CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

A Comprehensive Tool for Breaking Change Detection in C++ Development

Master's thesis in Computer science and engineering

Zelei Weng

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

A Comprehensive Tool for Breaking Change Detection in C++ Development

Zelei Weng

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

A Comprehensive Tool for Breaking Change Detection in C++ Development
Zelei Weng

Supervisor: Natasha Bianca Mangan
Advisor: María José Mera, King
Examiner: Staffan Björk

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2024

Contents

List of Figures

List of Tables

1
Introduction

This chapter presents the introduction of the thesis, including the purpose, aims, context, limitations, and stakeholders.

1.1 Purpose and Aims

In contemporary software development, a breaking change refers to a modification that disrupts previously established contracts, resulting in compilation errors and behavioral changes [1]. Software libraries are commonly utilized to enhance development productivity and facilitate functionality reuse. Ideally, updates to application programming interfaces (APIs) should maintain backward compatibility. However, in practical scenarios, application updates often introduce breaking changes, leading to improper code execution. Thus, it is crucial to accurately update documentation each time a breaking change occurs.
In our company[2], developers in different groups are responsible for various parts of the products. If a breaking change occurs in a library developed by one group, developers in other groups must be informed to modify their code and prevent crashes. Manual determination of breaking changes by developers is prone to errors and inefficiencies. To mitigate mistakes and enhance workflow efficiency, a dedicated tool will be developed as part of this master's thesis. The tool aims to detect changes between two code versions based on established guidelines and highlight any breaking changes found.

1.2 Context

During the review of related research, it was found that breaking changes are prevalent in modern software development and pose significant challenges to developers. Numerous studies report that breaking changes negatively impact developer productivity and workflow, often requiring substantial effort to identify and resolve. These findings motivate the need for improved mechanisms to detect and communicate breaking changes. In particular, they indicate that the development of an automated breaking change detection tool is both feasible and necessary to address these challenges effectively.
According to a comprehensive research[3], approximately 28% of API changes in software libraries lead to backward compatibility issues, with an average of 14.78% per library. The research spans five years, revealing a 20% increase in the frequency of breaking changes for the studied libraries, ranging from 29.02% to 49.14%. What's

more, the research also mentioned that libraries with higher breaking change frequencies are larger in size, more popular, and more active.
Another study by S. Raemakers, A. van Deursen, and J. Visser[4] focuses on the releases in Maven Central [5], a repository storing all required dependencies or artefacts, revealing that approximately one-third of all releases introduce breaking changes. The study advocates for adherence to semantic versioning principles to help users estimate modification times and understand changes more efficiently. However, it also notes that certain libraries may not adhere to semantic versioning in a proper way, posing a challenge for those developers who want to use those libraries. Given these findings, automated generation of documentation could alleviate the burden associated with writing and reading such documentation.
A separate study [6] identified the main reasons developers introduce breaking changes, including adding new features, API simplification, maintainability, and bug fixing, among others. The data above reveals that breaking changes are unavoidable during software development. According to the survey conducted in this research, developers tend to document detected breaking changes in various documents: 22% of the developers plan to use release notes, another 22% opt for Changelog, while JavaDoc accounts for 17%. The other choices are Websites, Migration Guides and Examples, each of which represents 11%, with README being the least chosen at 6%. This lack of standardisation regarding where to document breaking changes may confuse developers seeking to use them. Furthermore, an analysis of 110 Stack Overflow posts emphasises the practical significance of breaking changes. Notably, 45% of the questions were from clients seeking guidance on how to address specific breaking changes, highlighting the importance of addressing these issues during software development. Clearly, breaking changes consume a significant amount of developers' time.
In summary, breaking changes are common and inevitable during software development. The lack of unified standards for documenting breaking changes highlights the importance of detecting and clarifying them. This is crucial to alleviate the burden on developers responsible for documenting breaking changes and to provide clarity to developers who need to use these libraries. The necessity for a dedicated breaking change tool is evident in improving code efficiency and reducing mistakes. While existing tools primarily cater to Java libraries [7], with some detection tools available for Python packages [8] or Node.JS libraries [9], there is a notable scarcity of detection tools for C++. Although the mentioned tools are not specifically designed for C++, valuable insights and approaches can be drawn from them. Furthermore, considering the specific codebase of the USDK team at King [2] is written in C++, it is imperative to develop a tailored breaking change detection tool that accounts for the unique aspects of their codebase.

1.2.1 Research Question

Based on the situation discussed above, breaking changes are indeed unavoidable during software development. This raises the question of what can be done to address them. One might first ask broadly: how significant is the problem? Could the answer lie in better documentation standards? Or perhaps in a tool that helps

developers communicate changes more clearly? These are valid angles, but they ultimately circle back to a more fundamental concern: before anything else can be communicated or documented, the breaking change must first be *detected*.
This narrowing of focus leads to the question of automated detection specifically for C++. Existing tools predominantly target Java, Python, or JavaScript, leaving a notable gap for C++ codebases. Given the King team's C++ codebase and the practical need to flag breaking changes before they reach downstream developers, the central research question becomes:

How can breaking changes in C++ APIs be automatically detected?

To answer this fully, a supporting question is also considered: what are the implications of breaking changes in C++ in the first place? Understanding what kinds of changes break code, and why, is necessary context for designing a detection approach. This implication question therefore, serves the main one — it informs what the tool needs to detect and why certain categories of changes matter more than others.

1.3 Limitation

While developing a breaking change detection tool is viable, there are inherent limitations to consider. The sheer multitude of elements that can cause breaking changes makes it impossible to identify every instance, which means it would be hard to ensure the completeness of our tool. However, we can prioritize ensuring the accuracy of our tool. Given the extensive range of potential causes for breaking changes, it is unrealistic to expect a tool capable of detecting 100% of them. Nonetheless, the tool should accurately classify the breaking changes it identifies as such with 100% certainty.

1.4 Stakeholder

This study involves collaboration with King[2], a video game developer and publisher specializing in casual games. The research topic was proposed by the USDK team, who are responsible for the general SDK for the game at King. Their duties include maintaining and updating the library, incorporating new functionalities, and upgrading code to new C++ standards.
Within the company, client-side developers are not required to delve into the intricacies of API internals; instead, they focus on developing the client and understanding how to integrate APIs seamlessly. Their primary concern is identifying breaking changes in new API releases, as this determines necessary modifications to their code.
Currently, in the USDK team at King, developers manually assess whether a given update introduces breaking changes. A dedicated tool will be developed for this master's thesis to enhance efficiency and reduce errors.

It's important to note that no private data from King will be used or shared. However, the researcher retains the right to their code.

2
Phase I: Preparation

The initial phase of this project, referred to as the preparation phase, establishes the foundation for the development of the breaking change detection tool. The primary objective of this phase is to gain a comprehensive understanding of the project requirements, clarify the scope of changes to be detected, and gather relevant knowledge from existing literature and technical resources. This ensures that subsequent development is guided by well-defined goals and informed by previous research.
To achieve these objectives, the preparation phase is organized into several key activities. First, a demand analysis is conducted to clarify the specific needs of the internal team at King, including the types of changes that require detection and the priorities for the tool's functionality. Next, a literature review is performed to explore existing studies and tools for breaking change detection, identifying their methodologies, strengths, limitations, and applicability to C++ projects.
Following this, the chapter provides a detailed background on breaking changes, their definitions, impacts on software development, and the limitations of current detection approaches. Subsequently, the theory section examines the technical methods used in previous research, such as static analysis, refactoring detection tools, API-focused analysis, and dynamic testing approaches, which inform the design of the proposed method.
Finally, the chapter outlines the approach, planning, and a review of the existing codebase, laying the groundwork for the development phase.
Overall, this preparation phase ensures that the project is grounded in a clear understanding of requirements, informed by existing research, and technically feasible within the context of the C++ codebase.

2.1 Background

2.1.1 What is Breaking Change

A breaking change refers to an alteration in one part of a software system that has the potential to causes other components to fail. For example, changing the name of a function, as illustrated in Figure 2.1, is considered a breaking change because the compiler can no longer locate the function based on its previous name, resulting in compilation errors. Similarly, modifying the parameter type of a function , Figure 2.2, could also constitute a breaking change because the function signature has changed. However, breaking changes are often more intricate than these illustrative examples. Breaking changes in APIs can lead to compilation or runtime issues in

client code. For instance, the removal of functions in shared objects for C/C++ could trigger runtime linking errors.

Figure 2.1: Example for breaking change: Change the name of the function

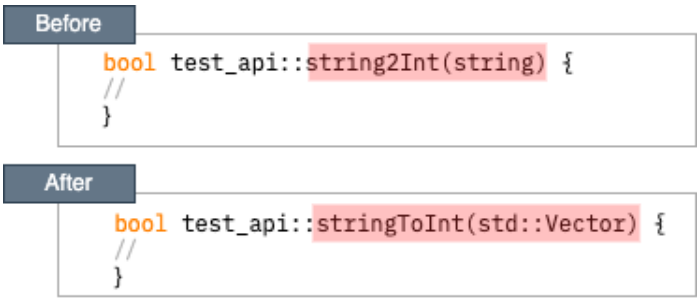

Figure 2.2: Example for breaking change: Change the paramater of the function

In software development, libraries are commonly used to enhance productivity and facilitate the reuse of functionality in client applications [10]. These functionalities are provided by Application Programming Interfaces (APIs). Ideally, APIs should maintain backward compatibility during updates, ensuring that code utilizing the APIs remains compatible with the updated library. However, the occurrence of breaking changes extends beyond mere alterations in function names, encompassing a variety of complex factors.
According to research conducted by Romain Robbes [11], they analyzed 577 deprecated methods and 186 classes across more than 2600 distinct systems involving over 3000 contributors. Their findings revealed that a considerable number of API changes resulting from deprecation can have a significant impact on the ecosystem. Additionally, they discovered that a small proportion of API changes may go unnoticed for an extended period, remaining undiscovered for more than three months, thereby posing a long-lasting hidden danger. Another study [12] conducted within the Pharo software ecosystem, comprising approximately 2600 distinct systems, corroborated these findings. It suggested that API alterations can have a substantial impact on client systems, methods, and developers, often taking time for client developers to identify these changes.
In practical software development, it is common for updates in libraries to result in breaking changes for client applications. According to research by Brito [1], 64% of API elements are deprecated per system. Xavier's group [3] conducted a study

involving 317 real-world Java libraries, comprising 9K releases and 260K clients. They found that libraries frequently introduce backward compatibility issues, with 27.99% of API changes breaking backward compatibility and an average of 14.78% of changes per library being breaking changes. Moreover, the frequency of breaking changes tends to increase over time, with a 20% rise observed between the first and fifth years of the studied libraries (from 29.02% to 49.14%). These findings underscore the dynamic nature of software libraries and highlight the importance of understanding breaking changes for client developers. Additionally, it is reasonable for developers to introduce breaking changes in certain situations. According to a survey conducted by Brito [6], the top four reasons for breaking APIs include implementing new features, simplifying and reducing API complexity, improving maintainability, and fixing bugs in the code. All of these events are inevitable and unavoidable in software development.

To mitigate the consequences and enhance the efficiency of software development, it is crucial to acknowledge breaking changes when libraries are updated. Brito's survey [6] reveals that developers tend to document detected breaking changes in various ways: 22% use release notes, another 22% opt for Changelog, while JavaDoc accounts for 17%. Other choices include websites, migration guides, and examples, each representing 11%, with README being the least chosen at 6%. Additionally, studies by Romain Robbes [11] have revealed that deprecation messages are not always present or clear, further complicating the issue. Moreover, Hora's research [12] highlights that knowledge of API changes may be concentrated among a limited number of developers, and API changes and deprecations may exhibit distinct characteristics, posing challenges for client developers in resolving issues. The lack of standardization regarding where to document breaking changes may confuse client developers seeking to use them, underscoring the need for clearer documentations for breaking changes.

While those inevitable breaking changes remain a significant challenge in software development, consuming substantial developer time, there are still opportunities to detect them automatically. Romain Robbes [11] suggests that replacements can be identified by analyzing the solutions implemented for specific methods within the ecosystem. Similarly, research by Hora [12] indicates that most API modifications can be integrated into static analysis tools, reducing adaptation time and increasing awareness of new or improved APIs among projects. Additionally, Brito [1] proposes the development of a recommendation tool to assist client developers by automatically inferring missing replacement messages, further streamlining the adaptation process. These insights highlight the feasibility and benefits of automated detection and resolution of breaking changes in software development.

2.1.2 Limitations of Current Approaches

Based on the research in the previous section, it is clear that breaking changes have a big impact on software development. It is vital to note it while a lot of libraries lack the documentation about breaking changes, it is important to address this main point, considerable efforts have been made in both research and practice to mitigate this problem. The following paragraph outlines the main approaches and actions

that have been taken to address the challenges posed by breaking changes.
RefDiff[13] is an automated method designed to identify refactorings between two code revisions within a Git repository. It utilizes a combination of heuristics based on static analysis and code similarity to detect 13 well-known refactoring types. Through an evaluation involving 448 known refactoring operations across seven Java projects, RefDiff achieved a precision of 100% and a recall of 88%.
APIDIFF[7] is a tool developed based on RefDiff[13]. It identifies both API breaking and non-breaking changes between two versions of a Java library using a combination of similarity heuristics and static analysis of Java source code. APIDIFF can detect the removal and addition of API elements, 13 well-known refactoring operations, and alterations in visibility, static and final modifiers, exceptions, field default values, superclasses, and deprecated APIs.
Type regression testing[9] can automatically assesses whether a library update affects its public interface types based on its usage by other npm packages. Experimental results on 12 widely used libraries demonstrate high accuracy in detecting type-related breaking changes, with at least 90% of updates correctly classified as major, minor, or patch, and 26 breaking changes identified among minor and patch updates.
AexPy[8] introduces an API-model-based systematic approach to address the challenge of breaking changes in Python packages. This method utilizes a Python-specific API model to categorize API modifications into various breaking levels. AexPy achieves an 86.9% recall rate on a dataset of 61 known breaking changes and identifies 405 high and medium breaking changes with a precision of 93.5% across the latest versions of 45 packages, uncovering 114 previously undocumented alterations. Additionally, AexPy reports 63 undocumented breaking changes to active package developers, with 31 confirmed.
More recently, Diagnose[14] proposes a dynamic approach to breaking change detection in JavaScript libraries. Unlike purely static or type-based methods, Diagnose constructs **dynamic object relation graphs** by exploring API usage and performing forced execution-based analysis. By comparing the graphs of different library versions, it identifies breaking changes more accurately, detecting a larger proportion of changes while producing fewer false positives. This approach highlights the potential of dynamic, usage-based analysis for languages with highly flexible and runtime-dependent APIs.
The tools mentioned above are all efficient and useful breaking change detectors. While RefDiff[13] is primarily designed for Java, APIDIFF[7] is specifically developed for Java libraries, type regression testing[9] targets Node.js libraries, AexPy[8] is tailored for Python packages, and Diagnose**diagnose2025** focuses on JavaScript. This underscores the notable scarcity of detection tools for C++. Considering the specific codebase of the USDK team at King, which is written in C++, it becomes imperative to develop a tailored breaking change detection tool that addresses the unique aspects of their codebase.

2.2 Demand Analysis

Since this tool is developed for internal use by the King group, it is essential to clarify the requirements before development. The topic of breaking change detection

is extensive. To address this, internal documentation was reviewed to refine the project's focus. According to these documents, the primary concern is whether the actual API has changed; if the APIs remain unchanged, there is no need to notify library users, as no modifications to their code would be required.

Furthermore, the internal documentation provides an initial classification of the types of changes, which are categorized as follows:

- **Fixes** that does no changes to the API.
- **Features** that introduce a new API.
- **Breaking changes** that modify the existing API.

Although there are three categories for the changes, only breaking changes have to be detected according to the requirements. This classification helps streamline the development process by clearly defining the scope of changes that need to be communicated to the users.

2.3 Literature Review

During this stage, relevant papers discussing the concept of breaking changes in software development were reviewed, allowing key insights and methods to be summarized.

An overall understanding of breaking changes was developed, including how and why they commonly occur during software development. A breaking change typically arises when a modification to one part of the code causes failures in other components of a software system. Common examples include alterations to function signatures, such as changes to function names or parameter types. Furthermore, it is acknowledged that the introduction of breaking changes is unavoidable during the software development lifecycle, particularly when implementing new features, simplifying or reducing API complexity, improving maintainability, or fixing bugs.

Subsequently, methods for detecting breaking changes, as described in the reviewed papers, were summarized. These include tools like RefDiff[13], APIDIFF[7], type regression testing[9], Refactoring Miner[15] and AexPy[8]. Although none of these detection tools are developed for C++ code, they provided significant insights into the detection process. For instance, RefDiff[13] helped us understand thirteen well-known refactoring types2.1, which serve as crucial references for developing this detection tools. Type regression testing[9] offers a novel approach which detects breaking changes based on its usage by other npm packages. The Refactoring Refactoring Miner [15] utilizes an abstract syntax tree (AST), a concept that is also incorporated in this project. Furthermore, differences between these existing methods and the proposed approach were observed. While most of the tools rely on static analysis, the method developed in this project diverges in its analytical techniques, potentially providing a novel contribution to the field of breaking change detection in software development.

2.4 Theory

This section delves deeply into the intricacies of breaking changes, exploring theoretical techniques utilized in other research and extracting insights that can inform this project.

2.4.1 Static analysis

Static analysis refers to the process of analyzing software code without executing it.[16] It involves examining the code's structure, syntax, and semantics to identify potential errors, security vulnerabilities, performance issues, or adherence to coding standards. Static analysis tools inspect source code or compiled binaries to detect issues such as syntax errors, unused variables, memory leaks, code smells, and compliance violations. This analysis is performed before the program is run, allowing developers to identify and address issues early in the development lifecycle.

