Universiteit Leiden

Master Computer Science

Predictive Test Advice System: Enhancing Software Testing Efficiency at ASML

Name:	Ritesh Sharma
Student ID:	S4158059
Date:	30/04/2025
Specialisation:	Artificial Intelligence
1st supervisor:	Dr. S. Salehkaleybar
2nd supervisor:	Prof. Dr. M.M. Bonsangue

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Contents

Acknowledgment

I would like to express my profound gratitude to my supervisors, **Dr. S. Salehkaleybar** and **Prof. dr. M. M. Bonsangue**, for their steadfast guidance, thoughtful feedback, and unwavering encouragement throughout both my internship and the master's thesis project. Their depth of knowledge and patient mentorship have been instrumental in shaping the direction and outcomes of this work.

I am equally indebted to the Team at ASML for entrusting me with a real-world data science challenge that allowed me to grow both technically and professionally. In particular, I want to thank my ASML supervisor, **Dr. Roman Shantyr**, for welcoming me into the complexities of lithography system performance and the analytical workflows that underpin it, as well as for his continual support and insights. My sincere appreciation also goes to my manager, **Burcu Babur**, for fostering a collaborative environment and offering invaluable counsel that guided my efforts.

I owe further thanks to **Ernest Mithun Xavier Lobo**, whose domain expertise and timely advice often propelled the analysis forward, as well as **C. Mocking** and **E. Dias dos Santos**, who provided essential feedback and readiness to share industry knowledge in addressing the challenges of throughput and quality assurance in a high-stakes manufacturing context.

Additionally, I extend my heartfelt appreciation to **S. N. van der Valk MSc**, my study advisor at LIACS, for helping me navigate the academic rigors of the Master's program and for offering crucial support whenever hurdles arose.

This thesis would not have been possible without the collective contributions and encouragement of these remarkable individuals. Their willingness to share their time, expertise, and enthusiasm made this journey both enlightening and deeply rewarding. I am truly grateful for their support and look forward to applying the lessons learned as I advance further in data science and software engineering.

Abstract

In today's time, software testing is one of the most important parts of the software development life-cycle, which ensures system reliability and software quality. In a semiconductor manufacturing environment, software testing is even more important due to the high cost associated with undetected failures. At ASML, where rapid software evolution and continuous integration are the norm, conducting exhaustive testing to validate software integration is an impractical approach. This is because of the enormous number of test scenarios present in test suites where most of the scenarios pass. In this thesis, we propose a predictive test advice system that is designed to prioritize test scenarios by leveraging historical test results, file changes, and code quality metrics. By a combination of these diverse data sources, we construct a rich feature set that includes static, dynamic and time-based rolling metrics, while addressing the challenges faced due to severe class imbalance in our data.

The methodology that we adopted is based on the use of CatBoost, an algorithm that uses gradient boosting and is capable of handling many categorical features and uses ordered boosting to prevent overfitting. The model is validated by using an expanding-window cross validation that respects the chronological order of the Qualified Baseline (QBL) data. A Qualified Baseline (QBL) represents a critical checkpoint in our continuous integration process where code changes that have passed extensive testing and thorough review become the stable foundation for all future development and integration efforts. This way, only the past data is used for the predictions, which is consistent with the way of continuous integration in the real world. The performance of the model was also improved by a comprehensive hyperparameter optimization, which led to the setting of 1000 iterations, tree depth of 7, learning rate of 0.01, and class weight ratio of [1, 25].

An evaluation on real data shows that the trained model has a high recall rate of about 80%. The precision is rather low, but this is an acceptable compromise given the fact that the absolute majority of errors in an industrial application should be avoided. The predictive framework is found to generalize well to unseen scenarios on new QBL data from 2025, which supports its potential as a robust tool for optimizing test execution and resource allocation.

Although the present system constitutes a considerable enhancement with regard to the effectiveness of testing at ASML, future work will involve refining the interdependency modeling process, enhancing drift detection, and integrating continuous retraining into the system in order to achieve higher levels of predictive accuracy in the context of an evolving software environment.

1 Introduction

Software Testing is considered as an important part of the software development life cycle to ensure that defects are rejected, new features are validated, and the product stability is maintained in the light of changing code and environmental conditions. In the semiconductor manufacturing industry, the software that regulates and supervises sophisticated machines is continuously being developed through enhancement of features, correction of bugs or updating of the software for compatibility. This pace of change coupled with the complexity of interactions between hardware parts demands frequent testing at each iteration. However, in large scale operations where there are thousands of possible test conditions, it would be impractical to execute all the checks within the desired time frame as it leads to wastage of resources. Hence, the problem is how to choose an optimal number of tests which are most likely to detect real failures without compromising on the product quality or coverage.

Predictive test selection leverages historical test execution results, component changes, and environmental factors to prioritize test cases likely to fail, reducing unnecessary executions. This data-driven approach is intended to preserve failure detection capabilities while improving the efficiency of the testing process, reducing the number of unnecessary runs, and saving resources. When used in conjunction with domain knowledge, predictive test selection can provide valuable information on the potential risks that are associated with a certain feature or subsystem due to a recent change

This thesis is about the development of a Predictive Test Advice System in the context of ASML, the world's leading manufacturer of photolithography equipment for the semiconductor industry. ASML's machines work on nanometer scale precision and rely on very complex and continuously changing software. Each change, whether an incremental modification to improve machine throughput or an entirely new feature to support next generation lithography processes, poses potential risks that have to be identified and verified through testing. With our proposed system, we aim to use historical test data and component change information to identify where those risks are most likely to be present, steering engineers toward the portion of the tests that are most likely to be defects or regressions.

This research has implications that go beyond simply finding out which test scenarios should be run. The consequences of reducing testing can be significant. For instance, the saved resources (in terms of CPU hours, or the time spent testing in a laboratory using dedicated equipment, or engineering effort) can be used for more sophisticated analyses and rapid iteration. Furthermore, the capability to predict failing scenarios is a proactive approach to quality. Potential failures can be investigated early in the development cycle, before the cost of fixing them or the downtime to fix them is high. In an environment such as ASML, where precision is a critical factor and even small defects can lead to a production efficiency loss, such improvements in testing effectiveness have a real effect on product reliability and development speed.

1.1 Problem Statement

Despite the importance of thorough testing, many organizations still employ exhaustive or static approaches that treat all scenarios equally, regardless of how likely they are to fail. This practice is costly and ignores possible subtle patterns of software changes and their relationships with potential failure. In particular, in ASML's semiconductor manufacturing environment, the absence of a predictive framework for test selection results in critical tests sometimes going undetected while developers spend time on uninformative test runs. This lack of targeted advice delays defect detection, increases testing costs, and leads to later feedback to development teams operating on tight iteration timelines.

This research addresses this need for an adaptive and data driven system that targets the most

likely to fail scenarios using historical data and component change data. We develop a Predictive Test Advice System that dynamically selects high-risk test scenarios, optimizing test coverage while reducing costs, thus improving software quality and operational efficiency in a high stakes manufacturing environment.

1.2 Structure of Study

In the following sections, we explain the fundamental concepts of predictive test selection, we explain the challenges of large-scale test management at ASML, and describe the end-to-end pipeline for developing, verifying, and deploying the predictive model. We first explored prior work on predictive test selection and software analytics (Chapter 2) before introducing the details of the ASML-specific data sources and feature engineering process (Chapters 3-4). We then present our model development and empirical validation (Chapters 5-7), demonstrating how our approach improves test efficiency. Finally, we discuss future enhancements and deployment considerations (Chapters 8-10).

In this thesis, we show how our Predictive Test Advice System for ASML can help realize selective and intelligent testing that not only helps save time and resources, but also enhances software quality in an industrial environment where each defect can pose a relatively high operational risk.

2 Literature Review

Testing is a critical phase in the software development life cycle to ensure that changes made do not cause regressions or failures [Rothermel et al., 2001]. Conventional test selection techniques involve either performing a comprehensive set of tests, which is costly in terms of computation time, or using static dependency-based selections that are inadequate to recognize complex behavioral patterns in the execution of the software. Predictive test selection, based on machine learning and past test execution information, is a recent innovative approach for improving the effectiveness of test execution by identifying critical risk areas with a reduced number of tests.

Data-driven strategies enhance testing efficiency without compromising fault detection rates. [Google AI Team, 2016] [Facebook Eng. Team, 2019] [Mozilla Hacks Team, 2021] have adopted machine learning-based test selection model on their respective data and implemented it to reduce testing time by up to 90% with high defect detection rates. The use of such methodologies in large-scale environments such as ASML's semiconductor manufacturing software is therefore a natural progression in optimizing the testing for complex systems.

2.1 Challenges in Traditional Test Selection

Conventional test selection techniques are based on static dependency analysis, which means that the tests are chosen according to build dependencies and file changes. Although efficient, these methods lead to an increase in the number of test executions, which are often unnecessary and result in the unnecessary expenditure of computational resources as well as prolong the time to feedback for developers. Furthermore, static analysis cannot capture the runtime behavioral changes that are brought about by software updates as it is unable to identify subtle defect patterns. Hence, there is a need to adopt a more refined approach that integrates both traditional techniques with adaptive strategies to enhance the effectiveness of test selection processes. [Williams and Kevin, 2024].

The exhaustive testing has been complemented by test case prioritization techniques in an attempt to improve upon them by ranking test cases according to a variety of metrics, including risk based, version based, and cost based prioritization. However, these approaches are typically based on manually defined heuristics and are therefore not very adaptable to changing software environments [Johnson and Miller, 2022]. There is a need for a more scalable and intelligent solution that adapts dynamically to historical test failure patterns and software change histories.

2.2 The Role of Machine Learning in Test Optimization

[Facebook Eng. Team, 2019] found that applying historical data and learned patterns to the training data set of a machine learning model can identify valuable test cases that are most likely to fail. It has also been shown that machine learning models can learn the set of tests impacted by a given code change to good approximation. This is important in large-scale continuous integration systems where the expense of running all tests is too much [Doe and Smith, 2023].

A systematic review of the use of machine learning in testing reveals that decision trees, neural networks, and reinforcement learning are used in conjunction with software testing to predict test case failures and to optimize the size of test executions. Hence, the classifiers learned from the previous test results are capable of identifying the relationship between code change and failure rates in order to reduce the number of tests while keeping a high level of failure detection [Sedighe et al., 2024] [Rongqi Pan, 2021]. But still the results achieved from neural networks, and reinforcement learning implementations were not upto industry standard. Decision-tree based implementations showed promising results on the open source data. But, it's important to note that the data used was different across the proposed solutions.

In a study on Facebook's continuous integration pipeline, researchers implemented a predictive test selection model that reduced test execution costs by 50% while preserving over 99.9% of defect detection capabilities [Facebook Eng. Team, 2019]. This success demonstrates the feasibility of deploying machine learning in industrial testing environments. At Facebook, they focused on the repeatedly changed code regions, along with the highly correlated and interconnected code. This gives Facebook an advantage. They are able to find regression from these regions, as they have data about which tests are related to a specific part of code. Our data at ASML is very vast and there is no such guidelines set to find the relationships between the test scenarios and changed files.

2.3 Predictive Test Selection in Practice

Real-world implementations of predictive test selection have demonstrated substantial gains in efficiency and defect detection. Mozilla, for instance, successfully integrated a machine learning-based test prioritization system that significantly reduced test execution times while maintaining fault coverage of their web browser development data [Mozilla Hacks Team, 2021]. Mozilla focused on historical performance of failed tests. They were also able to find a solution to the problem of flaky tests. A test is termed flaky when on same configuration the test results are different. Similarly, Launchable Inc. has developed a predictive test selection framework that dynamically selects high-value tests based on past failures, allowing developers to receive faster feedback without sacrificing test quality.

These approaches leverage features such as:

- **Code change history**: Frequent modifications to certain components increase the likelihood of failures.
- **Test failure history**: Tests with a history of failures are prioritized.
- **Dependency distance**: The proximity of a change to a given test case in the dependency graph influences selection.
- **Execution time constraints**: Tests are selected based on their runtime impact to balance coverage and efficiency.

2.4 Comparison of Approaches

In the literature, there are various approaches for effectively reducing the test suite. Table 1 shows the advantages and limitations of these approaches:

Table 1: Comparison of Test Selection Approaches

Approach	**Advantages**	**Limitations**
Exhaustive Testing	Ensures complete coverage	Computationally expensive, time-consuming
Dependency-based Selection	Captures direct code-test relationships	Cannot detect deeper behavioral dependencies
Heuristic Prioritization	Focuses on high-risk areas	Requires manual effort, lacks adaptability
Predictive Test Selection	Learns from historical patterns, optimizes execution	Requires data availability, initial model training effort

On the basis of these results, predictive test selection seems to be the most effective way of ensuring that the right number of tests are executed without compromising on the quality of the software.

This way, organizations can reduce costs, improve feedback time and make sure that the right tests are run for every software update by leveraging machine learning to guide test selection [Rongqi Pan, 2021].

The literature clearly supports the prediction of test selection over traditional test selection and prioritization techniques. Large scale deployments at the likes of Facebook, Google, and Mozilla show that predictive test selection cuts infrastructure costs and execution time without compromising on fault detection rates [Google AI Team, 2016] [Facebook Eng. Team, 2019] [Mozilla Hacks Team, 2021]. An intelligent, adaptive system is enabled by integration of historical execution data, component changes and dependency analysis for an efficient test set.

