An Intelligent Test Management System for Optimizing Decision Making During Software Testing

Albin Lönnfalt[a], Viktor Tu[a], Gregory Gay[a,b,*], Animesh Singh[c], Sahar Tahvili[c,d,*]

[a] *Chalmers University of Technology, Gothenburg, Sweden*
[b] *University of Gothenburg, Gothenburg, Sweden*
[c] *Ericsson AB, Stockholm, Sweden*
[d] *Mälardalens University, Västerås, Sweden*

Abstract

To ensure the proper testing of any software product, it is imperative to cover various functional and non-functional requirements at different testing levels (e.g., unit or integration testing). Ensuring appropriate testing requires making a series of decisions—e.g., assigning features to distinct Continuous Integration (CI) configurations or determining which test specifications to automate. Such decisions are generally made manually and require in-depth domain knowledge. This study introduces, implements, and evaluates ITMOS (Intelligent Test Management Optimization System), an intelligent test management system designed to optimize decision-making during the software testing process. ITMOS efficiently processes new requirements presented in natural language, segregating each requirement into appropriate CI configurations based on predefined quality criteria. Additionally, ITMOS has the capability to suggest a set of test specifications for test automation. The feasibility and potential applicability of the proposed solution were empirically evaluated in an industrial telecommunications project at Ericsson. In this context, ITMOS achieved accurate results for decision-making tasks, exceeding the requirements set by domain experts.

Keywords: Software Testing, Decision Support, Continuous Integration, Natural Language Processing, Machine Learning

*Corresponding author
Email addresses: greg@greggay.com (Gregory Gay), sahar.tahvili@ericsson.com (Sahar Tahvili)

1. Introduction

During software development, functional requirements are specified that define the behavior of the features of a product, as well as non-functional requirements that describe the quality-related properties of those features such as their performance or availability [1]. Quality assurance (QA) can be achieved by verifying that both the functional and non-functional requirements are met through software testing—where behavioral correctness is checked following the application of selected input to the system [2].

By thoroughly checking these requirements through testing, QA ensures that the software not only performs its intended functions but also operates effectively under expected conditions and constraints. Comprehensive testing helps to identify and fix defects early, resulting in a higher-quality product that meets user expectations and standards [3]. Such testing often is executed in Continuous Integration (CI), where automated pipelines build, package, and test applications, following defined configurations[1] [4, 5]. By automating the integration, verification, and delivery of code, CI enables software development teams to concentrate on fulfilling business requirements, all the while upholding standards of quality and security [4, 6, 7].

Ensuring that requirements are verified through testing requires *making various test management decisions*. For example, one may need to decide *whether a test should be automated or performed manually* or *which CI configuration* the tests related to a requirement should be executed under to ensure that the form of testing is appropriate to the requirement and that the requirement is verified in a correct, traceable, and efficient form.

Making effective decisions that balance quality and time-to-market is a challenging task in industries that require agile design, meeting coding standards, varied testing methods, and automated tools. In fact, several subject matter experts such as test managers, designers, developers, and integrators are generally involved in this process. Existing industrial practices typically require manual decision-making. However, employing a manual

[1] In this study, we define a *CI configuration* as a specific selection of execution environment (e.g., isolated or integrated microservices and systems), testing level (e.g., unit, integration, or system testing), build frequency, and other settings used during test execution in a CI pipeline.

demand-based decision-making process for testing different features of a software product is expensive and sensitive to product and customer changes. Previous studies highlight the risks of relying on manual test management, including inadequate coverage of requirements, a need for a deep understanding of the domain, higher costs and increased development time, lower levels of customer satisfaction, and potential reputational damage [1, 8, 9, 10, 11].

Due to the widespread availability of data in the software development process, the ability to monitor and automate test management decision-making through data analysis has become increasingly feasible. We propose that machine learning and natural language processing-based decision support can enhance the efficiency and effectiveness of software testing processes, ultimately contributing to product quality and reliability.

In this study, we propose an intelligent test management system called ITMOS that supports decision-making tasks. This framework is intended to be modular and expandable, to enable automation of an increasing number of test management decisions. In particular, as a starting point, we have implemented two forms of test management decision support:

(i) **Assigning the verification of requirements to different CI configurations**. Effective selection of CI configurations can improve efficiency and ensure that requirements are verified in the correct environment and with the correct form of testing [12]. However, incorrect configuration selection can cause failures or delay feedback [13, 14]. Therefore, it is important that the correct configuration be identified early in the test specification process.

(ii) **Recommending whether a particular test specification should be implemented as an automated or a manual test case, considering the requirements to be verified, the complexity of the feature-under-test, and the assigned CI configuration**. Because test automation requires an up-front investment, automation may not be more cost-effective than manual testing [15]. Further, automation may not always be possible or the best way to verify a requirement [16, 17]. Therefore, it is important to make the appropriate decision on how to implement a test specification.

The industrial case study conducted at Ericsson AB reveals that ITMOS effectively reduces the need for subject matter expert involvement in making

test management decisions. By automating and optimizing the test management process, ITMOS enhances both the correctness and efficiency of decision-making at Ericsson AB. This reduction in reliance on experts allows for more streamlined operations and potentially faster decision cycles, contributing to improved overall productivity. However, these findings are currently specific to Ericsson AB, and the generalizability of this benefit to other organizations remains to be validated through further studies in different contexts.

ITMOS differs from past research on both of the implemented decision-making tasks through its focus on efficient, correct, and traceable requirements verification. The majority of past approaches do not make predictions regarding specific requirements or test specifications, instead focusing on configuration tuning [14, 18, 19], the overall testing process [15], or high-level use cases [20]. ITMOS also incorporates factors other than cost—a focus of past research [14, 21, 18, 15, 20, 22]—into decision making.

We have implemented and evaluated ITMOS as part of an industrial case study at Ericsson AB, a telecommunications company in Sweden. ITMOS achieves an average F1-Score of 0.9163 for classifying CI configurations and 0.9463 for determining whether a test specification should be automated—both surpassing the requirements of surveyed domain experts. These experts further indicated potentially significant cost savings and reduction of mental fatigue that could result from the use of decision support tools such as ITMOS. These results offer an indication of the potential of this approach—and similar approaches—for increasing the effectiveness and efficiency of the test management process in development organizations.

The organization of this study is laid out as follows: Section 2 describes relevant background concepts and an overview of related research is described in Section 3. Our approach, ITMOS, is depicted in Section 4. We present an industrial case study, focusing on the accuracy of the framework, in Section 5. We present the results of a survey study on the correctness, speed, value, usability, interpretability, and stability of the framework in Section 6. Threats to validity are other implications of the case study are discussed in Section 7. Finally, Section 8 concludes the study and discusses future directions for this research.

2. Background

This section provides an initial overview of concepts important to this study, including requirements, testing, machine learning, and natural language processing.

2.1. Software Requirements

In software, requirements refer to the needs and constraints that the system-under-development must meet to be deemed ready to release [23]. Requirements can be specified over software, hardware, and the intersection between the two in the system-under-development, as well as on the development process itself [23]. In this study, we focus on requirements defined over software.

Requirements are typically classified into *functional* and *non-functional requirements* [23]. Functional requirements are meant to capture the intended behavior of the services, tasks, or functions of the software-under-development—collectively referred to as *features* [24]. Non-functional requirements, also known as quality requirements, refer to constraints on *how* a feature should deliver its capabilities [25]. Non-functional requirements are imposed over quality attributes including, e.g., reliability, performance, and usability [25].

2.2. Software Testing

Software testing is the act of dynamically executing a system-under-test to identify issues affecting its correctness or its ability to deliver services [2]. While many quality assurance techniques exist, testing remains the primary means of assessing whether the system-under-test meets its functional and non-functional requirements.

During testing, a *test suite*—a collection of one or more *test cases*—is executed on the system-under-test [2]. Tests are often first expressed as *test specifications*, natural-language descriptions of the scenarios to be executed. They are then concretely implemented as test cases. Test cases can be either implemented in an *automated* form—written as executable code—or in a *manual* form executed by human testers.

In this study, a test specification includes the following elements:

- Description: A high-level overview of the purpose of the test.
- Pass Criteria: Conditions that must be fulfilled for the test to pass.

- Test Procedure: Input actions, as well as actions taken to verify the correctness of the resulting software behavior.
- Pre and Post-Conditions: Conditions that must be true before the test procedure begins, or that must be true after the test procedure ends for the test to pass.
- Implementation Form: Whether the test will be implemented as an automated or manual test case.

Testing can be performed at multiple levels within a system, including *unit*, *integration*, and *system* testing [2]. Unit testing revolves around testing individual units—e.g., individual classes—of the system in relative isolation [26]. Integration testing focuses on testing the interactions between units and other components (e.g., services or subsystems) within the system-under-test [27]. During system testing, the full system-under-test is evaluated by applying input to a defined interface [2].

Generally, testing takes place in the order discussed. Integration testing is typically conducted under the assumption that all components already have been tested at the unit level, increasing the likelihood that discovered faults relate specifically to how the units interact [27]. Similarly, system testing is typically conducted under the assumption that integrations have already been tested [28].

2.3. Continuous Integration

Continuous Integration (CI) is a widely adopted development practice where multiple developers frequently integrate code changes into a shared codebase [4]. When code is committed, the updated codebase is built and tested as part of a "pipeline" or "workflow". A *CI pipeline* refers to an automated process that builds, tests, and deploys code after it is committed [5].

This practice has gained popularity due to its ability to ensure high code quality and promote efficient collaboration among team members [5]. By committing changes frequently, developers can ensure that everyone is working with the current code. The automated pipeline ensures that code can be compiled, tested, and deployed in a consistently repeatable manner, avoiding inconsistencies between developers and their development environments [4]. If a failure takes place at any stage of the pipeline, the codebase is reverted to the previous working state, and feedback is sent to the developer [29].

CI enables early detection of faults, reducing the overall cost of development [4, 6, 7]. CI also increases visibility into the progress of the development process, enabling more accurate estimation and planning [29]. Users can get access to new features more quickly, provide feedback, and be more integrated into the development process [29].

However, effective implementation and use of CI is not straightforward. Debbiche et al. identified challenges related to developer mindset and openness to changes in workflow, code review and integration tools, unstable or flaky test cases, integration of branching code, differences in interpretation and expectations between teams, code dependencies, and the need to break functionality into suitable increments for integration [6].

A major challenge is that, as the codebase grows in size and complexity, scalability issues begin to emerge in the CI pipeline [12]. Long build times can significantly impact the development workflow, reducing the frequency of commits and impeding the team's overall productivity. Developers often face frustrating delays in receiving feedback on their changes [12].

2.3.1. Continuous Integration Configuration

A *CI configuration* defines a set of specific settings and parameters for a CI pipeline. The CI configuration establishes build conditions, such as the operating system, disk size, compiler flags to utilize, required library dependencies, and other analogous properties [13]. In this study, we define a *CI configuration* as a specific selection of execution environment, testing level and focus, build frequency, and other settings used during test execution in a CI pipeline.