2.4.1.1 Refactoring Miner

Refactoring Miner[15] is a prominent approach used to detect breaking changes in source code based on the static analysis. It is introduced by Tsantalis [17] and then extended by Silva[18].
Refactoring Miner support generate AST diffs at the commit level, support multi-mappings, match AST nodes of different types, and provide semantic diff capabilities in a fully refactoring-aware manner. Static analysis is the main method it used to detect breaking changes. It starts by matching code entities in a top-down manner, seeking exact matches in names and signatures between two code revisions. Elements that have been removed or added between the two models are first matched based on the equality of their names, and then by the similarity of their members at the signature level. Subsequently, a specific set of rules is applied to identify various types of breaking changes. Through this process, Refactoring Miner can identify 14 high-level refactoring types, such as Rename Package/Class/Method, Move Class/Method/Field, Pull Up Method/Field, Push Down Method/Field, Extract Method, Inline Method, and Extract Superclass/Interface.

2.4.1.2 RefDiff

RefDiff [13] functions as a refactoring tool grounded in both static analysis and code similarity. It receives two versions of a system as input and produces a list of identified refactorings as output.
The detection algorithm consists of two phases: Source Code Analysis and Relationship Analysis. In the Source Code Analysis phase, the source code before and after changes undergoes parsing and analysis, resulting in the creation of two models, denoted as E_b and E_a. Each model represents a set of types, methods, and fields present in the source code. Specifically, the model before changes is represented as $E_b = (T_b \cup M_b \cup F_b)$, while the model after changes is represented as $E_a = (T_a \cup M_a \cup F_a)$. To enhance efficiency, only code entities from modified source

files (added, removed, or edited) are analyzed, thereby forming sets of types, methods, and fields in each model.
In the second phase, referred to as Relationship Analysis, the algorithm identifies relationships between source code entities before (E_b) and after (E_a) the code changes. This is achieved by constructing a bipartite graph composed of entities from the two versions, where edges represent relationships R between corresponding entities. For example, a method in the previous source code model may correspond to a method in the updated model with a different name, indicating a Rename Method refactoring. Through this process, RefDiff is capable of detecting 13 well-known refactoring types.

2.4.1.3 APIDIFF

APIDIFF [7] is a dedicated tool designed for recognizing API alterations, both breaking and non-breaking, across two versions of a Java library. Its overarching structure encompasses three pivotal modules: processing, refactoring, and analysis, depicted in 2.3.

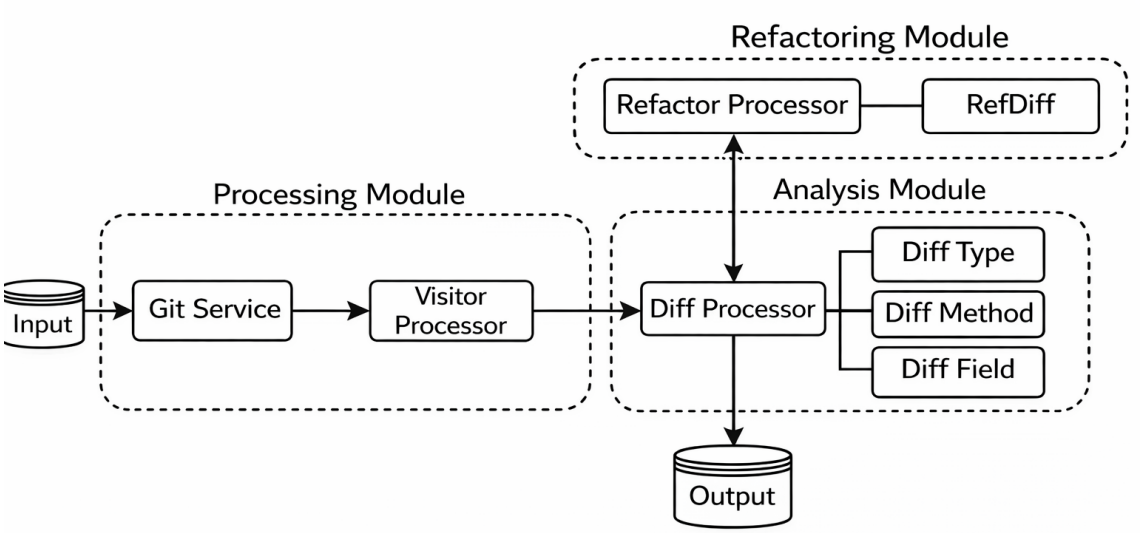

Figure 2.3: APIDiff Architecture

Processing Module handles the library's metadata and creates a commit history graph using Git. It utilizes the Visitor Processor to generate library instances for analysis.
The refactoring module is a reuse of REFDIFF [13], detects 13 common refactoring actions in library instances, including type, method, and field changes like moves and renames.

Subsequently, the Analysis Module identifies changes in the library's history and generates a comprehensive list of modifications to types, methods, and fields, including commit metadata and detailed information about the changes.

2.4.2 Type Regression Testing

Mezzetti introduce Type Regression Testing[9] to detect whether an update to a library implementation impacts the types of its public interface, considering its utilization by other npm packages. Leveraging the test suites of clients, this method employs dynamic analysis to construct models of the library interface. Comparing these models before and after an update enhances existing tests, exposing alterations that could impact clients.
There are two phases in the workflow of Type Regression Testing: Public API Discovery and Type Regression Detection.
In the Public API Discovery stage, all the packages with tests that depend on the old library version are retrieved from npm. Then, a public API model of both the old and the new library version is built using an instrumented interpreter that runs the tests of all the collected dependents. The models π use the notion of dynamic access paths to give types to the individual components of the library's API.
During the second phase, a second model $\pi'(p)$ is built by doing the same process on the new library version, then models π and π' are compared to identify type-related breaking changes.

2.4.3 Catalog of Changes

Each of the tools and methods mentioned above serves a distinct purpose in detecting specific categories of breaking changes. RefDiff [13], for instance, excels in identifying 13 well-known refactoring types, providing valuable insights into code evolution. Based on RefDiff, APIDIFF [7] offers a comprehensive analysis by detecting both breaking and non-breaking changes across types, methods, and fields. The compiled data on breaking and non-breaking changes, organized in the table below, serves as a valuable reference for this research.

2.4.4 Summary and Implications for This Project

This chapter reviewed several theoretical approaches and tools for detecting breaking changes, including static analysis techniques, refactoring detection tools such as Refactoring Miner and RefDiff, API-focused tools such as APIDIFF, and dynamic approaches such as Type Regression Testing. Together, these studies demonstrate that breaking changes can be effectively identified using both static and dynamic analysis, depending on the target language, ecosystem, and type of changes under investigation.
Static analysis–based tools, such as Refactoring Miner, RefDiff, and APIDIFF, offer the advantage of early detection without requiring program execution, making them suitable for integration into development pipelines. However, these tools are predominantly designed for Java and rely heavily on abstract syntax tree diffing

Element	Breaking Changes
Type	REMOVE TYPE, LOST VISIBILITY, CHANGE IN SUPERTYPE, ADD FINAL MODIFIER, REMOVE STATIC MODIFIER, RENAME TYPE, MOVE TYPE
Method	REMOVE METHOD, LOST VISIBILITY, CHANGE IN RETURN TYPE, CHANGE IN PARAMETER LIST, CHANGE IN EXCEPTION LIST, ADD FINAL MODIFIER, REMOVE STATIC MODIFIER, MOVE METHOD, RENAME METHOD, PUSH DOWN METHOD, INLINE METHOD
Field	REMOVE FIELD, LOST VISIBILITY, CHANGE IN FIELD TYPE, CHANGE IN FIELD DEFAULT VALUE, ADD FINAL MODIFIER, MOVE FIELD, PUSH DOWN FIELD

Table 2.1: Breaking Changes Detected by APIDIFF

Element	Breaking Changes
Type	ADD TYPE, GAIN VISIBILITY, REMOVE FINAL MODIFIER, ADD STATIC MODIFIER, ADD SUPERTYPE, EXTRACT SUPERTYPE, DEPRECIATE TYPE
Method	ADD METHOD, PULL UP METHOD, GAIN VISIBILITY, REMOVE FINAL MODIFIER, ADD STATIC MODIFIER, DEPRECIATE METHOD, EXTRACT METHOD
Field	ADD FIELD, PULL UP FIELD, GAIN VISIBILITY, REMOVE FINAL MODIFIER, DEPRECIATE FIELD

Table 2.2: Non-Breaking Changes Detected by APIDIFF

and refactoring pattern recognition, which limits their direct applicability to C++ projects. Additionally, refactoring-based approaches may overlook certain breaking changes that do not correspond to known refactoring patterns.
Dynamic approaches, such as Type Regression Testing, provide strong guarantees by observing real client usage, but they depend on the availability of client test suites and runtime environments. This makes them less practical for C++ libraries, where dependency management, build configurations, and test availability vary significantly.
Did-it-break builds upon insights from static analysis techniques while addressing the specific challenges of C++ library evolution. By generating code that simulates usage of public APIs through header files and leveraging compiler diagnostics, Did-it-break avoids reliance on language-specific refactoring patterns or client test suites. While this heuristic-based strategy may not detect all possible breaking changes, it prioritizes accuracy and practicality within the constraints of C++ development. The reviewed theories thus provide both motivation and justification for Did-it-break's design, highlighting the trade-offs between completeness, accuracy, and applicability.

2.5 Approach

This section presents the methodology adopted to design, implement, and evaluate the proposed breaking change detection tool, Did-it-break. The section first introduces the overall method and system architecture, followed by a description of the development process, tools, and technologies employed. Finally, the evaluation strategy used to assess the tool's functionality, usability, and extensibility is outlined.

2.5.1 Method

Did-it-break identifies breaking changes between two versions of code based on their header files. It employs a heuristic based on static analysis to detect breaking changes by generating a code file that simulates the use of all methods defined in the header files. 2.4 illustrates the high-level architecture of Did-it-break, which comprises the Generate Module and the Analysis Module.
In the Generate Module, the header file, before the change, is taken as input and transformed into an abstract syntax tree (AST). The module then traverses the AST to generate code that calls all the methods defined in the header file. Since this file is generated from the header file, it should result in no compile errors. However, the file cannot be compiled successfully if there are any breaking changes introduced during the change. This method helps determine whether breaking changes exist between the old and new versions.
If any compile errors occur during the compilation of the generated file with the header file after the change, it indicates the presence of detected breaking changes. The Analysis Module processes the compiler output, applies additional formatting to improve readability, which serve as the final output of the system.

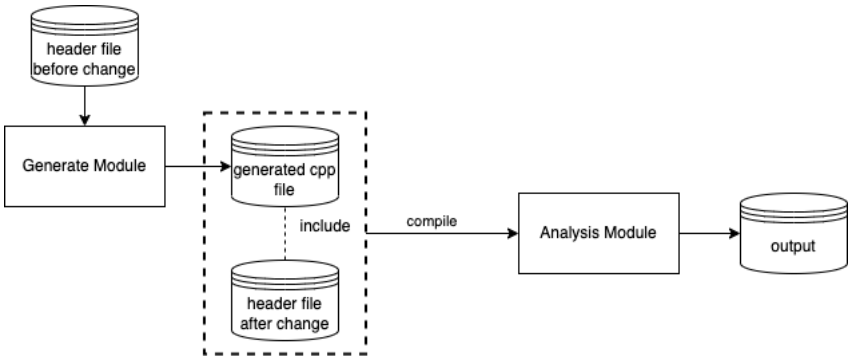

Figure 2.4: Architecture of Did-it-break

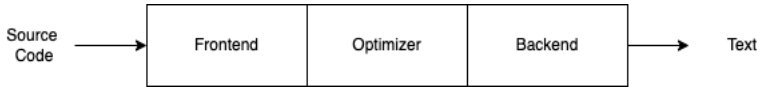

Figure 2.5: Three-stage compiler structure

2.5.2 Development

In this stage, the project embraces an Agile approach [19], which emphasizes iterative and adaptive development practices. Agile methodologies promote collaboration, flexibility, and responsiveness to change throughout the development process. By adopting Agile principles, customer satisfaction is prioritized, there is a welcome for changing requirements, and the aim is to deliver working software incrementally, ensuring continuous improvement and adaptability in development efforts.

2.5.2.1 Libclang

C++ is chosen as the main language for its emphasis on performance. Additionally, integration of Python to streamline user input parsing, enhancing the overall usability of the tool. The Generate Module as figured in 2.4, considered the core of the project, will utilize Libclang [20].
Clang is a compiler front end for the C language family built on top of LLVM. In a traditional three-stage compiler architecture, the front end is responsible for the analysis phase. During this phase, the source code is parsed according to its grammatical structure, syntactic and semantic errors are checked, and the code is transformed into an Abstract Syntax Tree (AST).
The AST serves as an intermediate representation that is later processed by the back end, which performs code generation (synthesis) to produce the target program. An optional optimization stage may be introduced between the front end and the back end to improve performance, as illustrated in Figure 2.5.
Libclang provides a C interface and offers functionalities like parsing source code into an Abstract Syntax Tree (AST), loading ASTs, traversing the AST, and associating source locations with elements within the AST. With these interfaces, parsing code into an AST becomes straightforward, and accessing node information such as variable types or parent nodes becomes convenient. With libclang, traversing header files using the cursor is simplified, especially with functions like *CXChildVisit_Continue*

or *clang_getCursorSemanticParent*. Additionally, functions like *clang_getCString* facilitate obtaining variable and function names' spellings, which streamlines code generation processes.

2.5.2.2 Docker

Furthermore, given that modules used in this project adhere to the latest C++ 20 standard, it's important to note that not all compilers currently support the complete C++ 20 standard. Therefore, the project will leverage Docker [21] to ensure a stable environment for executing the code.
Docker is a software platform designed to expedite the process of building, testing, and deploying applications. It achieves this by packaging software into standardized units known as containers, which encapsulate all the necessary components for the software to operate seamlessly, including libraries, system tools, code, and runtime environment. By utilizing Docker, applications can be deployed and scaled rapidly across diverse environments, with the assurance that the code will run consistently.

2.5.3 Evaluation

The evaluation of the developed tool hinges on two key features:

2.5.3.1 Functionality

This criterion assesses the tool's capability to accurately detect breaking changes. Smoke tests[22] and unit tests[23] will be conducted in this stage. These tests will scrutinize the tool's performance, ensuring that it functions correctly and efficiently. quantitative research will be conducted in this phase to statistic the accuracy and efficiency improvement of the tool. Quantitative Research[24] will be conducted to measure the accuracy and efficiency improvements achieved by the tool.

2.5.3.2 Ease of further development

The ease of further development is a critical aspect evaluated through considerations of code readability, structure, and modification ease. A well-structured and readable codebase ensures that subsequent developers can readily comprehend the tool's architecture and functions. Google C++ Style Guide[25] should be followed during the development.

2.6 Planning

2.6.1 Agile

After investigation, the Agile [19] software development framework was selected for the development stage. While Agile methodology typically involves six iterative stages, it has been slightly customized to better suit the project's specific requirements, as illustrated in Figure 2.6. Since the primary task involves integrating

new functionalities into the existing code framework, the adapted Agile framework consists of four stages: Plan & Design, Development, Test, and Review.

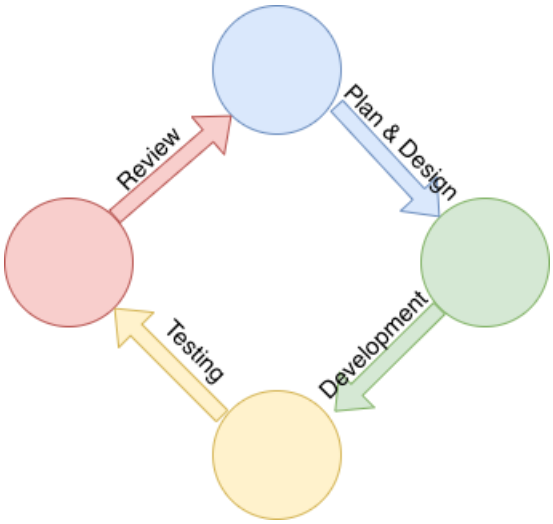

Figure 2.6: Four steps Agile framework

In the **Plan&Design** stage, the project goes through the existing codebase to figure out how to integrate the new function. The architecture should be considered carefully so that the implementation of the new function will not disrupt the existing ones.
In the **Development** stage, the project proceeds to implement the new function as planned. This involves writing new code and refactoring old parts if necessary.
In the **Test** stage, the project will run the test files to check whether the new functions deployed in the last stage work properly. This includes generating the correct code, and testing whether this generated code can detect the breaking changes properly under the test script.
The **Review** stage is critical for reflection and optimization. During this phase, the project reflects on the errors or issues that were encountered during the last stage and considers potential improvements to the solution. It also gathers tips that could accelerate future iterations, enhancing the efficiency and effectiveness of the development cycle.

2.6.2 Time Plan

This part outlines the planning aspects of the master's thesis, including the project timeline and ethical considerations. It presents a structured time plan for the research, development, and reporting phases, ensuring that the project progresses in a systematic and manageable manner. In addition, the chapter discusses the ethical principles guiding the study, with particular attention to data privacy, informed consent, and transparent documentation.
Following is the original project timeline which provided a structured plan for the research, implementation, and evaluation phases. However, during execution it became evident that the plan could not be followed strictly. Several factors contributed

to this deviation, including the complexity of configuring the development environment, the time required to investigate and resolve technical challenges, extended debugging efforts, and personal circumstances. Collectively, these factors required substantially more time than originally estimated.
Figure 2.7 is the initial Gantt chart plan for this study:

Figure 2.7: Gantt chart for this study

In January, Demand Analysis and Literature Review were conducted to elucidate the necessary functions of the tool, and relevant theories and references were collected for development. The code base was reviewed to identify reusable components. Moving forward, the introduction section of the report should be outlined, with the aim of finalizing it by mid-February, and the development stage has commenced.
The next phase in development primarily involves the span from the middle of February to the middle of May, with a subsequent reflection period. Progress documentation will be maintained to ensure the cohesion of the overall process.
As May and June approach, the emphasis will shift towards reporting the results and completing the report. Concurrently, preparations for the final presentation will be underway. This timeline ensures a systematic and well-structured approach to the master thesis, from initial research and development to final documentation and presentation.
[26]

2.6.2.1 Data Privacy

Any data collected or processed during the research must comply with pertinent privacy regulations to safeguard individuals' privacy and data protection rights. It is crucial to emphasize that company code as well as the data will be treated as confidential and will not be disclosed publicly to preserve proprietary information and intellectual property rights.