The literature demonstrates that predictive test selection outperforms traditional methods because it saves time and costs but it is crucial to recognize potential drawbacks. The predictive test selection approach depends on extensive historical data about test executions and failure patterns, yet it can generate cold start problems when there is insufficient data available. The inherent probabilistic nature of machine learning methods creates a risk that test cases will be misclassified which leads to incorrect test case result predictions. The probabilistic behavior of the system creates difficulties in explaining its decisions because stakeholders need clear reasons for model choices. The challenges from these issues can be reduced through ASML's large historical data collection and its continuous integration practices.

These insights can be leveraged to develop a Predictive Test Advice System tailored to ASML's semiconductor manufacturing software. The system will then use state-of-the-art predictive test selection methods to optimize test execution and create a scalable, data-driven approach to software quality assurance in a high-stakes industrial environment [Google AI Team, 2016] [Mozilla Hacks Team, 2021].

It is also important to note that the data used by the mentioned literature references are very different to what we have at ASML. Systematic Reviews are based on open source data, which doesn't match the data complexity of industry leaders like ASML. Big companies mentioned above do not provide data due to privacy concerns. Making their findings and results very hard to reproduce. Hence, It is safe to say that our data brings new and unseen challenges with itself, which require custom solutions as existing solutions will not work on our data directly. In the following sections of this thesis, we are going to discuss these challenges and propose our solutions.

3 Data Collection and Description

ASML is operating in a highly dynamic and complex software development environment, whereby several departments and development teams develop various components while working together. Since the expectation of the software quality is very high, there is a need for very strict and very thorough testing. In this regard, ASML schedules routine Qualified Baseline (QBL) testing runs, which are daily and weekly tests, for checking integrity and completeness of software features. A QBL functions as an essential milestone in our continuous integration process because it establishes a stable foundation for future development and integration efforts through extensive testing and thorough review of code changes. Weekly runs include running of a large test scenarios across all the functional clusters (FCs) while the daily runs only cover a small number of these test scenarios. With every QBL testing cycle, new scenarios are introduced and the testing coverage is gradually increased. However, the majority of test scenarios pass [Mohamed A. Shibl, 2021].

In each FC, dedicated teams perform small-scale tests of the modifications, which are specific to their area. Localized tests are critical to finding defects early in the development process, and provide rapid feedback on each code change. However, such tests are not sufficient to guarantee that the overall system will work properly when integrated as a whole. Therefore, all test scenario suites are run only after any changes are to be incorporated officially into a new QBL (as depicted in Figure 1). This last version of integration testing is meant to ensure that all the changes combined from multiple FCs work together without a surprise regression.

This dual level testing strategy where changes are first tested at the FC level and then subjected to a comprehensive QBL testing guarantees both local and system wide reliability. The growing complexity of the test scenarios (as depicted in Figure 2) not only shows the rising number of tests but also the ever-increasing resource requirements to sustain such a dense testing schedule. In the end, this integrated approach resonates with the ultimate goal of ASML's testing framework: to detect and fix defects early to ensure product quality and efficiency.

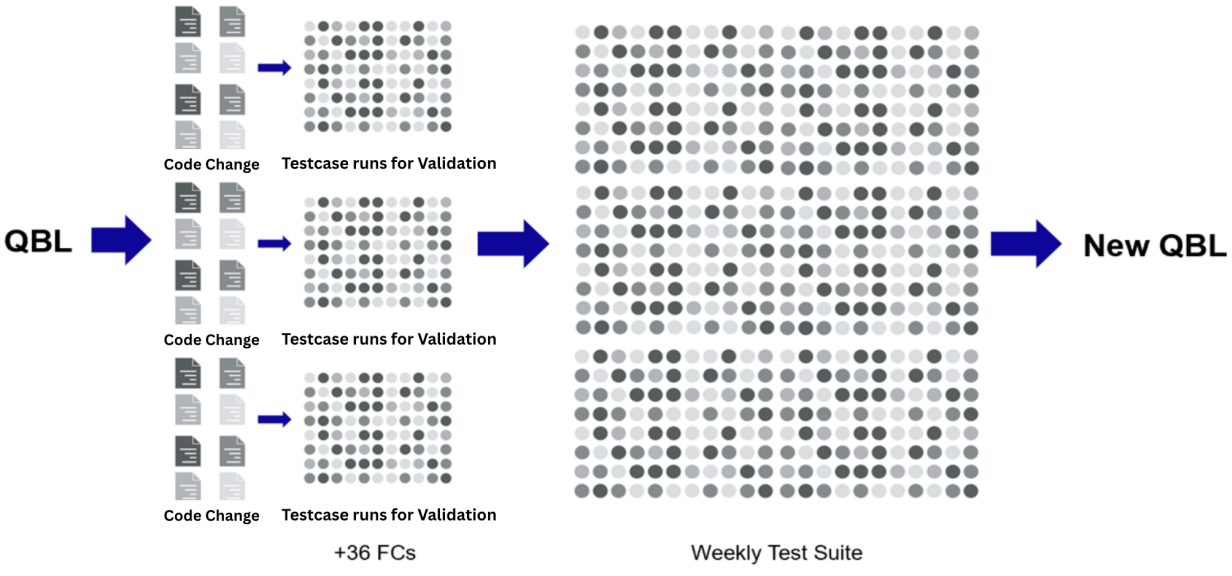

Figure 1: A schematic that shows how each FC runs its own test to validate code changes, which are then fed back into the full test suite to do broader integration testing. It shows how a baseline with code changes is tested before being accepted as the new QBL.

The software testing structure of ASML is based on a hierarchy of about 35 functional clusters, each of which combined have over 500 building blocks (BB). The BBs encapsulate several

components (approximately 3,000), which in turn are associated with more than 15,000 distinct test scenarios. Most scenarios are modeled after several configurations and test cases; a failure at the configuration level implies the failure of the entire test scenario. The main purpose of this research is to design a predictive model that can distinguish between scenarios that are likely to fail during weekly QBL test runs to ensure that resources are used to their fullest potential and tests are as effective as they can be [Kuhn and Johnson, 2013].

As depicted in Figure 2, the rate of executed test scenarios at ASML has been on the rise, especially for the last 3 years, which shows the growing complexity and increasing needs in terms of resources for the thorough testing throughout the period under consideration (2022-2024).

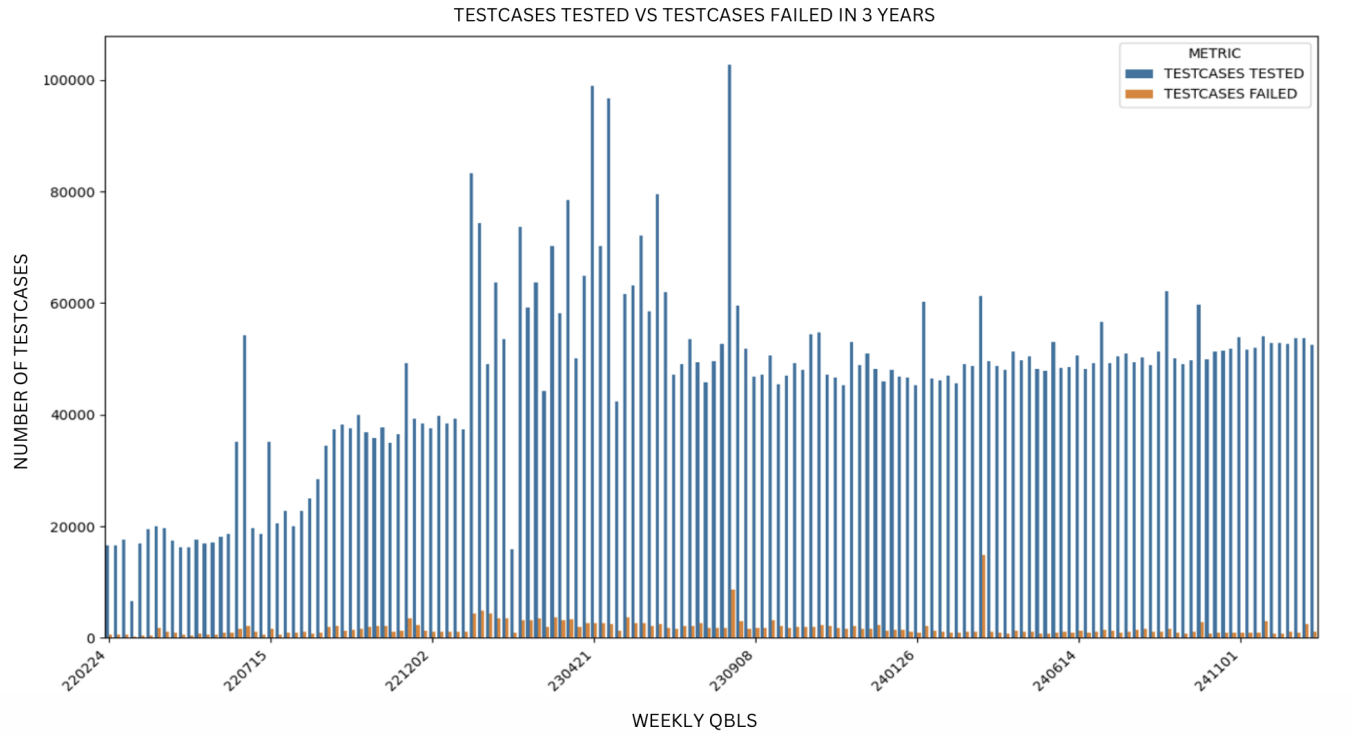

Figure 2: Growth in the number of test scenarios executed during QBL test runs at ASML from 2022 to 2024. This figure illustrates the increasing complexity and resource demands associated with maintaining comprehensive test coverage.

3.1 Data Sources

The development of predictive models which forecast test scenario failures requires study across different stages of development and testing pipeline. The study requires multiple interconnected data sources that provide distinctive and useful information. The predictive model uses extensive information from five main data sources: Historical Test Scenario Execution (Test Result Database), Integration Data, Interface Data, Polluter Data, and Code Quality Data. Each data source brings different aspects of the testing and development process and are all necessary for a detailed predictive analysis.

Historical Test Scenario Execution (Test Result Database)

The historical execution data is the core of the predictive framework. This data is sourced from the Test Result Database (TRD) and is from 2022-2024 and contains detailed logging of every

executed test scenario. The record is further enriched with other attributes like the Qualified Baseline identifier (source), functional cluster, building block, scenario name, configuration name, test level, machine type, target operating system, and the test environment (primarily Google Cloud Platform (GCP)). Furthermore, key data on execution outcomes (pass/fail) and their timestamps is captured, thus giving the data a temporal dimension.

Besides these basic properties, the dataset also contains records of identified known issues, i.e. recurring problems that were highlighted during the previous test cycles. Failures from previous QBL test runs, which are already known to us are named known issues. The presence of known issues allows the model to learn the tendencies of failure that are more likely to recur. The data set is extensive and longitudinal in nature, and thus, considerable time is spent on data collection, accuracy, homogeneity, and density. This basic dataset is used not only for the analysis of historical trends and performance, but also as a framework into which other features from other datasets are appended.

Integration Data

Integration Data is received through an internal Application Programming Interface (API) which provides detailed information regarding QBL and their snapshots. This dataset reflects the changing nature of software development by reporting the changes made between integrations. Each unique software delivery by teams and the number of modified files are also recorded. Further, the file modifications are subdivided into code-specific files, i.e., those with extensions .cpp, .py, .xml, .h, .hpp, .c, .robot, and .ddf, and non-code files, which include Make-files and Markdown documents.

The predictive model is able to contextualize the impact of software modifications on test outcomes by linking integration data to the corresponding QBL identifiers in the historical test logs. This detailed record of integration activities gives a nuanced view of the scale and scope of changes, and how small tweaks and big overhauls affect the probability of test failures. In other words, Integration Data is not only a measure of the volume of changes but also encapsulates the collaborative and iterative process of the software development process.

Interface Data

Interface Data is used to capture the complex inter dependencies among the software components. Because of the modular architecture of modern software, changes in one component can cause a drastic change in other components as well. This data source quantifies such interactions by extracting key metrics that highlight the connectivity and complexity of the system. It also measures the maximum dependency count among the components and the maximum frequency of nested changes (They are defined later in Section 4).

These metrics are helpful in determining the risk of cascading failures. For instance, a high dependency count indicates that a component is highly depended on by others, which means that even a small change can cause propagation through the system. Likewise, more frequent nested changes may indicate a more volatile part of the code base that is more likely to cause indirect failures. Thus, Interface Data is crucial in providing essential information on how small changes can cause big systemic events, and it is an essential part of the predictive analytics framework.

Polluter Data

Polluter Data is based on the idea of identifying the frequent causes of failure, also known as polluters. During the testing process, the initial failure of scenarios is checked. If a test fails, then the test is re-executed to ensure that the failure is consistent. If a test fails repeatedly, it is termed a polluter. After the review by the developers and confirmation of these scenarios as known issues, i.e., intractable problems that adversely impact the software stability to a large extent, they are

further categorized.

Received weekly through an internal API and in CSV format, Polluter Data offers a complete record of faulty scenarios. The data is crucial to the predictive model as it helps in identifying the frequent problems and their effects on the test suite. By identifying the failure patterns, Polluter Data helps the model distinguish between occasional test anomalies and the general problems that require further analysis. Therefore, it improves the predictive ability by focusing on the regions that are likely to cause other failures based on their likelihood.

