Code Compass: A Software change impact analysis tool using vector search.

Dr.T.Jalaja[1], Dr.T.Adilakshmi[2], K.Abhinav[3] ,M.Rakesh[4]

[1,2,3]Vasavi College of Engineering, Hyderabad, India

Email: jalaja.t@staff.vce.ac.in[1],t_adilakshmi@staff.vce.ac.in[2], kabhinav2007@gmail.com[3], rakeshmadugundu@gmail.com[4]

Abstract

Introduction: In modern software development, managing code dependencies efficiently is crucial for ensuring system stability and seamless implementation of changes. Traditional static analysis tools often miss semantic relationships and complex interactions across codebases. Code Compass addresses this challenge by introducing a novel approach combining code structure parsing with dual representation techniques and vector similarity search. By representing code components through both natural language descriptions and code-specific embeddings, the system captures both semantic intent and implementation details. This dual-modal approach enables developers to identify potential impact points in multi-language codebases.

Objectives: The objective of this project is to develop an AI-driven system that simplifies and enhances the management of software changes by providing actionable insights, comprehensive impact analysis, and automated documentation. By leveraging techniques like static and dynamic code analysis, dependency mapping, and natural language generation, the system aims to empower development teams to make informed decisions, mitigate risks, and streamline workflows, ultimately ensuring higher software quality and faster delivery.

Methods: The project involves development of a comprehensive code analysis system built on four key components. First, source code parsing leverages Tree-sitter to extract structured information from multiple programming languages, capturing hierarchical code structures with rich contextual metadata. Second, dual representation generation transforms each code structure into complementary text and code representations, converting naming conventions to human-readable form while preserving original syntax. Third, specialized embedding models generate vectors for each representation type—MiniLM-L6-v2 for text and jina-embeddings-v2-base-code for code snippets. Finally, Qdrant vector search implements Reciprocal Rank Fusion to combine results from both embedding spaces, enabling identification of both semantic relationships and implementation similarities across codebases.

Results: We conducted tests on a diverse codebase incorporating Python, JavaScript, TypeScript, and Java components. Code Compass demonstrated strong performance across all supported languages. The dual embedding approach proved effective across all languages, with Python showing the highest accuracy due to its more explicit import structure. Analysis of false negatives revealed limitations primarily with dynamic language

features such as Python's metaprogramming, JavaScript's prototype-based inheritance, and dynamic module loading patterns.

Conclusions: Code Compass enhances software change management by automating impact analysis through dual-modal code representation and vector similarity search. It improves dependency tracking accuracy, minimizes regression risks, and integrates seamlessly into development workflows. By identifying semantic relationships beyond explicit references, it provides developers with comprehensive insights into potential change impacts, ensuring more efficient and reliable software updates across multi-language projects.

Keywords: Vector search, AST parsing, Similarity scoring, Code embeddings

1.Introduction

In the rapidly evolving landscape of software development, managing code changes effectively has become essential for ensuring system stability and seamless deployment. The increasing complexity of modern applications, coupled with diverse programming languages and intricate dependency networks, means that even small modifications can have significant ripple effects. These changes, whether they are part of bug fixes, feature additions, or general refactoring, can unintentionally disrupt existing functionality or introduce new issues. Traditional methods for tracking these dependencies rely on static analysis tools or developer intuition, which often miss semantic relationships and complex interactions across codebases, making it difficult to predict the true impact of changes accurately [1].

To address these challenges, Code Compass has been developed to streamline and enhance software change impact analysis. By leveraging dual representation techniques, vector embeddings, and comprehensive code structure parsing, the system provides developers with a deeper understanding of how modifications propagate across projects. This ensures that development teams receive accurate insights into potentially affected components, even when traditional static analysis would miss these connections. The system's ability to uncover hidden relationships, those not explicitly mentioned or visible in the code, helps mitigate the risk of overlooked dependencies, offering a more comprehensive analysis of code changes than conventional methods.

One of the key innovations of Code Compass is its dual modal approach to code representation, capturing both semantic intent through natural language descriptions and implementation details through code specific embeddings. This combination of natural language understanding and structural analysis ensures that the system not only captures syntactic relationships but also grasps the deeper, more meaningful interactions between components. By doing so, it identifies relationships beyond explicit references, revealing components that share similar patterns or functional purposes across multiple programming languages including Python, JavaScript, TypeScript, and Java. This flexibility is especially crucial in polyglot environments, where multiple languages and technologies are often integrated to build modern applications.