2.6.2.2 Informed Consent

When conducting research involving human participants, especially in qualitative studies that may entail interviews or interactions, obtaining informed consent is paramount. Informed consent ensures that participants are fully aware of the research's purpose, objectives, and their role in it. They should be provided with comprehensive information about the study, including its scope, potential risks and benefits, confidentiality measures, and how their data will be collected, stored, and utilized. Participants should have the opportunity to ask questions and seek clarification before voluntarily agreeing to participate. Obtaining informed consent demonstrates respect for individuals' autonomy and ensures ethical research practices are upheld throughout the study.

2.6.2.3 Transparent Documentation

Transparent and clear documentation is essential throughout the entire research process to ensure replicability. Comprehensive documentation encompasses various aspects of the research methodology, tools utilized, and ethical considerations addressed. It enables readers to understand the technological infrastructure supporting the study and assess the validity and reliability of the results obtained through these tools.

2.7 Existing Code Review

Given the presence of an existing framework and codebase for this project, it is significant to understand both the current code and the methods employed to detect breaking changes. This understanding forms the foundation for further development.

2.7.1 Code Structure

Figure 2.8 shows the overall structure of the code.

```
did-it-break/
├── cmake/
│   ├── FindClang.cmake
│   └── FindLLVM.cmake
├── pugixml
├── source/
│   ├── ApisManager.cpp
│   ├── ApisManager.ixx
│   ├── ArgParse.ixx
│   ├── ClangTranslationUnit.cpp
│   ├── ClangTranslationUnit.ixx
│   ├── CppApiUsageGenerator.cpp
│   ├── CppApiUsageGenerator.ixx
│   ├── IGenerator.ixx
│   ├── XmlGenerator.cpp
│   ├── XmlGenerator.ixx
│   ├── clangutils.cpp
│   ├── clangutils.ixx
│   ├── dib.ixx
│   ├── format.h
│   ├── main.cpp
│   └── stdcompat.ixx
├── tests/
│   ├── AddPublicDataMemberShouldNotBreak.patch
│   ├── ChangeMemberParameterShouldBreak.patch
│   ├── ChangeMemberParameterToWiderShouldNotBreak.patch
│   ├── IBaseline.h
│   └── RenamePublicDataMemberShouldBreak.patch
├── CMakeLists.txt
├── Dockerfile
├── Docker-entrypoint
├── conanfile.txt
├── README.md
├── dib.py
└── docker-entrypoint
```

Figure 2.8: code structure of the project

2.7.1.1 cmake

This folder contains two CMake files, FindClang.cmake and FindLLVM.cmake, which are essential for configuring the compiler to recognize the appropriate versions and paths to Clang and LLVM. These CMake files play a vital role in ensuring that the build environment is correctly set up, thereby minimizing errors during configuration.
The findClang.cmake file is structured to define and set several key variables necessary for integrating Clang into the project:

- **CLANG_FOUND**: This variable is set to true if Clang is successfully found.

- **CLANG_INCLUDE_DIRS**: Represents the path where Clang's header files are located, essential for the compiler to resolve the include statements.
- **CLANG_LIBS**: Contains a list of library files associated with Clang, which are necessary for linking during the compilation process.

Similarly, the FindLLVM.cmake file facilitates the discovery and configuration of LLVM within the project:

- **LLVM_FOUND**: Set to true when LLVM is found.
- **LLVM_INCLUDE_DIRS and LLVM_LIBRARY_DIRS**: These variables specify the paths to LLVM's include files and library directories, respectively, guiding the compiler to these resources.
- **LLVM_CFLAGS and LLVM_LDFLAGS**: These are the llvm compiler flag and llvm linker flags.
- **LLVM_MODULE_LIBS**: Includes a list of LLVM library files needed for module operations, ensuring that those module-related functionalities are supported.

2.7.1.2 pugixml

This folder contains the pugixml library, a lightweight C++ library for parsing, manipulating, and writing XML files. In this project, it provides the core functionality for the XmlGenerator module, allowing C++ code structures to be converted into XML representations. The folder includes the implementation file (pugixml.cpp), header files (pugixml.hpp and pugiconfig.hpp) for library declarations and configuration, and a small utility (foreach.hpp) for iteration support. By integrating pugixml, the project can efficiently generate and save XML files representing parsed code structures.

2.7.1.3 source

In this folder, there are main codebase of the project which consists of both '.ixx' and '.cpp' files. The '.ixx' files represent a relatively new feature in C++ introduced with the C++20 standard known as the module interface unit. This concept is designed to modernize and improve the modularity and encapsulation aspects of C++ programming.
Traditionally, C++ has relied heavily on header files to manage declarations and include reusable code across different '.cpp' files. Header files help reduce code complexity and enable the reuse of functions by including them in multiple source files. When a header file is included in a program, the compiler replaces the '#include <FileName.h>' directive with the actual content or functionality defined in the header file.
Modules interface units, represented by '.ixx' files in this project, serve a similar purpose as header files but with enhanced capabilities. Each '.ixx' file is compiled into a binary representation that other parts of the program can use. One of the significant advantages of using modules over traditional header files is that modules are compiled only once. This singular compilation process can significantly speed up compile times, as the compiler does not need to recompile the same declarations each time they are included in different parts of the program.

The main functions of those files are listed as follows:

- **ApisManager.cpp**: This file is a warp-up for the generating process, managing the process of generating, the output of the compile messages, or showing the error if the file cannot be processed.
- **ArgParse.ixx**: This is a file that changes include order to fix clang errors.
- **ClangTranslationUnit.cpp**:This file is responsible for parsing C++ source files into Clang translation units. It utilizes the Clang C API to create and reparse a translation unit from a given source file, handling compiler arguments and file paths. The file ensures proper initialization of the translation unit and throws runtime errors if parsing fails, serving as a foundational component for subsequent code analysis and AST generation in the project.
- **CppApiUsageGenerator.cpp**: This file implements the ‘CppApiUsageGenerator‘ class, which generates C++ code snippets to test and validate the usage of APIs defined in a specific header file. It traverses the Clang AST, focusing only on symbols from the main file, and handles constants, enums, structs, unions, functions, constructors, destructors, protected members, and typedefs. For each entity, it generates ‘static_assert‘ statements, function calls, or derived class scaffolds to ensure type correctness, constructibility, and convertibility. The generated code is written to a source file, allowing automated compile-time validation of API usage within the project.
- **IGenerator.ixx**: This file defines the abstract interface ‘IGenerator‘ for generator classes within the project. It declares two pure virtual functions: ‘add‘, which accepts a Clang cursor to include in the generation process, and ‘save‘, which writes the generated output to a specified file path.
- **XmlGenerator.cpp**: This file implements the ‘XmlGenerator‘ class, which generates an XML representation of C++ source code using Clang cursors. It traverses the abstract syntax tree (AST) of the parsed code and creates XML nodes for various constructs, including constants, enums, structs, unions, functions, namespaces, and typedefs. Each node contains attributes such as name, type, location, and evaluated value when applicable.
- **clangutils.cpp**: This file provides utility functions to interact with the Clang C API, facilitating traversal and inspection of C++ source code. It includes functions to retrieve cursor and type spellings, fully qualified names, source locations, tokens, annotations, and member information. It also provides helpers for parsing function arguments, enum values, and const structs. Additionally, it offers mechanisms to extract diagnostics from a translation unit and to visit AST nodes, enabling other components like ‘CppApiUsageGenerator‘ and ‘XmlGenerator‘ to analyze and generate code or XML representations based on Clang’s AST.
- **dib.ixx**: This file serves as the main module interface for the project, aggregating and exporting the core components of the ‘dib‘ module.
- **format.h**: This file enables the use of ‘fmtlib‘ as an alternative formatting library for environments where "std::format" is not supported. "std::format" is a feature introduced in C++20, offering a type-safe and modern approach to string formatting that is similar to Python’s "str.format()" functionality. However, not all compilers or systems have implemented this feature yet.

- **main.cpp**: This is the entrance of the parse code project, which responsible for parse the input files with the clang tool library, and then output an xml file that is an abstraction of the parsed file.
- **stdcompat.ixx**: This file provides compatibility utilities for C++ range-based operations, ensuring consistent behavior across different compiler versions. It defines helper functions such as ‘to‘, ‘contains‘, ‘insert_range‘, and ‘append_range‘, which allow the project to manipulate containers and sequences using modern range-based syntax even on compilers that do not fully support the latest C++ standard.

2.7.1.4 tests

In this folder, there are the patch files and header file which will be used as input files during the test.

2.7.1.5 header file

The IBaseline.h file serves as the original input for the tests and contains most of the common code constructs typically encountered in C++.
At first, there are some **preprocessor directives**, such as "#pragma one" or "#include <string>", the former one is used to ensure that the header file is included only once during the compilation, and the later one enables the programmer to use the "std::string" class. Then, there is the definition of the **namespace**, which means all of the definitions that in the parentheses are enclosed within this namespace, which helps avoid name conflicts.
There are also struct and class definitions in this file:

In **struct Immovable**, there is a default constructor and destructor, while the copy and move constructors and assignment operators are explicitly deleted.
Struct PureDataContainer is a simple structure containing an "int" and a "std::string" variable.
Struct ImmovableDataContainer contains an instance of Immovable, this structure can be used to demonstrate the containment of non-copyable and non-movable objects.

The **object class** is a class with a public constructor and two public void functions. This class also contains a private int variable.
Class IBase is an abstract base class with a virtual destructor and a pure virtual function "overridable()". This class has a protected constructor, which means it is intended for use as a base in inheritance hierarchies, enforcing derived classes to implement the "overridable()" function.
Struct DataContainerWithProtected contains both default and protected integer members. In a struct, if there is no explicit declaration of "public", "protected" or "private" access modifiers, members are public by default. Therefore, the integer "number" is a public member.
There is one function definition in this file, **Immovable returnsImmovable()** is a function declared to return an Immovable object.

2.7.1.6 patch file

A patch is a text file that shows modifications made to a file compared to its previous version. In this project, both the original file and a corresponding patch file are required as inputs for test.

The format of the patch file's name is fixed, it is "operation" + "should/shouldnot" + "break". The former part of the file is just used to make what it done clear while the should(not)break is the expected test result, which will be automatically recognized by the test program and compared with the real test result.

The name format of those patch file is constructed as follows: "operation" is appended by "should" or "shouldnot" followed by "break". It is structures to clarify both the operation performed and the expected outcome of the test. The initial segment of the file name, "operation," specifies the action applied to the file, enhancing clarity regarding the nature of the change. The latter part, "shouldbreak" or "shouldnotbreak," indicates the expected result of the test. This is critical as it allows the testing program to automatically recognize and anticipate the desired outcome. The actual test results are then compared to these expected outcomes to verify if the detect tool behaves as predicted under test conditions.

```
#pragma once
#include <string>

constexpr int f() { return 0; }
constexpr int size = 10;
const int CONST_INT = 10;
int globalInt = 10;

namespace test_api
{
int anotherGlobalInt = 10;
struct Immovable
{
    Immovable() = default;
    Immovable(const Immovable&) = delete;
    Immovable(Immovable&&) = delete;
    Immovable& operator=(const Immovable&) = delete;
    Immovable& operator=(Immovable&&) = delete;
    ~Immovable() = default;
};
struct PureDataContainer {
    int number;
    std::string text;
};

struct ImmovableDataContainer {
    Immovable immovable;
};

Immovable returnsImmovable();

class Object {
public:
    Object() = default;
```

```
35
36      void process();
37      void setValue(int);
38
39  private:
40      int privateData_ = 0;
41  };
42
43
44  class IBase
45  {
46  public:
47      virtual ~IBase() = default;
48      IBase& operator=(const IBase&) = default;
49      constexpr int getValue() const { return 0; }
50      static constexpr int constValue = 10;
51
52      virtual void overridable() = 0;
53
54  protected:
55      IBase() = default;
56      virtual void overridable_protected() = 0;
57  private:
58      virtual void privateOverridable() = 0;
59  };
60
61  struct DataContainerWithProtected {
62      int number;
63  protected:
64      int anotherNumber;
65  };
66
67  enum EnumContainer {
68      firstNum, secondNum, thirdNum
69  };
70
71
72  class protectedBase {
73  protected:
74      void setValue(int);
75      protectedBase& operator=(const protectedBase&) = default;
76  };
77  }
```

Listing 2.1: Content for IBaseline.h

3

Phase II: Development and Iteration

3.1 Libclang

To generate the usage code, parsing the c++ code accurately is imperative. For this purpose, I explored various tools as recommended by Clang's official documentation[27], which provides a detailed descriptions of the tools and their respective advantages and disadvantages. After a thorough evaluation, I opted to use libclang, which is a C interface to Clang.
Although LibTooling, being a C++ interface to Clang, may initially be considered more appropriate for parsing C++ code, libclang stands out due to its stability. This aspect is explicitly mentioned on the official Clang website, where it is noted that there is a possibility of disruptions in LibTooling due to API modifications.
Libclang, a C interface to the Clang compiler, serves as a streamlined API that offers essential functionalities for handling source code. These include parsing source code into an abstract syntax tree (AST), loading pre-parsed ASTs, and traversing the AST to analyze and manipulate code structures. This section will introduce and explain several key concepts and methods commonly employed in libclang that have proven instrumental throughout this project.

3.1.1 Representation of AST in libclang

Suppose the following code block:

```
int foo(int x) {
    int res = 2 * x;
    return res;
}
```

The AST of this code can be generated using Clang as follows:

```
clang -Xclang -ast-dump -fsyntax-only foo.h
```

In this command, *-Xclang* means Pass the argument to clang -cc1, flag*-ast-dump* means enable the AST-dump mode, flag *-fsyntax-only* means run the preprocessor, parser and semantic analysis stages.
Here is the output:

```
TranslationUnitDecl 0x7ff184840208 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0x7ff184840a30 <<invalid sloc>> <invalid sloc>
   implicit __int128_t '__int128'
```

```
| `-BuiltinType 0x7ff1848407d0 '__int128'
|-TypedefDecl 0x7ff184840aa0 <<invalid sloc>> <invalid sloc>
   implicit __uint128_t 'unsigned __int128'
| `-BuiltinType 0x7ff1848407f0 'unsigned __int128'
|-TypedefDecl 0x7ff184840db0 <<invalid sloc>> <invalid sloc>
   implicit __NSConstantString 'struct __NSConstantString_tag'
| `-RecordType 0x7ff184840b80 'struct __NSConstantString_tag'
|   `-Record 0x7ff184840af8 '__NSConstantString_tag'
|-TypedefDecl 0x7ff184840e58 <<invalid sloc>> <invalid sloc>
   implicit __builtin_ms_va_list 'char *'
| `-PointerType 0x7ff184840e10 'char *'
|   `-BuiltinType 0x7ff1848402b0 'char'
|-TypedefDecl 0x7ff184841148 <<invalid sloc>> <invalid sloc>
   implicit __builtin_va_list 'struct __va_list_tag[1]'
| `-ConstantArrayType 0x7ff1848410f0 'struct __va_list_tag[1]' 1
|   `-RecordType 0x7ff184840f30 'struct __va_list_tag'
|     `-Record 0x7ff184840eb0 '__va_list_tag'
`-FunctionDecl 0x7ff184898ad0 <foo.h:1:1, line:4:1> line:1:5 foo '
   int (int)'
  |-ParmVarDecl 0x7ff184898a00 <col:9, col:13> col:13 used x 'int'
  `-CompoundStmt 0x7ff184898d20 <col:16, line:4:1>
    |-DeclStmt 0x7ff184898cc0 <line:2:5, col:20>
    | `-VarDecl 0x7ff184898be0 <col:5, col:19> col:9 used res 'int'
    cinit
    |   `-BinaryOperator 0x7ff184898ca0 <col:15, col:19> 'int' '*'
    |     |-IntegerLiteral 0x7ff184898c48 <col:15> 'int' 2
    |     `-ImplicitCastExpr 0x7ff184898c88 <col:19> 'int' <
   LValueToRValue>
    |       `-DeclRefExpr 0x7ff184898c68 <col:19> 'int' lvalue
   ParmVar 0x7ff184898a00 'x' 'int'
    `-ReturnStmt 0x7ff184898d10 <line:3:5, col:12>
      `-ImplicitCastExpr 0x7ff184898cf8 <col:12> 'int' <
   LValueToRValue>
        `-DeclRefExpr 0x7ff184898cd8 <col:12> 'int' lvalue Var 0
   x7ff184898be0 'res' 'int'
```

Listing 3.1: output of the syntax tree

This output helps in understanding how Clang intrerprets and process the code. Here is the breakdown of each part of the provided AST output.

- **TranslationUnitDecl:** This is the root node of the AST for a translation unit. It will always be the toplevel declaration in a translation unit.
- **TypedefDecl:** The TypedefDecl for '__int128_t', '__uint128_t', '__NSConstandString' and the other types are the internal declarations of clang.
- **FunctionDecl:** Declares a function named 'foo' returning an 'int' and taking an 'int' parameter. This function is defined in the source file 'foo.h' from code lines 1 to 4. The following definitation are all under this node.
  - **ParmVarDecl:** Declares a parameter named 'x' of type 'int'. The 'used' label indicates that the variable 'x' is utilized in the function body.
  - **CompoundStmt:** Represents the compound statement, in the other words, the function body enclosed in '', spanning from the definition of the function to its end.
    - **DeclStmt & VarDecl:** A declaration statement within the func-

tion body defines and initializes a variable 'res' of type 'int'. The initializaton is '2 x'.

