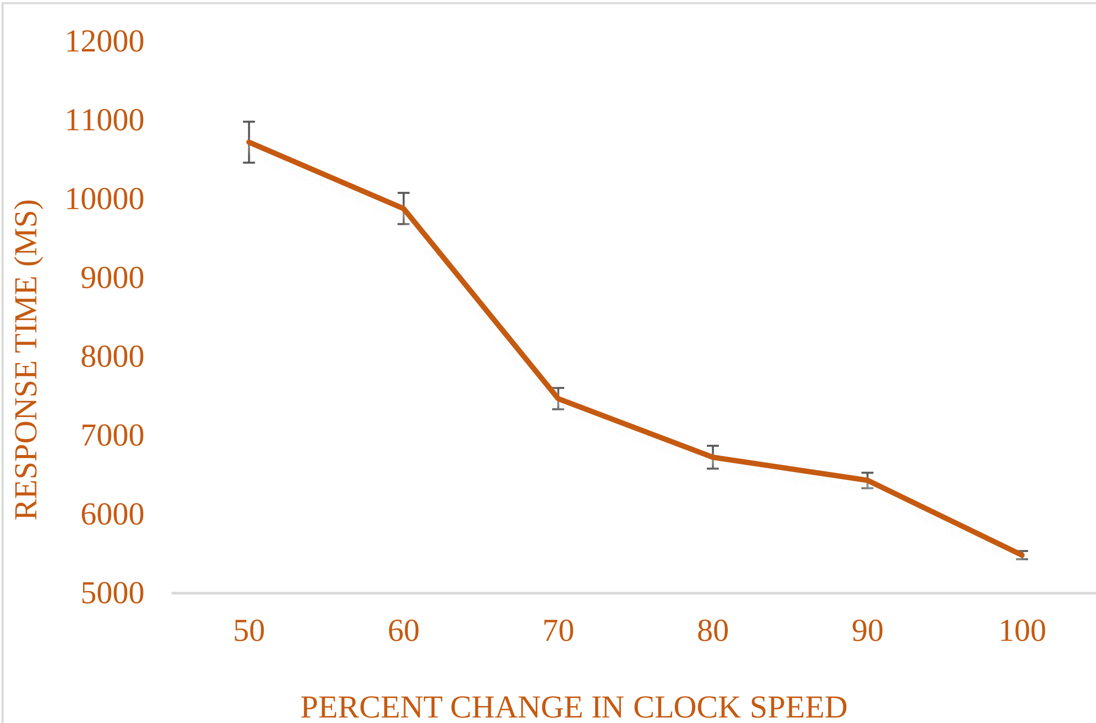

Figure 4.14: Impacts of percent change in clock speed on RT

After aggregation, it is identified that for each 10% increase in clock speed a mean 12.27% decrease (an improvement) in RT is observed. This aggregated decrease is used as a factor to modify the predicted RTs for all future releases to adjust RTs. Equation (6) is used for this estimation. The adjusted RTs are then used to predict modified RULs.

$$RT_n = RT_0\left[1 - \left(\frac{0.1227}{0.1}\right)\left(\frac{Hz_n - Hz_0}{Hz_0}\right)\right] \quad (6)$$

where, RT_n is the adjusted RT, RT_o is the previously predicted RT for future releases (Release 6.0.0 to Release 7.5.1), Hz_n is the new clock speed and Hz_o is the existing clock speed.

The first improvement in clock speed is made at Release 6.0.0 (2.0 GHz clock speed) and the second improvement is assumed to be made at Release 7.0.0 (2.4 GHz clock speed). As shown in Figure 4.15 and Table 4.10, the clock speed is upgraded from 1.80 GHz to 2.0 GHz at Release 6.0 and continued until Release 7.0.0. At Release 7.0.0, the processor is upgraded once more from 2.0 GHz to 2.4 GHz. Both the upgrades are made on 64-bit machine and resulted in a significant reduction in RTs that led to longer RUL.

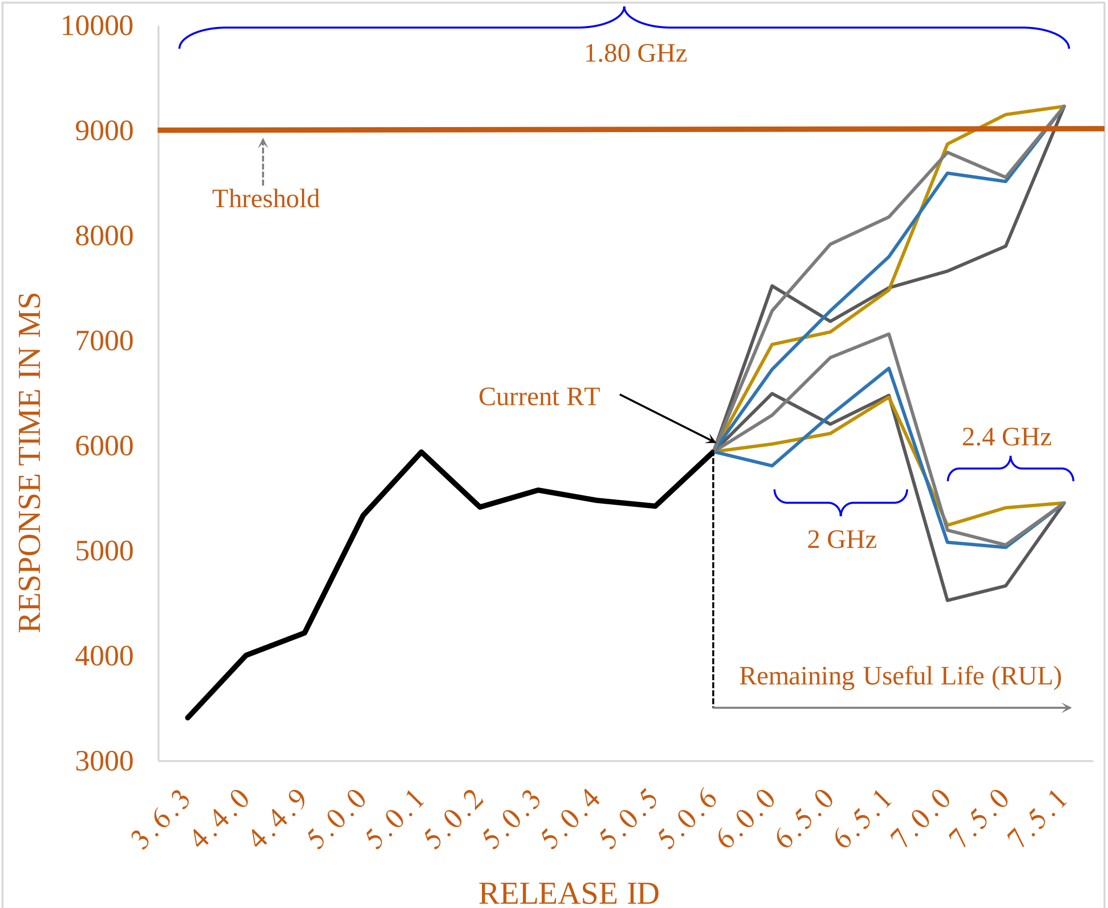

Figure 4.15: Adjusted RUL after upgrades in clock speed on a 64-bit machine

Table 4.10: Adjusted RT After Upgrades in Clock Speed on a 64-bit Machine

64-bit Processor										
Releases ID	**Test Bed Clock Speed (GH)**	**RT (ms) Combo -1**	**RT (ms) Combo -2**	**RT (ms) Combo -3**	**RT (ms) Combo -4**	**Upgrade in Clock Speed (GHz)**	**RT (ms) Combo -1**	**RT (ms) Combo -2**	**RT (ms) Combo -3**	**RT (ms) Combo -4**
Release 6.0.0	1.8	7524	6967	6728	7285	2.0	6498	6017	5811	6292
Release 6.5.0	1.8	7186	7086	7285	7921	2.0	6207	6120	6292	6841
Release 6.5.1	1.8	7504	7484	7802	8180	2.0	6481	6464	6739	7065
Release 7.0.0	1.8	7663	8875	8597	8796	2.4	4529	5245	5081	5198
Release 7.5.0	1.8	7901	9154	8518	8557	2.4	4669	5410	5034	5057
Release 7.5.1	1.8	9233	9233	9233	9233	2.4	5457	5457	5457	5457

Again, in Figure 4.16 and Table 4.11, RUL is estimated based on modified clock speed but on a 32-bit machine. Both 2.0 GHz and 2.4 GHz processors show better promises than RTs on 1.80 GHz processor on 32-bit machine. Although, in few combos, RTs still cross the threshold of 9 sec in Release 6.0.0 to Release 6.5.1 on 2.0 GHz processor, all the combinations of RTs are <9 sec once processor speed is increased to 2.6 GHz.