Code Quality Data

A specialized third-party tool is used to gather Code Quality Data which gives a better view of the technical health of individual software components. This data source has variety of metrics designed to capture both the structural quality and maintainability of the code. The insights are from the key metrics, namely, the Technical Quality Index (`tqi`) and the average cyclomatic complexity (`cycloxAverage`) that help understand the overall complexity and potential fragility of the code. Moreover, the dataset includes the percentage of duplicated code and the effective lines of code (`eloc`).

To further enhance this profile, the dataset also collects metrics on average code coverage and code churn rates (`changerate`). These metrics can be used to highlight parts of the code base that are being changed frequently or that may have low test coverage. For example, high churn rates or low code coverage may point to unstable or poorly maintained components that are more likely to fail. When the model incorporates Code Quality Data, it provides a more accurate risk assessment that relates technical health metrics to the probability of test scenario failures. All the features mentioned here are defined later in section 4.

3.2 Dataset Overview and Characteristics

The dataset processed from the source data sets includes data from 137 QBL test run files from 2022–2024. The size of the dataset is about 6 million records, and there is a strong class imbalance: 97% pass outcomes to around 3% failures. Such imbalance makes standard models face prediction challenges as the model tends to favor the majority class (Pass in our case) which produces poor sensitivity to the minority class (failures). As a result, it achieves high overall accuracy but fails to identify actual failure cases effectively. This class imbalance is addressed through targeted modeling and data sampling techniques discussed in detail later in the methodology section.

Each processed dataset file is tied to a specific QBL test run and are saved separately in Parquet format. The Parquet format was used instead of the traditional CSV because it provides data compression and faster read-write performance, optimized storage, and better scalability to a large amount of data typical in industrial applications.

Table 2 summarizes the approximate dimensions and complexity of the dataset utilized in this research:

Table 2: Overview of dataset statistics used in this study (2022–2024)

Attribute	**Approximate Count**
Qualified Baseline Runs (QBLs)	137
Test Scenarios	15,500+
Components	3,000+
Building Blocks	500+
Functional Clusters	35+
Configurations per Scenario	Multiple
Total Test Executions (Rows)	6,083,455
Passing Instances	5,882,740 (96.7%)
Failing Instances	200,715 (3.3%)

This structured approach to data collection and description is a foundation for strong and meaningful predictive analytics that is designed to greatly improve test scenario selection, failure detection accuracy, and test efficiency within ASML's software development ecosystem.

There were some challenges in integrating data from multiple departments. First, we had to secure access to the diverse datasets because each department had its own data repositories, standards, and protocols. We had coordinating meetings, negotiated data-sharing agreements, and aligned on a common format that could accommodate the heterogeneous nature of the data, from legacy systems to modern cloud-based platforms. We also had to develop bespoke extraction and transformation processes to harmonize these sources, because each department logged and stored, and managed the information differently [Zhu, 2017].

4 Data Preprocessing and Feature Engineering

It is essential to design a rich and carefully chosen feature set for detecting failure signs in large-scale, continuously evolving environments such as ASML's. In addition to the domain knowledge, we also used insights from extensive exploratory data analysis (EDA) to identify the specific attributes of how, why, and when a scenario may fail. The changes in the code base usually happen in a gradual manner and can be seen as increased activity in certain file extensions, or a dependency that was not accounted for in a highly interconnected component can worsen the impact of even a small change. However, it is crucial to monitor the historical performance of each scenario, component, or cluster in the last testing runs in order to find out whether the issues have been resolved, worsened, or appeared for the first time. It is only by combining these signals – code churn metrics, platform configurations, and time series rolling windows of failures – that we can build up a full picture of the dynamics that give rise to the unexpected. Furthermore, since data drift is a normal phenomenon in any long-lived software system, our rolling-window features are based only on the data from past runs, which avoids data leakage and preserves the practical significance of the prediction. Each feature, whether categorical or numerical, brings some extra information that helps the model distinguish between the robust tests and the ones that are likely to regress. Thus, we build up a coherent view of the testing environment based on the recent information about code refactors, system-wide dependencies, operating environments, and historical pass-fail patterns to enable the predictive model to pinpoint high-risk scenarios before they reach production [G Dong, 2018].

4.1 Categorical Features

scenario_name An identifier for the test scenario. EDA showed that some scenarios had higher failure rates consistently, which could be due to their complexity or the focus on a particular function. With `scenario_name`, the model can use historical performance data (e.g., repeated failures) without referring to the outcomes of the current QBL test run. This prevents data leakage and still learns scenario-specific behaviors.

configuration_name Defines the configuration in which a test scenario is to be executed. The data revealed that certain configurations, particularly those involving uncommon hardware or complex environment variables, are more likely to fail. By encoding `configuration_name`, the model identifies failure risks specific to the environment: what might be a stable scenario in one configuration can be a failure in another.

level Sets the hierarchical scope of a test, e.g. Component, BB, FC, or system (SY), etc. SY level tests are very important as these tests integration of different systems. The level feature also provides rolling statistics like `level_10QBL_fail_rate` (explained later in this section), which gives the model a more general view of how the entire test levels have been performing recently.

machine_type Refers to the hardware platform on which the test is running, e.g., a specific machine or hardware configuration set. Platform specific failures can be caused by hardware nuances, wrong driver versions and concurrency issues. With `machine_type`, the model can look at `machine_type_fail_rate_10QBL` (explained later in this section) and see if there has been any recent spike in failures attributable to certain machines.

target_os Defines the type of system (e.g. Wind River Linux, Red Hat Enterprise Linux Version 8) on which the test is performed. OS-level variations can lead to certain dependencies or driver conflicts that can result in failures that may not occur on other OS types. Hence, the `target_os_fail_rate_10QBL` (explained later in this section) rolling metric thus provides a time-phased measure of OS-specific reliability.

component_x Denotes the particular software component to be tested. Components differ in size, complexity, and volatility. Out of the components considered, some had reliably high failure

rates which may have been accrued through frequent refactoring. Having `component_x` allows us to build a rolling fail rate (`component_x_fail_rate_10QBL`, explained later in this section) such that the model can tell whether the risk posture of that component has altered lately.

building_block Within a FC, a fine grained subdivision is small enough to include several related components. Some building blocks are closely correlated with failure spikes.

functional_cluster Represents a more coarse-grained categorization of building blocks. Some clusters are developed more often or have experimental functions and have a higher failure rate. To this end, we incorporate the `functional_cluster` and calculate the failure rate of each `functional_cluster_fail_rate_10QBL` (explained later in this section) to determine whether a cluster and its constituent components have become unstable in the past 10 QBL test runs.

4.2 Numerical Features

total_developers This measures how many developers are currently involved in the QBL run. EDA revealed that as the number of developers increases, the complexity can result in more regressions. Including this feature gives the model a sense of organizational scale, telling the model when more collaboration may lead to instability.

total_extension_files_changed & total_non_extension_files_changed These features help to distinguish between file changes with known source extensions (e.g. .cpp, .py) and changes to other files without extensions (documentation, Make-files).
The existence of `non_extension_files_changed` may affect builds, but the effect on direct code failures is usually smaller.

if_component_changed A Boolean that indicates whether the test's associated component was changed during this QBL test cycle. EDA shows the majority of the failing scenarios do not have their component changed in the QBL.

.cpp, .py, .xml, .h, .hpp, .c, .ddf, .robot Over the course of three years' worth of data, eight extensions were implicated in the majority of file modifications at ASML. Understanding the type of extension helps the model distinguish if the changes are in test harness files (e.g. .robot) as opposed to core system code (e.g..cpp, .c).

age_days The number of days since the scenario was first introduced. This means that younger scenarios have a higher initial failure rate which can be attributed to the fact that the code has not been fully debugged or that there are new features that are not yet fully tested. The model thus distinguishes between a fully formed, thoroughly tested scenario and a recently introduced one that is likely to be riddled with undiscovered bugs.

weighted_age Refines `age_days` by also considering the frequency of test executions.

$$\text{weighted\_age} = \frac{\text{total\_runs}}{\text{age\_days} + \text{total\_runs}},$$

Here, `total_runs` refers to the total number of times a given test scenario has been executed across all testing runs.

It ensures that scenarios old but seldom run remain flagged as higher risk. Our processed data showed that simply being "old" is not necessarily correlated with stability; scenarios must also be exercised frequently to truly harden against failures.

fail_count_10QBL A simple count of how many times the current `scenario_key` has failed in the last 10 QBL test runs. EDA showed that historical performance in a rolling window is a good short term indicator of future fails. This way we do not use data leakage but still encode the notion of the recent memory.

Centrality_Component_x Tells how interconnected a component is from the system's dependency graph. If they break, components with high centrality can spread failures broadly. However, we believe that while the more central components are more aggressively monitored, they also fail more destructively when problems occur, thus increasing the probabilities of test-failure.

tqi (Technical Quality Index) A composite code-quality metric. We believe that components with low `tqi` contain more bugs, i.e., have technical debt. Including `tqi` in the model assists in pin pointing where poor code quality may lead to failed tests.

RunDate Captures the QBL's date. `RunDate` is usually a baseline for time comparisons, but it becomes important for cutting up historical windows (e.g., last 10 QBL test runs). `RunDate` is never used to look at run time outcomes directly, thus preventing leakage and ensuring purely time based grouping.

cycloxAverage The mean cyclomatic complexity of the functions in a component. High cyclomatic complexity can hide more corner cases and thus increases the likelihood of undiscovered bugs.

DuplicatedCode() Shows the percentage of the code that is similar. Similar code can lead to similar bugs and if the fixes are not done correctly then they can lead to similar errors.

eloc Effective lines of code (excluding comments and blank lines). A more extensive code base, however, increases the defect surface area. This feature is also employed in ratios like `eloc_per_fail` to demonstrate how failures occur rather 'densely' with respect to code size.

changerate Tracks how often a component's code is changing over time. Rapid iteration or refactoring indicates volatility. We believe that test scenarios targeting high churn components were more likely to fail, which could be due to bad merges or newly introduced regressions.

avgCodeCoverage A measure of overall test coverage (e.g., coverage of lines or branches) of the component. Lower coverage is a sign that many possible pathways are left untested, and hence, there are more potential bugs.

Max_DepCC_total_To The maximum dependency count of a component, or in other words the number of modules that depend on it. The effects can be far reaching; e.g., changes to heavily depended-on components. We saw that failures usually occur in these critical components as even modest defects have significant impacts downstream.

Max_Change_Freq_Nested_To Shows the frequency of changes in one module causing or containing changes in another. Highly nested or highly coupled modules have integrated complexity.

eloc_per_fail Ratio capturing how many lines of effective code exist per historical failure:

$$\texttt{eloc\_per\_fail} = \frac{\texttt{eloc}}{\texttt{fail\_count\_historical}}.$$

A small code base with many failures has a high "fail density," while large code areas with few failures appear more stable.

total_known_issues_last_time & if_last_time_known_issue Indicates whether or not a scenario (or however many times it) was identified as an 'known issue' in previous QBL runs. Recurring issues persist unless solved completely. These variables help the model identify potential risks by highlighting past issues.

if_flaky_count_last_QBL & if_flaky_tendency_last_QBL Indicates whether a scenario was considered flaky (inconsistent pass/fail) and how often it switched result in the previous QBL test run. It also prevents overemphasis on one off failures where a test is understood to be environment sensitive rather than indicating a real code defect.

polluter_count_last_QBL Displays the number of times the scenario was classified as a 'polluter' i.e. the test that if failed affects subsequent tests. EDA revealed that polluters are frequent offenders so including this metric is useful to help the model identify them as potential priority for retesting.

total_runs_all_data_last_QBL Historically up to the prior QBL, the cumulative number of times this scenario has been run. We want a product with hundreds of tests with little failure, but we don't want one that has been barely tested. This measure gives us a sense of how far each scenario has come.

4.3 Time-Series Rolling Features

pass_percentage_last15 & fail_percentage_last15 Calculated over the previous 15 QBL test runs, these percentages reflect medium-term behavior of a scenario:

$$\text{fail\_percentage\_last15} = \left(\frac{\text{Number of times the scenario failed in the last 15 QBL runs}}{\text{Number of times the scenario ran in the last 15 QBL runs}} \right) \times 100,$$

$$\text{pass\_percentage\_last15} = \left(\frac{\text{Number of times the scenario passed in the last 15 QBL runs}}{\text{Number of times the scenario ran in the last 15 QBL runs}} \right) \times 100$$

Crucially, they exclude the current QBL test run's results to avoid data leakage. By focusing on a rolling window, these features show whether a scenario's reliability has trended upward or downward over several weeks. It is necessary to add both of these features as a test result can be of multiple types (e.g. Pass, Fail, Untested, Unresolved, and Unsupported) [Christ et al., 2018].

Category-Level Rolling Fail Rates Each of these fail rates measures how often the respective category has failed in the last 10 QBL test runs:

$$\text{category\_fail\_rate\_10QBL} = \frac{\sum_{j=t-10}^{t-1} [\text{failure in category } c]}{\sum_{j=t-10}^{t-1} [\text{total runs in category } c]}$$

They reveal trends that may not be apparent on a per scenario basis. e.g. if `component_x` has started failing more often of late across a number of scenarios, the `component_x_fail_rate_10QBL` feature brings the model's attention to a potential issue with that component.