- **BinaryOperator:** A binary operation multiplying an integer '2' by 'x'. The multiplication is handled as an 'int' operation. Under this **BinaryOperator**, the **IntegereLiteral** stands for the integer value '2', the **ImplicitCastExpr** responsible for converting the lvalue 'x' to an rvalue, which is regerenced by **DeclRefExpr** and necessary for evaluating the expression.

* **ReturnStmt:** The function returns the values of 'res'. Similar to the use of 'x', **ImplicitCastExpr** which referenced by **DeclRefExpr** ensures the returned value is an rvalue, suitable for returning from the function.

3.2 Implementation of Docker

3.2.1 Difficulties of Porting the C++ code on a Different Machine

C++ is standardized by the International Organization for Standardization(ISO). This standardization ensures that compliant compilers provide the same core language features and standard library functions across different platforms. This standardization is what makes C++ a highly portable language, theoretically allowing the language itself, its standard library, and programs written in C++ to be compiled and executed on different hardware and operating systems without modification. However, due to several reasons, it is still challenging to port this project to the other platform.

The first challenge lies in the differences among compilers. Although the C++ standard aims to unify the language, different compilers may implement the standard in slightly different ways. For example, when new C++ features are introduced, compilers may not adopt them simultaneously. This project was initially developed on the Windows platform, where std::format was utilized. However, std::format was not supported by Xcode until version 15.3, which was released in May 2024[28]. The most commonly used compilers in the C++ ecosystem today include MSVC, GCC, Clang, and Apple Clang, each with its own unique characteristics and update schedules, making local implementation of the project more challenging.

The second challenge is the dependency on third-party libraries. During local implementation, one issue encountered involved the following error during the compilation process:

```
1 undefined reference to symbol '
    _ZN4llvm24DisableABIBreakingChecksE@@LLVM_17'
2 error adding symbols: DSO missing from command line
3 clang++-17: error: linker command failed with exit code 1 (use -v
    to see invocation)
```

Listing 3.2: Error encountered while compile the project

This error occurred because the project was linking against precompiled binary files using the target_link_libraries directive. The LLVM version being used during local

implementation differed from the one originally used to compile the project, leading to symbol resolution failures during the linking stage. Since LLVM versions can vary depending on the computer or device being used, ensuring consistent behavior across different environments becomes difficult.

3.2.2 Implementation of Docker

To address the challenges posed by compiler differences and third-party library dependencies, and to ensure a more reliable, consistent, and portable development process, Docker is used for the implementation environment.
Docker allows for the creation of a containerized environment that is consistent across different platforms, regardless of the underlying operating system or hardware. This means that once a Docker image is created with the required dependencies and configurations, it can be used to ensure that the project compiles and runs the same way on any machine, eliminating issues caused by variations in compiler implementations and system configurations.
Docker containers encapsulate all the necessary dependencies, including specific versions of compilers and third-party libraries like LLVM. This isolation ensures that the project uses the exact same versions of these tools every time it is built, avoiding the errors and inconsistencies that can arise when different systems have different versions of these dependencies installed.
The ease of setup also makes Docker stand out, allowing developers to configure the environment effortlessly, saving time and reducing the potential for setup-related errors.

3.3 Tools

During development, there are some functions might be used more than once. It is better to isolate and abstract these functions to promote code reusability and maintainability. By doing so, redundancy is reduced, debugging is simplified, and updating or modifying the code can be done more easily without affecting other parts of the project.

3.3.1 Clang_cursor Struct

The clang_cursor struct is a fundamental structure within the dib::clangutils namespace, designed to encapsulate a Clang cursor CXCursor. This struct provides a common interfact for various Clang entities and has been the base for the following manipulation.

3.3.2 Get Spelling

There are numerous instances where it's necessary to retrieve the name of a class, such as the name of an enum or the class name itself. Changes to these names can lead to breaking changes, so it's crucial to include them in the generated code.

Given the frequent need to obtain these names, the following methods are provided: "getSpelling", "getFullSpelling", and "getFullyQualifiedName".

The "getSpelling" method is implemented in two variations. One is designed to retrieve the spelling of a variable, while the other is for obtaining the spelling of a type. Retrieving the spelling of a variable is essential for testing whether the name of an enum has changed, as well as for accurately writing the namespace name or class name. Retrieving the spelling of a type is vital for testing whether a function can still operate correctly after modifications have been made. These methods ensure that any changes in names are properly detected and addressed, maintaining the integrity of the code.

The "getFullSpelling" method is designed to be used when a variable is defined within a class. In such cases, using "getSpelling" will only return the name of the variable itself. If the generated code is written using just this name, it may lead to errors due to the lack of context.

For example, consider a scenario where you have the following class:

```
class MyClass {
public:
    int myVariable;
};
```

Listing 3.3: Example for getFullSpelling

If "getSpelling" is used on "myVariable", it will return just "myVariable". However, attempting to reference this variable outside its class context without including the class name might lead to a compilation error because the compiler won't know which "myVariable" is being referred to, especially if there are multiple variables with the same name in different classes.

Instead, "getFullSpelling" would return "MyClass::myVariable". This method as well as the namespace solution, solved the issue of obtaining fully qualified name, ensuring that the generated code is accurate and free from such ambiguities, thereby avoiding potential errors during compilation.

3.3.3 Get Enum Values

Enum is a commonly used type in programming, often employed to define a set of named integer constants that represent values within a specific domain. Enums are particularly useful for improving code readability and maintainability by allowing the use of descriptive names instead of numeric values. However, changes to the names of enum values can introduce breaking changes in the codebase.

To address this, a function was developed that recursively retrieves the enum values and returns a vector containing a custom type called "enum_value". This type is designed to store both the name of the enum value and its corresponding integer value. Within this function, it is easy to write the generate code and test whether there are breaking changes due to the modificaiton on enum.

3.3.4 Get Function Argument

This function is designed to extract and return a list of arguments from a given Clang cursor representing a function.
It begins by initializing an empty vector of argument objects, which will store the extracted arguments.
The argument struct is a specialized structure within the dib::clangutils namesapce, designed to represent function arguments in Clang's abstract syntax tree. It inherits from the clang_cursor struct, which encapsulates a Clang cursor, providing a common interface for various Clang entities.
The function will retrieves the number of arguments of the function and traverse them one by one by skipping the null and invalid kind. If the checks pass, it creates an argument object with the argument cursor, the spelling and the type of it. After that, it pushs the newly-created object into the result and return the result vector with all the valid argument.

3.3.5 Generate Arg Values

This function is a template function designed to generate a string representation of argument calues for a given range of arguments.
The function starts by transforming the unput range of arguments using std::views::transform. Each argument is converted into a string representation using the format function, which calls std::declval on the argument;s type.
Next, if the C++ standard library supports std::views::join_with, which is introduced in C++23, it joins the transformed arguments with a comma and space. For the situation in which std::views::join_with is not available, the function will joins the transformed arguments manually.

3.4 The Namespace

3.4.1 Role of the Namespace

A namespace is a declarative region that assigns a specific scope to identifiers such as names of types, functions and variables. By placing symbols declared within a namespace into a distinct scope, namespaces help avoid confusion with symbols of the same name in different scopes. It is entirely acceptable to define multiple namespace blocks with the same name, as all declarations within these blocks are aggregated into the same named scope.
In this project, incorporating a namespace when generating code is vital. Omitting the namespace can lead to significant issues during the compilation phase, where otherwise valid code may be misinterpreted as incorrect. Consider the following code snippet, which illustrates the definition of the struct Immovable within the namespace test_api and a function that returns an instance of Immovable:

```
1 namespace test_api
2 {
3     struct Immovable
4     {
```

```
5           Immovable() = default;
6           Immovable(const Immovable&) = delete;
7           Immovable(Immovable&&) = delete;
8           Immovable& operator=(const Immovable&) = delete;
9           Immovable& operator=(Immovable&&) = delete;
10          ~Immovable() = default;
11      };
12      Immovable returnsImmovable();
13  }
```

Listing 3.4: Definition of the struct "Immovable" in the namespave "test_api" and the function which returns "Immovable"

When generating code, it is essential to check if the return value remains consistent before and after any modifications to the code. There are two methods can be used to accurately compare these values, the first one is reference the **fully qualified name** of the return type. **A fully qualified name** is a precise and unambiguous identifier that clarifies which object, function, or variable a call refers to, independent of the call's context. The second solution is enclose the code within the test_api namespace.
For instance, when generating the return value of the function returnsImmovable, the fully qualified name can either be used directly, such as test_api::Immovable, or the code can be placed within the corresponding namespace test_api. From the implementation perspective, the AST traversal is performed recursively. When a namespace node is encountered, its child nodes are traversed next. This approach simplifies the insertion of namespace braces before and after processing the nodes within the namespace, which is why this strategy was chosen for namespace handling in code generation.
From a technical standpoint, the process of navigating the Abstract Syntax Tree (AST) is recursive. Upon detecting a namespace, the traversal moves to its child nodes. This hierarchical approach facilitates the easy insertion of namespace delimiters (and) around the code contained within a namespace block. Opting for this method in the code generation phase simplifies the integration of namespace syntax and ensures the structural integrity of the generated code. Based on the reason mentioned above, enclosing the piece of code into namespace delimiters is chosen as the solution of the implementation of namespace.

3.4.2 Implementation of Namespace

The process of saving the output as a tree structure is the same as the code's traversal order, which also follows a tree structure. To navigate through the code, I employ a recursive approach. When a new namespace is detected during traversal, the system instantiates a namespace class to encapsulate all subsequent content found within that namespace3.1.
The namespace class is divided into four main sections: static assert, function scope, derived scope, and pointers to the encapsulated namespaces. Here's an expanded explanation of each section:

- **Static Assert:** In this section, static_assert is used to verify whether the types of function variables have changed and whether these changes could

potentially cause breaking changes. This part is also responsible for testing breaking change in aggregate types and enum types.

- **Function Scope:** This scope involves generating an instance of the class defined in the header file to check if the class remains initializable. Modifications such as deleting the constructor, altering its access specifier, or transforming the constructor into a virtual function might prevent class initialization. It also includes tests to check if the names of enumerations have been changed.

```
1     struct MyClass
2     {
3         int a;
4         double b;
5     };
6
```

Listing 3.5: Original code

```
1     struct MyClass
2     {
3         int a;
4         double b;
5         MyClass(const MyClass&) = delete;
6     };
7     MyClass obj1;
8     MyClass obj2 = obj1; // Error: copy constructor is deleted.
9
```

Listing 3.6: Change the access specifier of the constructor

```
1     struct MyClass
2     {
3         int a;
4         double b;
5         virtual void foo() = 0; // Once added  the virtual
    constructor, the class has been a abstract class
6     };
7     MyClass obj; // Error: Cannot instantiate abstract class
8
```

Listing 3.7: change the constructor into virtual

- **Derived Scope:** The project generates a derived class here to test the extendibility of the original class. This procedure verifies whether the class can still be inherited and helps identify breaking changes, such as the addition of a final specifier.
- **Namespace:** In the namespace class, there will be a vector which saves the pointers point to the child namespaces in this current namespace. This architectural choice is important for maintaining the hierarchical structure of namespaces, especially when dealing with complex or deeply nested codebases.

The namespace class is an important container to save all of the generated code that needs to be written into the file, it takes the changes of the structure of the generated code. Each section within the namespace class plays a vital role in ensuring that the modifications to the code maintain backward compatibility and do not introduce errors. This structured approach to organizing the generated code is particularly

beneficial as it facilitates a systematic and stable traversal of the code, line by line, regardless of the complexity or the number of nested namespaces involved.

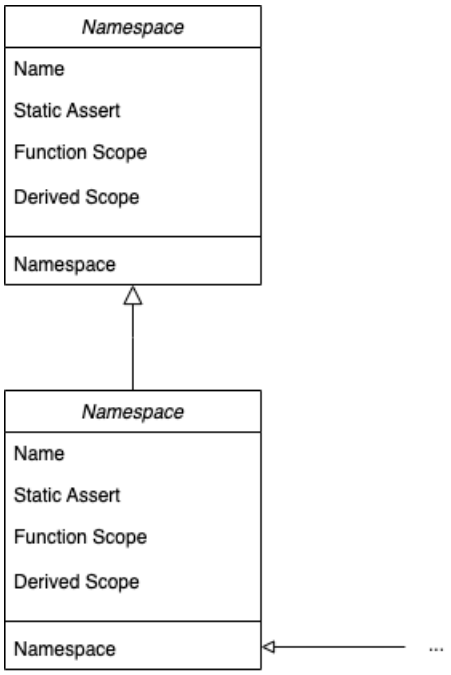

Figure 3.1: Structure of the Namespace class

3.5 Special Member Functions

In the C++ programming language, special member functions are functions which the compiler will automatically generate if they are used, but not declared explicitly by the programmer. The automatically generated special member functions are:

- **Default constructor:**
  Default constructors play a crucial role in C++ as they are automatically invoked in certain situations. An error will occur if a class lacks a default constructor in contexts where one is needed. For example, when an object is declared without any arguments, the default constructor is used to initialize the object. Similarly, if a container is defined with a user-defined class but no initial value is provided, the default constructor will be called to create instances of that class.
  The importance of default constructors in C++ cannot be overstated. If a class does not have an explicitly defined constructor, the compiler will implicitly declare and define a default constructor for it. This implicit constructor is equivalent to an explicitly defined one with an empty body, ensuring that the class can be instantiated without any arguments.
- **Copy constructor:**
  A copy constructor is a special constructor in C++ used to create a new object as a copy of an existing object. It is the standard mechanism for copying objects in the language. While the compiler automatically generates a default

copy constructor for each class, a user-defined copy constructor is typically necessary when an object contains pointers or non-shareable references. This custom copy constructor ensures that the copying process correctly handles these resources, avoiding issues such as shallow copying or unintended sharing of pointers, which could lead to memory management problems or other bugs.

- **Move constructor:**
  Move Constructor is a feature introduced in C++11 that optimizes the handling of temporary objects (often referred to as "rvalues") by allowing the transfer of resources from one object to another without the overhead of deep copying.
  For instance, when a temporary std::vector<T> was returned from a function, it had to be copied entirely into a new std::vector<T>, leading to significant memory and performance overhead.
  For example, in the move constructor of std::vector<T>, the internal pointer to the array is transferred to the new std::vector<T>, and the original pointer is set to nullptr. This prevents the original temporary object from deleting the array when it goes out of scope, avoiding unnecessary copying and deallocation.
- **Copy assignment operator:**
  The copy assignment operator is a special type of assignment operator used when the source and destination objects are of the same class type. Unlike the copy constructor, which initializes a new object by copying data from an existing object, the copy assignment operator is responsible for cleaning up the data members of the assignment's target and also have to correctly handle self-assignment situation.
- **Move assignment operator:**
  The move assignment operator, introduced in C++11, was designed to enhance the move semantics in C++. It functions similarly to the copy assignment operator but instead of copying the data, it transfers ownership of the data from the source object to the destination object without creating any additional copies. This operation is more efficient than copying, particularly for objects that manage dynamic resources, as it avoids the overhead of duplicating these resources. After the move assignment, the source object is left in a valid but unspecified state.
- **Destructor:**
  In object-oriented programming, a destructor is a special method that is automatically invoked just before an object's memory is released. This occurs in several situations: when the object's lifetime is bound to a scope and execution exits that scope, when the object is embedded within another object whose lifetime ends, or when the object was dynamically allocated and is explicitly deallocated. The primary function of a destructor is to free any resources that the object acquired during its lifetime, such as memory allocations, open files or sockets, database connections, and resource locks. Additionally, destructors may be used to deregister the object from other entities that maintain references to it. The use of destructors is integral to the Resource Acquisition Is Initialization (RAII) idiom, which ensures that resources are properly acquired and released in a controlled manner.

3.5.1 How Breaking Changes can be Introduced in the Special Member Function

Special member functions include the default constructor, copy constructor, move constructor, copy assignment operator, move assignment opertaor and destructor. Breaking changes in special number functions can have significant impacts on the behavior of a c++ class and its derived classes. Here are several common examples listed following:

```
class MyClass {
public:
    MyClass();
};

MyClass obj;  // Can run successfully
```

Listing 3.8: original code

3.5.1.1 Removing the Constructor

If a class that previously has a constructor, either it is user-defined or compiler-generated, mark it as delete will introduce breaking changes into the code.

```
class MyClass {
public:
    MyClass() = delete;  // Removing the default constructor
};

MyClass obj;  // Error: use of deleted function
```

Listing 3.9: code after delete the constructor

3.5.1.2 Changing the Accessible Specifier in the Constructor

If a class previously has a public constructor, changing the accessible specifier of the constructor might introduce breaking changes into the code.

```
class MyClass {
private:
    MyClass() = delete;  // Removing the default constructor
};

MyClass obj;  // Error: 'MyClass::MyClass()' is private within this
    context
```

Listing 3.10: code after mark the constructor as private

It is important to note that changing the access specifier of code from public to private or protected will introduce a breaking change. If code that was originally public is marked private or protected, it can no longer be accessed by any other part of the program, leading to compilation errors where that access was previously allowed.
If a piece of code is originally protected, it is accessible only within the class itself and its derived classes. If there is code that attempts to access the protected members

from outside the class, the original code will no longer compile. However, changing the access specifier of protected members can still introduce a breaking change, since the protected members may be accessed by derived classes. If the access specifier of these members is altered, it could break existing code that relies on those members being accessible in the derived classes. Therefore, even though protected members are not directly accessible outside their class hierarchy, changing their access specifier can still introduce breaking changes.
On the other hand, if a constructor is originally private, changing its access specifier does not typically introduce a breaking change. Since private constructors cannot be accessed from outside the class, they are not used externally, and therefore changing their access specifier does not affect existing code. However, there is an exception to this: if there is a private virtual function, it can be overridden in a derived class. This scenario must be carefully considered when designing the tools, as it can still lead to breaking changes in derived classes that rely on the ability to override such functions.
Another point that worth to mention is that if the special member function is marked as "deleted", it will not show up in the processed AST, which give a great convenient during development.

```
class MyClass {
public:
    MyClass() = default;
    virtual ~MyClass() {};
};

class MyClass_derived: MyClass {
    ~MyClass_derived() override {};
};

int main() {
    MyClass b;
}
```

Listing 3.11: private virtual function can be overrided

3.5.1.3 Providing a User-defined Constructor

If a user-defined constructor is added, the compiler will no longer generate a move constructor by default. This can introduce breaking changes if the code relies on the default-generated constructor, as it will no longer be available.

```
class MyClass {
public:
    MyClass(int i) {};
};

int main() {
    MyClass b; // error: no matching function for call to 'MyClass
    ::MyClass()'
}
```

Listing 3.12: code after adding user-defined constructor

As mentioned above, it is evident that introducing breaking changes into the code can occur easily. Detecting these changes is this project's primary mission. Additionally, there are some special cases that should be taken into account.