The testing level determines what type of test cases can be included in a test suite that is executed during a build (e.g., unit, integration, or system-level test cases). In addition, the test suite used in a particular configuration may have a particular focus, i.e., specific scenarios explored in that configuration (e.g., emulation of certain customer interactions). The build scheduling specifies how often builds are run and how commits may be bundled into one or more builds.

The execution environment, whether centralized or decentralized, determines whether microservices are tested independently or together, allowing for controlled testing of their interactions. Testing activities are also broken down in the configuration based on whether they occur before or after code changes are merged into the main branch of the version control system. Some forms of requirements (functional versus non-functional) may only be able

to be verified, and some testing levels may only be able to be targeted, in certain CI configurations. More details about the specific CI configurations used in this study can be found in Section 4.

Effective selection of CI configuration can counteract scalability issues, e.g., by executing a subset of a test suite appropriate for the changed code [12]. However, incorrect configuration selection can cause build failures or delay feedback [13, 14].

2.4. Supervised Machine Learning

Machine learning (ML) is a branch of artificial intelligence based on algorithmic inference of patterns from observed input data [30]. Specifically, in this study, we focus on *supervised learning*—a form of ML where a model is trained to make predictions for new input cases based on patterns identified in pre-labeled *training data* [31]. Supervised machine learning has been applied for many tasks in many fields, from hand gesture recognition to semiconductor fault detection to targeted advertising [30, 32]. Supervised learning has been applied to many aspects of the software testing process, e.g., to generate valid input or test oracles or to select a subset of test cases for execution [33].

2.5. Natural Language Processing (NLP)

NLP is a branch of artificial intelligence focused on algorithmic inference from natural language [34]. In this study, we utilize supervised learning to make predictions based on test descriptions written in natural language.

We perform this task using *word embeddings*—a NLP technique that represents words numerically, typically as a high-dimensional vector of real numbers [35]. The goal of word embedding is to capture the semantic meaning of words and the relationships between them [35]. Once a word embedding model is trained, it can be used to generate a numerical representation of any word in a vocabulary, which can then be used as input to supervised learning models. This is often more efficient and effective than using the raw text, as numerical embeddings capture meaningful semantic relationships between words that are difficult to infer directly [35, 36].

3. Related Work

Researchers have, to a limited extent, explored both automated CI configuration optimization and test procedure optimization—i.e., deciding whether

to automate a test. Below, we present relevant past work, and then outline the research gap that our work fills.

3.1. CI Configuration Optimization

Santolucito et al. performed static analysis to detect CI configuration errors at the code level [13]. They use a neural network to filter constraints with a low likelihood of being the root cause of CI configuration errors. Similarly, Vassallo et al. attempted to automatically identify smells in the configuration file using a rule-based approach [37]. Their focus is on error detection, not optimizing the selection of CI configurations.

Medvedev and Aksyonov proposed a multi-agent simulation intended to identify configuration options that optimize CI performance [14]. They explored the relationship between the number of service channels and the number of events per period. Simulation-based approaches are potentially expensive to execute, and the simulation must be customized for each configuration option modeled. However, it would be useful for cases where no past historical data exists to train a supervised learning model. If such data exists, a supervised learning model is faster and able to account for a wider range of configuration options.

Spieker et al. used reinforcement learning to select and prioritize test cases for execution in CI configurations to minimize the feedback time after commit [21]. Their approach makes predictions based on test duration, time since the last execution, and failure history.

Hwang et al. propose the use of supervised learning to improve the efficiency and deployment time of particular CI configurations [18]. Their approach identifies dependencies between development artifacts and then targets only changed artifacts and dependencies in a CI configuration. Our focus differs somewhat from the three approaches above, as we are interested not just in performance but also in reliability—ensuring that tests are executed under the correct configurations.

Bregman and Mattar have proposed a method of predicting which subset of execution platforms should be selected for a particular build within a CI configuration [19]. Their approach must be integrated into a live pipeline, where the current characteristics are inferred. After making this inference, a subset of the current pool of execution platforms is selected for the execution of a build as part of a CI configuration. This patent suggests that supervised learning could be used to make this selection. This approach is similar to

our own, but we focus on a larger set of configuration options and do not require active integration of the tool into the CI pipeline.

3.2. Test Procedure Optimization

Garousi and Pfahl use a process simulation model to propose the degree to which testing should be automated [15]. This model makes predictions based purely on the resources that would be consumed by automation, estimating based on the experience level of employees, training and communication overhead, and available person-hours. This model does not answer whether particular test specifications should be automated. Rather, it is used to estimate the percentage of testing efforts that should be automated over different phases of the testing process.

Amannejad et al. have proposed a search-based approach to predicting which testing activities—in this case, test design, scripting, execution, and evaluation—should be automated for a set of defined use cases [20]. The approach attempts to find a decision, automated or not, for each activity and use case that maximizes effort savings. Again, this approach is not based on specific test specifications but is at a higher use case level. One use case can correspond to many requirements and many test specifications. The decision is also based again purely on the cost of automation, while we base the decision on the content of the specification and on how similar test specifications were assigned in the past.

Flemström et al. also use a simulation model to prioritize a set of natural-language test specifications for automation, based on the estimated resources required to perform automation and the similarities between the test specifications—i.e., tests that differ substantially from others are prioritized, while tests that overlap are given a lower priority [22]. Our approach differs in that we do not offer a prioritization scheme, which assumes that all test cases should eventually be automated, but a clear recommendation on whether the test should be automated in the first place.

3.3. Summary

Our approach differs from past research in both areas in two primary ways. First, existing approaches lack traceability to the particular requirements to be verified, or even to particular test specifications intended to verify requirements. In the case of CI configuration optimization, there is a focus on tweaking individual parameters of configurations [14, 18, 19]. In the case of test procedure recommendations, most approaches make predictions at a

level higher than the specific test specification or requirement level, instead focusing on either the overall process [15] or high-level use cases [20]. We seek to provide decision support that supports efficient, correct, and traceable requirements verification—from requirement to test specification to code feature. This is a gap not filled in past research in this area.

Second, the past research is predominately focused on reduction of costs—either time to execute a CI configuration [14, 21, 18] or the effort to perform automation [15, 20, 22]—rather than the *correctness* of either a configuration or automation decision. Particularly in the case of automation, there is an implicit assumption that all test specifications could be automated. In reality, this may not always be the case [16, 17]. In the case of CI configuration assignment, improper decisions can cause costs that outweigh the potential cost savings of automation [13, 14]. Cost savings are important, but there is a research gap regarding making predictions using factors other than cost.

4. The Proposed Solution—ITMOS

This section outlines the structure of the proposed intelligent test management system, ITMOS. The initial implementation of ITMOS offers two forms of decision support, providing recommendations for the classification of requirements—either functional or non-functional—presented in natural text into the CI configuration where they should be verified and determining whether test specifications—based on those same requirements—should be implemented through automated or manual testing. Figure 1 provides a high-level overview of ITMOS.

As depicted in Figure 1, both functional and non-functional requirements will be captured, analyzed, and applied as input by the proposed intelligent test management system. Following the ISO (International Organization for Standardization) standards, such as ISO 25010 [38], is one way to fulfill the quality assurance standards for testing a software application [1]. ISO standards can guide the specification of non-functional requirements, and can be used by any organization, large or small, regardless of the organization's field of activity. Functional requirements will be provided by the users (e.g. operators) of the system. Both functional and non-functional requirements have an unstructured natural text format that might be changed based on the customer's needs and requests, considering different regions and quality standards.

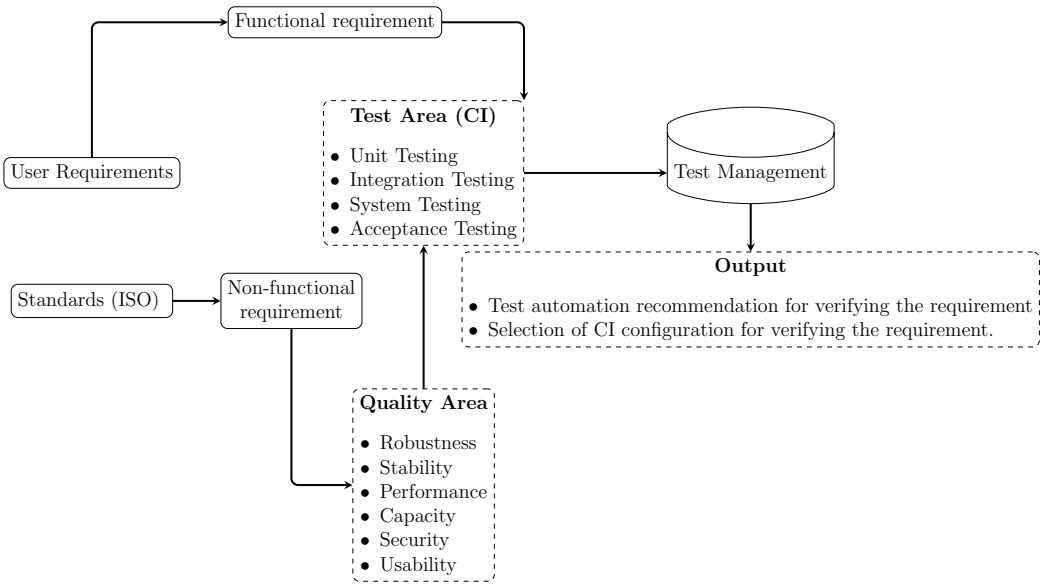

Figure 1: A holistic overview of the proposed intelligent test management system.

The proposed intelligent test management system consists of three modules, employing natural language processing and supervised machine learning. The first helps the user to define the test specification—based on the requirements—then the other two generate recommendations regarding the CI configuration for performing testing and whether the specification should be implemented through automated or manual testing. These recommendations are then propagated to the test management database. Figure 2 provides an overview of the three modules embedded in ITMOS, with details of each module provided in the following paragraphs.

Input: ITMOS captures both functional and non-functional test requirements as input (see Figure 1). Generally, the functional requirements are provided by the users (e.g. operators, customers), while the non-functional requirements need to be extracted from different standards, such as ISO standards. However, both functional and non-functional requirements have a textual format, usually written in unstructured natural text.