By providing clear visualization of code dependencies and potential impact points, Code Compass enables teams to make informed decisions while minimizing regression risks. This capability is especially valuable during complex refactoring operations, feature additions, bug fixes, and cross language integration efforts where understanding the full scope of impact is critical. The system's ability to visualize the interplay between different parts of a codebase ensures that developers can anticipate unintended consequences and mitigate risks early in the development process. This foresight is invaluable in large scale projects where tracking every change manually would be time consuming and error prone.

Beyond improving technical accuracy, the system enhances developer productivity by reducing the time spent manually tracing dependencies. The high precision vector search methodology ensures that developers can quickly identify relevant code components without wading through false positives. This increased efficiency not only speeds up development but also reduces the cognitive load on developers, allowing them to focus on more meaningful tasks. Additionally, by using AI powered techniques to automatically track and assess code dependencies, Code Compass provides a more scalable solution for large teams working on complex systems. As a result, the tool helps prevent bottlenecks that might otherwise slow down development or introduce errors into the system.

By addressing the multifaceted challenges of code dependency tracking, Code Compass not only enhances code quality but also supports faster, more confident software delivery. It provides real time insights into the potential effects of proposed changes, enabling developers to act with greater confidence and reducing the likelihood of introducing bugs or performance issues. Furthermore, the tool’s capacity for integrating with continuous integration continuous deployment (CI/CD) pipelines means that it can deliver its analysis at every stage of the development lifecycle, from initial coding to final deployment. This ensures that code quality remains high throughout the process, providing an ongoing safeguard against regression and other common issues.

It ensures that development teams can maintain high standards of stability while adapting to evolving requirements in modern, fast paced development environments. By offering a tool that can continually adapt to new coding practices, evolving technologies, and changing business needs, Code Compass helps development teams stay ahead of the curve. This adaptability, combined with its deep understanding of code relationships and potential impacts, makes it an invaluable resource for organizations seeking to improve their software development processes.

1.1 Objectives

The primary objective of Code Compass is to develop a comprehensive and AI powered software engineering tool that simplifies and significantly enhances the process of software change management. In the ever evolving landscape of software development, code changes whether minor bug fixes, refactoring, or large scale feature integrations can introduce unintended side effects that affect system performance, introduce bugs, or disrupt existing functionality. Traditional static analysis tools often fall short in understanding the full scope of these effects, particularly in large, distributed, or polyglot codebases. Code Compass addresses this gap by offering actionable insights into code dependencies and potential impact zones before any modification is made.

At the core of Code Compass lies an advanced dual embedding architecture combined with semantic vector search and deep structural code analysis. This multifaceted strategy allows the system to capture both the syntactic structure and semantic intent of source code elements such as functions, classes, and modules across various programming languages. These elements are converted into dense vector representations using specialized machine learning models, enabling the system to assess similarity and relevance not only based on surface level syntax but also on deeper behavioral and functional characteristics. By storing and indexing these embeddings in a high performance vector database like Qdrant, Code Compass supports high speed querying and real time comparison of code fragments against natural language inputs or other code snippets.

Through this architecture, Code Compass enables precise and meaningful impact analysis, a critical component in modern software lifecycle management. When a developer proposes a code change, the system analyzes it in the context of the surrounding codebase, automatically identifying related components that may be affected directly or indirectly by the change. This identification is not limited to explicit imports or static references, but also includes semantically similar implementations, usage patterns, and contextual clues that traditional analysis might overlook. As a result, development teams gain early visibility into possible side effects, enabling them to plan code modifications with greater confidence and reduced risk.

The system's ability to automatically detect semantically related components is especially valuable in large scale projects where dependencies are complex and documentation may be lacking or outdated. In such environments, unanticipated ripple effects are a common challenge. Code Compass mitigates this issue by surfacing hidden dependencies and usage links that could compromise the stability of the application if left unnoticed. This not only streamlines debugging but also significantly cuts down on the trial and error cycles that typically accompany regression testing and manual inspection.