3.5.2 Aggregate Type

Apart from normal constructors, there is another significant exception need to be taken into consideration when building this tool: aggregate types.
Once a class or struct is an aggregate type, it can still introduce breaking changes even if the signature of the constructor has not changed, which is different from the breaking changes detection typically associated with normal constructors.

3.5.2.1 What is an Aggregate Type?

An aggregate in C++ is defined as an array type or a class type that meets specific criteria, including having no user-declared or inherited constructors, no private or protected non-static data members, no virtual base classes, no private or protected direct base classes, no virtual member functions, and no default member initializers.[29]
The concept of the aggregate type was introduced to facilitate simple and efficient initialization of objects, particularly for plain data structures. This allows for object initialization without the necessity for explicitly defined constructors, which is particularly advantageous for structures that encapsulate multiple values. This design choice aligns with C++'s objectives of providing high performance, direct memory access, and enhancing code readability and maintainability.
It is important to note that the definition of what constitutes an aggregate has evolved significantly across different versions of modern C++. For example, prior to C++11, the criteria included "no user-declared constructors." This was revised in C++11 to "no user-provided, inherited, or explicit constructors," and in C++20, the criteria were updated to exclude "private or protected non-static data members." Additionally, the requirements related to base classes were also modified in C++17. Initially, aggregates could not have any base classes, but the modification now excludes only "virtual base classes and private or protected direct base classes."[30]
These frequent changes in the definition of an aggregate type can potentially lead to difficulties in testing the breaking changes, as the evolving criteria may overturn the way to test the aggregate type. This necessitates continuous updates and reviews of test code to ensure compatibility and correctness of the detective tool.

3.5.2.2 Designated Initializer

Aggregate initialization requires the use of brace-enclosed initializer lists to initialize all members of an aggregate. For instance:

```
1     int ar[] = {1, 2}
```

In earlier versions of C++, aggregate initialization was primarily applicable in basic scenarios, such as initializing arrays or structs with a sequence of values. However, the latest version of C++ has significantly expanded this feature, allowing for more

complex aggregate initialization, including the setup of non-static data members of a class.
There are several advantages to using aggregate initialization. It is concise and easier to read compared to assigning values to members individually. Additionally, it can save time and memory by bypassing the need to call constructors for object members. Aggregate initialization also allows programmers to initialize only certain members of an object while leaving others with their default values. Furthermore, it enhances the portability of code because aggregate initialization is widely supported by all major compilers and is integrated into the C++ standard.
For aggregate classes, it is also possible to use designated initializers, as described in the C++ standard[31]. However, this feature can be a primary reason of breaking changes. If a type is an aggregate and is used with designated initializers, certain changes to the class or struct—such as adding, removing, or reordering members—can break the code that relies on this initialization method.

```
struct Point {
    int x;
    int y;
};

int main() {
    Point p = {.x = 1, .y = 2};
    std::cout << p.x << std::endl;
}
```

Listing 3.13: designated initialization

3.5.2.3 Breaking Changes Related to Aggregate Types

- **Adding a User-Declared Constructor**
  If a user-declared constructor is added to a class that was previously an aggregate, the class ceases to be an aggregate, which means it will no longer accept an aggregate initialization, even the signiture of the constructor is not changed.

```
    struct Point {
        int x;
        int y;
        Point(int x, int y):x(x), y(y) {}
    };


    int main() {
        Point p = {.x = 1, .y = 2}; // error: designated
    initializers cannot be used with a non-aggregate type '
    Point'
     }

```

Listing 3.14: Adding a user-declared constructor will break the designated initialization

- **Making a non-static data member private or protected**
  Changing a public data member to private or protected disqualifies the class from being an aggregate.

```
    struct Point {
        int x;
    private:
        int y;
    };

    int main() {
        Point p = {.x = 1, .y = 2}; // error: designated
    initializers cannot be used with a non-aggregate type '
    Point'
    }

```

Listing 3.15: code after change a non-static data member private

- **Making a non-static data member private or protected**
  Changing a public data member to private or protected disqualifies the class from being an aggregate.

```
    struct Point {
        int x;
    private:
        int y;
    };

    int main() {
        Point p = {.x = 1, .y = 2}; // error: designated
    initializers cannot be used with a non-aggregate type '
    Point'
    }

```

Listing 3.16: code after change a non-static data member private

- **Adding private or protected Base Class**
  Introducing inheritance will also remove the aggregate nature of the class, which makes the original code using a designated initializer break.

```
    struct Point {};

    struct IPoint: private Point {
        int x;
        int y;
    };

    int main() {
        IPoint p = {.x = 1, .y = 2}; // designated initializers
     cannot be used with a non-aggregate type 'IPoint'
    }

```

Listing 3.17: code after add a private base class

- **Adding virtual functions**
  Adding a virtual function to the class introduces a virtual table pointer in the class layout, which disqualifies the class from being an aggregate.

```
    struct Point {
        int x;
        int y;
        virtual void play() {};
    };

    int main() {
        Point p = {.x = 1, .y = 2}; // error: designated
    initializers cannot be used with a non-aggregate type '
    Point'
    }

```

Listing 3.18: code after change a non-static data member private

All in all, the examples listed above point to a common cause of breaking changes: they alter the class in such a way that it is no longer considered an aggregate class. This change prevents the previously allowed designated initializer from being used, leading to breaking changes in the code that relied on aggregate initialization.

3.5.2.4 How to Test Aggregate Types

As mentioned before, with respect to aggregate types, a breaking change will only be introduced if an aggregate class is no longer considered an aggregate. In other words, for this scenario, the primary focus should be on testing whether the aggregate class remains an aggregate after the modification.
There are mainly two steps to generate the test case for the aggregate type:

- **Test whether the class or struct is an aggregate in the original header file.**
  The task of verifying whether a class or struct is an aggregate in the original header file was one of the most challenging aspects of this function. Although the C++ Standard Library provides a method std::is_aggregate that could be used to determine if a type is an aggregate, this method is not directly applicable in this scenario. The system utilizes libclang for code traversal, parsing all code information into a format that defined by libclang. Unfortunately, while std::is_aggregate requires a Type object as a variable, the type information obtained from libclang is in the form of an enumeration defined by libclang itself, not directly compatible with the requirement of the variable of this method.
  To address this challenge, a method is implemented to determine if a type is an aggregate, closely adhering to the criteria defined in the C++ standard. This custom implementation involves checking for:
  - No user-declared or inherited constructors.
  - No private or protected non-static data members.
  - No virtual base classes.
  - No private or protected direct base classes.
  - No virtual member functions.

  - No default member initializers.

  The conditions mentioned above are checked one by one, and return true if the class is aggregate.
- **If yes, generate the static assert to test whether the class or struct is still an aggregate in the modified header file.**
  If the class is an aggregate, the class should remain an aggregate after any modifications. Otherwise, breaking changes may be introduced, as aggregate initialization is no longer allowed for non-aggregate classes. The project uses methods implemented in clangutils to obtain the spelling of the struct or class and stores it in a static_assert within the corresponding namespace scope.

3.5.3 How to Detect the Breaking Change Introduced by Changing of the Specific Member Function?

Based on the discussion in the previous section, there are several special cases that need to be taken into consideration. To clarify these, Figure 3.2 is provided to clearly illustrate the logic behind generating the test cases when a specific member function is present. This visual representation helps to ensure that all relevant scenarios are accounted for and indicates the code logic of this part of the tool.

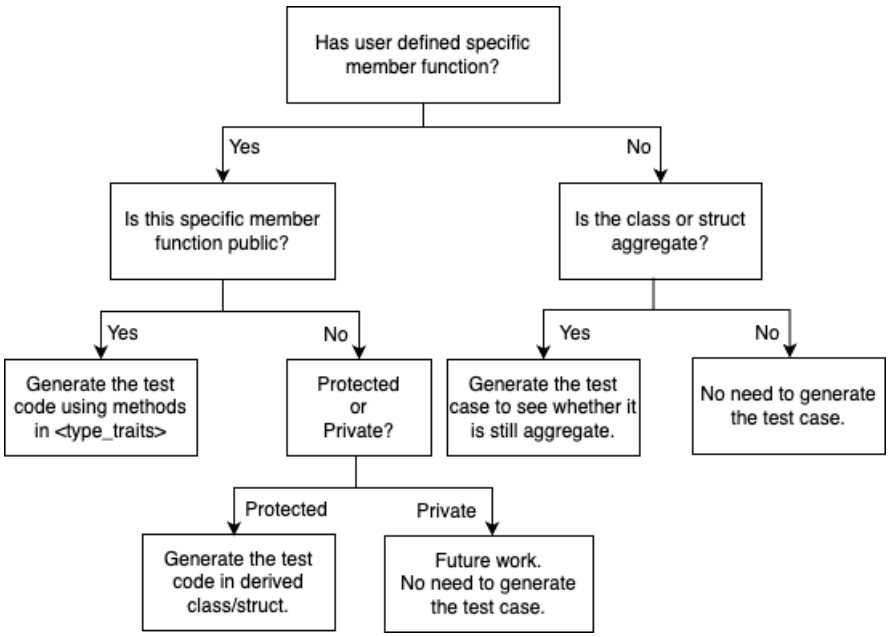

Figure 3.2: Method to test specific member functions

3.5.3.1 Public Specific Member Function

When traversing the Abstract Syntax Tree (AST) generated by libclang and encountering a user-defined constructor, the tool checks whether the access specifier for the constructor is public or protected. If the constructor is public, the tool invokes the *addConstructorUsage* function.

The *addConstructorUsage* function is specifically designed to generate and add static assertions for different types of constructor associated with a given Clang cursor that represents a constructor. The test code is generated using methods defined in *<type_traits>*.
The function first retrieves the name of the class and arguments of the constructor utilizing previously implemented functions such as *clangutils::getFullSpelling*, *clangutils::getSpelling*, and *clangutils::getFunctionArguments*. Based on the type of constructor (default, copy, or move constructor), the function then generates the appropriate static assertion using *std::is_constructible*, *std::is_copy_constructible*, or *std::is_move_constructible*.
Finally, the generated code, including these static assertions, is saved in the corresponding *Namespace* class, ensuring that the generated code can be read and written into the file during the final stage of the tool.

```
struct Immovable
{
    Immovable() = default;
    ~Immovable() = default;
};
```

Listing 3.19: sample code of Public specific member function

```
static_assert(std::is_constructible<Immovable>());
static_assert(std::is_destructible<Immovable>());
```

Listing 3.20: generated testing code for above header file

3.5.3.2 Protected Specific Member Function

Although protected specific member functions cannot be called outside of the class or struct, they can be accessed within derived classes or structs. Therefore, generating a derived class is the method used for creating test code. If the tool detects that the constructor is protected, it invokes the *addDerivedConstructor* function.
The *addDerivedConstructor* function is designed to generate and add the test code when the constructor is protected. It creates a class or struct derived from the original class, and within this derived class, it generates the test code so that the specific member functions can be tested in the protected field.
The function begins by retrieving the name of the class to which the constructor belongs using *clangutils::getSpelling*. It then constructs the name of the derived class by appending the class name to a prefix *derived_*. Depending on whether the original type is a struct or class, as indicated by the *isStruct* flag, the derived type is appropriately prefixed with either *struct* or *class*.
Next, the function retrieves the list of arguments for the constructor using *clangutils::getFunctionArguments* and generates the corresponding constructor code based on the type of constructor (default, copy, or move). Finally, the generated code is saved in the correct location within the *Namespace* class.

```
class IBase
{
protected:
    IBase() = default;
```

```
};
```

Listing 3.21: sample code of protected specific member function

```
class derived_IBase: IBase {
derived_IBase(): IBase() {};
};
```

Listing 3.22: generated testing code for above header file

3.5.4 Assignment Operator

The initial plan was to use the methods *clang_CXXMethod_isCopyAssignmentOperator* and *clang_CXXMethod_isMoveAssignmentOperator* provided by libclang to identify copy and move assignment operators in the code. However, these methods are unable to reliably distinguish copy assignment functions, treating them as regular methods instead. While this behavior is technically correct—since the copy assignment operator is indeed a form of operator overloading—it introduces significant inconvenience during tool development.
To effectively detect breaking changes in assignment operators while avoiding the use of the unreliable *clang_CXXMethod_isCopyAssignmentOperator*, the assignment operator will be treated as a regular method. To ensure comprehensive detection of breaking changes, the assignment operator will be handled based on its access specifier, code in protected and public fields will treated separately.

3.5.4.1 Public Field

For public copy or move assignment operators, they are treated as regular methods, and their functionality is tested by generating code that simulates their usage with *static_assert*.
First, the name and type of the cursor are retrieved. Then, the arguments of the function are obtained and combined into an expression that simulates a call on an instance of the function or method, ensuring that the generated code accurately reflects the use of the copy or move assignment operator.

```
class IBase
{
    IBase& operator=(const IBase&) = default;
};
```

Listing 3.23: sample code of public copy or move assignment operator

```
static_assert(std::is_same_v<decltype(std::declval<test_api::IBase
    >().operator=(std::declval<const IBase &>())), IBase &> || std::
    is_convertible_v<decltype(std::declval<test_api::IBase>().
    operator=(std::declval<const IBase &>())), IBase &>);
```

Listing 3.24: generated testing code for above header file

3.5.4.2 Protected Field

For protected copy assignment operators, they are still treated as regular methods and are tested by attempting to call them within a derived class. Similar to functions in the public field, the name of the function is obtained using the previously implemented function.
Next, the name of the derived class or struct is generated with a "derived_" prefix. The arguments and the name of the function are then retrieved and combined to form the appropriate call expression. Finally, the generated code is saved into the corresponding 'Namespace' class, ensuring it can be properly read into the file in the final stage of the tool.

```
class protectedBase {
protected:
    protectedBase& operator=(const protectedBase&) = default;
};
```

Listing 3.25: sample code of protected copy or move assignment operator

```
class derived_protectedBase: protectedBase {
protectedBase & operator=(const protectedBase & para0) {
   protectedBase::operator=(para0);};
};
```

Listing 3.26: generated testing code for above header file

3.6 Type and Name

Type and name changes are among the easiest elements to introduce breaking changes, given their broad impact across the codebase. A simple change in the type of a member variable can affect numerous parts of the program, leading to significant issues during compilation and execution.
Moreover, according to the 13 well-known breaking changes analysed by RefDiff[13] and the comprehensive breakdown across types, methods, and fields by APIDIFF 2.1, it is evident that a significant portion of breaking changes are related to name or type modifications. This further emphasizes the importance of thoroughly testing for type-related breaking changes. These analyses also provide valuable guidance, highlighting that type changes often impact function signatures, return types, and member variables, making them a critical focus during the evolution of the code.
According to APIDIFF, breaking changes related to methods include removing a method, losing visibility, changes in the return type, changes in the parameter list, changes in the exception list, adding a final modifier, removing a static modifier, moving a method, renaming a method, pushing down a method, and inlining a method. Most of these breaking changes stem from a common issue: a function or variable that was previously accessible can no longer be called or used in the same way. These changes are often unavoidable, perhaps due to optimizations in the function or the introduction of new features.
Most of the breaking changes listed above can be tested by generating code that simulates the usage of the method according to the old header file and then attempting

to compile it against the new header file. So that if there are breaking changes and the function can not be used according to the old header file, the generated code will help identify this breaking change by causing a compilation error.

3.6.1 How to Test Breaking Change Related to Type and Name

3.6.1.1 Public Field

As mentioned before, the generated code should generate a minimum test case that can call the function, it should focus on the according faces: Whether the arguments of the function change, whether the return type of the function changes, whether the name of the function or data member changes, etc.
Before generating the code, it is necessary to determine whether the element is a function or a data member. Although the generated code follows a similar logic for both cases, they are handled differently, as illustrated in Figure 3.3.