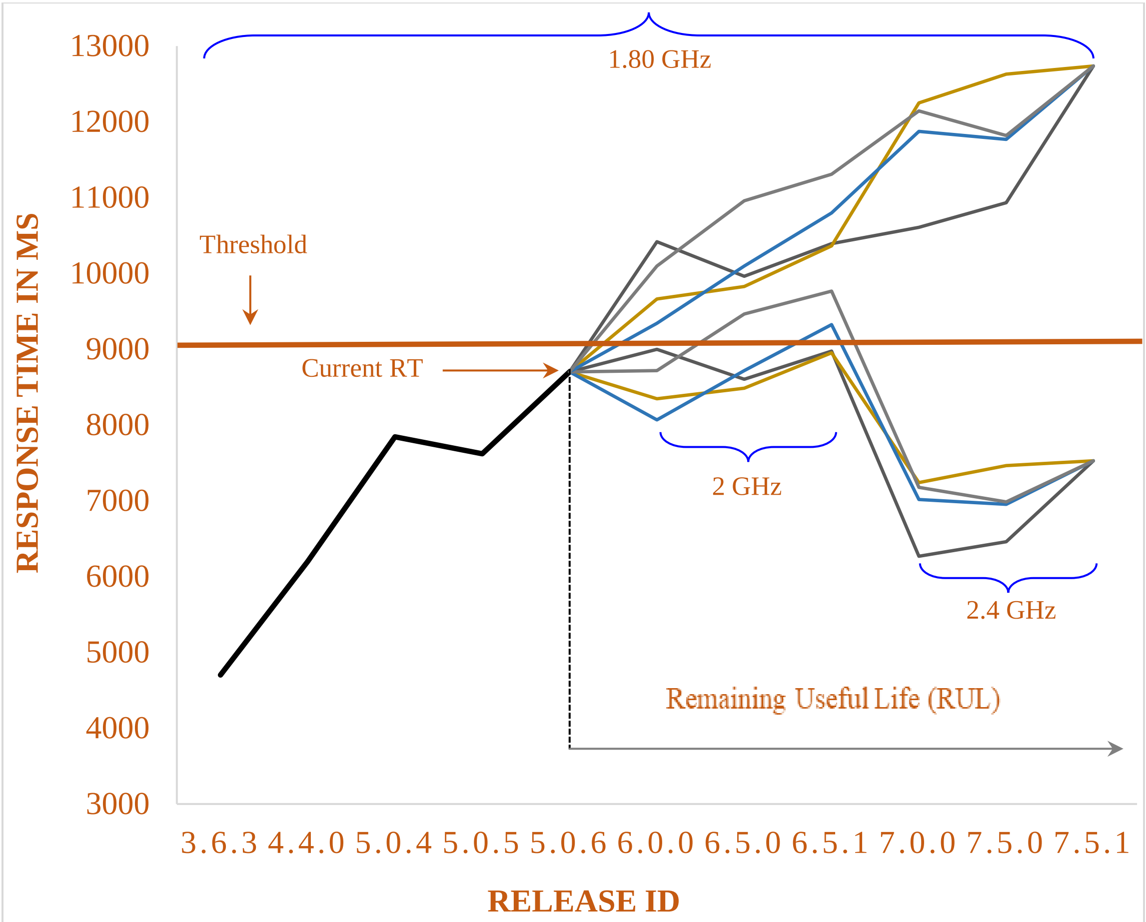

Figure 4.16: Adjusted RUL after upgrades in clock speed on 32-bit machine

Table 4.11: Adjusted RT After Upgraded Clock Speed on a 32-bit Machine

32-bit Processor										
Releases ID	**Test Bed Clock Speed (GHz)**	**RT Combo-1**	**RT Combo-2**	**RT Combo-3**	**RT Combo-4**	**Upgrade in Clock Speed (GHz)**	**RT Combo-1**	**RT Combo-2**	**RT Combo-3**	**RT Combo-4**
Release 6.0.0	1.8	10420	9665	9342	10096	2.0	8999	8348	8068	8720
Release 6.5.0	1.8	9961	9827	10096	10959	2.0	8604	8487	8720	9465
Release 6.5.1	1.8	10393	10366	10797	11309	2.0	8976	8953	9325	9767
Release 7.0.0	1.8	10608	12252	11875	12144	2.4	6269	7241	7018	7177
Release 7.5.0	1.8	10932	12629	11767	11821	2.4	6461	7464	6954	6986
Release 7.5.1	1.8	12737	12737	12737	12737	2.4	7528	7528	7528	7528

Furthermore, as shown in Figure 4.17, when the upgrade is made both in OS (from 32-bit to 64-bit) and clock speed (from 1.8 GHz to 2.0 GHz and 2.4 GHz, respectively) a significant improvement in RUL is observed across all the combinations of releases.

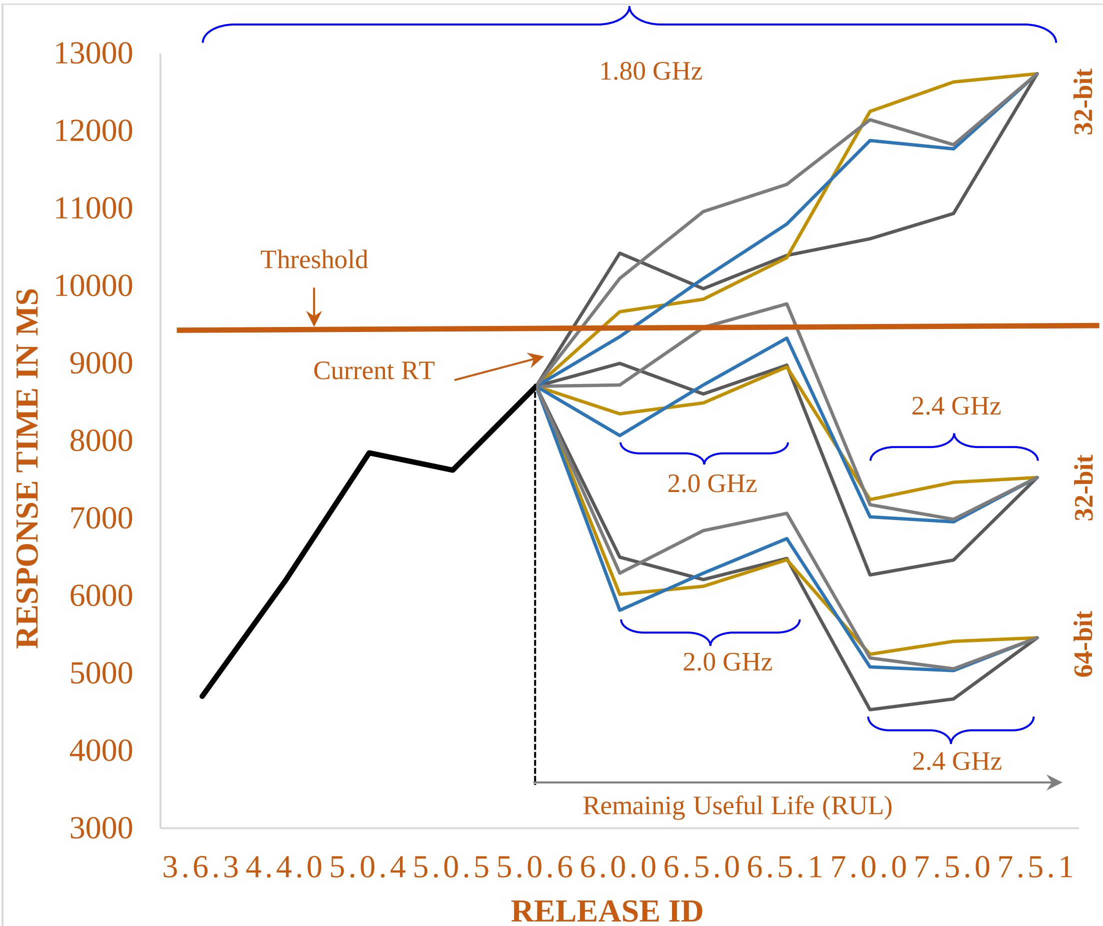

Figure 4.17: Adjusted RUL after upgrade in GHz and bit (32 versus 64)

Further research needs to be carried out to identify if and how other factors of the OS and processor, such as architecture, compatibility with other components; affect software performance.

Chapter 5: Summary and Contributions

This chapter summarizes the research performed in this dissertation, discusses the contributions, and proposes directions for future work.

5.1 Summary

This dissertation demonstrates that Prognostic Health Management (PHM) concepts and techniques can be utilized for software systems. Although more usage parameters can be explored, the release cycle appears to be an appropriate usage parameter for software systems, knowing that software degrades/decays and performance changes only when upgrades and updates are made over releases. Although this demonstration addresses only one performance parameter, which is the response time (RT), a similar methodology is applicable when other performance metrics are to be considered. Though, appropriate adjustments will be needed. If other performance parameters are to be used, the corresponding input variables that only affect them must be identified and mapped. Then the steps presented in the methodology should be followed to estimate the RUL with new performance metrics. Adjustments in data collection, processing, and analyzing data may be needed. In future research, other statistical models should be explored as more data becomes available, and accuracy should be compared.

Calculating RUL with multiple options ("combos") assists both the users and the development team in estimating and planning for the number of releases that they can have before software reaches its performance thresholds. At the same time such combos give them control over the quantity and contents of releases. Representation of RUL of the software system allows the development team to visualize past and future releases. This assists in planning and decision-making.