By leveraging its dual representation strategy which fuses both the implementation level structure and the intent level semantics of code Code Compass surpasses the limitations of conventional code analysis tools. Its rich metadata tracking, AST based parsing, and vector based similarity engine enable a broader and more accurate dependency mapping across multiple languages and architectural layers. This capability positions the system as an indispensable assistant in the software development pipeline, particularly during high stakes operations like refactoring, version migrations, and feature rollouts.

In summary, Code Compass empowers developers and project teams to make more informed decisions, reduce debugging and testing overhead, and ensure greater code reliability and maintainability over time. The tool provides a forward thinking approach to software evolution, combining the power of artificial intelligence with practical, real world needs of developers working within large, distributed codebases.

2.Proposed Methodology

System Overview

Code Compass operates through four primary stages. First, source code parsing extracts code structures and their relationships from the target codebase. Second, dual representation generation creates both natural language and code-specific representations of each structure. Third, embedding generation transforms these representations into vector embeddings. Finally, vector search utilizes these embeddings to identify potentially impacted code components based on similarity measures.

Source Code Parsing

We leverage Tree-sitter, a parser generator tool and incremental parsing library, to extract structured information from source code. Our implementation supports Python, JavaScript, TypeScript, and Java, with language-specific parsing strategies for each. For each supported language, we define specialized node type sets to identify function definitions, class definitions, and function calls with appropriate syntax patterns.

The parser recursively traverses the Abstract Syntax Tree (AST) to extract code structure definitions with full contextual metadata, call sites with caller-callee relationships, and code blocks with proper scope boundaries [3]. This approach enables us to capture the hierarchical structure of code, including nested definitions and their relationships.

Each extracted code structure is enriched with metadata including file path and module information, line number ranges, containing structures (e.g., class for methods), function signatures, and complete code snippets. This rich metadata provides critical context for later analysis and presentation of results.

Dual Representation Generation

We transform each code structure into two complementary representations to capture both semantic intent and implementation details. For the text representation, we apply several transformations beginning with converting naming conventions (camelCase, snake_case) to human-readable form using the inflection library. We then extract and incorporate available docstrings, include structural context such as module, file, and containing class information, and generate a unified text description that captures the semantic purpose of the code.

For the code representation, we preserve the original code snippet with full syntactic structure, including the complete function or class definition, all contained statements and expressions, and original formatting and comments. This preservation ensures that code-specific patterns and implementation details remain available for analysis.

Embedding Generation

We apply a dual-embedding approach, using separate embedding models optimized for different modalities. For text representations, we use the sentence-transformers/all-MiniLM-L6-v2 model, which produces 384-dimensional embeddings optimized for semantic similarity of natural language. This model effectively captures the functional intent and purpose of code structures as expressed in natural language.

For code representations, we use the jina-embeddings-v2-base-code model, which produces 768-dimensional embeddings specifically trained on code corpora [2]. This model captures programming patterns, API usage, and code syntax that might not be apparent in the natural language description. By using specialized models for each modality, we ensure that both semantic meaning and implementation details are properly encoded in the vector space.

Vector Search Implementation

We implement vector search using Qdrant, a vector similarity search engine that supports multi-vector indexing and complex search operations[4]. Each code structure is indexed with both embeddings: the text embedding in the "text" vector space and the code embedding in the "code" vector space. This dual indexing enables searches targeting either semantic meaning or code implementation patterns, depending on the analysis needs.

To combine results from both embedding spaces, we implement Reciprocal Rank Fusion (RRF), which queries both vector spaces independently and then combines rankings with preference for items ranking highly in both spaces. This approach balances semantic similarity with implementation similarity, providing more robust results than either embedding alone could achieve.

Our search implementation supports grouping results by modules to identify cross-module impacts, filtering by code type to focus on specific kinds of structures, and limiting results to manage information overload. These features make the tool practical for real world analysis scenarios where targeted results are more valuable than exhaustive ones.

Change Impact Analysis Process

Given a target code element (function, method, or class), Code Compass generates text and code embeddings for the target element and performs dual-vector search using RRF fusion. It then ranks potential impact points by similarity score and groups results by module to highlight cross module dependencies. Finally, it presents ranked results with relevant metadata and similarity scores to the user.