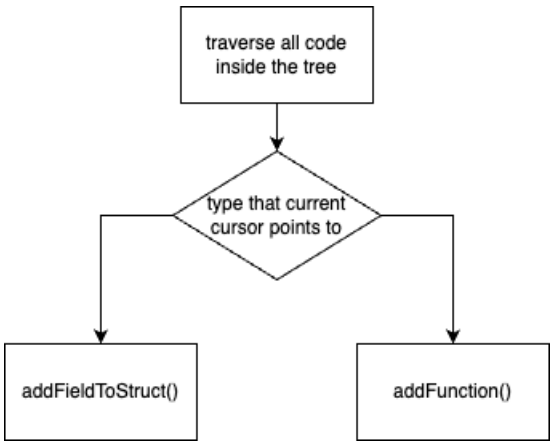

Figure 3.3: Logic in addStruct

If the cursor points to a data member, *std::is_same_v* and *std::is_convertible_v* are used to verify whether the type and name remain consistent between two versions of the code. For the first argument of *std::is_same_v* or *std::is_convertible_v*, an instance of the data member's type is generated within the class. The second argument corresponds to the type of the data member as retrieved from the previous version of the header file. This ensures compatibility and detects breaking changes related to type or name modifications in data members. For example, if the name of the data member has changed, an error occurs when generating the instance. Conversely, if the type of the data member has changed, an assertion failure is triggered based on *std::is_same_v* or *std::is_convertible_v*, highlighting the type inconsistency. This method effectively identifies changes that could introduce potential breaking issues.
If the cursor points to a method, the project still use *static_assert* along with *std::is_same_v* and *std::is_convertible_v*. The first argument of the function will be the instance of the method's return type, and the instances of the method's arguments will be generated to pass into the function. The second argument of *std::is_same_v* and *std::is_convertible_v* will be the return type of the method, as

retrieved from the previous version of the header file. Similarly, in the generated code for data members, changes to the variable name or return type will result in either compiler errors or assertion failures. Additionally, if the types of the method's parameters are modified, it will also lead to compiler errors, further helping to detect breaking changes.

```
class Object {
public:
    void setValue(int);
};
```

Listing 3.27: Sample code of public method in a class

```
static_assert(std::is_same_v<decltype(std::declval<test_api::Object
    >().setValue(std::declval<int>())), void> || std::
    is_convertible_v<decltype(std::declval<test_api::Object>().
    setValue(std::declval<int>())), void>);
```

Listing 3.28: generated testing code calling the function

```
class Object {
public:
    void setValue1(int);
};
```

Listing 3.29: Change function name in a class

```
error: 'class test_api::Object' has no member named 'setValue'; did
    you mean 'setValue1'?
   14 |     static_assert(std::is_same_v<decltype(std::declval<
    test_api::Object>().setValue(std::declval<int>(
```

Listing 3.30: Error tested by the generated code

3.6.1.2 Protected Field

For data members in the protected field, the test code is generated using the *addProtectedFunction* method, which is designed to generate and incorporate code for protected member functions of a class or struct. This method ensures that protected functions are correctly recognized and used by creating a derived class that exposes these functions, enabling them to be tested.
The method begins by retrieving the name of the class to which the function belongs using *clangutils::getSpelling* on the semantic parent of the cursor. It then constructs the name of the derived class by appending the class name to a prefix *derived_*. The derived class is prefixed with either *struct* or *class*, based on the *isStruct* flag. Next, the method retrieves the function name, its arguments, and its return type. It then generates the function call expression for the derived class, which calls the protected function of the base class from within the derived class or struct.
By invoking the method in the parent class or struct, any changes to the function signature will result in errors, thereby detecting breaking changes introduced in the parent class or struct.

```
class protectedBase {
protected:
    void setValue(int);
};
```

Listing 3.31: Sample code for the protected member function

```
class derived_protectedBase: protectedBase {
void setValue(int para0) {protectedBase::setValue(para0);};
};
```

Listing 3.32: Generated code for the protected member function

3.7 Virtual and Override

Function overriding is an object-oriented programming concept where a function in a derived class has the same name, return type, and parameters (signature) as a function in its base class. The function in the derived class overrides the behavior of the function in the base class, allowing the derived class to provide its own specific implementation of the function.
The purpose of function overriding is to allow the derived class to customize or completely change the behavior of the base class's function, while still maintaining the same interface.
In C++, a function must be declared as *virtual* in the base class to allow it to be overridden in derived classes. The overriding function in the derived class typically uses the *override* keyword to explicitly indicate that it is intended to override a base class function. Furthermore, the overriding function must have the same signature as the base class function to ensure proper behavior. A pure virtual function in C++ is a virtual function that does not have an implementation in the base class, and it must be overridden by any derived class that inherits from the base class. A class that contains at least one pure virtual function becomes an abstract class, meaning that it cannot be instantiated directly. Instead, the class can only be used as a base class for other classes that provide implementations for the pure virtual functions. The concept of virtual and override is a good reflection of polymorphism in C++.

3.7.1 Test for the Virtual Method in Public and Protected Fields

When the traversal cursor points to a method identified as a pure virtual function, the *addDerivedFunction* method is invoked to generate the test case for that function. This method is responsible for generating code for derived functions that override virtual functions in derived classes. It begins by retrieving the return type of the function using *clangutils::getSpelling* on the result type of the cursor, along with the function name and the class name to which the function belongs. Next, it constructs the name of the derived class by appending the class name to a prefix, *derived_*. The derived class is prefixed with either *struct* or *class*, depending on the *isStruct* flag. Finally, the method generates the function declaration for the derived class,

ensuring that it includes the *override* keyword, indicating that the function overrides the virtual function from the base class.

```
class IBase
{
public:
    virtual void overridable() = 0;
protected:
    virtual void protectedOverridable_protected() = 0;
private:
    virtual void privateOverridable() = 0;
};
```

Listing 3.33: Sample code for the pure virtual member function

```
class derived_IBase: IBase {
void overridable() override {};
void protectedOverridable() override {};
void privateOverridable() override {};
};
```

Listing 3.34: Generated code for the pure virtual member function

3.7.2 Special Case for Function Overriding

As mentioned above, there will not be generated test cases for the methods in the private field because they cannot be accessed outside the class. However, there is a special case: private virtual functions.

When a private function is defined in a class, it can only be accessed by instances of that class. Other classes, including those derived from the original class, cannot access it. However, if the private function is declared as virtual, it can still be overridden by derived classes.

To address this special case, private methods that are also pure virtual are not filtered out. In such cases, the *addDerivedFunction* method will still be invoked to handle the generation of test cases for these private virtual functions, ensuring that they are correctly overridden in derived classes.

```
class IBase
{
private:
    virtual void privateOverridable() = 0;
};
```

Listing 3.35: Sample code for the private pure virtual member function

```
class derived_IBase: IBase {
void privateOverridable() override {};
};
```

Listing 3.36: Generated code for the private pure virtual member function

3.8 Enum

An enum, which is also called enumeration, is a user-defined type that consists of a set of named integral constants in C++. Enums are typically used to represent a collection of related constants with human-readable names, improving code readability and maintainability. Renaming enum values, removing enum values or renaming the enum class will cause breaking changes.

3.8.1 Generated Code for Enum

The generated code will focus on generating an enum variable and assign all of the enum that appeared in the previous header file. Once an enum is detected, the *addEnum* function will be invoked. This method is designed to handle enum in the C++ code being processed, ensuring that the enumerations are correctly recognized and assigned to a variable one by one.

```
enum EnumContainer {
    firstNum, secondNum, thirdNum
};
```

Listing 3.37: Sample code for Enum

```
static void __gen_func() {
EnumContainer temp;
temp = firstNum;
temp = secondNum;
temp = thirdNum;
}
```

Listing 3.38: Generated code for Enum

3.9 constexpr and const expression

The **constexpr** keyword, introduced in C++11, allows the compiler to evaluate the value of a variable or the result of a function at compile time, aiming to improve performance by enabling optimizations during compilation.
On the other hand, **const** is a keyword in C++ that marks a variable as immutable after initialization. Unlike **constexpr**, **const** does not require the value to be known at compile time, allowing it to be initialized at runtime. **const** can be applied to both variables and function parameters, ensuring they remain unchanged after being set.
Breaking changes introduced in **constexpr** and **const** variables or functions follow similar patterns to those seen with data members or methods in classes or structs. These include changing the name of the variable, changing its type, or altering the return type of a function. Such changes can disrupt existing code that relies on the previous signatures, making it important to track these modifications.

3.9.1 Generated Code for Constexpr and Const Expression

As mentioned above, the approach to testing breaking changes in **constexpr** and **const** variables is similar to the method used for testing data members and functions.
The function *addConstStaticVar* is invoked when the cursor points to a variable. This method begins by retrieving the type and name of the variable using *clangutils::getSpelling* on the cursor. It then generates the corresponding static assertion, which is added to the current namespace. This ensures that the variable's type and name remain consistent and that any changes will trigger a static assertion, helping to detect breaking changes in the code.

```
constexpr int f() { return 0; }
constexpr int size = 10;
const int CONST_INT = 10;
```

Listing 3.39: Sample code for constexpr variable constexpr function and const variable

```
static_assert(std::is_same_v<decltype(f()), int> || std::
    is_convertible_v<decltype(f()), int>);
static_assert(std::is_same_v<decltype(size), const int> || std::
    is_convertible_v<decltype(size), const int>);
static_assert(std::is_same_v<decltype(CONST_INT), const int> || std
    ::is_convertible_v<decltype(CONST_INT), const int>);
```

Listing 3.40: Generated code for constexpr

3.10 Summary

This chapter documented the development of a tool for detecting breaking changes in C++ code through automated analysis. The implementation utilized libclang for reliable code parsing and Docker for ensuring consistent cross-platform execution environments.
The core architecture organizes generated test code hierarchically within namespace classes, systematically testing:

- Special member functions (constructors, destructors, assignment operators)
- Aggregate types and designated initializers
- Type and name changes in functions and data members
- Virtual methods and polymorphism
- Enumerations and const expressions

Key implementation insights include prioritizing stable parsing interfaces, ensuring environment consistency through containerization, and generating hierarchical test code that mirrors the original code structure. The tool demonstrates that automated breaking change detection requires careful handling of C++ semantics and scope management while providing a foundation for practical compatibility testing.

4

Phase III: Test and Performance

In this chapter, we evaluate the proposed tool through a series of tests conducted in a real-world development environment. Unlike the preliminary tests performed during development, which relied on simplified and relatively independent header files, the tests in this phase aim to reflect realistic usage scenarios encountered in industrial C++ projects. Such projects typically involve complex directory structures, extensive inter-file dependencies, and, in some cases, dependencies across multiple repositories.

The primary objective of this testing phase is to examine whether Did-it-break can function correctly when applied to real project files, and to identify potential limitations arising from environmental or configuration factors. During the testing process, several issues were observed that did not manifest in the earlier development-stage tests. These issues mainly stem from missing dependencies, project-specific compilation settings, and differences between the assumed and actual compilation environments.

By systematically analyzing these issues and applying corresponding solutions, we refined the testing setup to more accurately represent the real compilation context of the project. This process not only validates the correctness of Did-it-break under realistic conditions, but also provides insight into its robustness and practical applicability in real software development workflows.

4.1 Real-World Test Setup

During the initial development stage, the test files used were deliberately simplified to facilitate rapid iteration and debugging. In these tests, header files were largely independent and rarely included other project-specific headers. While this approach was sufficient for validating the core logic of Did-it-break, it does not accurately reflect real-world development scenarios.

In practical software projects, header files often depend on multiple other headers, and it is common for a project to rely on external repositories through mechanisms such as Git submodules. Therefore, testing Did-it-break in a realistic environment is essential to assess its applicability beyond controlled experimental conditions.

4.1.1 Dependency Management and Context Reconstruction

As shown in Listing 8.1, the compilation terminated with a fatal preprocessor error due to an unresolved include directive in IRootDirProvider.h. Specifically, the compiler was unable to locate the header file, which is provided by a dependent repository and is normally resolved via project-specific include paths. Because the file was compiled outside of its original build context, the required include directories were not propagated to the compiler invocation. Consequently, the preprocessor failed during header expansion, preventing further semantic analysis and code generation. Reconstructing the complete repository hierarchy restored the expected include search paths, allowing all transitive dependencies to be resolved correctly.

To resolve the dependency-related issues, it was necessary to restore the correct compilation context for the tested files. The most straightforward solution was to clone the entire target repository along with all its dependencies. This ensured that the directory structure and relative include paths matched those used in the original development environment.

After confirming the approach with team members, the main repository and its dependent repositories were cloned locally. The tested header files were then placed back into their original locations within the project structure. By reconstructing the full context in which the files are normally compiled, the compiler was able to correctly resolve all include directives, allowing Did-it-break to proceed with code generation as intended.

```
1 Diagnostic: ./tests_realfile/IRootDirProvider.h:13: fatal error: '
    king/filesystem/path.h' file not found [Lexical or Preprocessor
    Issue]
2 ERROR: Can't process file: "./tests_realfile/IRootDirProvider.h"
```

Listing 4.1: Can not find file error

4.1.2 Compiler Configuration and C++ Standard Compatibility

In addition to dependency-related issues, another problem was identified during testing related to compiler configuration. The default compiler settings used by Did-it-break did not support the C++20 standard, which is required by the project. As a result, modern C++ features such as `concept` could not be recognized, leading to additional parsing errors.

To address this issue, an explicit compilation flag `-std=c++20` was added to Did-it-break's invocation. This ensured that the compiler was configured consistently with the project's requirements and was able to correctly parse all C++20-specific language constructs. After applying this configuration, Did-it-break successfully processed the test files without encountering further standard-related issues.

4.2 Test Results

This section presents and analyzes the outcomes of applying the proposed compilation-based breaking change detection tool. Its structure is designed to closely align with the research objectives and the development process described in the preceding chapters. The chapter begins by restating the primary evaluation goals derived from the research questions: assessing whether Did-it-break can correctly detect breaking changes in C++ APIs and whether it remains effective when applied to realistic codebases with complex dependencies and modern C++ language features.

4.2.1 Evaluation Objectives

The primary objectives of the evaluation are twofold. First, it aims to verify the functional correctness of the proposed tool in detecting breaking changes through compilation failures. Second, it evaluates the robustness of the approach when applied to real-world projects, where build environments, dependency structures, and compiler configurations introduce additional complexity.

4.2.1.1 Functional Correctness of Breaking Change Detection

After resolving dependency-related issues and aligning compiler configurations with the original project environments, the proposed tool was able to successfully generate usage source files and perform compilation-based breaking change detection on real-world project headers.
Using ChangeMemberParameterShouldBreak.cpp as an example, Listing 4.2 illustrates the output generated during a single test execution. Each test file follows a consistent naming convention that encodes both the type of API modification applied to the IBaseline.h (e.g., ChangeMemberParameter) and whether the change is expected to introduce a breaking change (ShouldBreak).
The initial part of the output visualizes the test workflow, including invocation of the code generation command, construction of the translation unit, and execution of the compilation step. This information confirms that Did-it-break successfully processes the input header, generates the corresponding usage source file, and compiles it within the configured environment.
The latter portion of the output shows the compiler diagnostics that reveal the detected breaking change. In this example, the parameter type of the member function setValue has been modified from int to PureDataContainer. As a result, the generated usage code attempts to call setValue with an int argument, which is no longer convertible to the updated parameter type. The compiler therefore reports a type conversion error, indicating that the previously valid usage is no longer supported.
This compilation failure directly reflects a real breaking change that downstream users would encounter after upgrading the API. Since the test is designed to expect such a failure, the final test result is reported as successful.

```
1 -- Running test ChangeMemberParameterShouldBreak --
2 Running generation command: ['/did-it-break/build/did-it-break', '-
    i', '/usdk/kstl/include', '-i', '/usdk/usdk/api/cpp-api/include'
```

```
    , '/did-it-break/tests/ChangeMemberParameterShouldBreak.h', '-o'
    , '/did-it-break/tests/ChangeMemberParameterShouldBreak.cpp']
libclang API: 0.64
Translation Unit for: "/did-it-break/tests/
    ChangeMemberParameterShouldBreak.h"
Diagnostic: /did-it-break/tests/ChangeMemberParameterShouldBreak.h
    :1: warning: #pragma once in main file [Lexical or Preprocessor
    Issue]
-- Processing "ChangeMemberParameterShouldBreak"
  -> Output file: "/did-it-break/tests/
    ChangeMemberParameterShouldBreak.cpp"
Hunk #1 succeeded at 34 (offset 6 lines).
Running compile command inside /did-it-break/tests: ['clang++-17',
    'ChangeMemberParameterShouldBreak.cpp', '-c', '-std=c++20', '-I/
    did-it-break/tests']
ChangeMemberParameterShouldBreak.cpp:23:81: error: no viable
    conversion from 'int' to 'PureDataContainer'
    23 | static_assert(std::is_same_v<decltype(std::declval<test_api
    ::Object>().setValue(std::declval<int>())), void> || std::
    is_convertible_v<decltype(std::declval<test_api::Object>().
    setValue(std::declval<int>())), void>);
       |
                        ^~~~~~~~~~~~~~~~~~
./ChangeMemberParameterShouldBreak.h:21:8: note: candidate
    constructor (the implicit copy constructor) not viable: no known
     conversion from 'int' to 'const PureDataContainer &' for 1st
    argument
    21 | struct PureDataContainer {
       |        ^~~~~~~~~~~~~~~~~
./ChangeMemberParameterShouldBreak.h:21:8: note: candidate
    constructor (the implicit move constructor) not viable: no known
     conversion from 'int' to 'PureDataContainer &&' for 1st
    argument
    21 | struct PureDataContainer {
       |        ^~~~~~~~~~~~~~~~~
./ChangeMemberParameterShouldBreak.h:37:36: note: passing argument
    to parameter here
    37 |     void setValue(PureDataContainer);
       |                                    ^
ChangeMemberParameterShouldBreak.cpp:23:186: error: no viable
    conversion from 'int' to 'PureDataContainer'
    23 | static_assert(std::is_same_v<decltype(std::declval<test_api
    ::Object>().setValue(std::declval<int>())), void> || std::
    is_convertible_v<decltype(std::declval<test_api::Object>().
    setValue(std::declval<int>())), void>);
       |


    ^~~~~~~~~~~~~~~~~~
./ChangeMemberParameterShouldBreak.h:21:8: note: candidate
    constructor (the implicit copy constructor) not viable: no known
     conversion from 'int' to 'const PureDataContainer &' for 1st
    argument
    21 | struct PureDataContainer {
       |        ^~~~~~~~~~~~~~~~~
./ChangeMemberParameterShouldBreak.h:21:8: note: candidate
```

```
   constructor (the implicit move constructor) not viable: no known
    conversion from 'int' to 'PureDataContainer &&' for 1st
   argument
29   21 | struct PureDataContainer {
30      |        ^~~~~~~~~~~~~~~~~
31 ./ChangeMemberParameterShouldBreak.h:37:36: note: passing argument
   to parameter here
32   37 |     void setValue(PureDataContainer);
33      |                                   ^
34 2 errors generated.
35 --- ChangeMemberParameterShouldBreak: Success
```

Listing 4.2: Output for the test

4.2.1.2 Detected Breaking Change Categories

The results presented in Table 4.1 demonstrate that, when executed within an appropriate project context and with correct compiler settings, Did-it-break can operate reliably on realistic codebases that include complex dependencies and modern C++ language features. It is important to note that Table 4.1 is not an exhaustive list of all possible breaking changes in C++; rather, it reflects the categories that were explicitly modelled and validated through the testing conducted in this project. The space of possible breaking changes in C++ is inherently open-ended, and additional categories may be incorporated as new patterns are encountered in practice.