Other benefits include the ability to mitigate component failure/ineffectiveness through proper planning and maintenance. Timely identification of RUL trends allows developers to anticipate and avoid issues that could limit software performance and/or reduce software system life. When a frequent cause is observed, developers can adjust their responses accordingly to minimize problems. Also, over time, the developer's knowledge of the software system(s) is enriched by continuously architecting release details. Such knowledge, over time, can enhance the accuracy of future diagnoses and prognoses (Agarwal et al., 2015).

This methodology, with appropriate adjustments, can apply to other performance parameters. However, to apply this predictive and target variables need to be identified and aligned carefully, and CPV needs to be estimated. Methods to estimate CPV for other performance parameters could vary depending on input data types and such methods should be explored in future research.

The methodology described in this dissertation potentially applies to all software types that have performance requirements, especially RT. And almost all the software types have performance requirements. For example, even real-time programs need to guarantee *a response* within specified *time constraints* that are often referred to as “deadlines” (Mordechai, 1990). Such responses are often understood to be in the order of milliseconds and sometimes microseconds. As an example, a pacemaker is a real-time system that needs to operate fast enough to guarantee the health and safety of the patient (Hillary, 2005). Therefore, RT is one of the most critical performance criteria even for any real-time system. Similarly, embedded software is designed to meet real-time system requirements. Such software requires precise real-time responses to the microsecond in a distribution system, and the system is expected to be fault tolerant under strict timing requirements (Happe, 2005).

The demonstrated methodology is a fusion between data-driven, model-based, and expert judgment. It takes account of the descriptions, frequencies, and impacts of faults/enhancement requests and inherent properties and data categories. The demonstrated methodology is independent of the inherent complexity and architecture of the software. Therefore, this methodology is independent to the types of coding language(s), numbers and types of integrated components, and types of associated hardware. To apply the proposed fusion methodology, the development team is not required to be experts on the inherent architecture and the complexity of any software system. Though that would be beneficial. While variations in estimated values of CPV and output parameters (e.g., RT) are expected even for the same software system in different environments, due to environmental attributes this methodology will still work equally well for a software application hosted on a local machine (e.g., a local server-based software system) or in a cloud environment. This is because, in both environments software systems will incur faults and enhancements. Even though their numbers and estimated CPV may vary, the demonstrated methodology will be applicable if data is available on faults and enhancements. The same is true when it comes to changes in clock speed or other attributes.

5.2 Contributions

The contributions of this dissertation include the following:

- This work proposed a definition of RUL for software.
- This is the first work to propose a methodology that can prognostically and dynamically predict the RUL of a software system based on new usage parameters and multiple performance parameters (e.g., response time, throughput, availability, and others).
  - A new usage parameter (*Software release* instead of *Calendar Time*) has been introduced and is found to be the most appropriate measure for RUL for a software system.
  - Dynamically estimated RUL can be used for decision-making on software upgrade/update/integration, the addition of new features, improving existing features, the timeframe for such changes, system re-engineering, predictive maintenance, budgeting, and others.
  - Data Quality Assessment Metrics (DQAM) and fault/enhancement requests categorization metrics have been introduced.

    - A Natural Language Processing (NLP) based story point estimation technique has been introduced.

- A detailed demonstration of the methodology on the Bugzilla software application has been performed. This demonstration includes the first reported prediction of RUL for software based on software release schedule.

5.3 Future Work

Though this work demonstrated that PHM-based RUL could be applied to a software system, there are many open issues that should be addressed in future work, including:

- In this dissertation, RUL is estimated based only on one performance parameter (target parameter) and that is response time (RT). Other performance parameters, in addition to RT, are followed in the industry, as discussed in Section 3.3.1. They are execution time, throughput, availability, utilization, and others. The methodology developed in this dissertation can also be applied to other target variables. However, measuring historical data on different performance parameters may require different types of health monitoring and performance measurement tools. A few health monitoring tools are listed in Section 3.5, and other tools are available in the market. Some of the target variables are measured in continuous values and the rest are measured in categorical values, e.g., the availability of software could be either ‘Yes’ or ‘No’. More research is needed in this area.

- The virtual machine used for this demonstration is based on Windows 10 operating system. More research is needed to identify how the software under test performs on the operating system in use at the time of the Bugzilla version’s use, such as Windows 8.1, Windows 8, Windows 7.
- There is no ground truth for validation of the prognostic model for future releases, except data-driven prediction based on historical data. More research needs to be done to build ground truths for future releases based on the future releases that can be verified and validated.
- The demonstration in this dissertation utilizes regression analysis to predict the target variable (RT) in the prognostic section. As more data becomes available other predictive algorithms, such as models that are already being used in software reliability estimation should be explored. Even machine learning algorithms such as artificial neural networks are candidates. Ensemble techniques should be explored as well.
- The data quality assessment matrix has four dimensions, associated sub-dimensions, and criteria questions. As the volume and variety of data grows, research should be carried out to bring more granularity into the data quality assessment matrix.
- Since changes in the software system are carried out in releases, the release cycle has been selected as the usage parameter. However, more research should

be carried out to identify if there are other better candidates. CPV takes account of estimated story points of the input variables and impacts of individual fault fixes/enhancements on the performance parameters. More research should be carried out to identify if more factors (such as variations in RAM and HDD) need to be incorporated into the CPV. Research can also be carried out to identify if there are better alternatives than the CPV.

- Research should be carried out to divide and sub-divide faults/enhancements categories and sub-categories. More granularities can assist in better estimation of story points.
- Even though NLP is used for the automated estimation of story points, the estimation process is still highly dependent on expert judgement. Research on more methods for automation should be carried out.
- Other clustering algorithms could be explored, such as Long-Deep Recurrent Neural Net (LD-RNN).
- Though there are no industry standards for thresholds, an acceptable range of thresholds could be built based on different software classes. Building Pareto charts could help in identifications of thresholds or something close. The threshold is not static and future research can be carried out to identify how, over time, application-specific user requirements impacting thresholds is expected to evolve. Also, factors that would motivate such evolution could be identified. Prospective evolution of thresholds can be grouped according to different types of software.

- Since the requirements and the priority of performance parameters may vary from software system to system and user to user, keeping the visualization of RUL separate for easier understanding and making educated tradeoffs make sense at this point. However, research should be carried out to determine if a consolidated performance parameter should be considered.
- In this research, faults/enhancement requests are categorized based on environments, architecture, modifications of syntax and others. Further granularity in the categorization matrix could be useful.
- This dissertation utilizes a selected model validation technique due to the limited availability of test and training data (elbow diagram, k-fold validation, adjusted R2, and *p*-value). As more data becomes available, model performance should be evaluated based on other validation techniques.
- Four different impact factors are chosen in this dissertation. Further segmentation of the impact factors should be carried out that could bring more precision in estimating impacts of faults/enhancements on performance variables.
- Bugzilla is chosen for this demonstration because configuration files for many historical releases are open-source and functional. In addition, all the historical and future faults and enhancements data are public for Bugzilla. Demonstrations could be performed on other software. However, finding software with publicly available information and histories to test on is non-trivial.

- This demonstration is designed based on four different combos for six future releases. More combination and more releases could be architected.
- This method could be applied to multiple software systems and compared.
- Research could be carried out to factor in other hardware and software usage parameters such as degradation and maintenance attributes such as restoration.
- Software durability is the solution's ability of serviceability of software and meet user needs for a relatively long time. Research can be carried out to identify how PHM can be applied to improve software durability. User viewpoint needs to be incorporated when improving durability.

5.4 Publications from this Dissertation to Date

Islam, M.R., & Sandborn, P. (2022), "Demonstration of a Response Time Based Remaining Useful Life (RUL) Prediction for Software Systems," *Journal of Prognostics and Health Management,* https://doi.org/10.22215/jphm.v2i2.0000, ISSN: 2563-6685, Accepted and in production for publications.

Islam, M.R., & Sandborn, P. (2022), "Application of Prognostics and Health Management (PHM) to Software System Fault and Remaining Useful Life (RUL) Prediction," *Proceedings of the ASME International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, August 17-20, Virtual.

Appendix A – Detailed RUL Analysis

This appendix provides detailed documentation on how the RUL calculations were carried out in this dissertation, including relevant Python and R scripts.

After input (faults and enhancements) data is quality-verified and categorized, they are individually estimated in terms of story points (as described in Section 4.2.2) using a Fibonacci sequence (1,2,3,5,8). This estimation is made in combination with expert judgment and by an automated NLP process. To train the NLP model, manual estimation is carried out and is divided into training and testing data set. The training data set trains the Naïve Bayes-based classification algorithm (for NLP), and testing data verifies its performance.