Module 1 (Defining Test Specification): In the first module, ITMOS prompts the user to define a test specification given the feature that will be tested. ITMOS guides the user to provide input such as test instructions and quality areas. A mix of structured natural language and predetermined

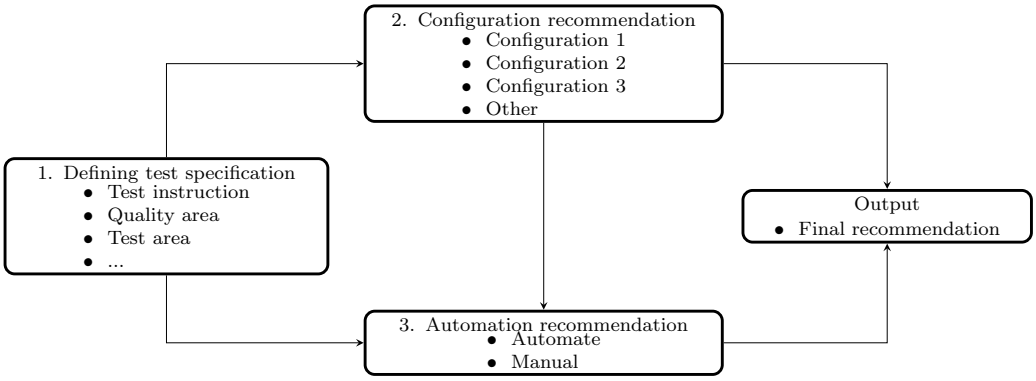

Figure 2: An overview of the embedded three AI-based modules in the proposed intelligent test management system.

categorical options is used to gather the data needed for ITMOS to make recommendations.

Module 2 (CI Configuration Recommendation): In the second module, the test specification is used to recommend a CI configuration for the specific test case.

As highlighted before, for testing any software application, different CI configurations are needed. Some empirical studies have demonstrated that CI [39] facilitates early defect detection [40], enhances developer productivity [41], and accelerates release cycles [42]. Therefore, CI has gained widespread adoption both in the industry [43] and in open-source projects [40].

In this study, we make predictions based on CI configurations in use at Ericsson, each consisting of different execution environments (testbeds and platforms for executing test cases), test suite levels and contents, build schedules, and testing activities performed. Each configuration encompasses a collection of characteristics specified in the ISO 25010 standard. Classifying the CI configuration for a test case, therefore, indirectly predicts which parts of the ISO 25010 standard will be covered. The three CI configurations utilized in this study, detailed in Table 1, were selected due to their significance and commonality in the industry. This selection does not imply that only these three configurations are important or that other configurations are less relevant. Instead, these examples serve to illustrate a broad spectrum of possible configurations. ITMOS is designed to be flexible and is not restricted to these specific configurations or to any fixed number of configurations. It is capable of adapting to and supporting a wide range of

CI configurations beyond those highlighted in this study.

CI Configuration	ISO Characteristics
Configuration 1	**Execution Environment**: Decentralized environment where microservices execute in isolation. **Test Levels and Suite Focus**: Due to the decentralized environment, suited for covering the lowest test levels, e.g., unit testing. Focus on functional requirements and validation of legacy functionalities. **Build Scheduling**: After every commit. **ISO Characteristics**: robustness, stability, traffic, performance, capacity, upgrade.
Configuration 2	**Execution Environment**: Centralized environment where microservices are tested simultaneously and interact. **Test Levels and Suite Focus**: Fulfillment of functional requirements related to basic and life cycle management functionality. Unit, integration, and system tests. **Build Scheduling**: Can be executed either on a single commit or overnight, where all commits since the previous build are bundled. The testing process for a single commit should be short (under one hour) to ensure quick feedback. **ISO Characteristics**: upgrade system operation, robustness, traffic functionality, security.
Configuration 3	**Execution Environment**: Centralized environment. **Test Levels and Suite Focus**: Complex scenarios that verify at overall system level. Includes emulating customer-like environments and interactions. **Build Scheduling**: Executed weekly. **ISO Characteristics**:stability, capacity, accessibility, mobility, integrity, resilience.
Other	The organization uses additional configurations in some situations. These are labeled as **Other** in the dataset.

Table 1: Overview of CI configurations used in this study.

The classification is accomplished by using different supervised learning approaches. The module consists of a rule-based component and three sub-models that are integrated into an ensemble architecture. The rule-based component analyzes the length of the test instruction and overrides the sub-models for given situations. The three sub-models can be categorized into natural language-based and categorical-based models.

1. Analyzes the semantics of test instructions to classify the configuration.
2. Utilizes the distribution of words in the test instruction to classify.
3. Uses categorical data from the test specification to classify.

Table 2 provides an overview of a selected test specification, utilized in this study, that verifies a functional requirement. Each requirement is verified by a test specification, which serves as input to ITMOS.

ID	Description	Pass Criteria	Test Procedure	Pre/Post-condition	Test Implementation
TC_1	The Purpose of TC_1 is to verify the A4 measurement timer interaction with PDU session addition.	No new alarms, errors, or crashes detected.	Action: Check that QFI=4 is MN terminated, if not define it as MN terminated Verify: Verify that QFI is set according to the above parameters get NRDC Termination Action: Attach Ue with only one PDU session (MN terminated, QFI=4) Verify: Verify that Ue is attached, and presented only on MgNB (e.g. NRDC not set) Action: Add second SN terminated PDU session Verify: Verify that event A4 is reported in RRCReestablishmentRequest message	The NRDC feature is active, and all associated MO objects are configured. Configure the simulator to enable support for multiple PDU sessions.	Manual Testing

Table 2: Example of a test specification that verifies a functional requirement.

Module 3 (Test Automation Recommendation): This module uses an ensemble model that utilizes two sub-models to generate automation recommendations. The first sub-model uses the semantic representation of the description of the test specification. The second sub-model uses categorical data from the test specification and the recommended CI configuration provided by Module 2 to determine whether a test case should be automated. For this task, supervised learning is used.

Output: The recommendations provided by ITMOS serve as an intelligent test management system tool where it assigns test cases to CI configurations and decides whether a test case should be automated. This decision support is integrated into the operational process, enabling decision-makers to improve the efficiency of the task and minimize the number of misclassifications.

By leveraging the recommendations generated by ITMOS, decision-makers can better allocate their time and resources, resulting in a more streamlined and effective testing process. This, in turn, leads to a faster and more reliable software development lifecycle.

4.1. Implementation

The proposed solution in Figure 1 can be implemented in various ways. In this study, to dynamically capture end-user inputs, a graphical user interface (GUI) was developed as a web portal using JavaScript, HTML, Python, and CSS. Incorporating a GUI into the proposed solution in this study facilitates prompting the end user for both required and optional data during the input of test specifications.

Module 2 (CI Configuration Recommendation): This module adopts an ensemble-based architecture to generate recommendations for the CI configuration. This architecture, depicted in Figure 3, integrates a rule-based

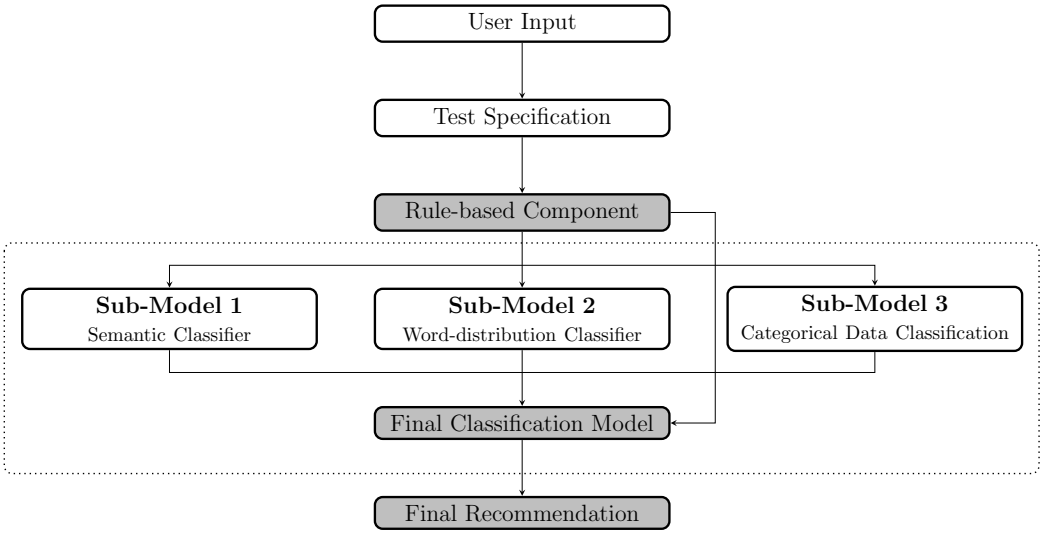

Figure 3: An overview of Module 2 (CI Configuration Recommendation).

component alongside three sub-models, each specializing in handling a distinct aspect of the input. The input to the ensemble model comprises test specifications derived from end-user input, as previously described. These test specifications encompass both natural language test instructions and categorical data.

The rule-based component of the ensemble assesses the length of the test instructions within the test specification. Sub-Model 1 focuses on extracting the semantic meaning embedded in the test instructions, Sub-Model 2 analyzes the statistical distribution of words within them, then Sub-Model 3 leverages the categorical data provided in the test specification. The outputs from each sub-model are then fed into the final classification model within the ensemble, which generates the recommendation.

- *Rule-based Component:* This component assesses the character count of each subpart within the test instruction provided in the test specifications. This component categorizes test specifications for various CI configurations based on pre-defined conditions. For instance, if the character count exceeds 1000 in the pre/post-condition section or 800 in the pass criteria section of the test instruction, the test specification is directed to a specific CI configuration. This rule was derived from the manual analysis of the dataset. The thresholds for these rules may vary when implementing the solution in different domains.

- *Sub-Model 1 (Semantic Classifier):* The semantic classifier receives input comprising four sub-parts of the test instruction: test setup, pre/post-conditions, test procedure, and pass criteria, all presented in a natural language format. To capture both semantic and syntactic information embedded in these texts, the sub-parts are transformed into word embeddings utilizing pre-trained FastText embeddings [44], which yield 300-dimensional vectors.

  These embeddings are then individually inputted into Random Forest classifiers [45], with one classifier dedicated to each subpart of the test instruction. Each specific Random Forest classifier employs the Gini impurity as a cost function to facilitate predictions for its corresponding subpart. Consequently, the semantic model generates four three-element vectors, one for each subpart, wherein each element reflects the confidence level that a given test specification pertains to a particular CI configuration.

- *Sub-Model 2 (Word-distribution Classifier):* The word-distribution classifier also takes input from the four sub-parts of the test instruction. This model examines the distribution of words within each subpart of the test instruction. The model computes the conditional probability of a bag of words given a specific CI configuration to classify test specifications. It achieves this by multiplying the conditional probabilities of each word in the bag given the CI configuration, which was precomputed and stored during training. While this approach assumes word independence, which may not always hold true, its impact on the model's accuracy is typically negligible.

  The conditional probability of the word bag given a CI configuration is computed for all three configurations. Subsequently, the conditional probability of a CI configuration given a word bag is determined by dividing the conditional probability of the word bag given a particular configuration by the sum of the conditional probability of the word bag given each configuration, multiplied by the probability of that CI configuration.