All of the features are deliberately chosen to represent a distinct factor that contributes to the instability of the test. These features in combination define the map of the code changes dynamics, component complexity, organizational effort, and historical performance can lead to scenario failing. To prevent data leakage, rolling-window features (for example, 10 or 15 QBL windows) are limited to *historical* data only, while keeping an accurate understanding of ongoing data drift. The large feature set of this model includes categorical labels, numeric indicators, and time-series rolling metrics to increase the model's effectiveness in identifying the highest risk test scenarios in every new QBL test cycle [Isabelle and André, 2003].

5 Addressing Class Imbalance and Feature Skewness

It is a challenge to achieve statistical significance in the context of an industrial scale software testing pipelines where we get many more instances of successful test executions than failed ones. As discussed in Section 3, only 3.3% of the scenarios that were recorded in the dataset are failed, and 96.7% are passed. This situation creates a typical *class imbalance* problem: There is a lot of data, but most of it is from the majority class (passing scenarios). A naïve strategy would be to build a model that would mostly predict *pass* as it minimizes the overall error; however, such model would be practically useless in identifying the real failures. In this section we discuss the strategies used to address class imbalance and feature skewness in the predictive modeling process and explains why some other strategies, such as SMOTE or under-sampling, were inapplicable to this particular field [Fernández et al., 2018].

5.1 Nature and Impact of Class Imbalance

In an environment that is continuously integrated such as ASML's, developers and validation engineers rely on predictive insights to recommend high risk test scenarios to be injected [Fowler and Foemmel, 2006]. But when only 3.3% of runs fail, a model trained on an unmodified dataset will likely predict all scenarios as likely to pass because it provides high accuracy. However, such a model fails to provide vital risk assessments: it does not capture the real value; where and when the next failure is most likely to occur.

From a business and engineering perspective, precise failure prediction helps teams focus their time and resources on the tests most likely to have regressions or new bugs. If we fail to identify these failures, we delay the discovery of the defect and can introduce the problem at a later stage or even in the production environment. Hence, addressing class imbalance is not a theoretical exercise but a necessity to ensure that the model is fit for practical industrial use.

5.2 Challenges of Oversampling and Under-sampling in a Time-Series Context

Oversampling using SMOTE. One commonly suggested remedy for class imbalance is SMOTE (Synthetic Minority Over-sampling Technique). SMOTE generates additional examples for the small class (failing scenarios) by drawing lines from the existing minority-class instances [Chawla et al., 2002]. While SMOTE and other oversampling techniques can help in some classification tasks, they have certain drawbacks in a dynamic, time series-based industrial setting:

- *Loss of Temporal Integrity.* In a software testing pipeline, each QBL represents a real chronological step. Introducing synthetic failures that never actually occurred disrupts the continuity of these records, making it difficult to interpret when and why failures happen.
- *Risk of Unrealistic Data Points.* As for the real software changes or the interactions of test configurations, code dependencies, and execution environments, it may not capture the entire scenario in synthetic scenarios. That could lead the model to be wrong about actual failure signatures.
- *Evolution of Software and Tests.* Because our dataset is of a constantly changing code base, the patterns learned from older QBL runs might be somewhat inaccurate. Since historic minority instances are fake data created, it might randomize synthetic data that represents instances that are not so typical anymore.

Undersampling the Majority Class. At the other end of the spectrum, undersampling the majority (passing) class means that the data size is also reduced by passing records being discarded randomly until the two classes are balanced. This approach is also inconsistent with the time-series and domain-driven nature of our data:

- *Potential Loss of Crucial Information.* Passing tests are not a 'black and white' issue. There are important signals they carry: about stable regions of the code, tested functionalities that did not fail under particular file changes, or slowly worsening scenarios. Discarding them might strip crucially temporal information on the software evolution process.
- *Distortion of Realistic Pass/Fail Ratios.* Maintaining the real-world ratio of test outcomes is crucial for ensuring that model's prediction aligns with actual scenarios. Avoiding overfitting the artificially balanced dataset is unrealistic for real (future) data, which might have much higher false positive rates.
- *Preserving Chronological Progression.* Randomly removing pass instances can break the coherence of QBL run, which can remove entire sequences that show gradual changes or shifts in the code over time.

Due to these issues, using SMOTE or under-sampling would compromise both the temporal integrity of the data and the accuracy of the model's predictions. Maintaining the precise order and frequency of pass/fail events is crucial to the model.

5.3 Rationale for Using Class Weights

To address the limitations of SMOTE or under-sampling, we use class weighting in our training process for our models. Class weights are a way of adjusting the loss function so that, for instance, misclassify minority-class (failing) instances are penalized much more heavily, effectively forcing the model to really weigh those rare but critical outcomes more heavily. Advantages:

- *Preservation of Data Integrity.* Unlike oversampling or under-sampling, class weighting retains the entire time series. Every passing and failing instance remains in the dataset with its full contextual detail, ensuring chronological continuity.
- *No Synthetic Instances Introduced.* The model trains strictly from real events (real pass/fail scenarios) so that domain consistency is maintained. This approach is particularly important when code changes are special and not often repeated exactly like in industrial projects.
- *Parameterization Emphasis.* We can tune the weight ratio to balance the predictive objectives. When tested on multiple class weight ratios (e.g. [1,10], [1,15] ,[1,20], [1:25], [1:30]) we found that a fail-to-pass weight ratio of 1:25 was both practical and effective. This means a single misclassified failing scenario is treated as severely as 25 misclassified passing ones.

Incorporating class weights into the model's cost function directly encodes the reality of the engineering problem: It is much more costly to fail to detect a real failing scenario than it is to falsely identify a passing one. From a business continuity perspective, "false positives" – incorrectly tagged high risk scenarios – are more easily managed than "false negatives" – undetected failing scenarios.

5.4 Accounting for Feature Skewness

Besides the problems of class imbalance, some numerical features are heavy tailed or skewed. `Fail_count_10QBL` is one such example where most of the tests have no failures in the last 10 QBL runs, but a few have very high failure counts. The same applies to the `changerate` feature where, while some components might have frequent changes, others may have none.

Log Transformations & Outlier Capping. To stabilize these skewed features, we employ limited transformations:

- **Logarithmic Scaling.** Using log(1+x) can decrease the influence of the big outliers and help the model to distinguish between the minor but significant variations in the components that have moderate changes.

- **Winsorizing.** To ensure that a handful of extraordinary data points do not skew the model's gradients, we cap or trim extreme values at the 99th or 99.5th percentile.

Both transformations are applied carefully to preserve model interpretability. For instance, if we have an extreme outlier in `fail_count_10QBL` then we might cap it at a quantile that is reasonably high to avoid the outlier affecting the model while still capturing the idea that a situation with many failures is more dangerous than one with no or very few failures.

Respecting Temporal Continuity. Another important consideration is that any transformation or outlier treatment should be consistent through time. We prevent the use of dynamic thresholds that could randomly shift between QBL cycles, which could otherwise bias the historical data.

5.5 Practical Implementation and Lessons Learned

The decision to use class weights rather than oversampling or under-sampling was informed by extensive domain feedback and pilot experiments:

- *Pilot Runs.* When SMOTE and random under-sampling were first used, they destroyed the temporal order and created pathological pass/fail rates. This is because predictive accuracy on real QBL data actually worsened, highlighting the significance of chronological accuracy.
- *Weight Tuning.* We tested different weight ratios (e.g. $[1:10], [1:25], [1:50] etc$) and checked them against a hold-out set of QBL runs to find which ratio gave the optimal sensitivity and specificity for the failing class. We finally settled on $[1:25]$ as a fair and stable selection.
- *Influence on Model Architecture.* As a result, for example, some model architectures, such as tree based ensembles, learn to handle imbalance more easily than linear or unregularized models. To make sure the model does not overfit the minority class, we had to optimize hyperparameters (e.g. learning rate, maximum tree depth).

Furthermore, the transformations for feature skewness were useful to even out extreme values in some of the numeric fields. Without these modifications, some features like `fail_count_10QBL` or `changerate` could have outliers that would dominate the training of the model and produce inaccurate predictions in more normal ranges. By applying slight capping or logarithmic scaling, we achieved more even gradients, leading to better generalization across typical value ranges [Miguel Carvalho, 2025].

It is crucial to handle heavily imbalanced data and skewed features for robust predictive modeling in industrial testing. Using actual pass/fail data without creating synthetic or discarding real examples maintains the structure and chronology of the QBL pipeline. Class weights are a simple yet effective way of telling the model to pay more attention to the minority class (failing scenarios) when calculating the loss. In parallel, small feature transformations ease the extreme values effect, which helps the model distinguish between relevant signals spread across the vast majority of 'normal' data points.

6 Model Development

Many high cardinality categorical inputs (e.g., `scenario_name` or `component_x`) and frequent changes in the software under test pose specific challenges in the setting of failure prediction in large industrial scales [Zhang and Chen, 2023]. Although techniques like Random Forest, XGBoost and LightGBM have shown effectiveness on tabular data, factors such as the native support for categorical features, robustness to transient signals, and computational complexity dictate which of these algorithms is best. Our thesis further compares different tree-based approaches. Rationale for choosing Catboost is too discussed in this section [Prokhorenkova et al., 2017].

6.1 Rationale for Using CatBoost

The table below highlights the strengths and drawbacks of commonly used tree-based classification methods, focusing on how they handle high-cardinality categorical data and evolving distributions:

Algorithm	Advantages	Drawbacks
Random Forest		
	• Straightforward to implement	• Requires external categorical encoding
	• Often robust against moderate overfitting	• Can be slower on large datasets
	• Good baseline for structured data	• May struggle with complex feature interactions
XGBoost		
	• Highly optimized; proven performance	• Manual or one-hot encoding for categoricals
	• Well-established in competitions	• Overfitting if not carefully regularized
	• Extensive hyperparameter support	• Can require intensive tuning
LightGBM		
	• Very fast (histogram-based splits)	• Binning of categories may lose detail
	• Memory-efficient	• Sensitive to some hyperparams
	• Built-in GPU support	• Overfitting possible if misconfigured
CatBoost		
	• Native handling of categoricals	• Slower unless GPU is used
	• Resilient to short-lived patterns	• Higher memory usage for wide data
	• Strong performance with minimal tuning	• Less common than XGBoost

Table 3: Comparison of Common Tree-Based Methods for Evolving, High-Cardinality Data

CatBoost is particularly compelling because it is one of the few algorithms that can directly handle categorical features (`cat_features`) without the need for encoding or binning and thus avoids the potential inflation of the feature dimension. Its 'ordered boosting' is another way of combating overfitting to noisy or short-lived trends, a crucial advantage in contexts where bugs that lead to failure one week might be fixed by the next week.

Key advantages in an Industrial testing context include:

Efficient Handling of High-Cardinality Categories. There are columns like *scenario_name* that have thousands of unique values. CatBoost encodes these internally without needing la-

borious transformations to preserve meaningful distinctions and reduce engineering overhead [CatBoost.ai, 2025].

Reduced Overfitting via Ordered Boosting. Transient bugs tend to induce ephemeral failure patterns in software tests. CatBoost prevents the model from latching onto fleeting correlations by permutation-driven approach that limits target leakage [CatBoost.ai, 2025].

Balanced Performance with Minimal Tuning. For example, other gradient boosting libraries can achieve similar results, but they generally require complicated hyperparameter tuning, especially for categorical feature encoding. CatBoost typically sets a good baseline that is difficult to improve upon without extensive fine-tuning, which is useful for large teams developing their software at speed [CatBoost.ai, 2025].

Support for GPU Acceleration. For iterative workflows, however, because of the high cost of training time for model with deep trees and many boosting rounds, CatBoost's parallel processing enables efficient handling of large datasets, particularly when using GPU acceleration (`task_type="GPU"`) [CatBoost.ai, 2025].

6.2 Overall Pipeline Overview

Figure 3 shows a high level view of how diverse data sources are ingested into our Predictive Test Advice System, and how it produces a CatBoost model that is ready to be deployed on future QBL runs. First, multiple repositories such as test result logs, integration data, interface metrics, polluter flags, and code quality measures are combined into one large dataset. The data collection phase was also rather complex and involved significant data cleaning and alignment to ensure that each QBL file is associated with the right changes, dependencies and historical impacts.

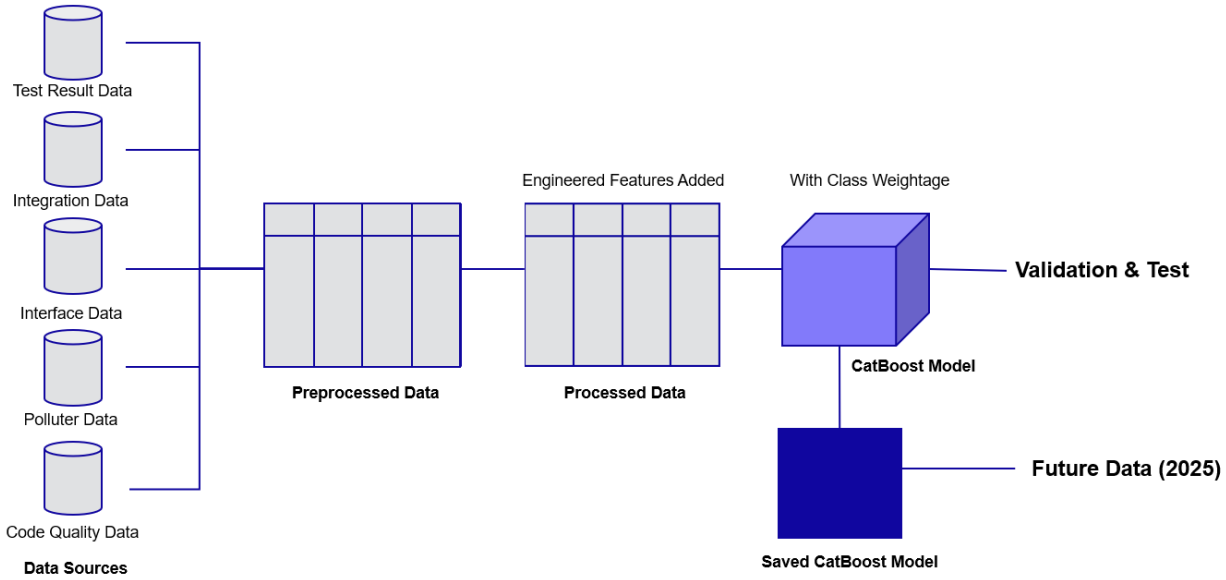

Figure 3: Data is merged and preprocessed from multiple sources and then fed to CatBoost for training, validation, and testing. Class weighting is incorporated *during training* to manage class imbalance without changing the base dataset. When validated, the final model is saved and used on new QBL runs (2025) in sync with ASML's practice of gradual integration.