Story Point Estimation

The Python code for NLP is:

```
# -*- coding: utf-8 -*-
'''Copyright 2022: Mohammad Rubyet Islam '''

import numpy as np
import pandas as pd
import nltk
import string
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn import naive_bayes
from sklearn.metrics import roc_auc_score

df = pd.read_csv('source_file.csv')


#STOPWORDS
stopset = set(stopwords.words('english'))

#VECTORIZE
```



```
vectorizer = TfidfVectorizer(use_idf=True, lowercase=True, strip_accents='ascii', stop_words=stopset)

y= df.category
x= vectorizer.fit_transform(df.summary)

print (y.shape)
print (x.shape)

x_train, x_test, y_train, y_test = train_test_split(x, y, random_state=42)

clf = naive_bayes.MultinomialNB()
clf.fit(x_train, y_train)

y_pred = clf.predict_proba(x_test)
print(y_pred)
print(y_test)
print(y_train)
roc_auc_score(y_test, y_pred, multi_class='ovo')

#CALCULATE CONFUSION MATRIX
from sklearn.metrics import confusion_matrix
confusion_matrix(df['category'],clf.predict(category_vector))

auc = pd.read_csv('source_file.csv')
act = auc.actual
pred = auc.predicted
roc_auc_score(act, pred, multi_class='ovo')

x= vectorizer.fit_transform(df.summary)

category_array = np.array(["Provide Encrypted WAN (R2)"])
category_vector = vectorizer.transform(category_array)
print (clf.predict(category_vector))
```

The estimated story points are multiplied (Equation (1)) with impact factors from the impact scale (Table 3.2), to get the weighted predictive variable (*PV*) for individual faults/enhancements. The weighted predictive variables for individual faults/enhancements could either be positive or negative depending on their impacts on RT. And such directions (positive/negative) of the impacts are identified through expert judgements. After summing the PVs for individual faults/enhancements following

Equation (1), the total weighted *PVs* for individual releases are estimated. Equation (2) is used to accumulate *PV* (CPV) for consecutive releases. Table A.1 shows the example estimation of SP, IF and PV for Bugzilla Release 5.0.4:

Table A.1: PV Estimation for Bugzilla Release 5.0.4

Bugzilla ID (Faults/ Enhancements)	**Description**	**Impact Factor (IF)**	**Impact Direction (+/-)**	**Estimated Story Points (SP)**	**Weighted Faults/ Enhancements (PV)**
1533368	Perl script checkconfig.pl ignores value in local config file. I have 2 servers running on Centos 7. Issue happens when executing the checksetup.pl perl script. When running checksetup.pl, the script triggers an error.	0.25	Positive	1	0.25
1545043	Graphical Reports Features Not Available in this Version of Bugzilla. May be a positive effect on response time	0.5	Negative	-8	-4
1473226	There was an error connecting to MySQL:install_driver(mysql) failed: Attempt to reload DBD/mysql.pm aborted. It looks like DBD::MySQL has been compiled using another version of MySQL, maybe 5.5 which is the original version available in RHEL/CentOS 7.	0.5	Positive	2	1
1480378	Error on sending Email from Bugzilla through Exchange server	0.25	Positive	2	0.5
1592129	Bugzilla failed after mysql upgrade to 8.0.17 due to "groups" keyword in mysql	0.25	Positive	2	0.5
1508531	Enhancement Request: REST API for searching attachments. There is an API to search the bugs, but not for attachments. Such an API should be there and would be quite useful.	0.25	Positive	1	0.25
Security	Via the image generation in report.cgi, a malicious site using the presence of certain images, could extract potentially confidential information if the victim was logged in and could access the bug.	0.25	Positive	1	0.25
1310728	editflagtypes.cgi would crash when classifications are enabled, and the user did not have global edit components privileges.	0.25	Positive	2	0.5
Total Weighted Predictive Variable (PV) in Release 5.0.4					**-0.75**

PV for Release 5.0.4:

$$PV_{5.0.4} = SP_1 \text{ x } IF_1 + SP_2 \text{ x } IF_2 + SP_3 \text{ x } IF_3 + SP_4 \text{ x } IF_4 + SP_5 \text{ x } IF_5 + SP_6 \text{ x } IF_6 + SP_7 \text{ x } IF_7 + SP_8 \text{ x } IF_8$$

$= [1 \times 0.25] + [-8 \times 0.5] + [2 \times 0.5] + [2 \times 0.25] + [2 \times 0.52] + [1 \times 0.25] + [1 \times 0.25] + [2 \times 0.25]$

$= (-0.75)$

CPV for release 5.0.4:

$CPV_{5.0.4} = PV_{3.6.3} + PV_{4.4} + PV_{4.4.9} + PV_{5.0} + PV_{5.0.1} + PV_{5.0.2} + PV_{5.0.3} + PV_{5.0.4}$

$= 7.75+10+1.5+9.75+2.5+1+5.25+(-0.75)$

$= 37$

CPVs for all the other historical and future releases are estimated as shown in Table A.2. For example, Release 3.6.3 to Release 5.0.6 are historical releases, and Releases 6.0 to Release 7.5.1 are future releases. Future releases are architected with faults/enhancement requests to be fixed and deployed.

Table A.2: Combined Predictive Variable (CPV) for All the Releases

Release #	Weighted Faults/ Enhancements (Weighted PV)	Combined Predictive Variable (CPV)
3.6.3	7.75	7.75
4.4.0	10.00	17.75
4.4.9	1.50	19.25
5.0.0	9.75	29.00
5.0.1	2.50	31.50
5.0.2	1.00	32.50
5.0.3	5.25	37.75
5.0.4	-0.75	37.00
5.0.5	-0.50	36.50
5.0.6	4.00	40.50
6.0.0	19.00	59.50
6.5.0	-4.25	55.25
6.5.1	4.00	59.25
7.0.0	2.00	61.25
7.5.0	3.00	64.25
7.5.1	16.75	81.00

Elbow Diagram

The following Python script is used to build the elbow diagram to identify the optimum number of clusters (shown in Figure A.1):

```
# -*- coding: utf-8 -*-
"""Copyright 2022: Mohammad Rubyet Islam"""

import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

#Check Directory
import os
current_dir = os.getcwd()
```



```
print(current_dir)

#import data
df = pd.read_csv("cluster-1.csv")
df

# import library
import sklearn.cluster as cluster

K=range(1,17)
wss = []
for k in K:
    kmeans=cluster.KMeans(n_clusters=k,init="k-means++")
    kmeans=kmeans.fit(df)
    wss_iter = kmeans.inertia_
    wss.append(wss_iter)

mycenters = pd.DataFrame({'Clusters': K, 'Within-Cluster-Sum of
Squared' :wss})
mycenters

sns.lineplot(x = 'Clusters', y = 'Within-Cluster-Sum of Squared',
data = mycenters, marker="+")
```

Using the above codes an elbow diagram (Figure 4.6 in section 4.2.2) is generated to identify number of clusters.

Clustering of Analogous Releases

The following Python script runs k-means clustering to identify analogous releases based on 2 clusters on Google Colab, and the following Figure A.1 shows the resulting clusters of releases:

```
#K-Means

k =2 # Number of cluster
model = KMeans(n_clusters=k,init='k-
means++',max_iter=100,n_init=1)
model.fit(features)

df['cluster'] = model.labels_
```



```
#Output the result to a text file
clusters = df.groupby('cluster')
for cluster in clusters.groups:
  f = open('cluster'+str(cluster)+ '.csv','w') # create csv file
  data = clusters.get_group(cluster)[['bugzilla_release','avg_res
ponse_time', 'release_information']] #get title and overview colu
mns
  f.write(data.to_csv(index_label='id')) # set index to id
  f.close()
```