  This summation encompasses all possible CI configurations. When given a test instruction as input, the word distribution model generates four three-element vectors as output, corresponding to each subpart of the test instruction. Each element signifies the confidence level that a

test specification aligns with a specific CI configuration. If an unseen word arises during prediction, the model assigns minimal probabilities to all associated words.

- *Sub-Model 3 (Categorical Data Classification):* This sub-model utilizes the categorical information supplied by the end user in the test specifications to generate recommendations. This implementation employs a one-hot encoder [46] to encode the categorical data. The encoder generates binary columns for each potential value of the categorical data present in the test specification. Subsequently, the resulting one-hot encoded categorical data is inputted into a random forest model, which utilizes the Gini impurity as the cost function for recommendation generation. The output of the categorical data sub-model is a three-element array, with each element denoting the probability that the test specification aligns with each CI configuration.

- *Final Classification:* The final model in the ensemble accepts a 27-element vector as input, which is formed by concatenating the outputs from the sub-models. This concatenated vector is subsequently supplied to a machine-learning classifier. In this implementation, a random forest with Gini impurity serves as the cost function. The output of the final model is a three-element vector, indicating the model's confidence in assigning the test specification to each CI configuration.

Module 3 (Test Automation Recommendation): Module 3 uses an ensemble-based architecture, which is depicted in Figure 4. The ensemble is comprised of two sub-models that specialize in handling different aspects of the input. The input to the ensemble model is in the form of test specifications created based on the information provided by the end user. Sub-Model 4 captures the semantic meaning of the test specification description, while Sub-Model 5 utilizes the categorical data in the test specification. The final recommendation is obtained by using the output from each sub-model as input to the final classification model in the ensemble.

- *Sub-model 4 (Semantic Classifier):* The semantic classifier takes in a test specification described in natural language. To capture the underlying meaning of the text, the description is first converted into word embeddings using pre-trained FastText embeddings. These embeddings are then fed into a random forest that utilizes the Gini impurity as the

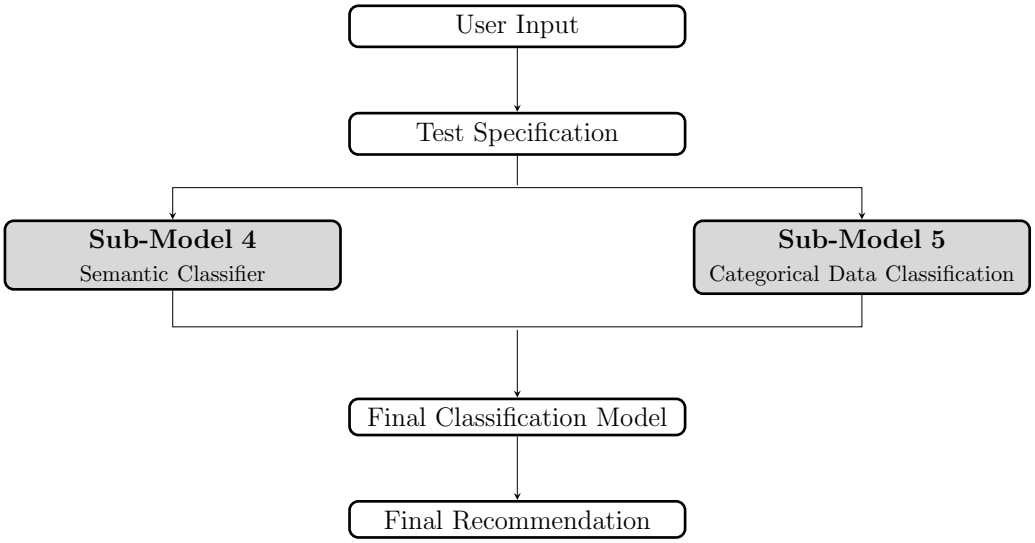

Figure 4: An overview of Module 3 (Test Automation Recommendation).

cost function to make predictions about whether the test specification should be automated or not. The output from the semantic classifier for automation is a list with two elements, indicating the level of confidence for automating or not automating the specific test specification.

- *Sub-Model 5 (Categorical Data Classification):* This sub-model utilizes categorical data from the test specifications, provided by the end user via the GUI, along with the predicted CI configuration, to generate recommendations for automation. The data undergoes encoding using a one-hot encoder. Subsequently, the encoded data is inputted into a random forest classifier, which employs the Gini impurity as the cost function. The output from the random forest classifier is a two-element array, where each element signifies the confidence level regarding whether a test case should be automated.

- *Final Classification:* The final model accepts a 6-element vector as input, which is generated by concatenating the outputs from sub-models 4 and 5. This concatenated vector is then supplied to a support vector machine [47]. In this implementation, a support vector machine with C-SVM serves as the cost function. The output of the final model is a two-element vector, representing the model's confidence in automating the test specification or not.

4.2. Experimental Setup

Overview of Training Data: Table 3 presents an overview of the dataset for conducting this case study. The dataset comprises 31 variables (input features), with four of them "Pass Criteria", "Test Procedure", "Pre/Post-condition", and "Test Setup" represented as unstructured natural text. The remaining 27 input features consist of categorical textual data. Additionally, the dataset includes 2 target variables: CI Configuration and Automation.

	Pass Criteria	Test Procedure	Pre/Post-condition	Test Setup	CI Configuration
Count	1838	1838	1838	1838	1838
Unique	641	977	534	508	4
Frequency	693	592	771	624	1180

Table 3: Overview of the dataset used in the case study.

We utilized the *describe function* to conduct a descriptive analysis, enabling a comprehensive understanding of the dataset and obtaining essential statistical information. The *pandas.DataFrame.describe*() method is a powerful tool for summarizing the data, giving a clear and concise statistical overview. It generates descriptive statistics of the DataFrame's data, providing a quick overview of the dataset's distribution and central tendency. The following three key pieces of information were extracted:

- Count: the total number of observations in the feature list,
- Unique: the total number of unique values in the feature list,
- Frequency: the frequency of the most common value in the feature list.

	CHS Verification	Network Automation SW	SUT	Testbed	Automation
Count	1838	1838	1838	1838	1838
Unique	2	2	2	4	2
Frequency	1662	1784	1624	1674	1251

Table 4: Overview of the categorical dataset used in the case study.

Table 4 represents some of the utilized categorical datasets in the case study. In this table, "CHS" stands for Characteristic Specification, which refers to a set of requirements that define the desired characteristics or attributes of a product. "SUT" refers to the System Under Test, and "Testbed" denotes the test environment and infrastructure.

Training of Sub-Model 2: During the training of ITMOS, statistics about the training data were collected. This sub-model estimates the conditional probability of a specific word occurring given a CI configuration. It also calculates the distribution of CI configurations in the training data. Both the conditional probability of words and the distribution of CI configurations are calculated and stored separately for each of the four sub-parts of the test specification. These distributions were calculated from the training data using the following equations:

- The proportion of each CI configuration relative to the total number of configurations:

$$P(c) = \frac{N(c)}{\sum_{c' \in C} N(c')} \tag{1}$$

  where:

  - $N(c)$ is the number of occurrences of CI configuration c in the dataset.
  - $\sum_{c' \in C} N(c')$ is the total number of CI configurations in the dataset.
  - C is the set of all possible CI configurations in the dataset.

- Estimating the probability of a word in a subpart of a test instruction given a CI configuration can be calculated as:

$$P(w \mid c, S_i) = \frac{N_{S_i}(w, c)}{\sum_{w' \in V_{S_i}} N_{S_i}(w', c)} \tag{2}$$

  where:

  - $P(w \mid c, S_i)$ is the probability of the word w occurring in subpart S_i given CI configuration c.
  - $N_{S_i}(w, c)$ is the number of occurrences of the word w in documents with the CI configuration c in subpart S_i.
  - $\sum_{w' \in V_{S_i}} N_{S_i}(w', c)$ is the total count of all words in documents with the CI configuration c in subpart S_i.
  - V_{S_i} is the vocabulary (the set of unique words) in subpart S_i.

As mentioned before, if a word that has not been seen during the training phase occurs during prediction, the model will assign all probabilities for that word to a minimal number. To estimate the probabilities, the following equations are used:

- To calculate the probability of a combination of words (Word Bag) in the subpart of a test instruction given a CI configuration:

$$P(\text{Word Bag} \mid c, S_i) = \prod_{w \in \text{Word Bag}} P(w \mid c, S_i) \tag{3}$$

  where:

  - $P(\text{Word Bag} \mid c, S_i)$ is the probability of the Word Bag occurring in subpart S_i given CI configuration c.
  - $P(w \mid c, S_i)$ is the probability of the word w occurring in subpart S_i given CI configuration c.
  - If w has not been seen during training, $P(w \mid c, S_i)$ is assigned a minimal probability, denoted as ϵ.

  The probability $P(w \mid c, S_i)$ can be calculated as:

$$P(w \mid c, S_i) = \begin{cases} \frac{N_{S_i}(w,c)}{\sum_{w' \in V_{S_i}} N_{S_i}(w',c)} & \text{if } w \text{ is seen in training} \\ \epsilon & \text{if } w \text{ is not seen in training} \end{cases}$$

  where:

  - $N_{S_i}(w, c)$ is the number of occurrences of the word w in documents with the CI configuration c in subpart S_i.
  - $\sum_{w' \in V_{S_i}} N_{S_i}(w', c)$ is the total count of all words in documents with the CI configuration c in subpart S_i.
  - V_{S_i} is the vocabulary (the set of unique words) in subpart S_i.
  - ϵ is a small probability assigned to words not seen during training.

- Calculating the probability of the test instruction being assigned to a particular CI configuration ($CI\ config_1$) based on the subpart of the test instruction can be done using Bayes' Theorem:

$$P(CI\ config_1 \mid \text{Word Bag}, S_i) = \frac{P(\text{Word Bag} \mid CI\ config_1, S_i) \cdot P(CI\ config_1 \mid S_i)}{P(\text{Word Bag} \mid S_i)} \tag{4}$$

where:

- $P(CI\ config_1 \mid \text{Word Bag}, S_i)$ is the probability of the test instruction being assigned to $CI\ config_1$ given the Word Bag in subpart S_i.
- $P(\text{Word Bag} \mid CI\ config_1, S_i)$ is the probability of the Word Bag occurring in subpart S_i given $CI\ config_1$.
- $P(CI\ config_1 \mid S_i)$ is the prior probability of $CI\ config_1$ in subpart S_i.
- $P(\text{Word Bag} \mid S_i)$ is the total probability of the Word Bag occurring in subpart S_i.

4.3. Parameters

For our experimental study, we selected various parameters for each sub-model recommended by the state-of-the-art. Table 5 presents the parameters utilized in this study.