Element	Breaking Changes Detected by Did-it-break
Type	REMOVE TYPE, LOST VISIBILITY, CHANGE IN SUPERTYPE, ADD FINAL MODIFIER, CHANGE IN CONSTRUCTIBILITY, CHANGE IN DESTRUCTIBILITY, CHANGE IN AGGREGATE PROPERTY
Method	REMOVE METHOD, LOST VISIBILITY, CHANGE IN RETURN TYPE, CHANGE IN PARAMETER LIST, ADD FINAL MODIFIER, REMOVE STATIC MODIFIER, CHANGE IN OVERRIDABILITY
Field	REMOVE FIELD, LOST VISIBILITY, CHANGE IN FIELD TYPE, CHANGE IN CONST QUALIFICATION

Table 4.1: Breaking Changes Detected by Did-It-Break

The evaluation confirms that Did-it-break effectively applies its compilation-based detection strategy to identify breaking changes as intended. By reconstructing expected API usage patterns and validating them through compilation, Did-it-break exposes incompatibilities introduced by API evolution that would affect downstream users. These findings indicate that the approach is not limited to isolated or synthetic examples, but is applicable to practical development scenarios.
Based on the generated test code and associated compile-time assertions, the proposed tool detects breaking changes across three main API element categories: *Type*, *Method*, and *Field*, altogether identifying 19 distinct breaking change types.

The distribution of detected breaking changes across these three categories provides insight into the strengths of the proposed approach. Type-level changes account for a substantial portion of the detected cases, reflecting the fact that many breaking changes in C++ APIs originate from modifications to class properties, inheritance relationships, and object construction semantics. Changes such as altered constructibility, loss of aggregate properties, or the introduction of `final` specifiers are particularly impactful, as they can invalidate large classes of client code without altering function signatures.
Method-level breaking changes are primarily identified through changes in callable behavior, including return type modifications, parameter list changes, and loss of overridability. The results show that Did-it-break is effective at detecting these changes by attempting to compile representative usage patterns, such as invoking member functions, overriding virtual functions in derived classes, and validating static versus non-static access. This demonstrates that the compilation-based strategy naturally captures semantic incompatibilities that are difficult to identify through purely syntactic analysis.
Field-level breaking changes are detected through type trait checks and direct member access in generated test code. Although fewer in number compared to type- and method-level changes, these results highlight Did-it-break's ability to identify subtle API evolutions, such as changes in const qualification or data member types, which can silently break binary or source compatibility in dependent codebases.
Across all categories, the observed compilation failures directly correspond to realistic breakage scenarios that downstream users would encounter when upgrading library versions. Rather than relying on predefined rules alone, Did-it-break validates API stability by enforcing actual usage constraints at compile time. This reinforces the practical relevance of the detected breaking changes and supports the claim that the proposed method aligns closely with real-world developer experience.

4.2.1.3 Applicability to Real-World Codebases

Beyond controlled test cases, Did-it-break was evaluated on real project headers with deep dependency chains and external libraries. After resolving environmental issues, Did-it-break successfully generated usage code and detected breaking changes in these settings.
The results demonstrate that the proposed approach scales beyond isolated examples and can be applied to realistic development scenarios. This confirms the practical viability of compilation-based breaking change detection for integration into real-world workflows.

4.2.1.4 Coverage and Limitations of Detection Logic

Finally, the results highlight the coverage boundaries of the current implementation. Did-it-break performs as expected for explicitly modeled breaking change patterns across type, method, and field elements. However, the space of possible breaking changes in C++ is inherently open-ended, and changes that do not correspond to predefined usage patterns may remain undetected.

This limitation reflects an inherent characteristic of rule-based and usage-driven detection approaches rather than a flaw in the method itself. The results therefore motivate continuous extension of the detection logic as new breaking change patterns are encountered.

4.3 Discussions

The evaluation demonstrates that the proposed compilation-based approach provides a practical balance between detection accuracy and implementation complexity. By encoding expected API usage into generated source files and relying on the compiler to validate compatibility, Did-it-break leverages the C++ type system to detect breaking changes with high fidelity. This strategy significantly reduces false positives, as reported breaking changes correspond directly to compilation failures that would be encountered by real API consumers.

At the same time, the results highlight an inherent limitation of the approach: the space of possible breaking changes in C++ is open-ended. The current implementation is therefore most effective at detecting breaking changes that have been explicitly modeled in the detection logic. Changes that fall outside these predefined patterns may not be identified automatically. However, this limitation is not unique to the proposed tool and reflects a broader challenge in rule-based and usage-based detection systems. Importantly, the design remains extensible, allowing new detection rules to be incrementally introduced as additional breaking change patterns emerge in practice.

From a performance standpoint, the primary cost introduced by Did-it-break arises from additional compilation steps. This overhead is comparable to standard build processes and scales predictably with project size and dependency complexity. Since the approach does not require runtime instrumentation or dynamic analysis, it incurs no execution-time overhead and is well suited for integration into continuous integration and regression testing pipelines.

The evaluation also reveals that the effectiveness and reliability of Did-it-break depend on accurately reconstructing the original build environment. Missing dependencies or mismatched compiler configurations can limit the applicability of the analysis or lead to misleading results. This dependency illustrates a trade-off inherent to compilation-based techniques: while they provide precise and realistic detection of breaking changes, they require closer alignment with project build systems than purely static comparison methods.

Overall, the results indicate that the proposed approach is particularly well suited for pre-release validation and regression testing scenarios, where correctness and compatibility guarantees are prioritized over minimal analysis cost. As the detection logic evolves alongside the project and the C++ language itself, Did-it-break is expected to maintain predictable performance characteristics, supporting long-term maintenance and adoption in real-world development workflows.

5
Results

This chapter presents the results of the project, describing Did-it-break as a completed tool, summarizing the breaking change categories it detects, and answering the research question posed in the introduction. The chapter then provides guidelines for three audiences: C++ developers managing API evolution, users of Did-it-break, and developers who wish to extend or improve the tool.

5.1 Did-it-break: The Tool

Did-it-break is a static analysis tool designed to automatically detect breaking changes between two versions of a C++ library, based on their header files. Rather than comparing abstract syntax trees directly or relying on known refactoring patterns — as tools like RefDiff [13] and APIDIFF [7] do for Java — Did-it-break takes a compiler-driven approach. It generates a C++ source file that simulates realistic usage of all public APIs defined in the old header file, then attempts to compile that file against the new header. If compilation fails, the errors directly identify breaking changes.

The tool is organised into two modules, as described in Section 2.5.1. The Generate Module parses the old header file into an Abstract Syntax Tree (AST) using libclang [20] and traverses it to produce test code that exercises constructors, member functions, field accesses, virtual overrides, enum values, and const expressions. The Analysis Module then compiles this generated file against the new header using Clang, processes the resulting diagnostics, and formats them into readable output for the developer. The tool is containerised using Docker [21] to ensure a consistent C++20-compatible build environment across platforms.

Did-it-break accepts two inputs: the old header file and the new (patched) header file, along with any necessary include paths and compiler flags. Its output is a list of compiler diagnostics that correspond directly to breaking changes — changes that would cause compilation failures for any downstream developer who had been using the API as defined by the old header. Because the tool uses the compiler itself as the arbiter of compatibility, it produces no false positives: every reported error corresponds to a genuine incompatibility.

The tool is implemented in C++20, uses libclang [20] for AST traversal, Python for input parsing, and pugixml for XML generation. It follows the Google C++ Style Guide [25] and is structured to be extensible, allowing new detection rules to be added incrementally as new breaking change patterns are encountered.

5.2 Detected Breaking Change Categories

Based on the testing conducted during this project, Table 5.1 summarizes the categories of breaking changes that Did-it-break successfully detects. As noted in previous Section, this table reflects the results of the testing performed and is not intended as an exhaustive catalog of all possible breaking changes in C++. It represents the coverage achieved through the current implementation and serves as a baseline for future extension.

Element	**Breaking Changes Detected by Did-it-break**
Type	REMOVE TYPE, LOST VISIBILITY, CHANGE IN SUPERTYPE, ADD FINAL MODIFIER, CHANGE IN CONSTRUCTIBILITY, CHANGE IN DESTRUCTIBILITY, CHANGE IN AGGREGATE PROPERTY
Method	REMOVE METHOD, LOST VISIBILITY, CHANGE IN RETURN TYPE, CHANGE IN PARAMETER LIST, ADD FINAL MODIFIER, REMOVE STATIC MODIFIER, CHANGE IN OVERRIDABILITY
Field	REMOVE FIELD, LOST VISIBILITY, CHANGE IN FIELD TYPE, CHANGE IN CONST QUALIFICATION

Table 5.1: Breaking Changes Detected by Did-it-break

Comparing this to the broader catalog of breaking changes identified by APIDIFF (Table 2.1), it is clear that Did-it-break covers a substantial and practically significant subset. Notably, the compiler-driven approach naturally captures several categories that are difficult to detect through purely syntactic diffing, such as changes to aggregate properties, constructibility, and overridability. These are categories that emerge from C++'s rich type system and are tightly coupled to how APIs are actually used, rather than how they appear syntactically.

5.3 Answering the Research Question

The central research question posed in this thesis is:

How can breaking changes in C++ APIs be automatically detected?

The results of this project demonstrate that breaking changes in C++ APIs can be automatically detected by generating representative API usage code from header files and leveraging the C++ compiler to validate compatibility. This approach — implemented in Did-it-break — does not require runtime instrumentation, client test suites, or language-specific refactoring catalogs. It is grounded in the observation, supported by the literature review, that breaking changes are ultimately defined by whether existing usage patterns continue to compile: if code that was valid against the old API fails to compile against the new one, a breaking change has occurred.
The supporting implication question — what kinds of breaking changes occur in C++ and why do they matter — is addressed through the combination of the literature review and the test results chapter. Breaking changes in C++ are shown to

be frequent, multi-dimensional, and often semantic in nature. Empirical studies [1], [3], [6], [11] indicate that breaking changes affect source compatibility in 20–30% of API modifications, and that some changes may remain undiscovered for months, creating hidden risks. They arise from changes to constructibility, inheritance, overridability, type signatures, and aggregate properties, all of which can invalidate downstream code even when public interfaces appear superficially unchanged. Understanding this landscape directly informed the detection categories implemented in Did-it-break, confirming that the implication question is not a separate concern but a necessary foundation for answering the main question.

5.4 Guidelines

The following guidelines are drawn from the research, development, and testing conducted in this project. They are organized into three categories: guidelines for C++ developers managing API evolution, guidelines for users of Did-it-break, and guidelines for developers who wish to extend or improve the tool.

5.4.1 For C++ Developers Managing API Evolution

The most direct way to manage breaking changes in a C++ codebase is to use Did-it-break as part of the regular development workflow. The tool automates the detection of the most common breaking change patterns, and its output gives developers a concrete starting point for both fixing and documenting any incompatibilities introduced. The following guidelines build on this foundation, offering advice on how to get the most out of automated detection and how to handle the cases that go beyond what the tool currently covers.

5.4.1.1 Treat breaking changes as inevitable and document them explicitly

As established in Section 2.1, breaking changes are a normal and unavoidable part of software evolution. Research by Brito [6] identifies the top reasons developers introduce breaking changes as implementing new features, simplifying API complexity, improving maintainability, and fixing bugs — all of which are routine activities. Yet the same research shows that documentation of breaking changes is inconsistent and fragmented: 22% of developers use release notes, 22% opt for changelogs, 17% use JavaDoc, and others use websites, migration guides, or README files, creating confusion for downstream developers. Hora [12] further shows that knowledge of API changes is often concentrated among a small number of developers, and that deprecation messages are not always present or clear [11], making undocumented changes a persistent hidden risk.

The practical advice here is to adopt a consistent, team-wide convention for documenting breaking changes at the point of introduction, rather than retrospectively. Automated tools like Did-it-break can assist by detecting changes that might have been overlooked, but they complement rather than replace deliberate documentation. Running Did-it-break as part of a code review or pull request process ensures

that breaking changes are flagged before they reach downstream developers, giving authors the opportunity to document them immediately.

5.4.1.2 Be aware that breaking changes in C++ are often semantic, not just syntactic

The testing results in Chapter 4 and the literature reviewed in Section 2.4 together demonstrate that a significant portion of breaking changes in C++ do not involve obvious interface removals or renames. Changes to aggregate properties (Section 3.5.2), constructibility (Section 3.5.1), overridability (Section 3.7), and const qualification (Section 3.9) can all break downstream code without altering the visible public interface. Tools that rely solely on syntactic diffing — comparing function names and signatures — will miss these categories entirely, as discussed in the comparison with APIDIFF [7] and RefDiff [13] in Section 2.4.4.
Developers should therefore apply particular caution when modifying class properties that affect the type system indirectly: adding user-declared constructors to previously aggregate types, adding final specifiers, changing access specifiers of protected members, or modifying virtual function signatures. Did-it-break's generated test code specifically targets these scenarios through compile-time assertions using `std::is_constructible`, `std::is_aggregate`, `std::is_same_v`, and derived class scaffolds, providing coverage that purely syntactic approaches cannot offer.

5.4.1.3 Ensure the compilation environment matches the production environment

One of the most significant practical challenges encountered during this project, documented in Section 4.1, was that missing dependencies, unresolved include paths, and mismatched compiler flags could prevent the tool from functioning correctly or produce misleading results. This is not a limitation unique to Did-it-break; it reflects a fundamental characteristic of compilation-based analysis: its results are only as reliable as the environment in which it runs. The same issue would affect any developer compiling their code locally with incomplete dependencies.
The advice is to run breaking change detection in an environment that accurately reproduces the production build context. In this project, Docker [21] was used to address this requirement, as described in Section 3.2. For teams integrating Did-it-break into a CI/CD pipeline, the Docker container should be configured with the same compiler version, C++ standard flags, and include paths as the project's primary build system. This ensures that detected breaking changes correspond to real incompatibilities rather than environmental artifacts.

5.4.2 For Users of Did-it-break

5.4.2.1 Provide accurate include paths and compiler flags

Did-it-break requires that the compilation environment be correctly configured to process the header files under analysis. As demonstrated in Section 4.1.1, failing to provide the correct include paths results in fatal preprocessor errors that prevent

the tool from generating or compiling test code at all. Similarly, as shown in Section 4.1.2, the C++ standard flag must match the standard used by the project being analyzed — for this project, `-std=c++20` was required.

When invoking Did-it-break, users should supply all include directories that the header file depends on using the `-i` flag, and ensure that the compiler flags passed to the tool match those used in the project's own build system. The easiest way to verify this is to confirm that the old header file compiles without errors before running the detection step. If the generated file cannot be compiled against the old header, the issue lies in the environment configuration rather than in a breaking change.

5.4.2.2 Treat the tool's output as a starting point, not a complete verdict

Table 5.1 represents the breaking change categories that Did-it-break currently detects, based on the testing conducted in this project. As noted in Section 4.2.1.4, this is not an exhaustive list. The C++ language is sufficiently complex that no finite set of detection rules can cover every possible breaking change. Users should therefore treat a clean result — no compilation errors — as evidence that the most commonly modeled breaking change patterns have not been introduced, rather than as a guarantee of full backward compatibility.

In practice, this means Did-it-break is most effective as a safeguard against known, well-modeled breaking changes, and as a first pass before manual review. If the tool reports errors, those errors are reliable: every compilation failure corresponds to a genuine incompatibility. If the tool reports no errors, it is still good practice to review changes that fall outside the currently modeled categories, such as behavioral changes that preserve type signatures but alter runtime semantics. These second-order breaking changes — where the API compiles but behaves differently — are a natural direction for future extension of the tool, as discussed in Chapter 7.

5.4.2.3 Use Did-it-break as part of a regular development workflow, not only at release

The value of Did-it-break is greatest when it is used continuously rather than only before a release. Breaking changes that are detected early — at the point of a pull request or during a development iteration — are far cheaper to address than those discovered after downstream developers have already upgraded their dependencies. This principle is consistent with the Agile development approach adopted in this project [19], which emphasizes iterative validation and early feedback.

The tool is well suited for integration into CI/CD pipelines, as described in the future work section. Running Did-it-break automatically on every pull request that modifies a public header file ensures that breaking changes are never silently merged. Combined with the documentation practices described in the developer guidelines above, this creates a workflow in which breaking changes are both detected automatically and communicated clearly to all stakeholders.