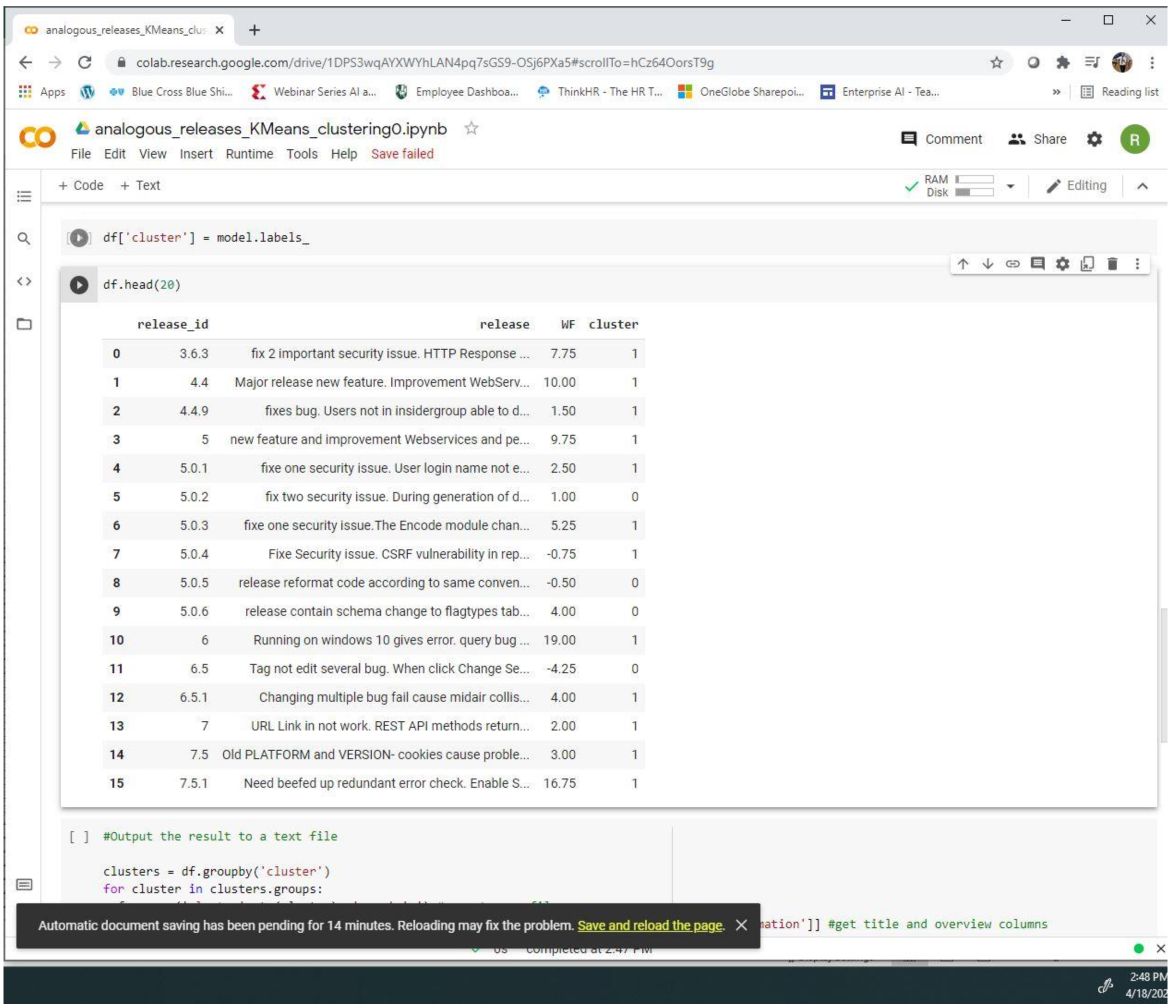

Figure A.1: Clustering of analogous releases

PROGNOSIS

Correlation

An R script is used to identify if any correlation exists between input variables:

```
#import .csv
dm <- read.csv("source_file.csv")
dm

# Correlationship among variables
# Install library 'psych'
install.packages("psych")
library(psych)
pairs.panels(dm,
             gap = 0,
             pchr = 15)
```

Prediction

Prediction of response time (RT) for future releases is determined based on analogous releases using the following linear regression model (R code):

```
#LinearModel
LinearModel = lm(response_time~CPV, data = dm)
summary(LinearModel)
```

The above scripts estimate the intercept (b) and slope (m) and predicts RT with following relationship:

$$y = mx + b \tag{4}$$

where, *y* is the target (RT) variable, and *x* is the predictor (CPV).

As an example, RT for future Release 6.0:

m = 79.51

b = 2792.87
And *x* (Combo-1) = 59.5

Hence,

RT (*y*) = 79.51 x 59.5 + 2792.87 = 7524 ms

Upon prediction of RT for the other releases, they are plotted against releases and the RUL is estimated (Figure 4.9).

Cross Validation

2-fold cross validation is carried out using the following Python script that rendered a mean value of 0.961:

```
# -*- coding: utf-8 -*-
""" Copyright 2022: Mohammad Rubyet Islam """

import os
import pandas as pd
mydata = pd.read_csv("source file.csv")
import numpy as np
X = np.array(mydata[['CPV']])
X.shape

y = np.array(mydata['RT'])
y.shape

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,y, 
test_size = 0.3)
X_test.shape
X_train.shape

from sklearn.linear_model import LinearRegression
model = LinearRegression()
mymodel = model.fit(X_train, y_train)

from sklearn.model_selection import cross_val_score
scores = cross_val_score(mymodel, X_train, y_train, scoring = 
'r2', cv =2)
scores

print (np.mean(scores))
```

```
from sklearn.model_selection import cross_val_predict
pred = cross_val_predict(model,X_test,y_test, cv =2)
pred

# 2 fold cross validation

scores_test = cross_val_score(model, X_test, y_test, cv =2)
scores_test

print(np.mean(scores_test))
```


Appendix B – Controlling Environmental Influences on the Test Beds

The experiments in this dissertation were carried out in controlled test bed environments. The following are among that variables that could influence the test beds:

- **Temperature:** Increased performance and increased heat generation are interdependent. Excessive heat can lead to throttling, which results in slowed performance (SVALT). Results from experiments carried out on Intel cores show that Intel CPUs are impressively stable, and stable performance can be expected from an Intel CPU if the temperature is kept below 212°F (100° C) (Best, 2015).

  The test bed machine has an Intel CPU with a built-in cooling system called Hyperbaric Cooling, which pulls cool air in, blows it over the motherboard and CPU, and then pushes it out of the vents. This cooling system is, also, theoretically very effective. The test machine is kept on a hardwood surface where ample air can pass all around the machine. Also, excessive humidity or dry air can exaggerate the effects of extreme temperatures on computer components (Smathers, 2019; Nadig, 2022). A room temperature of appx. 75^0 F is maintained with a humidity of appx. 65%. Hence, no bias on the test bed is assumed due to temperature or humidity.

- **Connectivity** (such as internet connection): Varying connectivity speed results in the varying performance of the application under test. To avoid bias, both the host computer and the virtual machines are disconnected from the internet/intranet during the test, and Bugzilla is configured to run on the local host. However, another reason for running on a local host is that not all the versions of Bugzilla are available on the web.

- **Storage:** Across the virtual machines, RAM and HDD capacity is kept the same.
- **Browser and versions:** Microsoft Edge (Version 88) is the browser used to open Bugzilla across all the test beds.
- **Selection and configuration of performance testing tool(s)**: Apache JMeter-Version: 5.4, is selected to measure RT for all the versions of Bugzilla.
- **Isolation of the test bed:** It is essential to ensure that no other processes were running when a test is in progress. However, if any processes needed to run in the background, it is necessary to make sure that the same processes are running across all the test beds. A dedicated VM ensures such isolation. The processes that were running in the background across all the test beds during the test are: JDK Platform, XAMPP, and a command line, Microsoft Edge. The utilization of RAM and HDD is kept the same. Figure B.1 shows the processes running across all the test beds while JMeter took readings on RT.

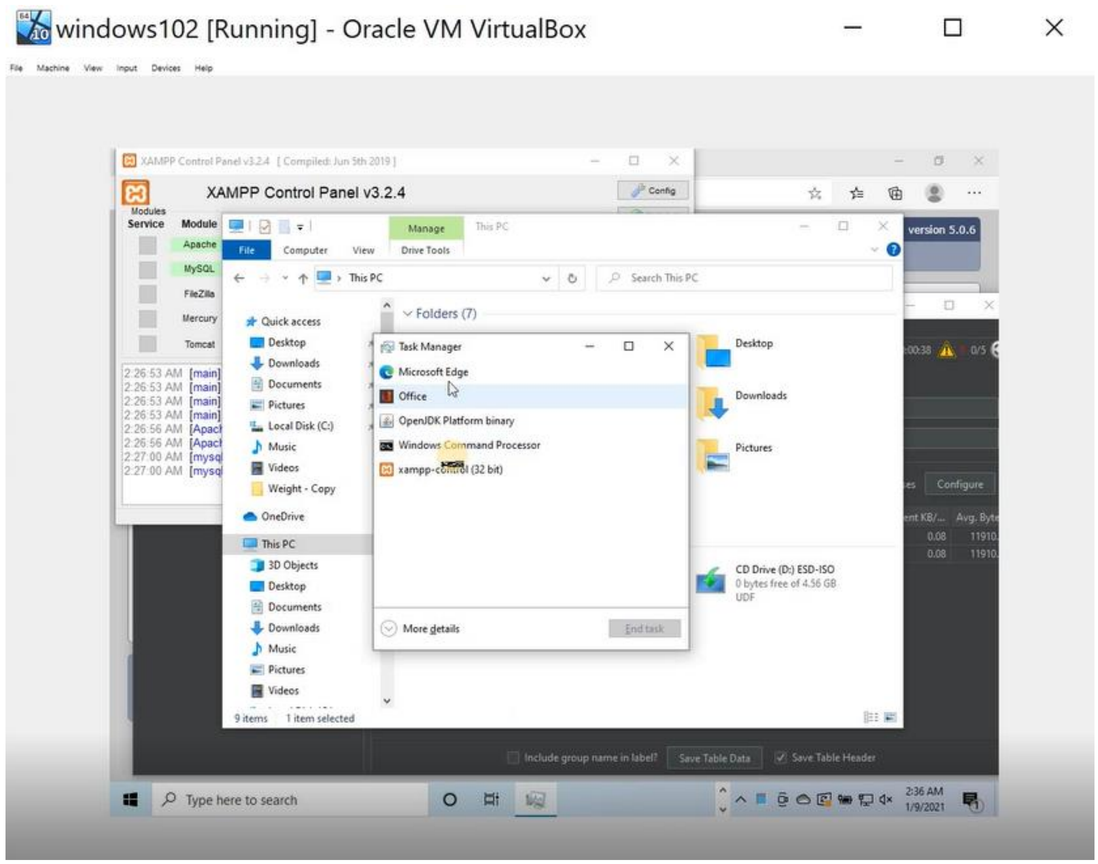

Figure B.1: Processes running in the test bed during the experiments.