		N_estimators	**Max_depth**	**N_booststrap**	**Other**
Sub-model 1	Test setup	500	35	2	
	Pre/post condition	250	20	40	
	Test procedure	500	45	5	
	Pass criteria	250	15	2	
Sub-model 2	No explicit parameters				
Sub-model 3	Random forest classifier	250	45		
Final Classification Model	Random forest classifier	50	15		
	Support vector classifier				Defaults
	Logistic regression				Defaults
Sub-model 4	Random forest classifier				Defaults
Sub-model 5	Random forest classifier	250	45		
Final Classification Model	Support forest classifier				kernel: linear
	Logistic Regression				Defaults
	Random forest classifier				Defaults

Table 5: Utilized parameters recommended by state-of-the-art methods in this study.

As previously discussed, conventional classification methods often exhibit bias toward the most prevalent class [48]. This bias stems from the optimization of global metrics, such as error or accuracy, which do not consider the distribution of instances across classes. Consequently, precision is notably achieved in the predominantly represented class, while instances belonging to the less represented class tend to be inaccurately classified. In all methods employed, we adhere to the parameters recommended by the respective authors in this study.

Considering confidentiality, intellectual property protection, and data privacy and security, we are unable to provide the concrete implementation of ITMOS or the actual data used as part of training or evaluating the framework without disclosing proprietary information or specific datasets. However, we have made a synthetic dataset available, containing data in the same format, which can be found on Figshare (see: [49]). Together with the method, algorithms, and design principles in Section 4, we believe that a similar framework to ITMOS could be implemented at a different company or by researchers.

5. Experimental Study

We have implemented and evaluated the proposed framework, ITMOS, as part of an industrial case study at Ericsson AB in Sweden. Ericsson is an international company that develops products in the telecommunications industry. To evaluate the performance, in the field, of ITMOS, we have conducted a performance study. In particular, we have addressed the following research question:

- **RQ1:** What is the effectiveness achieved by ITMOS when identifying CI configurations to verify requirements under and identifying whether to automate a test specification for a requirement?

While the study draws upon Ericsson data for both model training and evaluation purposes, its findings are anticipated to extend beyond the telecommunications industry. In the context of Ericsson AB, assigning new features to different CI configurations involves detailed and systematic processes. These include determining the appropriate test levels, selecting relevant test suites, and configuring the CI infrastructure to ensure comprehensive verification and validation of new features. Such processes are critical in maintaining the high standards required for telecommunications software. However, it is necessary to recognize that not all industries may have the same level of access to software development data or the same operational workflows. For example, in the Safety-critical system, surveillance, or booking systems might face different challenges and limitations. The availability of software development data, the complexity of integrating new features, and the specific requirements for testing and deployment can vary significantly.

The guidelines outlined by Runesson and Höst [50] have been adhered to in formulating this case study.

5.1. Unit of Analysis

The units of analysis in the case under study are test specifications used in different CI configurations to test cloud radio access network (C-RAN) products at Ericsson. C-RAN is a cloud-native software solution handling compute functionality in the RAN. C-RAN is intended to increase the versatility of network buildouts to address a variety of 5G use cases. C-RAN provides communications service providers (CSPs) with enhanced flexibility, accelerated service delivery, and improved scalability in network operations.

Compared to traditional RAN products like the Baseband Unit (BBU), C-RAN presents complex verification challenges due to its virtualized architecture, distributed nature, scalability requirements, interoperability challenges, and security considerations. Successfully deploying and managing C-RAN networks demands expertise in network virtualization, software-defined networking, radio access technologies, and network security [51]. Consequently, the testing process for C-RAN is more intricate than for other products. This complexity arises from the need for comprehensive validation of virtualized functions, intricate coordination between distributed components, and rigorous testing of interoperability with legacy systems. The dynamic nature of virtualized resources, complex communication protocols between distributed units, and the necessity to ensure integration with existing network infrastructure further contribute to the testing challenges of C-RAN.

One primary component of the C-RAN architecture, namely the virtualized distributed unit (vDU), has been chosen as the focus of this study. The term "virtualized" refers to the use of virtualized network functions (software) running on top of general-purpose computer hardware [51]. A total of 1838 test specifications have been extracted from the internal test management database at Ericsson. The opinions of test managers regarding the appropriate CI configurations by ITMOS have been collected and analyzed through a survey study, as presented in Section 6.

The CI configurations in this case study, outlined in Table 1, are context-specific and considered the most critical pipelines. These configurations are tailored to the specific needs of the products, considering factors such as required test levels, test suite focuses, the system under test, and the CI infrastructure. The decision to prioritize these pipelines stems from their importance in ensuring successful software delivery. However, ITMOS is not restricted to the configurations or test specification formats used in this study and can be applied to other organizations and product domains.

5.2. Experimental Evaluation

We have separately analyzed the accuracy of the recommendations for CI configuration and test automation. First, we assessed the effectiveness of our proposed solution in classifying CI configurations using a dataset containing 1838 test specifications (Table 3). Then, we assessed the effectiveness of our solution for classifying whether a test specification should be automated or not. For both analyses, we employed an 80/20 train-test split and repeated the evaluation process 100 times with varying train-test splits to ensure robustness. Across these evaluations, we calculated the precision, recall, accuracy, and F1-score—the harmonic mean of the precision and recall—following the standard definitions of each in supervised learning. Additionally, to address the issue of data quality assurance, we have implemented rigorous data preprocessing steps to ensure the quality and reliability of the training data. We recognize that missing or inaccurate data during the training process can lead to an inaccurate or generalized model. The various data preprocessing techniques we have employed include:

1. Correcting errors: implementing case sensitivity measures to rectify inconsistencies.

2. Dropping duplicate data: removing redundant records to maintain data integrity.

3. Handling missing data: dropping empty data fields to ensure completeness.

4. Encoding categorical variables: converting categorical variables into a suitable numerical format.

These steps are part of our comprehensive strategy to mitigate the risks associated with faulty or inaccurate data and to enhance the robustness and reliability of our model.

5.3. Experimental Results

Table 6 provides a summary of the effectiveness evaluation of ITMOS in this study for the CI configuration classification. Table 7 summarizes the effectiveness evaluation for the test automation classification. In both cases, ITMOS was highly effective in performing the classification tasks.

To address the class imbalance issue, the average='macro' approach is employed when calculating the F1 score. This method computes the F1 score

Metric	Value
Precision	0.9079
Recall	0.9093
Accuracy	0.9233
F1-score	0.9163

Table 6: Average effectiveness for CI configuration classification.

Metric	Value
Precision	0.9509
Recall	0.9424
Accuracy	0.9532
F1-score	0.9463

Table 7: Average effectiveness for test automation classification.

independently for each class and then averages the scores, ensuring that each class contributes equally to the final metric. This approach is particularly useful for imbalanced datasets, as it treats each class with equal importance, irrespective of the number of instances (support) belonging to that class. By doing so, we ensure that the performance metric is not biased towards the majority class and provides a meaningful evaluation across all classes. The use of average='macro' for the evaluation metric is one such countermeasure.

These results surpass the performance requirements established by surveyed domain experts at Ericsson (see Section 6). Nevertheless, it is essential to acknowledge that a more robust ML model could have been attained with better data quality and increased quantity.

An analysis of the ensemble model has indicated that certain parts of the test specifications play a crucial role in determining the assignment outcome. Specifically, features such as "Tagged Microservice", "Test Framework", "Traffic Model", "Test Configuration", "SW Track", "Delegated Test Cases", and "System Under Test" appear to be highly influential. However, it is important to note that drawing definitive conclusions is challenging due to incomplete data, and the specific features that are correlated with particular classification outcomes will likely depend on the product domain, the organization providing the training data, and even the specific contents of the dataset.

6. Survey Study

The results in Section 5 offer an indication, out of context, of the performance of ITMOS for automating certain test management decisions. We are additionally interested in examining the suitability, more broadly, of incorporating tools such as ITMOS into real-world test management. In particular, we have addressed the following research question:

- **RQ2:** What is the applicability of ITMOS in practice?

A survey study was conducted to answer this research question. The survey concentrated on ITMOS, and specifically on the task of assigning CI configurations to the verification of a requirement. Nevertheless, many of the questions can be generalized to other use cases of ML models to automate test management decision-making.

6.1. Survey Setup

The survey was conducted online, including both closed and open-ended questions. The choice of the online survey was motivated by its low cost and time efficiency, compared to face-to-face interviews [52]. The population targeted for the survey are people with domain knowledge of software testing in the context of Ericsson. A mixture of purposive and convenience sampling was used to identify participants. Test managers at Ericsson C-RAN in Sweden and Canada were approached and invited to participate in the survey study. A total of 10 test managers took part, who utilize ITMOS in their daily work.

The survey included 27 questions, divided into six categories regarding correctness, speed, value, usability, interpretability, and stability. These categories are inspired by a set of six characteristics that suggest whether a problem domain is well-suited to the application of machine learning [53, 54][2].

The survey instrument is presented in Appendix A. The correctness section focuses on understanding the performance requirements for ITMOS in its intended domain, while the speed-related questions assess the necessary speed for effective decision support. The value section focuses on investigating the investment case for implementing a decision support system similar to

[2]The applicability of ITMOS to automated test management decisions is further discussed in Section 7.1.

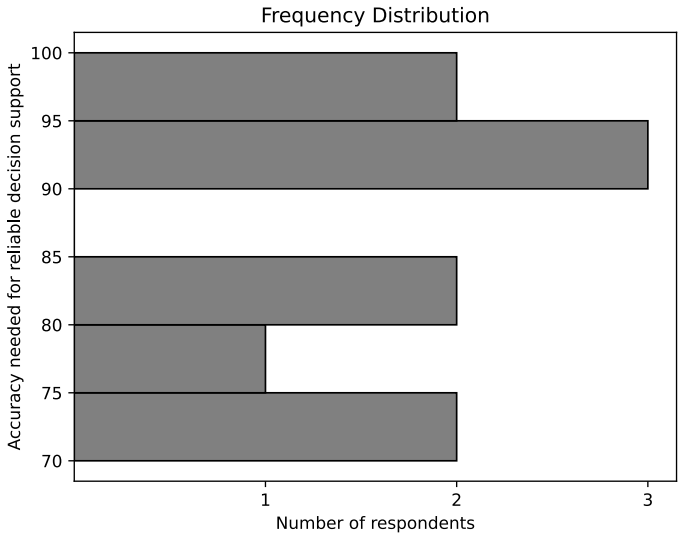

Figure 5: Survey results regarding the required accuracy for ITMOS to serve as a decision support tool.

ITMOS, whereas usability questions gauge end-user perspectives on the usability of ITMOS. Interpretability queries address the need for interpretability in AI systems within the domain, as well as ITMOS's interpretability level. Lastly, the stability questions examine the underlying stability of this problem domain and its implications for ITMOS's applicability.

6.2. Survey Evaluation

The survey responses then underwent thematic analysis, a qualitative research method that identifies patterns and themes within the data. This involves familiarizing with the responses, coding relevant concepts, generating initial themes, reviewing and refining them, and finally defining and naming them. The thematic analysis enables a systematic and comprehensive exploration of the survey data to uncover meaningful insights and understand underlying meanings. In addition, we use descriptive statistics to analyze the quantitative data.