After merging the data, we perform feature engineering to extract both static and time-based signals. This step includes constructing rolling window metrics (e.g., `fail_count_10QBL`,

`pass_percentage_last15`), domain-specific transformations (e.g., `eloc_per_fail`), and encoding various categorical features, which CatBoost handles natively. Class weighting is applied within CatBoost during model training, rather than altering the dataset itself. This means we are upgrading a weight for failing scenarios as part of the training configuration, and that is assisting the model to pay closer attention to these minority but crucial instances [Zheng and Casari, 2018].

After the feature rich dataset is made, it is divided chronologically into three sets; training, validation and test. The training subset is iterative and improves CatBoost's learning process, while the validation set differs from the training data to enable early stopping and hyperparameter tuning. We then test performance on a strictly newer QBL slice to gauge real world predictive power on final model selection. Once the fully trained and validated model is saved, it is deployed on upcoming QBL data (in 2025) to ensure that predictions are based solely on historical knowledge

As shown in Figure 3, this pipeline ensures that each stage–from data ingestion and feature engineering, to class weighted model training and to eventual deployment–aligns with the temporal nature of ASML's QBL cycles and the industrial necessity of accurate and timely defect prediction.

6.3 Training, Validation, and Testing Strategy

The training sets are based on the previous QBL runs, validation sets are taken from the central part of the subsequent QBLs, and test sets are obtained from the most recent QBL files. The approach is consistent with the actual setting where the models are expected to predict failures *before* they happen. This arrangement also enables the use of the most current data (which is likely to contain the most up-to-date component) for hyperparameter tuning.

- **Train on Older QBL Files:** Covered a wide historical range of passes and missteps, so the model could gain general trends.
- **Validate on Intermediate QBL Files:** Guarantees that alterations to hyper-parameters (e.g., *depth*, *iterations*, *learning_rate*) are set properly for fairly new, but not the latest data. These validation metrics — including *recall* to ensure failing scenarios are detected — help avoid overfitting while retaining generalization to the next wave of changes.
- **Final Test on the Newest QBL Files:** This is achieved through testing small changes to the code base, simulating how the classifier performs on genuinely unseen iterations of the code base, thus reflecting emergent bug fixes or expansions of test coverage.

Use of GPU Acceleration

Using the attribute specifying `task_type` as GPU, parallel computing resources are leveraged. The construction, generation and encoding of trees occur at a significantly higher rate, which is a significant advantage when retraining many times on datasets that have millions of entries. This is because of the necessity of having short feedback loops to continuously integrate new QBL data or tune features.

Methodology

1. **Data Aggregation:** The first, Individual QBL parquet files for each window of test executions are read and concatenated in chronological order. The columns are retained according to previous feature engineering steps, to make sure both numeric and categorical attributes are standardized.
2. **Partitioning by Time:** The consolidated dataset is divided into three parts: The older data is used to train, the next block is used for validation and the latest data is saved for final testing. This maintains the chronological order and prevents the future QBL information from leaking into the past.

3. **Model Configuration:** The key hyperparameters are chosen to avoid overfitting while capturing nuanced interactions without too many iterations, depth or learning rate. Processing of categorical features is enabled by CatBoost, for which a `cat_features` index is specified. Early stopping is controlled via `early_stopping_rounds`, which stops boosting if validation recall flattens.

4. **Training on Historical Data:** The model is trained on the training subset and every iteration is evaluated on the validation subset as we fit the model. It shows whether the chosen setup still identifies newly emerging failure patterns through this continual feedback loop.

5. **Validation and Early Stopping:** If validation recall does not improve for a certain number of consecutive iterations, training is stopped. This safeguard is to avoid overfitting to transient anomalies in the older QBL data.

6. **Final Testing:** After convergence of training, the real world performance is checked using the newest QBL portion. Recall, precision, F1-score and accuracy are the metrics that quantify the effectiveness of the classifier to generalize to scenarios that are not seen during training.

7. **GPU Acceleration:** The configuration `task_type=GPU`, which parallelizes computationally heavy tasks (for example, tree splitting, histogram generation) and reduces model-fitting time significantly for an environment that receives QBL data in a stream.

As shown in Figure 4, we divide the data based on time from 2022 up to the end of 2024 and the last 30 QBL runs we use for validation (15 QBLs) and test (15 QBLs). The rest is used for training. When the final model is trained and tested, it is then used on the new QBL data that comes in the year 2025, which means that the predictions for future runs are based only on historical information.

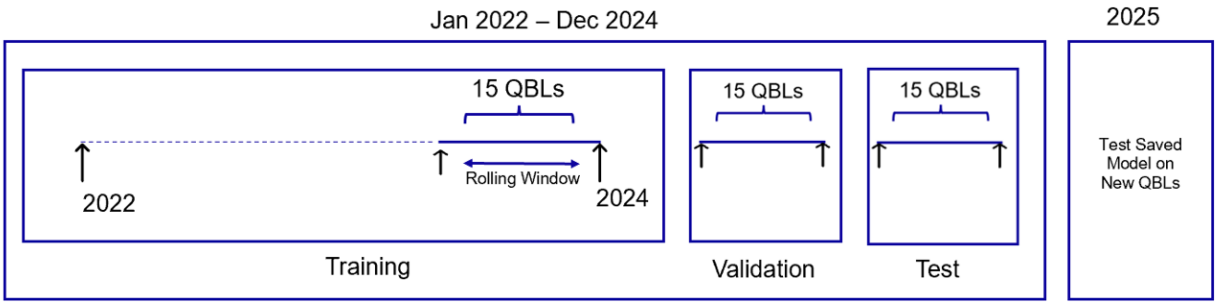

Figure 4: Timeline-based partitioning of QBL data from 2022 to 2024 into training, validation, and test sets, with the final saved model deployed on new QBL data in 2025. This forward-only scheme aligns with real-world continuous integration practices at ASML.

In addition, to building on time specific data splits, utilising CatBoost's strong performance with respect to categorical variables and exploiting GPU based training; this methodology is in line with the large scale, continuous integration pipelines needs. Thus, chronological integrity is maintained to ensure that the performance metrics actually reflect the true predictive value as far as upcoming QBL runs are concerned.

7 Empirical Results

In this section, we present the results of our predictive model. We have divided our results of our work into three main parts: (1) Hyperparameter optimization, (2) Model performance on validation and test sets, and (3) Feature importance analysis. Together, these results demonstrate the effectiveness of our approach in identifying failing test scenarios while considering the problems of class imbalance and the evolution of the software under test.

First, we conducted an extensive grid search to identify the best setting for our CatBoost model. The hyperparameters chosen are listed in Table 4. These settings—e.g., a large number of boosting iterations, moderate tree depth, and low learning rate—help the model learn complex, nonlinear relationships without overfitting. Furthermore, the use of class weighting (set to [1, 25]) means that rare failing scenarios are highlighted during training, and early stopping (with 200 rounds) stops the training process once the validation recall plateaus.

Next, we used validation and test sets which we derived from our QBL data which was chronologically split for the evaluation of the model. The performance metrics for these sets are presented in Tables 5 and 6 respectively. These results show that the model has a high recall of 0.820 on validation and 0.806 on test, which is important because missing a failing test scenario can have a significant impact on operations. While the test precision (0.107) is low, this is acceptable given the context in which this is a trade off between precision and recall, and detecting failures is the primary concern. The confusion matrix in Figure 5 confirms that the model is good at distinguishing between failing and passing scenarios.

Finally, we use CatBoost's internal evaluation to analyze feature importance. Figure 6 shows a horizontal bar plot of the top features. The results also show that features such as changes in files ending with .h, `eloc_per_fail`, and `fail_count_10QBL` are among the most influential. These findings support our feature engineering strategy and offer actionable insights for future refinement of test selection processes.

7.1 Hyper-parameter Optimization Results

We conducted a comprehensive grid search to determine the optimal configuration for our CatBoost model. The best hyper-parameters obtained are summarized in Table 4.

Table 4: Optimal Hyperparameter Configuration for CatBoost

Parameter	**Optimal Value**
Iterations	1000
Depth	7
Learning Rate	0.01
Loss Function	logloss
Class Weights	[1, 25]
L2 Leaf Regularization	0
Random Seed	42
Use Best Model	True
Early Stopping Rounds	200
Evaluation Metric	Recall
Random Subspace Method (RSM)	0.8

From our hyperparameter search, we identified several key settings that allow the model to capture complex, high-dimensional relationships while mitigating overfitting. We set the number of iterations to 1000 and a tree depth of 7. This allows the model sufficient capacity to learn

intricate patterns in the data, and the relatively low learning rate of 0.01 ensures that the model updates its weights gradually, fostering stable convergence even in the presence of noise and severe class imbalance. The use of early stopping with a patience of 200 iterations further guards against overfitting by stopping training when the recall metric on the validation set stops improving.

Additionally, we use the `logloss` loss function, which penalizes confident but incorrect predictions, making it well-suited for our binary classification task. For instance, to overcome the pronounced class imbalance where the failure cases are only about 3.3% of the data, we use class weight ratio of [1, 25]. This effectively forces the model to pay attention to the minority class by penalizing the model more for failing to identify a failing test during the training process. The RSM (Random Subspace Method) is also enabled (`RSM` = 0.8) to randomly sample 80% of the features in every iteration, preventing the model from over-relying on specific predictors in the training dataset.

Other parameters such as fixing the random seed at 42, to make sure that our experiments are reproducible, and setting `L2` leaf regularization to 0 since our robust early stopping mechanism proved sufficient. Finally, we use the best model as determined by the recall metric on the validation set, to guarantee that the final model is optimized for detecting failing scenarios, which is crucial in our industrial application.

All in all, the hyper-parameter settings identified in this study, as presented in Table 4 and discussed above, form a balanced configuration that is a foundation of the predictive power of our CatBoost model. They help us to accurately and reliably detect failing test scenarios, so that our system is working well for us in the challenging, data-intensive environment at ASML.

7.2 Performance Metrics

We trained the model on a different validation and test set taken from chronologically partitioned QBL data. The following tables present the key performance metrics.

Table 5: Validation Metrics

Metric	Value
Recall	0.820
Precision	0.339
F1 Score	0.480
Accuracy	0.964

Table 6: Test Metrics

Metric	Value
Recall	0.806
Precision	0.107
F1 Score	0.189
Accuracy	0.845

The high recall values on both the validation (0.820) and test (0.806) sets indicate that the model effectively identifies failing scenarios. The test precision is fairly low (0.107), but this is an acceptable trade off in our context as the cost of missing failures is greater than the inconvenience of some false alarms.

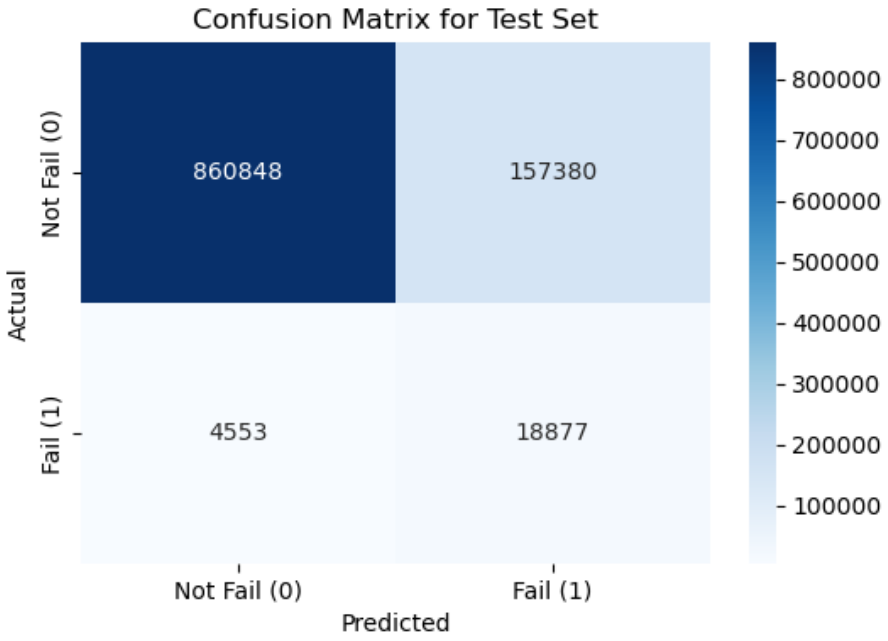

Figure 5: Confusion matrix for the test set, indicating the distribution of true positives, false positives, true negatives, and false negatives.