- **Calculating the mean values to deliver actionable metrics:** Multiple readings are taken for each module, and their mean values are calculated.
- **Other components** such as operating system, databases, servers: Due to the use of the same computer, all the hardware components remain the same across all the test beds (Stackify, 2021).

A new and dedicated computer is used to configure all the dedicated Virtual Machines (VM). Each version of Bugzilla was installed on each of the VMs.

Host Machine Specifications

- HP Spectre x360 Convertible 15-bl1XX
- Processor: Intel(R) Core (TM) i7-8550U CPU @ 1.80GHz
- Installed RAM: 16.0 GB (15.9 GB usable)
- System type: 64-bit operating system, x64-based processor
- Edition: Windows 10 Home

A room temperature of 75º F (approximately) and humidity of 65% (approximately) are maintained.

Oracle Virtual Machine (VM) Specifications

- Microsoft VM ware (Windows 10)
- RAM: 4 GB
- HDD: 50 GB
- Operating System (OS): 64-bit & 32-bit
- Processor: Intel(R) Core (TM) i7-8550U CPU @ 1.80GHz
- Available space on HDD after installing all the required software and tools 25 GB.

To be noted here, while running tests, only one VM at a time that is under test is activated, and the rest are powered off. Figure B.2 shows the list of virtual machines used.

Figure B.2: Oracle virtual machine

Apache JMeter Version 5.4 is installed across all the virtual machines to measure response time. Figure B.3 shows a screenshot of Apache JMeter while running tests.

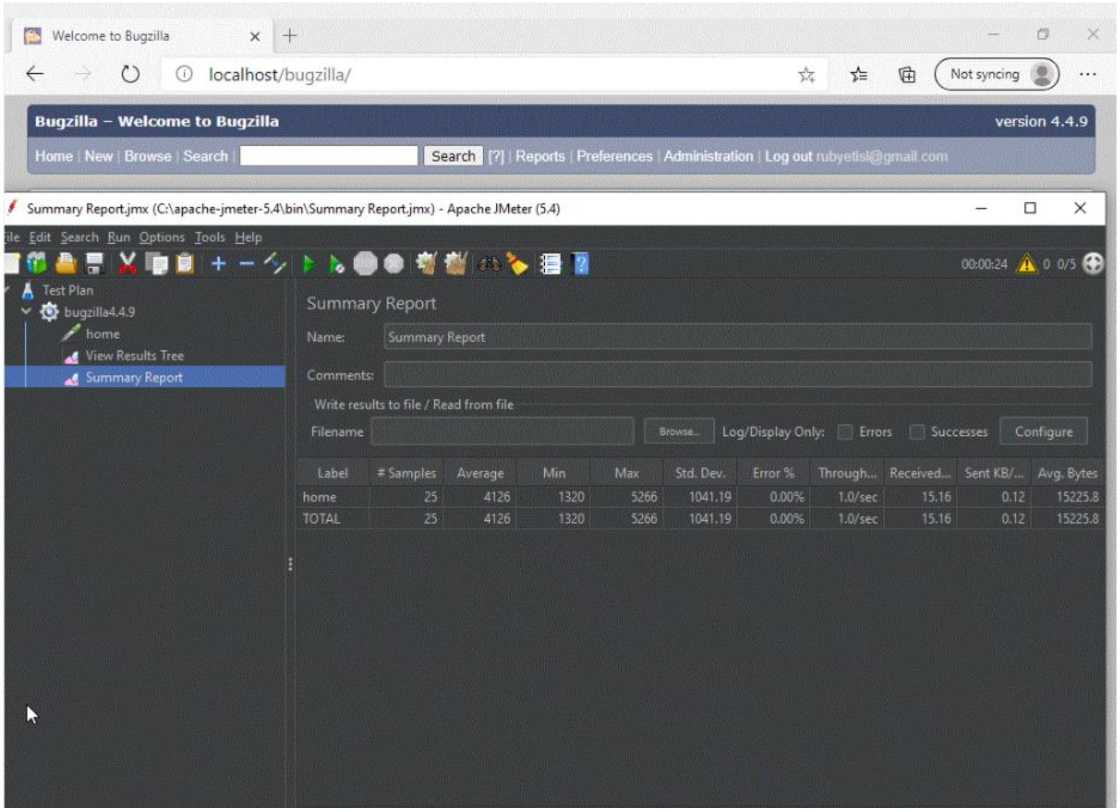

Figure B.3: Apache JMeter version 5.4

Adjustment of Clock Speed

The Virtual box has a setting (Figure B.4) called *Execution Cap* that can be used to adjust clock speed of the virtual machine. Location of *Execution Cap* is in: Setting>System>Processor> Execution Cap

Execution Cap can be adjusted by scrolling through a scale of 1% to 100%. As an example: A setting of 50% implies a single virtual CPU can use up to 50% of a single host CPU.

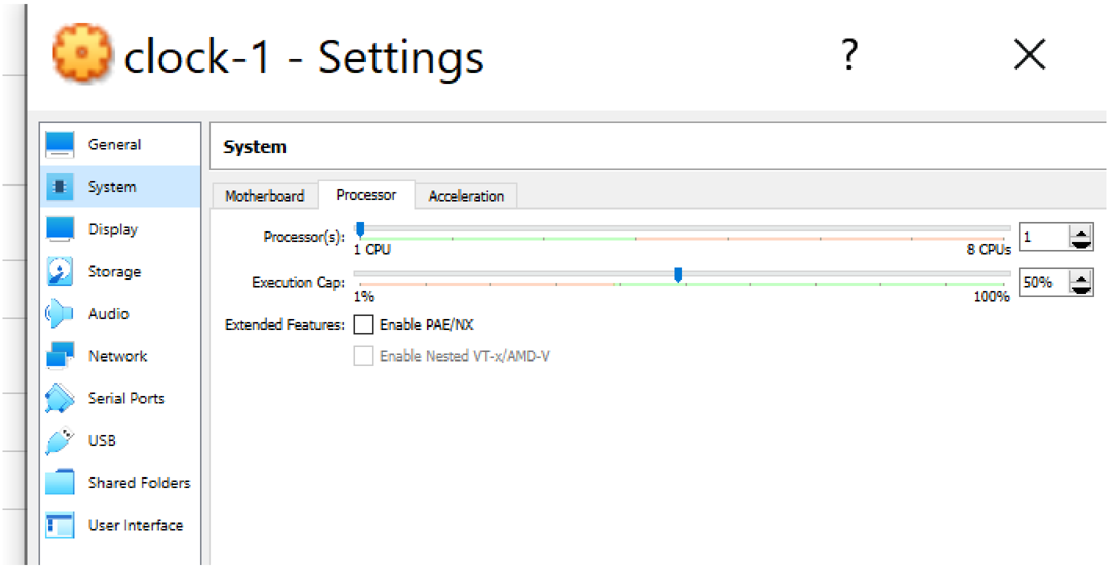

Figure B.4: Changing *Execution Cap* of the virtual machine to alter clock speed

References

Agarwal, V., Lybeck, N., Pham, B.T., Rusaw, R., & Bickford, R., (2015), Prognostic and health management of active assets in nuclear power plants, *International Journal of Prognostics and Health Management*, ISSN 2153-2648, 013, doi: 10.36001/ijphm.2015.v6i3.2270.

Albu, R.D., & Popentiu-Vladicescu, F., (2013), A comparative study for web services response time prediction, *The 9th International Scientific Conference eLearning and Software for Education*, April, Bucharest, doi: 10.12753/2066-026X-13-107.

Alonso, J., Belanche, L., & Avresky, D.R., (2011), Predicting software anomalies using machine learning techniques, *Proceedings of the 10th International Symposium on Network Computing and Applications*, August 25-27, 163-170, MA, USA, doi: 10.1109/NCA.2011.29.

An., D., 2018, Find out how you stack up to new industry benchmarks for mobile page speed, *Article published on Think with Google*, February, https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-page-speed-new-industry-benchmarks/

ANSI/IEEE, (1991), *Standard Glossary of Software Engineering Terminology*, STD-729-1991, ISBN 1-55937-067-X, New York, NY 10017, USA.

Antoniol, G., (2012), *Software Reliability*, A presentation available at https://moodle.polymtl.ca

Arora, I., Tetarwal, V., & Saha, A., (2015), Open issues in software defect prediction, *International Conference on Information and Communication Technologies*, 906-912, India, doi: 10.1016/j.procs.2015.02.161.

Arpaci-Dusseau, R., & Arpaci-Dusseau, A., (2014). *Operating Systems: Three Easy Pieces, Chapter: Hard Disk Drives*, Book, Arpaci-Dusseau, 16th February, ISBN. CreateSpace Independent Publishing Platform, 198508659X, 9781985086593.

Atamuradov, V., Medjaher, K., Dersin, P., Lamoureux, B., & Zerhouni, N., (2017), Prognostics and Health Management for Maintenance Practitioners- Review, Implementation and Tools Evaluation, *International Journal of Prognostics and Health Management*, ISSN 2153-2648, 2017 060, doi:10.36001/IJPHM. 2017.V8I3.2667.