6.3. Survey Results

Correctness: Measuring the correctness of ITMOS can help assess the accuracy and reliability of its recommendations. This involves evaluating IT-

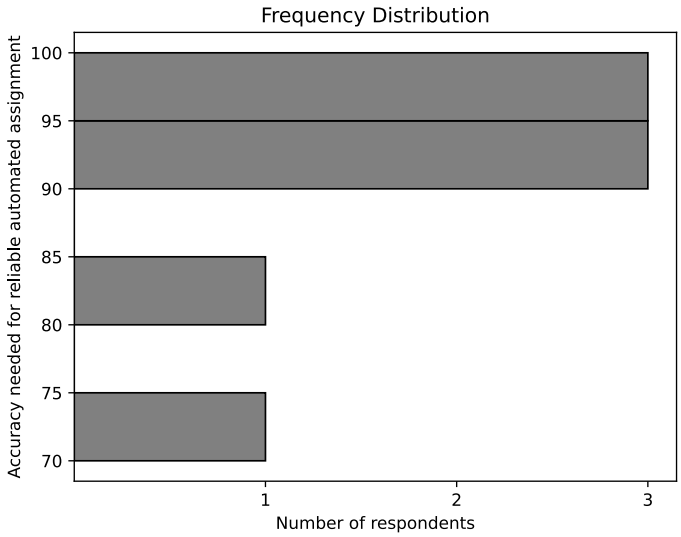

Figure 6: Survey results regarding the required accuracy for ITMOS to serve as a fully automated CI configuration assignment tool.

MOS's ability to consistently generate correct decisions across different CI configurations as well as the level of accuracy necessary for ITMOS to be considered effective.

Figures 5 and 6 show opinions on the level of accuracy required for ITMOS to be used to aid in decision making and the level required for it to serve as a fully automated tool—i.e., without a human involved in making a decision. The results reveal that consensus is lacking. While the accuracy needed for reliable decision support spans between 70% to 99%, with a median requirement of 85%, higher accuracy levels of around 90% are preferred for fully automated assignment. Additionally, respondents highlighted the significant consequences of misassignment, with 40% rating its importance as 4 out of 5 and 35% rating it as 5 out of 5. This indicates its perceived importance. Interestingly, there is disagreement on whether misassignment consequences are uniform across all CI configurations, with 60% claiming that all misassignments are equally problematic, while 40% suggested that misassignments were more serious for particular CI configurations than others.

Speed: The majority of respondents expressed the opinion that ITMOS should generate predictions within a few minutes or even seconds to serve as a decision support tool without negatively impacting user willingness to utilize

the tool. Specifically, four respondents believe the recommendation should be generated within a few seconds, while four others think that a few minutes would still be acceptable without negatively impacting usability. However, some variations in opinions exist. One respondent expressed satisfaction with receiving recommendations within a few hours, while another highlighted the importance of the instant generation of recommendations to enhance the users' sense of awareness and trust in the ML models.

Ninety percent of respondents believed that sacrificing some speed to improve accuracy would be beneficial. Some respondents express a strong preference for a more accurate model at the expense of the speed at which recommendations can be generated. Others point out that it would be worth trading off some speed to increase accuracy performance until the generation time of the model is approximately one second.

Value: Respondents acknowledge the challenge of accurately estimating the time saved through the use of ITMOS for assigning CI configurations, compared to a fully manual assignment. However, there is a consensus among respondents that significant time savings could be achieved. For example, one respondent mentioned potential savings of a couple of hours per test specification, while another suggested a saving of one hour per test specification. Yet another respondent emphasized that the savings would be substantial.

Additionally, the survey results highlight that the implementation of ITMOS could effectively reduce coordination efforts within Ericsson. Consequently, decisions could be made days faster compared to scenarios where models like ITMOS are not utilized.

However, the survey results indicate disagreement among respondents regarding the extent to which misassignment of CI configurations would be reduced when ITMOS is implemented in practice. Figure 7 displays the distribution of responses. Upon analyzing the answers, it becomes apparent that the majority of respondents are optimistic about the potential of ITMOS to significantly reduce misassignments. However, two respondents appear to be skeptical, believing that the reduction will be below 10%. The median estimate for the reduced number of misassignments stands at 60%.

Usability: Based on the survey responses, the task of manually assigning CI configurations to test specifications was perceived to be moderate to highly challenging, with 30% rating the difficulty as 4 out of 5, and 25% rating it as 5 out of 5, where 5 represents a high level of difficulty. However, respondents suggested that having access to a decision support system like ITMOS could

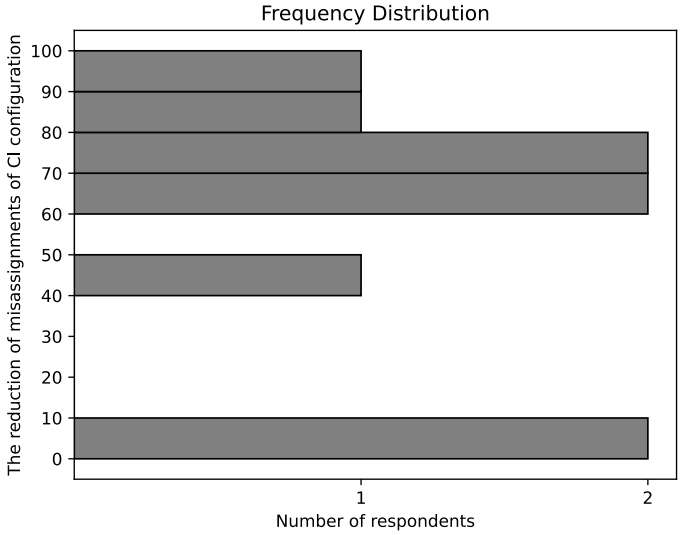

Figure 7: Survey results regarding the estimated reduction of misassignment of CI configurations.

significantly alleviate this difficulty, as 40% rated the difficulty as 2 out of 5, and 35% rated it as 1 out of 5 for the level of difficulty in assigning CI configurations to test specifications with the help of ITMOS. The majority of respondents expressed willingness to actively use ITMOS and were confident in its compatibility with their current workflow.

Those who indicated they would not use ITMOS either did not find assigning CI configurations to be difficult or felt that the system did not integrate well into their workflow. Overall, respondents demonstrated a strong interest in ML-based systems and their practical applications in the workplace.

Interpretability: The survey results reveal that interpretability is highly valued. Respondents rated the importance of interpretability as very high, with 50% rating it as 5 out of 5, and 35% rating it as 4 out of 5, where 5 indicates a very high importance. This suggests that the users of a decision support system like ITMOS in this domain are keen on understanding how ML models arrive at their recommendations. Opinions on the actual level of interpretability of ITMOS varied among respondents. Most responses fell within the range of 2 to 4 on a scale of 1 to 5, where 5 represents "strongly agree". Specifically, 25% rated the interpretability as 2, 30% rated it as 3, and 35% rated it as 4.

Stability: Respondents generally agree that the distribution of CI configurations will change in the future. Specifically, 40% rated the likelihood of change as 4 out of 5, and 35% rated it as 5 out of 5, where 5 indicates a high likelihood of change. However, respondents were less certain about the stability of test specifications, with 30% rating the stability as 4 out of 5, and 25% rating it as 5 out of 5 on the same scale. Regarding potential modifications to the CI configurations, opinions varied among respondents. While 50% of respondents rated the likelihood of changes as 3 out of 5 on the scale, indicating a moderate probability, the remaining respondents perceived a high likelihood of changes to the specific CI configurations.

7. Discussion

The main goal of this study is to introduce, implement, and evaluate ITMOS, an intelligent test management system that is intended to optimize decision-making during the software testing process. To this end, we make the following contributions:

- We have proposed an intelligent test management system that can analyze both functional and non-functional requirements.
- The proposed intelligent test management system can generate recommendations regarding the CI configuration and also test automation.
- The proposed intelligent test management system is integrated into a Python-based tool called ITMOS.
- The evaluation of ITMOS was performed by applying it to an industrial testing project in the telecommunication domain in Sweden.
- ITMOS achieves an average F1-Score of 0.9163 for classifying CI configurations and 0.9463 for determining whether a test specification should be automated—both surpassing the requirements of domain experts

However, the reliance on free-form text input for test instructions may lead to inconsistent or incomplete data entry, affecting the accuracy and reliability of the solution. The end users may omit crucial details or provide ambiguous instructions, resulting in sub-optimal recommendations. Employing other natural language processing (NLP) techniques to parse and extract key information from free-form text input more effectively can be considered

as a future direction of this study. This approach could involve sentiment analysis to gauge the clarity and completeness of instructions, as well as entity recognition to identify important components such as test setup, conditions, procedures, and criteria.

Practitioners within the company utilized a Flask-based GUI to access and interact with the tool. The provided survey in section 6 offers additional insights into the tool and the feedback obtained from the testing team. Additionally, the future direction of this study involves developing a more intuitive and interactive GUI to guide users through the input process effectively, while providing real-time feedback and suggestions for comprehensive and accurate data entry. Furthermore, employing clustering techniques to identify common themes and topics in test instructions may allow for more personalized and relevant recommendations.

As we evaluated ITMOS, focusing on correctness, speed, value, usability, interpretability, and stability, these insights, though promising, underscore the need for careful consideration regarding their generalizability. Therefore, while ITMOS shows significant potential, making broad claims about its universal value and interpretability should be approached cautiously and supported by further validation in diverse contexts. Future research should prioritize external validation to confirm these benefits across different organizational settings.

7.1. Applicability of Machine Learning to Test Management Optimization

As part of this study, we are interested in assessing whether supervised learning is appropriate for determining test management tasks such as CI configuration. To help make this determination, we have identified six characteristics that suggest whether a problem domain is well-suited for machine learning [53, 54].

1. **Tolerance to errors:** achieving 100% accuracy with supervised learning is virtually impossible. Therefore, an ML-friendly domain must be capable of tolerating a certain degree of error in predictions [54].

2. **Inapplicability of conventional approaches:** a problem must not be easily solvable using traditional models or algorithms, e.g., where determining a precise solution is not possible or sufficiently efficient [54].

3. **Low interpretability requirements:** supervised learning models often lack interpretability—that is, they do not explain the rationale

behind a particular prediction. Thus, ML is most suitable to domains where such rationale is not required [54].

4. **Mathematical expressibility of the objective function:** models are trained to optimize an objective function, quantifying the desired outcome or performance measure. The objective function for a problem should be mathematically expressible to enable optimization and parameter tuning [54].

5. **Stability of the objective function:** the underlying problem or task for which the model is being developed should not undergo frequent or significant changes. Stability allows for the development of reliable and robust models that generalize well to new data [54].