See figure 5 for the confusion matrix of the test set, offering a detailed breakdown of true positives, false positives, true negatives, and false negatives. The top-left cell (True Negatives) is 860,848 stable scenarios called *not fail* correctly. Out of that, 157,380 stable scenarios ended up in the top-right cell (False Positives), which means these were misidentified as failing. The bottom-left cell, False Negatives, contains 4,553 failing scenarios that the model failed to capture, while the bottom-right cell, True Positives, contains 18,877 failing scenarios that the model could correctly identify.

This breakdown makes clear that the model's high recall is gained by identifying the majority of failing scenarios correctly (True Positives) at the cost of labeling a moderate number of stable scenarios as failing (False Positives). This trade-off is justified in our industrial context, where missing failures poses a greater risk than investigating false alarms. However, this is also the cause of the lower precision seen in our test metrics since precision is directly proportional to the percentage of correctly identified failing scenarios among all the predicted failures. Overall, the confusion matrix captures how the model focuses on identifying the majority of failing tests — essential in ASML's high risk manufacturing environment — while acknowledging that future work may seek to decrease false positives without sacrificing recall.

7.3 Feature Importance Analysis

Internal evaluation was used to obtain feature importance by CatBoost. Although, the internal model generates a full table of feature scores, Figure 6 presents a horizontal bar plot of the most influential features.. It also has some high ranking features like file modifications in `.h` files, `eloc_per_fail`, and `fail_count_10QBL` indicating their importance in predicting failing test scenarios.

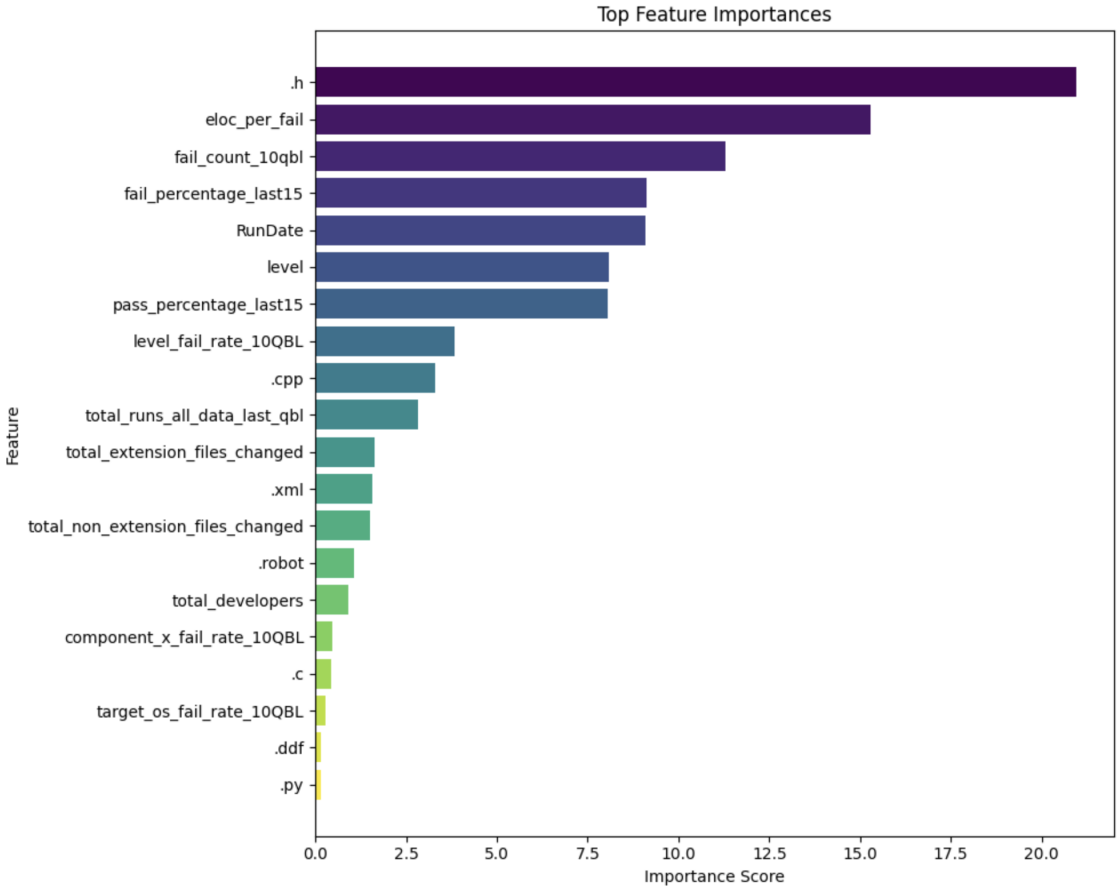

Figure 6: Horizontal bar plot of the top features by importance, highlighting the most influential variables in the predictive model.

Therefore, from our empirical results it can be concluded that the Predictive Test Advice System is effective and robust. The recall performance is also high (about 80%) and the feature importance analysis is insightful. Achieving a recall rate of 95% would be ideal but remains challenging in an imbalanced and dynamic environment such as ours. Thus, achieving approximately 80% recall represents a strong starting point and a step in the right direction.

It should also be highlighted that the validation and test sets are based on data from 15 QBLs (15 weeks), which introduces naturally variability into our performance metrics. Because of the weekly test outcomes variability and the complexity of the software under test, the current recall value indicates that our model identifies most failures, thus reducing the risk of undetected defects that could have a critical impact on ASML's operations.

In summary, further work in this area will continue to focus on further boosting recall and precision, while there is still room for improvement especially in enhancing precision the results achieved so far underscore the potential of our approach. While there is room for improvement—especially in precision, our results are encouraging and demonstrate the model's potential to enhance ASML's testing process in a high-stakes production environment.

8 Time-Aware Cross-Validation

Predictive modeling in our semiconductor manufacturing environment depends on identifying changes in software and testing conditions from multiple QBL runs. Simple random cross validation – where data is mixed arbitrarily – might lead to the so called 'future' code information being incorporated into the training data, overestimating the ability of the model to identify previously unseen defects. To this end, we employ an expanding window cross validation strategy that respects the chronological order of the QBL data. This section explains the rationale for using an expanding window, defines the folds of our dataset, presents the per-fold results and average metrics, and finally discusses how this approach is suitable for large-scale software systems that are continually evolving [Bergmeir and Benítez, 2012].

8.1 Expanding Window Folds

See figure 7 for how each fold is created in our expanding window setup. We first define an initial block of QBLs (e.g., 15 QBLs from early 2022) as the training set. In the first fold, this block is used to train the model and the next QBL is held out for validation. When the validation step is done, the newly validated QBL is then "folded" into the training set, thus increasing the length of the historical data. This process is then repeated incrementally such that every QBL is used once as a validation set.

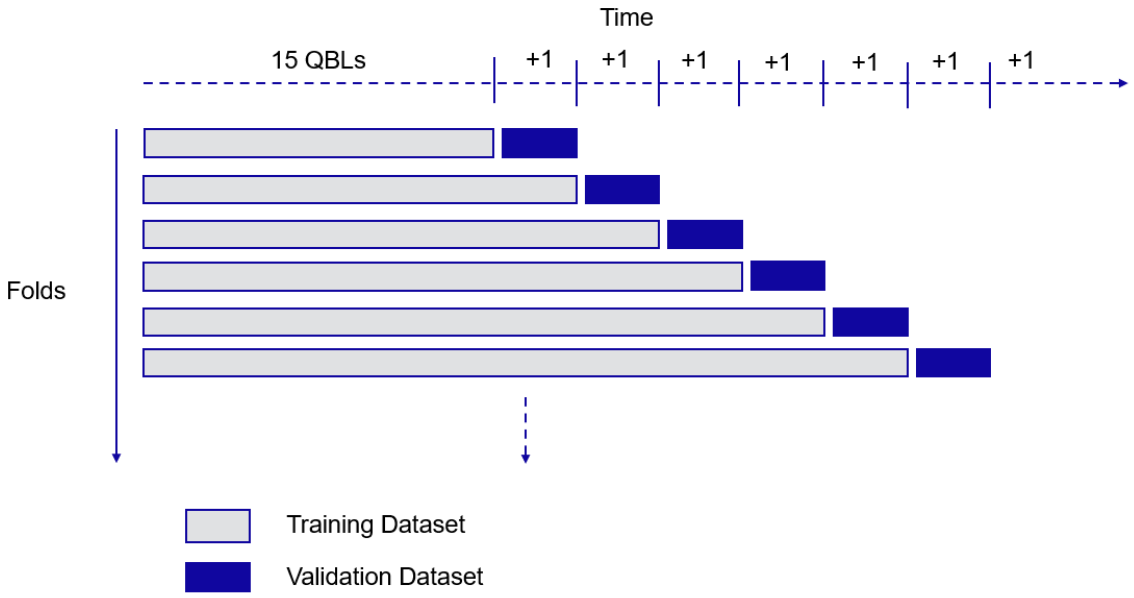

Figure 7: Schematic of expanding window cross-validation. Each fold adds a new QBL to the validation set while merging previously validated QBLs back into the training set, preserving temporal order and avoiding data leakage.

In this forward only manner, where no future data is made available to the model, we replicate real continuous integration (CI) workflows for testing. At ASML, new code merges and QBL runs arrive chronologically, and the model never "sees" future QBL data while training on the past, the true causal flow of development and testing events.

8.2 Implementation Details in Our Dataset

Initial Window. We use the first 15 QBL files (chosen to cover some months in 2022). This block includes the initial pass/fail patterns and helps to set a baseline for what constitutes a failing scenario for the model. Picking too few QBLs might prevent the model from ever seeing

the variety of failures that are typical in large industrial software, but using a large initial block could mean that subsequent folds would be relatively short.

Sequential Validation Folds. In each iteration, the next QBL files in the chronological order are used as the validation set. Recall, precision, F1-score and accuracy are reported as performance metrics. We then incorporate the newly validated QBL(s) to the 'training window' expanded to simulate the way of dealing with code changes and testing in production.

Cumulative Historical Knowledge. In addition, the training set is continually expanded by appending newly validated QBL data from early 2022 up to the final QBL runs that we keep for our ultimate test set. This enables the model to 'remember' historical defects that might recur due to reintroduced libraries or reverted code commits, e.g., older bugs or features. This also helps the model to catch the drift in the data where failing scenario patterns change frequently.

8.3 Per-Fold Validation Results

See figure 8 for a bar plot of validation *recall* across each fold, which shows how accurately the model detects failing scenarios at different points in the expanding window. A few QBLs have notably lower recall (suggesting code change or test coverage differences), but the overall pattern is stable. This suggests that the model is robust to incremental changes in the software environment, even when new QBLs are added to the validation set and then folded in to training.

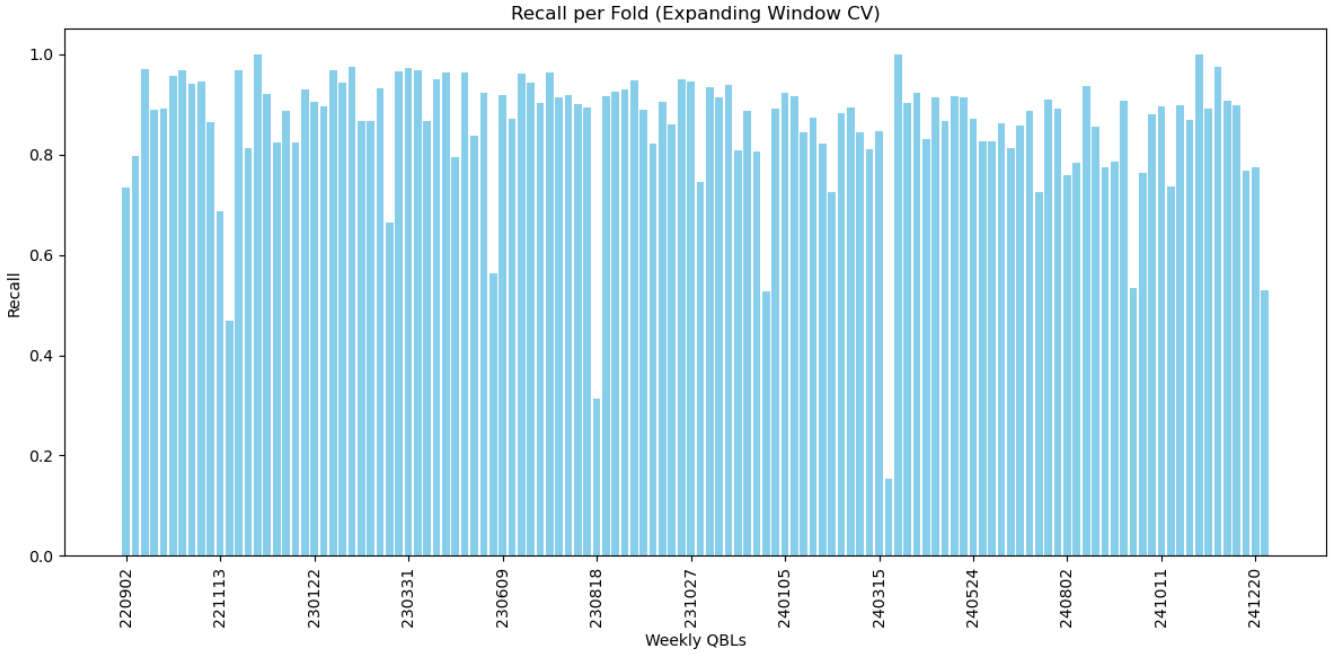

Figure 8: Bar plot of *recall* for each validation fold in the expanding window cross-validation.