Aumann, C.A., (2007), A methodology for developing simulation models of complex systems, *Ecological Modelling*, Volume 202, Issues 3-4, Page 385-396, DOI: 10.1016/j.ecolmodel.2006.11. 005.

Balnojan, S., (2020), Data Strategy: Good Data vs. Bad Data, Article published on towards data scienct, September, https://towardsdatascience.com/data-strategy-good-data-vs-bad-data-d40f85d7ba4e

Banks, J., & Merenich, J., (2007), Cost benefit analysis for asset health management technology, *53rd Annual Reliability and Maintainability Symposium*, 12 March, pp. 95–100, Orlando, FL, USA, DOI: 10.1109/RAMS.2007.328097.

Barajas, L.G., & Srinivasa, N., (2008), Real-time diagnostics, prognostics health management for large-scale manufacturing maintenance systems. *Proceedings of the ASME International Manufacturing Science and Engineering Conference*, MSEC2008, Evanston, IL, United States.

Barbieri, F., Hines, J.W., Sharp, M., & Venturini, M., (2015), Sensor-based degradation prediction and prognostics for remaining useful life estimation: validation on experimental data of electric motors, *International Journal of Prognostics and Health Management*, ISSN 2153-2648, 019, doi: 10.36001/ijphm. 2015.v6i3.2285.

Bellcore, (1990), *Reliability Prediction Procedure for Electronic Equipment*, Tech. Rep. TR-NWT-000332, 83-86.

Best, B., (2015), The Effect of Heat on Processors: A Challenge to Computer Usage in Northern Nigeria Rural Colleges, *Journal of Global Research in Computer Science*, 9th September, ISSN-2229-371X.

Bouwmeester, W., Twisk, JWR., Kappen, TH., Van Klei, WL., Moons, KGM., Vergouwe, Y., (2013), Prediction models for clustered data: comparison of a random intercept and standard regression model. *BMC Medical Research Methodology*; https://doi.org/10.1186/1471-2288-13-19

Brooksby, R., (2000), *Perforce Defect Tracking Integration Architecture*, August 10, Ravenbrook Limited. https://www.ravenbrook.com/project/p4dti/version/0.4/design/architecture/

Cohelo, E., & Basu, A., (2012), Effort estimation in agile software development using story points, *International Journal of Applied Information Systems (IJAIS)*, ISSN: 2249-0868, Vol.3, No.7, August, DOI:10.5120/ijais12-450574.

Dell'Aversana, P., (2018), Machine Learning for rock classification based on mineralogical and chemical composition. A tutorial. *Research Gate,* October, https://www.researchgate.net/publication/328020065

Diwaker, C., & Tomar, P., (2017), Identification of factors and techniques to design and develop component-based reliability model, *International Journal of Scientific Research in Computer Science and Engineering*, (E-ISSN: 2320-7639, Vol.5, Issue.3, pp.107-114), June.

Engel, S.J., Gilmartin, B.J., Bongort, K., & Hess, A., (2000), Prognostics, the real issues involved with predicting life remaining, *Proceedings for the IEEE Aerospace Conference*, March 25-25, Big Sky, MT, USA, DOI: 10.1109/AERO.2000.877920.

Everts, T., (2014), The Real Cost of Slow Time vs Downtime, *Performance & Capacity, Annual Conference*, Nov 5th, CMG, GA, United States.

Fortier, PJ., Michel, HE, (2003), *Computer Systems Performance Evaluation and Prediction*, Book, Elsevier Inc., June 25th, ISBN 978-1-55558-260-9, https://doi.org/10.1016/B978-1-55558-260-9.X5000-0

Halili, E.H, (2008), *Apache JMeter: A Practical Beginner's Guide to Automated Testing and Performance Measurement for Your Websites*, Packt Publishing, Birmingham, UK.

Hanmer, R.S., (2000), Software "reliability"? *Journal of Electronic Packaging*, 122(4): 357-360, doi: 10.1115/1. 1289630.

Happe, J., (2005), Performance prediction for embedded systems, *Seminar "Dependability Engineering",* Carl von Ossietzky University of Oldenburg, Oldenburg, Germany, 26th September.

Harris, R., (2018), *Tricentis Software Fail Watch Report* (5th annual), January 31.

Hess, A., Calvello, G., & Frith, P., (2005), Challenges, issues, and lessons learned chasing the "Big P". Real predictive prognostics. Part 1, *Proceedings of the IEEE Aerospace Conference*, March 5-12, Big Sky, MT, USA, doi: 10.1109/AERO.2005. 1559666.

Hillary, N., (2005), Measuring Performance for Real-Time Systems, *Freescale Semiconductor*, GRNTEEPFRMNCWP Rev. 0, November.

Hu, C., (2004), Toward Reliable Engineered System Design: Reliability Based design and Prognostic Health Management (PHM), *System Reliability & Safety Laboratory (SRSL), Symposium*, Iowa State University, Minnesota, USA.

Intel, What is clock Speed, *A newsletter published by Intel*, (Accessed October 9th, 2022) https://www.intel.com/content/www/us/en/gaming/resources/cpu-clock-speed.html#:~:text=The%20clock%20speed%20measures%20the,helps%20understand%20a%20CPU's%20speed

ISO/IEC, (1995), *Information Technology—Software Life Cycle Processes*, ISO/IEC 12207. International Standards Organization: Geneva, Switzerland.

Ivanescu, A., Li, P., George, B., Brown, A., Keith, S., Raju, D., & Allison, D., (2016), The importance of prediction model validation and assessment in obesity and nutrition research, *International Journal of Obesity*, 887-894, June, London, doi:10.1038/ ijo.2015.214.

Jaiswal, A., & Malhotra, R., (2018), Software reliability prediction using machine learning techniques, *International Journal of System Assurance Engineering and Management*, 9 230-244.

Kerr, L., & Goethel, D., (2014), Simulation modeling as a tool for synthesis of stock identification information, *Stock Identification Methods*, December, DOI:10.1016/B978-0-12-397003-9.00021-7.

Kothamasu, R., Huang, S., & VerDuin, W., (2006), System health monitoring and prognostics - a review of current paradigms and practices. *The International Journal of Advanced Manufacturing Technology.*, 28:1012–1024, July.

Krasner, H., (2018), The cost of poor-quality software in the US: A 2018 Report, *Consortium for IT Software Quality (CISQ), September 26, www.it-cisq.org*

Krasner, H., 2021, The Cost of Poor Software Quality in the US: A 2020 Report, *Consortium for Information & Software Quality, January 1st, www.it-cisq.org*

Lannino, A. & Musa., J., (1990), Software Reliability, *Advances in Computer*, *Academic Press Inc. by Elsevier,* Volume 30, Page 85-170, https://doi.org/10.1016/S0065-2458(08)60299-5

Lazer, D., Kennedy, R., King, G. and Vespignani, A., (2014), The Parable of Google Flu: Traps in Big Data Analysis, *Science*, 14 Mar, Vol 343, Issue 6176, pp. 1203-1205, DOI: 10.1126/science.1248506).

Lee, J., Ghaffari, M. & Elmeligy S., (2011), Self-maintenance and engineering immune systems: Towards smarter machines and manufacturing systems, *Annual Reviews in Control.* 2011;35(1):111–122, April.

Ly, C., Tom, K., Byington, C.S., Patrick, R., & Vachtsevanos G.J., (2009), Fault diagnosis and failure prognosis for engineering systems: A global perspective, *Proceedings of the IEEE International Conference on Automation Science and Engineering CASE* (108–115), August 22-25, doi: 10.1109/ COASE.2009.5234094.

Malaiya, Y.K., (2010), Software reliability management, *Encyclopedia of Library and Information Sciences*, 3rd Edition, February 17, 1: 1, 1-22, London, UK: Taylor & Francis.

Mancoridis, S., (2015), *Topics in software reliability*, CS 576 Study Notes for Computer Science, Drexel University.

McCarter, D., Shumaker, B., McConkey, B., & Hashemian, H., (2015), Nuclear power plant instrumentation and control cable prognostics using indenter modulus measurements, *International Journal of Prognostics and Health Management*, 6(3) ISSN 2153-2648, 2014 016.

Medjaher, K., & Zerhouni, N., (2013), Hybrid prognostic method applied to mechatronic systems, *The international Journal of Advanced Manufacturing Technology*, October, doi: 10.1007/s00170-013-5064-0.

Microprocessor Chronology, (2022), 21st October, In Wikipedia., https://en.wikipedia.org/ wiki/Microprocessor_chronology

Moallemi, C., & Saglam, M., (2013), The Cost of Latency in High-Frequency Trading, *Social Science Research Network (SSRN)*, February 5, https://ssrn.com/abstract=1571935

Modarres, M., Amiri, M., Jackson, C., (2017), *Probabilistic Physics of Failure Approach to Reliability: Modeling, Accelerated Testing, Prognosis and Reliability Assessment*, Wiley & Sons, NJ, USA, ISBN: 978-1-119-38863-0.