5.4.3 For Developers Extending Did-it-break

5.4.3.1 Follow the existing code structure and add detection rules modularly

Did-it-break is organized around a namespace class hierarchy that mirrors the structure of the C++ code being analyzed, as described in Section 3.4.2. New detection rules should be added as new cases within the AST traversal logic in `CppApiUsageGenerator.cpp`, following the existing patterns for handling constructors, functions, fields, enums, and virtual methods. Each new detection rule should generate compile-time assertions that directly exercise the usage pattern being tested, using `std::is_constructible`, `std::is_same_v`, `std::is_convertible_v`, or derived class scaffolds as appropriate.
The Google C++ Style Guide [25] should be followed throughout, and new code should be accompanied by corresponding patch-based test cases in the `tests/` directory, following the naming convention described in Section 2.8.1.6. A patch file named `operationShouldBreak.patch` or `operationShouldNotBreak.patch` allows the test runner to automatically validate that the new detection rule behaves as expected. Adding both a positive and a negative test case for each new rule ensures that the rule detects breaking changes when present and does not produce false positives when absent.

5.4.3.2 Prioritize extending coverage for second-order and behavioral breaking changes

The current implementation of Did-it-break detects first-order breaking changes: changes that cause compilation failures. As discussed in Section 4.2.1.4, the space of possible breaking changes extends beyond compilation failures to include behavioral changes that preserve type signatures but alter runtime semantics — for example, a function that previously returned a sorted collection but no longer does, or a constructor that previously initialized a field to zero but now leaves it uninitialized. These are cases where the API compiles but behaves differently, and they represent a natural direction for future extension.
Extending Did-it-break to detect such second-order breaking changes would require moving beyond pure compilation checking toward runtime validation or deeper semantic analysis. One approach, inspired by Type Regression Testing [9], would be to generate not only compilation tests but also runtime assertions based on observed behavior in the old version. Another direction would be to incorporate output type checking: verifying not only that a function can be called with the expected arguments, but that its return value satisfies the same type constraints and semantic contracts as before. Developers extending the tool should treat Table 5.1 as a baseline and systematically work through the broader APIDIFF catalog (Table 2.1) as a reference for uncovered categories.

5.4.3.3 Address the current IDE and workflow limitations

As noted in the future work section, Did-it-break currently lacks Language Server Protocol (LSP) support [32], which limits IDE integration and makes the develop-

ment experience less productive than it could be. Each test currently requires a full rebuild within the Docker [21] environment, which slows down iteration. Developers extending the tool should prioritize integrating LSP support to enable syntax highlighting, autocompletion, and incremental compilation within the Docker context, as this will significantly improve the experience for future contributors.
Additionally, the tool currently operates as a standalone command-line utility. Packaging it for easier installation and providing clear documentation for the `-i` include path flags and compiler flag options would lower the barrier for new users and make integration into external build systems more straightforward. The Docker setup described in Section 3.2 provides a solid foundation, but a simplified entry point script and example CI/CD configuration would make the tool more accessible to teams beyond the original **king** context.

6

Discussion

6.1 Compilation-based Detection and Its Trade-offs

The core design decision in Did-it-break — using the C++ compiler as the arbiter of breaking changes — provides both its greatest strength and its primary limitation. On the strength side, every reported breaking change corresponds directly to a compilation failure that a downstream developer would encounter, meaning the tool produces no false positives. On the limitation side, breaking changes that do not manifest as compilation failures — behavioral changes, semantic shifts, or second-order incompatibilities — fall outside the tool's current detection scope. This trade-off is inherent to the approach and is not a flaw so much as a design boundary that future work should extend.

6.2 First-order vs. Second-order Breaking Changes

Did-it-break currently detects what can be termed first-order breaking changes: modifications to a C++ API that cause code, which previously compiled against the old header, to fail compilation against the new one. However, a broader class of breaking changes exists beyond this. A second-order breaking change is one where the API still compiles but the return value or output behavior has changed in a way that breaks downstream logic — for example, a function that previously returned values in sorted order that now returns them unsorted, or a constructor that previously zero-initialized a field that now leaves it unspecified. A more unusual but real example is a function whose return type changes to `std::variant`, allowing it to return one of multiple types. While this is rare and generally considered poor API design — since having a function return two different types is confusing and goes against common software conventions — it is something C++ allows, and it could silently break code that assumes a single fixed return type. The current implementation does not check for any of these levels. EExtending Did-it-break to cover second-order changes would require more careful thought about how to use compile-time testing tools to detect these kinds of breaking changes, this is an important direction for future development.

6.3 Generalizability Beyond C++

Did-it-break was designed specifically for C++ and relies on libclang [20] for AST traversal and Clang as the compilation backend. However, the underlying approach — generating usage code from an old interface definition and compiling it against a new one — is not inherently C++-specific. A similar strategy could in principle be applied to any compiled language with a stable compiler interface, such as Rust or Go. The degree to which this project's approach generalizes depends heavily on the type system of the target language. C++ is particularly amenable because its header files provide a clean, well-defined interface boundary, and because the compiler enforces type compatibility strictly at compile time. However, for languages where types are checked at runtime rather than compile time — such as Python — a compile-time only approach is no longer sufficient. In these cases, detection tools would need to incorporate runtime testing as well, making the problem significantly more complex. This discussion suggests that the compiler-driven approach is most directly transferable to statically typed compiled languages, while dynamically typed languages require a broader detection strategy that goes beyond what compile-time checking alone can offer.

6.4 Limitations and Challenges

Throughout the project, several challenges and limitations were encountered at different stages:

- **Dependency and Build Environment:** When implementing the project, reconstructing the original compilation context was often complex due to missing dependencies, unresolved include paths, and incompatible compiler flags. This issue was ultimately resolved by using Docker [21].
- **C++ Code Structure:** During development, nested namespace structures made obtaining fully qualified names very challenging. To address this, we decided to abandon the idea to get the `getQualifiedName` and mirrored the namespace nesting in the generated test code instead, as described in Section 3.4.
- **Detection Coverage:** While developing the tool, it was observed that Did-it-break reliably detects explicitly modeled breaking changes. However, the set of possible API modifications in C++ is inherently open-ended, and breaking changes that do not match predefined usage patterns may remain undetected. This highlights the need for continuous extension and refinement of detection rules to maintain comprehensive coverage.
- **Tool Implementation and Scalability:** When trying to test the tool in a real environment, we found that applying it to a large library with thousands of files posed performance and scalability challenges.

Despite these challenges, the project successfully demonstrated the feasibility of compilation-based breaking change detection and provided a foundation for further extension and adoption in real-world C++ development workflows.

6.5 Performance Considerations

From a performance perspective, the testing results indicate that the primary cost of the proposed approach lies in the compilation rather than the complexity of the analysis. Since the tool relies on standard compilation processes and compile-time type traits, its overhead scales with project size and dependency depth. However, this cost is consistent with real-world build processes and does not introduce additional runtime overhead for end users, as described in the namespace section of Phase 2. This project leverages C++20 *modules*, as discussed in the code review section of Phase 1, to further mitigate compilation overhead. Traditionally, C++ relies heavily on header files to manage declarations and include reusable code across different `.cpp` files. Each `#include <FileName.h>` directive essentially copies the content of the header into the compilation unit, which can lead to repeated compilation of the same declarations. In contrast, module interface units (`.ixx` files in this project) are compiled into a binary representation once, which can then be used across multiple compilation units. This approach reduces redundant work and can significantly improve compile times, especially for large projects with deep dependency trees. Importantly, the observed compilation cost is justified by the accuracy of detection. By leveraging the compiler as the final authority on correctness, Did-it-break avoids false positives that often occur in purely static or rule-based approaches. Consequently, the trade-off between performance and detection reliability favors practical applicability, particularly in continuous integration and pre-release validation workflows.

6.6 Ethical Considerations

Ethical considerations are fundamental pillars that underpin the integrity and credibility of this research [26]. Several ethical considerations are relevant to this project, as introduced in Section 2.7.1. Regarding data privacy, the development and testing of Did-it-break involved access to internal C++ codebases from King **king**. All company code and associated data were treated as confidential throughout the project and are not disclosed in this report. The header files used in published test cases are either synthetic or anonymized to preserve proprietary information and intellectual property rights. Regarding informed consent, the project involved collaboration with developers at King, particularly the USDK team. Any feedback gathered from team members about requirements, tool usability, or development priorities was collected with their knowledge and in the context of a professional collaboration, with no personal data collected or stored. Regarding transparent documentation, the methodology, implementation decisions, test cases, and results are documented in this report to the extent permitted by confidentiality constraints. The goal is to ensure that the approach is sufficiently described for replication and extension by other researchers or developers, in keeping with the principles of transparent and reproducible research.

6.7 Suggestions for Improvement and Future Work

Based on the experience gained during the development and testing of Did-it-break, several directions for improvement and future work have been identified:

- **Improve the Development Workflow:** Currently, the project does not use the Language Server Protocol (LSP) [32], which results in limited IDE support, such as the absence of syntax highlighting, keyword differentiation, and variable colorization. This negatively affects code readability and developer productivity. Furthermore, real-time testing is not supported, as each test execution requires a full rebuild within the Docker [21] environment, leading to inefficient development iterations. Integrating LSP would enable the IDE to communicate more effectively with the Docker-based build system, allowing features such as syntax highlighting, autocompletion, intelligent code suggestions, and breakpoint debugging without requiring a complete rebuild for every test.
- **Enhance Breaking Change Detection Capabilities:** The current version of Did-it-break detects a predefined and limited set of breaking changes. However, breaking changes in APIs can manifest in many forms and are not bounded to a fixed set of patterns. Future work could extend the detection logic to cover a broader range of breaking changes, including more subtle API modifications such as semantic changes, behavioral differences, or changes in implicit contracts. This could be achieved by incorporating additional static analysis techniques [16], rule-based heuristics, or learning-based approaches.
- **Integrate the Tool into the CI/CD Workflow:** At present, Did-it-break is executed as a standalone tool. A natural extension would be its integration into existing Continuous Integration and Continuous Deployment (CI/CD) pipelines. By embedding the tool into the development workflow, API-breaking changes could be detected automatically during pull request validation or before deployment. This would allow developers to identify and address breaking changes earlier in the development cycle, improving software reliability and reducing the risk of introducing regressions into production environments.

7
Conclusion

This thesis investigated the problem of detecting breaking changes in C++ and demonstrated that such changes are both common and difficult to identify using purely syntactic approaches. Through empirical evidence and a comprehensive literature review, it was shown that breaking changes frequently arise from routine API evolution — including changes to function signatures, type definitions, and inheritance structures — and can have significant downstream impacts on client code [1], [3], [6], [11], [12], making early and reliable detection a critical concern for maintainable software development.

To address this challenge, Did-it-break was proposed and implemented as a compilation-based detection framework. By generating representative API usage code from existing header files and leveraging the C++ compiler [20] as the final authority on compatibility, the approach directly validates whether API changes disrupt real usage scenarios. The evaluation results confirm that this method effectively detects a broad class of practical breaking changes at the type, method, and field levels, as summarized in Table 5.1.

The central research question posed in this thesis — how breaking changes in C++ APIs can be automatically detected — is answered by the tool itself: by generating representative usage code from old header files and compiling it against new ones, Did-it-break reliably identifies breaking changes as compilation failures with no false positives. The supporting implication question, concerning what kinds of breaking changes occur in C++ and why they matter, is addressed through the literature review in Section 2.1 and the test results in Chapter 4, and directly informed the detection categories implemented in the tool.

While the current implementation has limitations in terms of detection coverage, scalability, and workflow integration, these constraints primarily reflect the inherent complexity of the C++ language rather than fundamental shortcomings of the proposed approach. The results demonstrate that compilation-based detection provides a robust and precise foundation upon which more comprehensive and scalable solutions can be built, with second-order behavioral breaking changes and CI/CD integration identified as the most impactful directions for future work.

Overall, this work contributes both a functional tool and a set of practical guidelines for automated breaking change detection in C++. It shows that a detection method based on compiler validation and generated usage scenarios is both feasible and effective, and that realistic usage-based verification is essential for meaningful detection. The insights and framework presented in this thesis lay a solid foundation for future research and tool development aimed at improving API stability and supporting sustainable evolution in large-scale C++ software systems.

Bibliography

[1] G. Brito, A. Hora, M. T. Valente, and R. Robbes, "Do developers deprecate apis with replacement messages? a large-scale analysis on java systems", in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, 2016, pp. 360–369. DOI: 10.1109/SANER.2016.99.

[2] *King.com*, www.king.com. [Online]. Available: https://www.king.com/.

[3] L. Xavier, A. Brito, A. Hora, and M. T. Valente, "Historical and impact analysis of api breaking changes: A large-scale study", in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 138–147. DOI: 10.1109/SANER.2017.7884616.

[4] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning versus breaking changes: A study of the maven repository", in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, 2014, pp. 215–224. DOI: 10.1109/SCAM.2014.30.

[5] *Mvn repository*. [Online]. Available: https://mvnrepository.com/.

[6] A. Brito, M. T. Valente, L. Xavier, and A. Hora, "You broke my code: Understanding the motivations for breaking changes in apis", *Empirical Software Engineering*, vol. 25, pp. 1458–1492, Nov. 2019. DOI: 10.1007/s10664-019-09756-z. (visited on 01/29/2021).

[7] A. Brito, L. Xavier, A. Hora, and M. T. Valente, "Apidiff: Detecting api breaking changes", in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 507–511. DOI: 10.1109/SANER.2018.8330249.

[8] X. Du and J. Ma, "Aexpy: Detecting api breaking changes in python packages", in *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*, 2022, pp. 470–481. DOI: 10.1109/ISSRE55969.2022.00052.

[9] G. Mezzetti, A. Møller, and M. T. Torp, "Type Regression Testing to Detect Breaking Changes in Node.js Libraries", in *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, T. Millstein, Ed., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 109, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018, 7:1–7:24, ISBN: 978-3-95977-079-8. DOI: 10.4230/LIPIcs.ECOOP.2018.7. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2018.7.

[10] D. Konstantopoulos, J. Marien, M. Pinkerton, and E. Braude, "Best principles in the design of shared software", in *2009 33rd Annual IEEE International Computer Software and Applications Conference*, vol. 2, 2009, pp. 287–292. DOI: 10.1109/COMPSAC.2009.151.

[11] R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to api deprecation? the case of a smalltalk ecosystem", in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12, Cary, North Carolina: Association for Computing Machinery, 2012, ISBN: 9781450316149. DOI: 10.1145/2393596.2393662. [Online]. Available: https://doi.org/10.1145/2393596.2393662.

[12] A. Hora, R. Robbes, N. Anquetil, A. Etien, S. Ducasse, and M. Tulio Valente, "How do developers react to api evolution? the pharo ecosystem case", in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 251–260. DOI: 10.1109/ICSM.2015.7332471.

[13] D. Silva and M. T. Valente, "Refdiff: Detecting refactorings in version histories", in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017, pp. 269–279. DOI: 10.1109/MSR.2017.14.

[14] D. Kong, J. Liu, C. Ni, D. Y.-A. Lo, and L. Bao, "More effective javascript breaking change detection via dynamic object relation graph", *Proceedings of the ACM on Software Engineering*, vol. 2, no. ISSTA, pp. 2340–2361, 2025. DOI: 10.1145/3728980.

[15] N. Tsantalis, V. Guana, E. Stroulia, and A. Hindle, "A multidimensional empirical study on refactoring activity", in *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, ser. CASCON '13, Ontario, Canada: IBM Corp., 2013, pp. 132–146.

[16] B. Chess and J. West, *Secure programming with static analysis*. Pearson Education, 2007.

[17] N. Tsantalis, V. Guana, E. Stroulia, and A. Hindle, "A multidimensional empirical study on refactoring activity.", in *CASCON*, 2013, pp. 132–146.

[18] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of github contributors", in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016, Seattle, WA, USA: Association for Computing Machinery, 2016, pp. 858–870, ISBN: 9781450342186. DOI: 10.1145/2950290.2950305. [Online]. Available: https://doi.org/10.1145/2950290.2950305.

[19] J. Highsmith and A. Cockburn, "Agile software development: The business of innovation", *Computer*, vol. 34, no. 9, pp. 120–127, 2001. DOI: 10.1109/2.947100.

[20] *Clang: Libclang: C interface to clang*, clang.llvm.org. [Online]. Available: https://clang.llvm.org/doxygen/group__CINDEX.html#details (visited on 12/03/2023).

[21] Docker, *Enterprise application container platform | docker*, Docker, 2018. [Online]. Available: https://www.docker.com/.

[22] V. K. Chauhan, "Smoke testing", *Int. J. Sci. Res. Publ*, vol. 4, no. 1, pp. 2250–3153, 2014.

[23] M. Olan, "Unit testing: Test early, test often", *Journal of Computing Sciences in Colleges*, vol. 19, no. 2, pp. 319–328, 2003.

[24] R. Watson, "Quantitative research", *Nursing standard*, vol. 29, no. 31, 2015.

[25] *Google c++ style guide*. [Online]. Available: `https://google.github.io/styleguide/cppguide.html`.

[26] M. Mohseni, B. Liebold, and D. Pietschmann, *Game research methods [lankoski björk]*, Apr. 2015.

[27] *Choosing the right interface for your application*. [Online]. Available: `https://clang.llvm.org/docs/Tooling.html`.

[28] *Xcode release notes15.3*. [Online]. Available: `https://developer.apple.com/documentation/xcode-release-notes/xcode-15_3-release-notes#C++-Standard-Library`.

[29] *Aggregate definition from cpp reference*. [Online]. Available: `https://en.cppreference.com/w/cpp/language/aggregate_initialization`.

[30] *Aggregate initialization*, Accessed: 2026-02-02, cppreference.com — C++ reference. [Online]. Available: `https://en.cppreference.com/w/cpp/language/aggregate_initialization`.

[31] *Designated initializer*. [Online]. Available: `https://en.cppreference.com/w/cpp/language/aggregate_initialization#Designated_initializers`.

[32] *Language server protocol*, Accessed: 2026-02-04, Microsoft Learn. [Online]. Available: `https://learn.microsoft.com/en-us/visualstudio/extensibility/language-server-protocol?view=visualstudio`.