From the validation folds, we summarize the results and present the overall performance in Table 7. The model has a high overall recall of 0.856, which is a key metric in industrial settings where missing a defective scenario can result in costly consequences. However, the precision is low at 0.260, which indicates that the model is likely to label some stable scenarios as fail. Some folds are not as robust, but the average of the overall average is still a good start for ASML's high-risk, highly dynamic continuous testing stream.

Table 7: Average Performance Metrics Across All Validation Folds

Metric	Value
Recall	0.856
Precision	0.260
F1-Score	0.380
Accuracy	0.868

The model reaches a high recall (0.856) which means it identifies most of the failing scenarios, an important goal in ASML's testing environment where missing a failure is costly. Although the precision is relatively low (0.260), resulting in more false positives, this trade-off is acceptable in this context. It is better to over-predict potential failures (and investigate further) than to risk missing critical issues. However, the precision could still be improved through further model tuning or post-processing, especially to reduce unnecessary test executions without compromising the recall.

8.4 Significance for Long-Lived Software Testing

Our methodology delivers practical advantages for enduring software testing environments. The following factors demonstrate its ability to sustain performance through time:

- **Ensuring No Forgotten Bugs:** ASML's code base can go back to older libraries or states. The model keeps a record of past defects by folding in continuously historical QBL data to the training set, so it can tell regressions that are similar to past failures.
- **Mimicking Real Processes:** All the QBLs are tested only after the results of previous QBLs are finalized. This growing scheme mirrors real-world development, where each QBL is validated before being incorporated into the training set, similar to how code merges and test data accumulate over time.
- **Capturing Evolving Failures:** Introducing new hardware integrations, environment changes or bug fixes can lead to the appearance of completely new failure modes. A forward-only cross validation scheme tells you whether the model is having trouble with newly introduced patterns, which calls for tweaking of feature engineering or hyper-parameters.
- **Time-Based Drift Assessment:** We can detect performance metric drift across folds to see if the model remains stable or declines, we can see this in figure 8. Such declines may be an indication of large code merges or environment modifications that the model has not yet learned how to handle. This enables us to detect drift in performance metrics across folds, whether the model stays stable or declines in recall or precision. Such declines may be an indication of large code merges or environment modifications that the model has not yet learned how to handle.

Therefore, expanding window cross-validation is a core part of our approach for sequentially indexed test data. To do this, we train the model on the past QBLs and validate on each new QBL before folding it back in, thus mimicking the real-life scenario in which the only available information is the data of the tests that have already been completed. Figures 7 and 8 highlight how the training region grows from 2022, so that early failing components, changes in the hardware or previous failures are included in the model's history when detecting the failure in the future.

9 Ablation Study

The Predictive Test Advice System underwent an ablation study to measure the effect of our designed features on its performance. The goal was to separate and measure the contribution of each feature subset to show that our advanced feature engineering including those from historical test results, integration data, interface metrics and time-aware signals significantly enhances the model's ability to predict failing test scenarios.

We analyzed three different configurations of the feature set:

1. **Categorical Features Only:** The configuration only includes the basic categorical attributes that are provided directly through ASML's APIs such as `scenario_name`, `configuration_name`, `level`, `machine_type`, `component_x`, `target_os`, `functional_cluster`, and `building_block`. These features describe the static properties of the test scenarios.

2. **Categorical + Code Quality Features:** The dataset includes both categorical features and code quality metrics (e.g., `tqi, cycloxAverage, DuplicatedCode(), and eloc`) that are also provided through APIs. These metrics give an idea about the maintainability and structural health of the software.

3. **All Features (Including Engineered):** This comprehensive set adds our custom-engineered features—such as rolling-window statistics (`fail_count_10QBL, pass_percentage_last15`), integration and interface metrics, known issue flags, and other time-aware signals—to the previous two groups. These features are designed to capture dynamic trends and historical patterns that static attributes cannot reveal. All feature definitions are provided in Section 3.

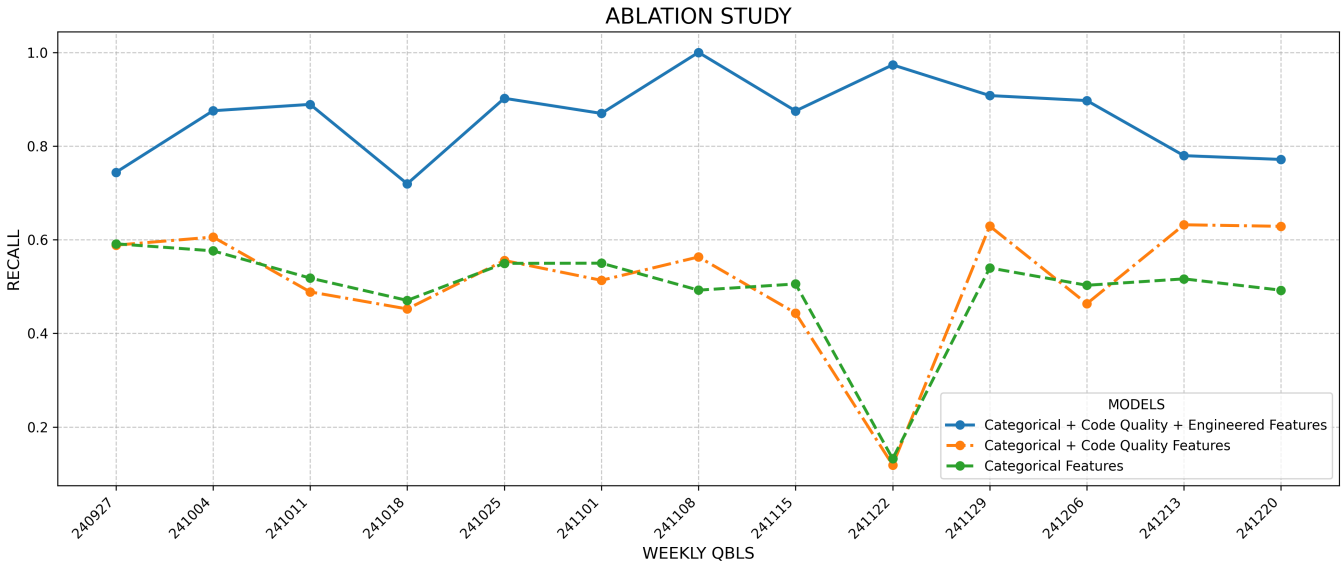

Figure 9: Comparing different models with different sets of features, showcasing the results achieved across all the test set QBLs.

Table 8: Ablation Study: Impact of Feature Subsets on Model Metrics

Feature Subset	**Average Recall**	**Average Precision**
Categorical Only	0.402	0.037
Categorical + Code Quality	0.450	0.060
All (Including Engineered)	0.806	0.107

The results reveal several key observations:

- The **Categorical Only** configuration, although useful as a baseline, only reaches an average recall of 0.402 and 0.037 precision. This shows that using only static categorical attributes is not enough to capture the complex dynamics of test failures.
- Adding **Code-Quality** features slightly improves recall to 0.450 and precision to 0.060, indicating a limited but noticeable effect This indicates that code quality metrics are somewhat useful, but the overall gain is rather small.
- The average recall increases to 0.806 when **All Features** (including our engineered features) are incorporated. This significant boost shows the effectiveness of our feature engineering efforts. Also precision improved to 0.107 too. Precision is still low but it varies on different data, table 5 showed precision of 0.339 on the validation set.

The engineered features originate from different data sources which include historical test results together with integration and interface information and time-dependent signals that detect temporary trends and developing patterns in software. The additional features deliver an enhanced understanding of testing conditions which allows the model to identify faint failure indicators that static data cannot detect [Kohavi and John, 1997]. The substantial increase in recall from 0.450 to 0.806 demonstrates how engineered features enhance the predictive system's overall performance.

The significant rise in recall from 0.450 to 0.806 demonstrates that the engineered features detect essential dynamic information which static categorical and code quality features fail to detect. The engineered features enable detection of time-dependent test failure patterns. These features excel at detecting new failure patterns that stem from recent code modifications and hardware updates and dependency changes which static attributes cannot detect. The analysis reveals that time-series rolling metrics such as `pass_percentage_last15` and `fail_percentage_last15` provide the most significant boost to recall improvement among all engineered features. The model shows robust performance across most QBLs yet there are specific challenging cases which produce lower detection rates due to atypical or transient failure patterns. Precision too requires more research to be improved.

Hence, the ablation study shows that while basic categorical and code quality features provide a baseline, the additional engineered features are necessary to achieve robust predictive performance. The complete feature set improves the model's ability to capture the dynamic and complex nature of test failures, which leads to a more effective and reliable Predictive Test Advice System.

10 Deployment on 2025 Data

After finalizing the model through expanding window cross-validation on QBL data up to late 2024, we now apply the saved model to new QBL runs from 2025. This step verifies how well the model generalizes to genuinely unseen test scenarios that may incorporate novel code merges, hardware updates, or unanticipated regressions [Li and Zhao, 2020].

Figure 10 shows the model's recall for each weekly QBL in the 2025 dataset, illustrating the extent to which failing scenarios are captured at different points in the new year. Although individual weeks fluctuate—likely reflecting variations in code changes and testing scope—the overall trend remains reasonably stable, underscoring the model's adaptability to ongoing development.

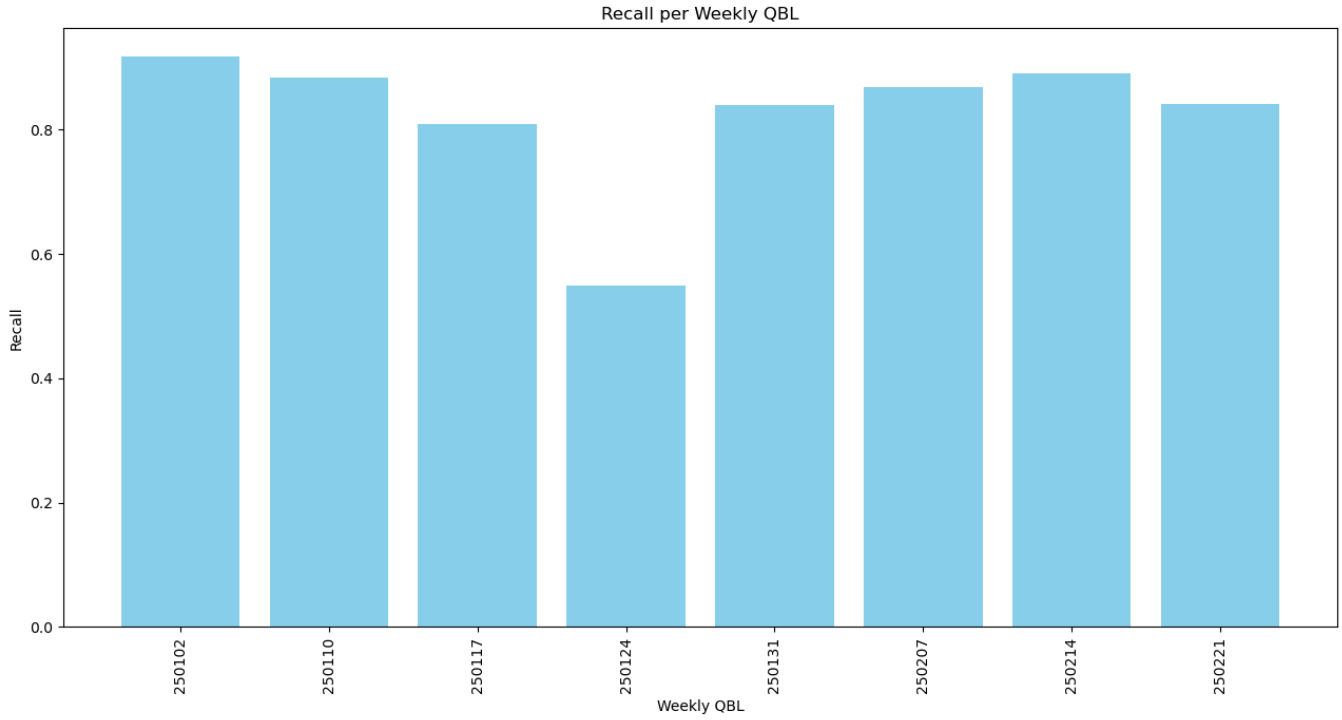

Figure 10: Recall per weekly QBL in 2025. Despite natural week-to-week variation, the model maintains a reasonably high recall.

To quantify the model's overall effectiveness on the 2025 data, we compute the average performance across all weekly QBLs in this timeframe. Table 9 summarizes the key metrics:

Table 9: Average Performance Metrics on 2025 QBL Data

Metric	Value
Recall	0.825
Precision	0.209
F1-Score	0.329
Accuracy	0.932

The average recall (approximately 0.825) reaffirms the model's strength is in identifying failing scenarios, which is a priority in ASML's production environment. The F1-score (0.329) and accuracy (0.932) further illustrate how the model balances the competing objectives of comprehensive

defect detection and minimizing false alarms.

In this context, low precision and F1-scores are acceptable because the primary objective is to maximize recall—ensuring that nearly all failing test scenarios are detected. In high-stakes environments like ASML, missing a failing test (a false negative) can have critical operational consequences, so the trade-off of having some additional false positives (resulting in lower precision) is justified.