Sebastian, R., and Mojolicious contributors, *(2021), Mojolicious Conventions*, https://docs.mojolicious.org/#CONVENTIONS

Mordechai, B., (1990), *Principles of Concurrent and Distributed Programming*, ch.16, page 164, ISBN 0-13-711821-X, Prentice Hall, New Jersey.

Nadig, S., (2022), How To Effectively Prepare "Test Bed" And Minimize The Test Environment Defects (Part 1), *Newsletter published on Software Testing Help*, 24th September, https://www.softwaretestinghelp.com/test-bed-test-environment-management-best-practices/

Neilsen, J., (1993), *Usability Engineering*, Academic Press, Inc, MA, USA.

Olenski, S., 2016, Why Brands Are Fighting Over Milliseconds, *Forbes*, Nov 10th, https://www.forbes.com/).

Patrick, R., Smith, M.J., Zhang, B., Byington, C.S., Vachtsevanos, G.J., & Rosario, R.D., (2009), Diagnostic enhancements for air vehicle HUMS to increase prognostic system effectiveness, *Proceedings of the IEEE Aerospace Conference*, March 7-14, Big Sky, MT, USA, doi: 10.1109/AERO.2009.4839653.

Pfleeger, S.L., (1992), Measuring software reliability, *IEEE SPECTRUM*, Volume:29, August, DOI: 10.1109/6.144538.

Porru, S., Murgia, A., Demeyer, S., Marchesi, M., & Tonelli, R., (2016), Estimating story points from issue reports, *Proceedings of the 12th International Conference on Predictive Models and Data Analytics in Software Engineering*, Spain, September, doi: 10.1145/2972958. 2972959.

Qiao, G., & Weiss, B.A., (2016), Advancing Measurement Science to Assess Monitoring, Diagnostics, and Prognostics for Manufacturing Robotics, *International Journal of Prognostics and Health Management* 7 (Spec Iss on Smart Manufacturing PHM), December, DOI:10.36001/ijphm. 2016.v7i3.2410.

Ramakrishnan, A., & Pecht, M., (2003), A life consumption monitoring methodology for electronic systems, *IEEE Transactions on Components and Packaging Technologies* 26, 625–634, doi: 10.1109/TCAPT.2003.817654.

Redman, T. C., (2018), If Your Data Is Bad, Your Machine Learning Tools Are Useless, *Harvard Business Review*, April 02, https://hbr.org/2018/04/if-your-data-is-bad-your-machine-learning-tools-are-useless

Riaz, M., Sulayman, M., & Naqvi, H., (2009), Architectural decay during continuous software evolution and impact of 'design for change' on software architecture, *Proceedings of the International Conference on Advanced Software Engineering, and Its Applications,* 119-126. Doi: 10.1007/ 978-3-642-10619-4_15.

Rish, I., (2001), An empirical study of the naive Bayes classifier, *Research Gate*, January, page 41-46, IBM New York.

Roemer, M.J., Nwadiogbu, E, & Bloor G., (2001) Development of diagnostic and prognostic technologies for aerospace health management applications, *Proceedings of the IEEE Aerospace Conference*, 10-17th March, pp. 3139–3147, Big Sky, MT, USA, DOI: 10.1109/AERO.2001.931331.

Salzberg, S., (2014), Why Google Flu Is A Failure, *An article published on Forbes.com*, March 23, https://www.forbes.com/sites/stevensalzberg/2014/03/23/why-google-flu-is-a-failure/?sh=713e5a715535

Saxena A., Celaya J., Saha B., Saha S., & Goebel, K., (2010), Metrics for offline evaluation of prognostic performance. *International Journal of Prognostics and Health Management.*; 22nd March, DOI:10.36001/ijphm.2010.v1i1.1336.

Schumann, J., & Mengshoel, O.J., (2011), *ISWHM: Tools and Techniques for Software and System Health Management*, Intelligent System Division, USRA: Research Institute for Advanced Computer Science, September, SHM TIM, NASA.

Schwabacher, M., (2005), A Survey of Data Driven Prognostic, *American Institute of Aeronautics and Astronautics*, September, DOI:10.2514/6.2005-7002.

Serrano, N., & Ciordia, I., (2005), Bugzilla, ITracker, and other bug trackers, *IEEE Software* 22(2), doi: 10.1109/MS.2005.32.

Shafiabady, A., Mahrin, M.N., & Samadi, M., (2016), Investigation of software maintainability prediction models, *Proceedings of the 18th IEEE International Conference on Advanced Communications Technology*, January 31-February 3, Korea, doi: 10.1109/ICACT.2016.7423558.

Shukla, H.S., & Verma, D.K., (2015), A review on software defect prediction, *International Journal of Advanced Research in Computer Engineering & Technology*, 4(12), December.

Si, X., Wang, W., Hu, C., & Zhou, D., (2011), Remaining useful life estimation – A review on the statistical data driven approaches, *European Journal of Operational Research*, 16th August, PP 1-14, doi: 10.1016/j.ejor.2010.11.018.

Sims, I., (2021), *Factors that affect the performance of your computer*, Newsletter published on TECHVERA, 14th July, https://techvera.com/factors-that-affect-the-performance-of-your-computer/

Smathers, M., (2019), How Does Temperature Affect The Performance of Computer Components?, *Newsletter published on CHORN*, 11th January, https://smallbusiness.chron.com/temperature-affect-performance-computer components-28197.html

Stackify, (2021), *The Ultimate Guide to Performance Testing and Software Testing: Testing Types, Performance Testing Steps, Best Practices, and More*, 16th April, https://stackify.com/ultimate-guide-performance-testing-and-software-testing/#:~: text=Performance%20tests%20are%20best%20conducted,used%20for%20quality%20assurance%20testing.

Sun, B., Zeng, S., Kang, R., & Pecht, M. (2010), Benefits analysis of prognostics in systems, *Proceedings of the Prognostics and System Health Management Conference*, February, Macao, China, doi: 10.1109/PHM.2010.5413503.

SVALT, Throttling, *A Newsletter published by SVALT*, (Accessed October 3rd, 2022) https://svalt.com/blogs/svalt/76623745-laptop-heat-slows-speed.

Takata S, Kirnura F, Van Houten F, Westkamper E, Shpitalni M, & Ceglarek D, (2004), Maintenance: changing role in life cycle management, *CIRP Annals-Manufacturing Technology*, 53(2):643–655.

Teradata, (2018), *Teradata Tools and Utilities for IBM z/OS Installation Guide*, Oct 2018, Release 16.20, https://docs.teradata.com/reader/406hisxXWkO93Vmmzp13ig/rcQ2DFwIek988nW2c9KsGQ

Tian, Z., Lin, D., & Wu, B. (2012), Condition based maintenance optimization considering multiple objectives, *Journal of Intelligent Manufacturing*, 23(2) 333–340,April, DOI:10.1007/s10845-009-0358-7.

Tidriri, K., Chatti, N., Verron, S., & Tiplica, T., (2016), Bridging data-driven and model-based approaches for process fault diagnosis and health monitoring: A review of research and future challenges, *Annual Reviews in Control*, Volume 42, Pages 63-81, Angers, France DOI: 10.1016/j.arcontrol. 2016.09.008.

Ting, S.L., Ip, W.H., & Tsang, A.H.C., (2011), Is Naïve Bayes a good classifier for document classification? *International Journal of Software Engineering and Its Applications*, Vol. 5, No. 3, July, 5(3).

Trivedi, K. (2019), Software fault tolerance via environmental diversity, *Annual Meeting of the Reliability Engineering Association of Japan*, May 31, Hiroshima Japan.

Venkatasubramanian, V., (2005), Prognostic and Diagnostic Monitoring of Complex Systems for Product Lifecycle Management: Challenges and Opportunities, *Proceedings of the 14th European Symposium on Computer Aided Process Engineering*, 15th May, Volume 29, Issue 6, Pages 1253-1263, Lisbon, Portugal, doi.org/10.1016/j.compchemeng.2005.02. 026.

Vogl, G., Weiss, B., & Helu, M., (2019), A review of diagnostic and prognostic capabilities and best practices for manufacturing, *Journal of Intelligent Manufacturing*, June, DOI 10.1007/s10845-016-1228-8.

Wu, C., & Buyya, R., (2015), *Cloud Data Centers and Cost Modeling: A Complete Guide To Planning, Designing and Building a Cloud Data Center*, ISBN-13: 978-012801413-4, Morgan Kaufmann Publications, 220 Wyman Street, Waltham, MA 02451, USA.

Xu, P., Wang, Z., & Li, V., (2010), Prognostics and Health Management (PHM) System requirements and validation, *Prognostics and System Health Management Conference*, January 12-14, 10.1109/PHM.2010.5413560.

Yakimovich, D., (2001*), A Comprehensive reuse model for COTS software products*, PhD Dissertation, http://www.cs.umd.edu/projects/SoftEng/ESEG/papers/ Dissertations/Yakimovich%20dissertation.pdf

Yousef, A. (2014), A Tool Suite for Estimation and Prediction of Software Dynamic Defect Models, *International Conference on Computational & Experimental Engineering and Sciences*, Cairo, December, DOI: 10.1109/ICCES.2014. 7030975.