6. **Availability of sufficient training data:** supervised learning relies heavily on large amounts of labeled training data to make accurate predictions. An ML-friendly domain should have access to sufficiently large and representative training datasets [53, 54].

We hypothesize that tasks such as determining CI configurations or the form of testing are suitable for supervised learning. Currently, such decisions are often made manually, resulting in significant costs. We propose that implementing an ML model would lead to substantial cost savings, rendering the traditional approach inefficient (Factor 2).

Further, the "objective function" to be optimized could be expressed as the accuracy of the prediction model (Factor 4). Furthermore, the objective function is stable, as long as, e.g., the set of CI configurations does not change frequently (Factor 5). Many companies log test specifications and CI configurations in a central test management system, allowing for the efficient gathering of training data from many organizations (Factor 6).

The exact error (Factor 1) and interpretability (Factor 3) tolerances in this domain are not clear. However, we hypothesize that some degree of error tolerance and acceptance of a lack of interpretability would be acceptable as long as the model is sufficiently accurate that the cost of fixing prediction errors does not outweigh the cost savings of having such a model. In addition, the existing process is potentially prone to error as well, as domain expertise is required to make correct decisions.

7.2. Threats to Validity

In this subsection, we discuss the validity threats, the research limitations, and the challenges in conducting the present study. However, while there are clear indicators of the solution's potential for broader use, the nuances of organizational culture, existing processes, and bespoke requirements underscore the need for further studies to validate and adapt the solution in diverse contexts.

Internal Validity: Some degree of non-determinism is expected when evaluating machine learning models, which can introduce variability in the results. To mitigate this threat, we executed the models 100 times and took the average of the results. Another potential threat arises from the limited scope of hyperparameter tuning, which was constrained by computational limitations. This constraint may have led to suboptimal model performance. However, the observed performance was still sufficient in the experiments.

There are many ML models. We could have potentially attained better results through other models, such as neural networks. However, the models employed are common in ML research and have been applied to similar problems. In addition, there is a threat of potential faults in our implementation. We minimized this risk by leveraging open-source implementations. In adherence to best practices in software development, we have meticulously documented the dependencies on open-source software components within the Implementation section of our project. The major set of libraries utilized includes scikit-learn (sklearn), NumPy, pandas, Matplotlib, and SciPy. Each of these libraries plays a pivotal role in various aspects of our implementation, ranging from data preprocessing and analysis to model training and visualization. By explicitly specifying these dependencies, we uphold transparency and facilitate reproducibility, enabling fellow developers to effortlessly comprehend the technological framework underpinning our project. Furthermore, this practice ensures compliance with licensing requirements associated with the utilization of open-source software. As our project evolves, this comprehensive documentation will serve as a valuable reference point for maintaining and updating our software components effectively.

External Validity: Our study was performed at a single company (Ericsson AB) in a single product domain (telecommunications). There is a risk that our results do not generalize. However, the formulations of test specification and CI configurations employed in this study are relatively generic—with few industry-specific aspects—suggesting that the findings in this study may

have broader applicability. We hypothesize that organizations that employ a similar methodology of assigning test specifications to pre-determined CI configurations are likely to achieve similar results, regardless of the industry. In future studies, we will explore the applicability to other domains. It is also important to note that the aim of industrial case studies is not necessarily to generalize findings to a broader population but rather to provide in-depth, context-specific insights that can inform theory-building, practice, and further research directions. Researchers often employ complementary methods, such as multiple case studies or quantitative analyses, to corroborate and extend the findings of single case studies.

Generalizability: The deployment of ITMOS at Ericsson AB showcases several benefits that could extend to other organizations. Survey responses from Ericsson AB highlight substantial gains in technical performance, including enhanced processing speed, accuracy, and system reliability. These quantifiable improvements are likely to be relevant to other companies with similar technical environments. Additionally, common patterns observed in user adoption and system integration suggest that ITMOS could be effectively applied by engineering teams across different organizations. However, it is crucial to note that certain findings may be specific to Ericsson AB. For instance, cultural factors influencing the acceptance of ITMOS and unique integration challenges related to Ericsson AB's existing infrastructure might not be encountered elsewhere. Furthermore, bespoke features designed to meet Ericsson AB's specific requirements could limit the solution's broader applicability.

Limitations: The proposed ITMOS framework relies on training data from historical test specifications and decisions on test automation. It is important to address the assumption that these historical categorizations and decisions were correct. We acknowledge that past data may contain inaccuracies or biases, which could affect the performance of the classifier. To mitigate this risk, we have incorporated mechanisms for continuous validation and adjustment of the model. These mechanisms include:

- Regular reviews and audits: we conduct regular reviews and audits of past data to identify and correct any inaccuracies.
- Expert feedback integration: we integrate feedback from subject matter experts to refine and validate the model's outputs.

- Adaptability and learning: we ensure the model can adapt and learn from new data to improve its accuracy over time.

By adopting these practices, we aim to enhance the reliability of our approach and ensure that the model's decisions are as accurate and effective as possible.

8. Conclusions

This study presents a novel test management system, ITMOS, that utilizes natural language processing and supervised machine learning to support decision-making during the test management process. Specifically, in this study, we utilized ITMOS to automatically select an appropriate CI configuration to perform requirement verification and to recommend whether a test specification to verify a requirement should be automated or performed manually. ITMOS is modular and can be expanded in the future to support other test management decisions.

To validate the effectiveness of our approach, we conducted a survey study involving domain experts from Ericsson, who provided valuable feedback on the proposed decision support system. This study explored the applicability of ML models and showcased our system's potential benefits in real-world scenarios. We have shown that it is possible to achieve an accuracy of 0.9139 and an F1-Score of 0.9075 using supervised ML, which surpasses the performance requirements established by most domain experts.

Implementing supervised ML in this domain brings significant potential business value, manifesting in several tangible benefits. Firstly, this implementation could lead to a reduction in misassignments—mistakes made by humans who perform test management. This improvement could minimize errors and enhance the overall quality of the testing process. Secondly, the use of supervised ML could result in a considerable reduction in the time required for making test management decisions. By semi-automating this process, the testing process becomes more efficient and allows engineers to focus their efforts on other critical tasks, thereby increasing productivity. Furthermore, applying supervised ML could reduce fault slip-through, meaning potential issues or faults are identified and addressed promptly. This proactive approach mitigates the risks of shipping bugs and other defects to customers.

Further research into the applicability of implementing a fully automated test management tool—encompassing a broader set of management decisions—would be of interest. We propose conducting a comprehensive case study that

implements a fully automated system in collaboration with one or more companies, as this would provide real-world insights and practical implications for replacing manual decision-making with automated or semi-automated decision-making. In addition, we would like to explore whether more complex machine learning and natural language models, such as deep neural networks or large language models could achieve even higher accuracy on decision-making tasks during test management.

References

[1] S. Tahvili, L. Hatvani, Artificial Intelligence Methods for Optimization of the Software Testing Process With Practical Examples and Exercises, Elsevier, 2022.

[2] M. Aniche, Effective Software Testing: A developer's guide, Simon and Schuster, 2022.

[3] L. Chung, B. A. Nixon, E. Yu, J. Mylopoulos, Non-functional requirements in software engineering, volume 5, Springer Science & Business Media, 2012.

[4] M. Meyer, Continuous integration and its tools, IEEE software 31 (2014) 14–16.

[5] M. Shahin, M. A. Babar, L. Zhu, Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices, IEEE access 5 (2017) 3909–3943.

[6] A. Debbiche, M. Dienér, R. Berntsson Svensson, Challenges when adopting continuous integration: A case study, in: Product-Focused Software Process Improvement: 15th International Conference, PROFES 2014, Helsinki, Finland, December 10-12, 2014. Proceedings 15, Springer, 2014, pp. 17–32.

[7] A. Miller, A hundred days of continuous integration, in: Agile 2008 conference, IEEE, 2008, pp. 289–293.

[8] D. Galin, Software Quality Assurance: From Theory to Implementation, Pearson Education, 2004.

[9] W. E. Lewis, Software Testing and Continuous Quality Improvement, Wiley, 1999.

[10] G. M. Weinberg, Quality Software Management: Systems Thinking, Dorset House Publishing, 1992.

[11] E. Dustin, Effective Software Testing: 50 Specific Ways to Improve Your Testing, Addison-Wesley Professional, 2002.

[12] R. O. Rogers, Scaling continuous integration, in: Extreme Programming and Agile Processes in Software Engineering: 5th International Conference, XP 2004, Garmisch-Partenkirchen, Germany, June 6-10, 2004. Proceedings 5, Springer, 2004, pp. 68–76.

[13] M. Santolucito, J. Zhang, E. Zhai, J. Cito, R. Piskac, Learning ci configuration correctness for early build feedback, in: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2022, pp. 1006–1017.

[14] D. Medvedev, K. Aksyonov, The development of a simulation model for assessing the ci/cd pipeline quality in the development of information systems based on a multi-agent approach, in: MATEC Web of Conferences, volume 346, EDP Sciences, 2021, p. 03095.

[15] V. Garousi, D. Pfahl, When to automate software testing? a decision-support approach based on process simulation, Journal of Software: Evolution and Process 28 (2016) 272–285.

[16] D. Graham, M. Fewster, Experiences of test automation: case studies of software test automation, Addison-Wesley Professional, 2012.

[17] B. Marick, When should a test be automated, Proceedings of The 11th International Software/Internet Quality Week (1998) 1–20.

[18] J. Hwang, M. Bulut, A. Canso, S. Nadgowda, Dynamic automation of pipeline workpiece selection, China Patent 115668129A, Jan. 2023.

[19] A. Bregman, S. Mattar, Execution platform assignments in ci/cd systems, 2023. US Patent App. 17/370,305.

[20] Y. Amannejad, V. Garousi, R. Irving, Z. Sahaf, A search-based approach for cost-effective software test automation decision support and an industrial case study, in: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops, 2014, pp. 302–311.

[21] H. Spieker, A. Gotlieb, D. Marijan, M. Mossige, Reinforcement learning for automatic test case prioritization and selection in continuous integration, in: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2017, pp. 12–22.

[22] D. Flemström, P. Potena, D. Sundmark, W. Afzal, M. Bohlin, Similarity-based prioritization of test case automation, Software quality journal 26 (2018) 1421–1449.

[23] K. Wiegers, J. Beatty, Software requirements, Pearson Education, 2013.

[24] R. Malan, D. Bredemeyer, et al., Functional requirements and use cases, Bredemeyer Consulting (2001).

[25] M. Glinz, On non-functional requirements, in: 15th IEEE international requirements engineering conference (RE 2007), IEEE, 2007, pp. 21–26.

[26] P. Runeson, A survey of unit testing practices, IEEE software 23 (2006) 22–29.

[27] P. C. Jorgensen, C. Erickson, Object-oriented integration testing, Communications of the ACM 37 (1994) 30–38.