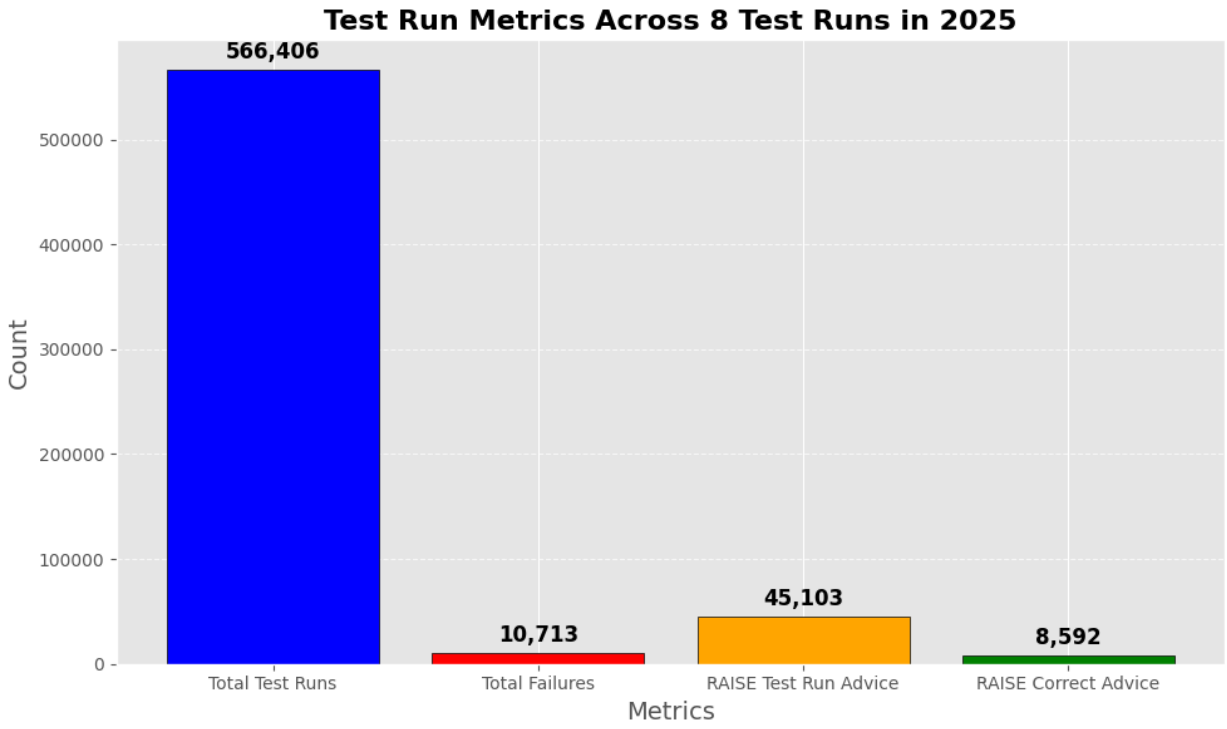

Figure 11: Comparison between the total test runs across 2025 data and the advice provided by our model.

Table 10: Models performance across 2025 QBLs

Metric	**Value**
Test Set Reduction	92%
Failure catching capability	80.2%

In Figure 11 and Table 10, we can see that the total test runs across 8 QBL test runs were 566,406 and the advice generated by our model is 45,103 tests. This results in a reduction of 92% from ASML's test suite. In terms of failures, 10,713 tests failed across the 8 QBL test runs in 2025 and our Predictive test selection system was able to catch 8,592 failure. This means our model was able to catch 80.2% of the total failures. This result is consistent with our results obtained on our expanding validation and test set.

11 Future Work

While the Predictive Test Advice System demonstrates promising results in identifying failing scenarios at ASML, several avenues remain open for further enhancement:

Refining Interdependency Modeling. Although our present feature set includes basic dependency metrics (e.g. `Max_DepCC_total_To`, `Max_Change_Freq_Nested_To`), future work could focus more on the structure of software modules. For example, we believe creating more detailed dependency graphs could help to understand how small changes in one component spread across complex interlinked subsystems.

Dynamic File Change Analysis. Our analysis does include file level metrics (.cpp, .h etc.) but more specific characteristics might be available from a comparison of the particular lines or functions altered over time. Further, using automated diff based parsing could help in understanding the kind of changes being made and may help in identifying some patterns which are strongly correlated with failing scenarios.

Advanced Drift Detection. For instance, the model now updates itself to new code merges using the expanding window cross-validation strategy. However, there is no need for explicit drift detection techniques to notify the engineers when historical patterns are no longer relevant. Making it possible to detect and highlight critical change, e.g., a sharp decrease in recall for new QBLs would force a more urgent need for retraining or feature re-engineering so that the model stays current with the constantly changing software.

Continuous Integration and Automated Retraining. Although our pipeline has a mechanism to retrain and validate on newly arrived QBL test run data, there is further automation possible. Scheduling frequent, incremental updates to the model—especially when development is intense—would help to maintain high recall on new features and bug fixes.

Extended Hyper-parameter Tuning. As for now, our grid search has yielded effective hyperparameters for CatBoost, but other tuning approaches (say, Bayesian optimization) might have been even better. Getting a more solid search strategy integrated could actually help reveal subtler parameter interactions as well, especially when the dataset grows or new features are added.

Exploring Additional Machine Learning Architectures. Although CatBoost was good at working with categorical data and high cardinality features, there are other advanced methods that may pose some additional benefits. Transformers excel at capturing long-range dependencies and contextual relationships in sequential data. While they are traditionally used in NLP, in predictive testing they could be employed to model sequences of code changes, test outcomes, and time-series trends. This could enable the system to understand subtle patterns over long periods—for instance. We believe examining such approaches could help to uncover some other ways to improve recall or precision.

12 Conclusion

In this thesis, a comprehensive *Predictive Test Advice System* has been designed for ASML's high-stakes, continuously evolving software development, testing, and integration. Central to this effort was the integration of multiple data sources — from historical execution logs, to detailed file change records, to code quality metrics — into a unified, time-aware pipeline. By leveraging CatBoost's native handling of categorical features and applying class weighting to address the pronounced imbalance between passing and failing test scenarios, our model prioritizes those test scenarios which are most likely to fail. This approach not only streamlines the allocation of testing resources but also protects against the possibility of missing critical defects in a domain where even minor oversights can have substantial operational consequences.

A key strength of the system is its time aware cross validation, which conserves the temporal coherence of the QBL data. This strategy ensures that the model is trained only on the historical runs before validating on the newer QBLs, just as it does in real-world continuous integration. Subsequent deployment on 2025 data and multiple validation folds yields high recall (typically greater than 80%) from the model in detecting the majority of failing scenarios. Precision is lower, but this is a reasonable trade off with respect to the industrial priority of avoiding false negatives (undetected defects). Thus, the system captures the majority of the failing scenarios early, providing a good safety net to avoid the cost of many defects appearing in production environments.

The system shows promising results but multiple challenges and limitations persist. The severe class imbalance in QBL test run data remains difficult to handle because passing test executions dominate the data. This results in low precision rates as stable scenarios are being misclassified as failures. The system requires additional development of feature engineering methods and advanced hyperparameter tuning and alternative machine learning architectures to maintain precision levels without compromising the achieved recall rates. The solution of these problems is crucial for enhancing the reliability and practical use of the predictive test selection system in industrial continuous integration environments.

Nevertheless, a fully optimized predictive test selection system is not yet complete. This could be achieved through a deeper dependency analysis of the data to understand the subtle interactions among software components and how small changes to one module can cause failures. Further, adaptability could be improved through more advanced drift detection mechanisms that can identify abrupt changes in code. Strategies for continuous retraining or incremental learning are needed to ensure that the model stays current with new QBLs as it learns to handle emergent bugs, features, and hardware integrations in near real time. At the same time, examining line-level code diffs or other more sophisticated graph-based techniques for modeling inter-module dependencies may reveal predictive signals that are currently unexploited by the feature set.

While CatBoost has proven highly effective for large-scale tabular data with many categorical features, other algorithms (e.g., transformer-based or graph neural networks) might capture the nuances of evolving code bases even more accurately. Finally, bridging these predictive insights with ASML's existing development workflows—such as automatically suggesting test subsets in a continuous integration pipeline—would further reduce unnecessary test executions, speed up feedback loops, and ensure that engineering efforts focus on genuinely high-risk areas.

In sum, the *Predictive Test Advice System* developed in this thesis marks a significant step toward more intelligent and resource-efficient testing at ASML. By identifying failing scenarios early, it helps avert production risks, saves engineering time, and maintains the high standards of quality essential in semiconductor manufacturing. Yet, as software complexity continues to grow, so too must our predictive strategies evolve. Continued research and iterative improvements promise to elevate recall, precision, and overall robustness, contributing to ASML's continuous efforts in delivering reliable lithography solutions.

References

[Bergmeir and Benítez, 2012] Bergmeir, C. and Benítez, J. M. (2012). On the use of cross-validation for time series predictor evaluation. *Information Sciences, Volume 191.*

[CatBoost.ai, 2025] CatBoost.ai (2025). Catboost is a high-performance open source library for gradient boosting on decision trees. *CatBoost.ai.*

[Chawla et al., 2002] Chawla, N., Bowyer, K., Hall, L., and Kegelmeyer, P. (2002). Smote: Synthetic minority over-sampling technique. *Journal Of Artificial Intelligence Research, Volume 16.*

[Christ et al., 2018] Christ, M., Braun, N., Neuffer, J., and Kempa, W. (2018). Time series feature extraction on the basis of scalable hypothesis tests (tsfresh – a python package). *Neurocomputing Volume 307.*

[Doe and Smith, 2023] Doe, J. and Smith, J. (2023). A systematic review of machine learning for test case prioritization. *Systematic Review.*

[Facebook Eng. Team, 2019] Facebook Eng. Team (2019). Scaling predictive test selection at facebook. *Facebook Engineering Team.*

[Fernández et al., 2018] Fernández, A., García, S., Galar, M., Prati, R. C., Krawczyk, B., and Herrera, F. (2018). Learning from imbalanced data sets. *Textbook, Springer Nature.*

[Fowler and Foemmel, 2006] Fowler, M. and Foemmel, J. (2006). Continuous integration. *Continuous Integration-martinFowlwer.com.*

[G Dong, 2018] G Dong, H. L. (2018). Feature engineering for machine learning and data analytics. *Book, CRC Press.*

[Google AI Team, 2016] Google AI Team (2016). Taming google-scale continuous testing. *Google Research Team.*

[Isabelle and André, 2003] Isabelle and André (2003). An introduction to variable and feature selection. *Journal of Machine Learning Research 3 (2003).*

[Johnson and Miller, 2022] Johnson, E. and Miller, R. (2022). A survey of machine learning-based software testing techniques. *International Journal of AI.*

[Kohavi and John, 1997] Kohavi, R. and John, G. H. (1997). Wrappers for feature subset selection. *Artificial Intelligence, Volume 97.*

[Kuhn and Johnson, 2013] Kuhn, M. and Johnson, K. (2013). Applied predictive modeling. *Applied Predictive Modeling textbook, Springer.*

[Li and Zhao, 2020] Li, F. and Zhao, H. (2020). Assessing model generalization in continuous integration systems. *Model Generalization.*

[Miguel Carvalho, 2025] Miguel Carvalho, A. J. P. . S. B. (2025). Resampling approaches to handle class imbalance: a review from a data perspective. *Journal of Big Data volume 12, Article number: 71.*

[Mohamed A. Shibl, 2021] Mohamed A. Shibl, I. M. A. H. . S. A. M. (2021). System integration for large-scale software projects: Models, approaches, and challenges. *Proceedings of International Conference on Emerging Technologies and Intelligent Systems. ICETIS.*

[Mozilla Hacks Team, 2021] Mozilla Hacks Team (2021). Testing firefox more efficiently with machine learning. *Mozilla Hacks.*

[Prokhorenkova et al., 2017] Prokhorenkova, L., Gusev, G., Vorobev, A., Dorogush, A., and Gulin, A. (2017). Catboost: Unbiased boosting with categorical features. *Advances in Neural Information Processing Systems 31 (NeurIPS 2018).*

[Rongqi Pan, 2021] Rongqi Pan, Mojtaba Bagherzadeh, T. A. G. . L. B. (2021). Test case selection and prioritization using machine learning: a systematic literature review. *Empirical Software Engineering, Volume 27, article number 29.*

[Rothermel et al., 2001] Rothermel, G., Untch, R., Chu, C., and Harrold, M. (2001). Prioritizing test cases for regression testing. *IEEE-Prioritizing test cases for regression testing.*

[Sedighe et al., 2024] Sedighe, Amirhossein, Mehdi, Mostafa, and Abbas (2024). A systematic review of machine learning methods in software testing. *Applied Soft Computing Journal.*

[Williams and Kevin, 2024] Williams, Michael, L. and Kevin (2024). Test case prioritization techniques and metrics. *Williams, Michael, Lee and Kevin.*

[Zhang and Chen, 2023] Zhang, W. and Chen, L. (2023). Challenges in machine learning-based software testing. *IEEE-ML.*

[Zheng and Casari, 2018] Zheng, A. and Casari, A. (2018). Feature engineering for machine learning: Principles and techniques for data scientists. *Feature Engineering for Machine Learnings, O'Reilly Media.*

[Zhu, 2017] Zhu, M. S. M. A. B. L. (2017). Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access, vol. 5.*