[28] R. Binder, Testing object-oriented systems: models, patterns, and tools, Addison-Wesley Professional, 2000.

[29] M. Fowler, M. Foemmel, Continuous integration, 2006.

[30] M. I. Jordan, T. M. Mitchell, Machine learning: Trends, perspectives, and prospects, Science 349 (2015) 255–260.

[31] P. Cunningham, M. Cord, S. J. Delany, Supervised learning, Machine learning techniques for multimedia: case studies on organization and retrieval (2008) 21–49.

[32] P. P. Shinde, S. Shah, A review of machine learning and deep learning applications, in: 2018 Fourth international conference on computing communication control and automation (ICCUBEA), IEEE, 2018, pp. 1–6.

[33] A. Fontes, G. Gay, The integration of machine learning into automated test generation: A systematic mapping study, Software Testing, Verification and Reliability (2023) e1845.

[34] G. G. Chowdhury, Natural language processing, Fundamentals of artificial intelligence (2020) 603–649.

[35] Y. Meng, J. Huang, G. Wang, C. Zhang, H. Zhuang, L. Kaplan, J. Han, Spherical text embedding, Advances in neural information processing systems 32 (2019).

[36] J. C. Young, A. Rusli, Review and visualization of facebook's fasttext pretrained word vector model, in: 2019 international conference on engineering, science, and industrial applications (ICESI), IEEE, 2019, pp. 1–6.

[37] C. Vassallo, S. Proksch, A. Jancso, H. C. Gall, M. Di Penta, Configuration smells in continuous delivery pipelines: a linter and a six-month study on gitlab, in: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020, pp. 327–337.

[38] International Organization for Standardization (ISO), Iso/iec 25010, 2011. URL: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010.

[39] F. Zampetti, S. Geremia, G. Bavota, M. Di Penta, Ci/cd pipelines evolution and restructuring: A qualitative and quantitative study, in: 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2021, pp. 471–482.

[40] M. Hilton, T. Tunnell, K. Huang, D. Marinov, D. Dig, Usage, costs, and benefits of continuous integration in open-source projects, in: Proceedings of the 31st IEEE/ACM international conference on automated software engineering, 2016, pp. 426–437.

[41] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, V. Filkov, Quality and productivity outcomes relating to continuous integration in github, in: Proceedings of the 2015 10th joint meeting on foundations of software engineering, 2015, pp. 805–816.

[42] P. M. Duvall, S. Matyas, A. Glover, Continuous integration: improving software quality and reducing risk, Pearson Education, 2007.

[43] L. Chen, Continuous delivery: overcoming adoption challenges, Journal of Systems and Software 128 (2017) 72–86.

[44] P. Bojanowski, E. Grave, A. Joulin, T. Mikolov, Enriching word vectors with subword information, Transactions of the association for computational linguistics 5 (2017) 135–146.

[45] G. Biau, E. Scornet, A random forest guided tour, Test 25 (2016) 197–227.

[46] S. Okada, M. Ohzeki, S. Taguchi, Efficient partition of integer optimization problems with one-hot encoding, Scientific reports 9 (2019) 13036.

[47] W. S. Noble, What is a support vector machine?, Nature biotechnology 24 (2006) 1565–1567.

[48] S. Tahvili, L. Hatvani, E. Ramentol, R. Pimentel, W. Afzal, F. Herrera, A novel methodology to classify test cases using natural language processing and imbalanced learning, Engineering Applications of Artificial Intelligence 95 (2020) 1–13.

[49] S. Tahvili, A. Singh, Intelligent Test Management System (ITMOS) (2024). URL: https://figshare.com/articles/software/Intelligent_Test_Management_System_ITMOS_/26097232. doi:10.6084/m9.figshare.26097232.v1.

[50] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empirical software engineering 14 (2009) 131–164.

[51] M. Makhanbet, X. Zhang, H. Gao, H. A. Suraweera, An overview of cloud ran: Architecture, issues and future directions, in: P. Fleming,

N. Vyas, S. Sanei, K. Deb (Eds.), Emerging Trends in Electrical, Electronic and Communications Engineering, Springer International Publishing, Cham, 2017, pp. 44–60.

[52] L. K. Owens, Introduction to survey research design, in: SRL fall 2002 seminar series, volume 1, 2002.

[53] D. Rafique, L. Velasco, Machine learning for network automation: overview, architecture, and applications [invited tutorial], Journal of Optical Communications and Networking 10 (2018) D126–D143.

[54] R. M. Morais, On the suitability, requisites, and challenges of machine learning, Journal of Optical Communications and Networking 13 (2021) A1–A12.

Appendix A. Survey Instrument

	Question wording	Response options
1	What percentage of correct assignments is necessary for the system to effectively function as a decision support tool?	Text box
2	What percentage of correct assignments is necessary for the system to operate as a fully automated system?	Text box
3	How would you rate the significance of the consequences of misassignment in this domain? (5= very significant, 1= insignificant)?	$1, 2, 3, 4, 5$
4	Do the consequences of misassignment vary across all CI configurations?	Yes, No
5	If not, how do the consequences vary across different CI configurations?	Text box

Table A.8: Survey responses evaluating the Correctness.

	Question wording	Response options
6	What is the acceptable time frame for the model to generate a recommendation without impacting your willingness to use the tool?	Text box
7	Given that the model currently generates recommendations with an accuracy level of X and an F1-score of Y, while operating at a speed of Z seconds, would it be beneficial to trade off some speed to enhance the accuracy or F1-score?	Yes, No
8	If so, what degree of improvement would be required to justify the slower speed? Could you provide an example of a favorable trade-off between accuracy and speed?	Text box

Table A.9: Survey responses evaluating the Speed.

	Question wording	Response options
9	How much faster do you anticipate the tool will enable you to assign CI configurations to test specifications? What is the estimated time savings per test specification?	Text box
10	What is your best estimate of the reduction in misassignments of CI configurations that can be achieved using the proposed tool? (Please provide your answer in percentage)	Text box
11	How many misassignments of CI configurations are estimated to occur annually?	Text box
12	Do you agree with the estimation that a misassignment of a CI configuration leads to an average feature delay of two months in delivery?	Yes, No
13	If you answered "Yes" to the previous question, please skip this one. Otherwise, on average, how long of a feature delay do you estimate one misassignment results in?	Text box
14	What is your best estimate of the cost associated with this feature delay? (Please provide your estimate in SEK (the national currency of Sweden)	Text box
15	How many fewer fault slip-throughs (i.e., undetected bugs) do you expect to occur when using ITMOS compared to the current process?	Text box
16	What is your best estimate of the average cost associated with a fault slip-through? (Please provide your estimate in SEK)	Text box
17	Do you believe that using ITMOS will reduce the requirement for domain expertise in assigning CI configurations?	Yes, No

Table A.10: Survey responses evaluating the Value.

	Question wording	Response options
18	Currently, what level of difficulty do you perceive in assigning CI configurations to test specifications? (5= more difficult, 1= less difficult)	$1, 2, 3, 4, 5$
19	How difficult do you anticipate it would be to assign CI configurations to test specifications using ITMOS? (5= more difficult, 1= less difficult)	$1, 2, 3, 4, 5$
20	How well do you perceive ITMOS integrates with your existing workflow? (5= very well, 1= less well)	$1, 2, 3, 4, 5$
21	If ITMOS were fully deployed in the Ericsson ecosystem, would you actively utilize it? (5= very inclined to use it, 1= not inclined to use it)	$1, 2, 3, 4, 5$
22	Overall, to what extent do you trust Artificial Intelligence -based systems for practical use in the workplace? (5= high trust, 1= low trust)	$1, 2, 3, 4, 5$

Table A.11: Survey responses evaluating the Usability.

	Question wording	Response options
23	How important do you consider interpretability, i.e., understanding how ITMOS generates its recommendations, to be? (5= very important, 1= not important)	$1, 2, 3, 4, 5$
24	Do you believe that the current method used by ITMOS to present its recommendations adequately addresses interpretability concerns? (5= strongly agree, 1= strongly disagree)	$1, 2, 3, 4, 5$

Table A.12: Survey responses evaluating the Interpretability.

25	How likely do you think it is that the distribution of the various CI configurations will change in the future? (5= very likely, 1= very unlikely)	$1, 2, 3, 4, 5$
26	How likely do you believe it is that the current structure of test specifications will change in the future? (5= very likely, 1 = very unlikely)	$1, 2, 3, 4, 5$
27	How likely do you perceive the different CI configurations to be subject to future modifications? (5= very likely, 1= very unlikely)	$1, 2, 3, 4, 5$

Table A.13: Survey responses evaluating the Stability.

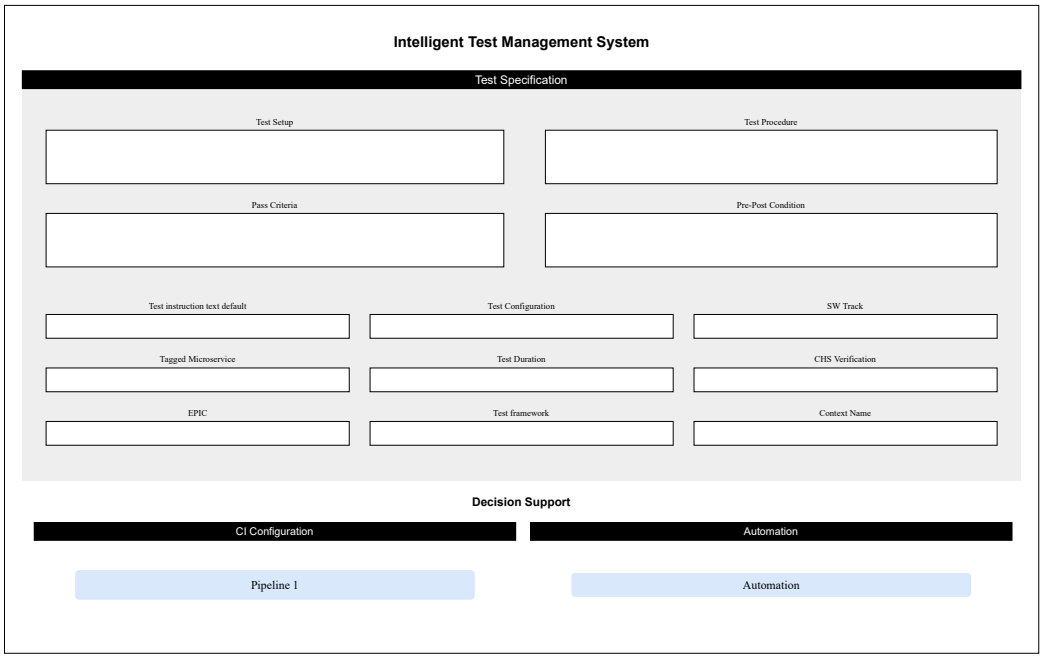

Figure A.8: The graphical user interface (GUI) of ITMOS.