This methodology enables identification of semantic relationships and potential impact points that traditional static analysis tools might miss, particularly in codebases with dynamic language features or complex dependency patterns. By combining syntactic analysis with semantic understanding, Code Compass provides a more comprehensive view of code dependencies than either approach alone could achieve.

3.Proposed Algorithm

Algorithm 1 Code Understanding and Search System Based on AST and Vector Embeddings

```
Require: URL – GitHub repository URL
Require: Languages – Supported languages (Python, JavaScript, TypeScript,
    Java)
Require: Model – Pre-trained embedding model
Require: Qdrant – Vector database for indexing embeddings
Ensure: Searchable, indexed semantic representations of source code
Clone URL using Git
Extract all source files matching Languages
for each source_file in repository do
    Parse source_file using Tree-sitter
    Extract:
    Function definitions
    Class definitions
    Function calls
    Class instantiations
    Store structured info in JSONL
for each code_chunk in JSONL do
    Generate embedding ← Model(code_chunk)
    Attach metadata ← {file path, language, symbol name, line number}
    Store in Qdrant ← (embedding, metadata)
while user query received do
    Embed query ← Model(user_query)
    Retrieve top-k vectors from Qdrant using cosine similarity
    Rank results ← (semantic score, usage frequency)
    Display results ← {function/class, file, code snippet}
```

Fig. 1. Pseudo code

See Fig. 1 for the pseudo code for our proposed system. We propose a system for semantic code understanding and retrieval, capable of parsing source code, extracting key elements, embedding them into a vector space, and performing similarity search using natural language queries. The goal is to enable intuitive, accurate, and context-aware code search, especially across large and complex codebases. The core of our method relies on Abstract Syntax Tree (AST) based code analysis, embedding generation using pre-trained language models, and efficient vector similarity search using the Qdrant engine.

The system begins by cloning a GitHub repository and extracting source files based on supported programming languages (e.g., Python, Java, JavaScript, TypeScript). For each source file, we use Tree-sitter to parse the file into an Abstract Syntax Tree (AST), which allows us to extract well-structured representations of code components such as function definitions, class declarations, function calls, and class instantiations. These extracted elements are then saved in a structured JSONL format with additional metadata like file path, symbol name, and line number, offering a rich dataset for downstream semantic processing.

Next, we use a pre-trained code/text embedding model (e.g., jina-embeddings-v2-base-code) to generate dense vector representations for each extracted item. These vectors, which capture the semantic meaning of code irrespective of exact syntax, are then stored in Qdrant a high-performance vector database along with their corresponding metadata. When a user issues a natural language query, the system converts the query into a vector using the same embedding model and performs a similarity search within Qdrant. The nearest neighbors are returned as results, ranked by semantic closeness. Each result includes the original code snippet, the type of element (e.g., function or class), and usage context, enabling users to quickly locate and understand relevant parts of the codebase. This approach makes the search more intuitive and context-aware compared to traditional keyword-based solutions.

4.Results

We conducted tests on a diverse codebase incorporating Python, JavaScript, TypeScript, and Java components. Code Compass demonstrated strong performance across all supported languages:

Metric	Value
True Positives (TP)	51
False Positives (FP)	3
False Negative (FN)	7
Accuracy	0.944
Recall	0.879
F1 Score	0.911

The dual embedding approach proved effective across all languages, with Python showing the highest accuracy due to its more explicit import structure. JavaScript and TypeScript exhibited similar performance profiles, while Java's static typing provided strong structural information for impact analysis.

Analysis of false negatives revealed limitations primarily with dynamic language features such as Python's metaprogramming, JavaScript's prototype-based inheritance, and dynamic module loading patterns. These edge cases highlight areas for future refinement while confirming the approach's robustness for typical development scenarios. The results validate our multi-language parsing strategy and demonstrate that Code Compass provides reliable impact analysis across heterogeneous codebases, making it particularly valuable for projects employing multiple programming languages.

```
--------------------------------------------------
Name:         detect_encoding
Code Type:    FunctionCall
Signature:    detect_encoding(csv_file_path)
Docstring:    None
Lines:        22 - 22
Score:        0.8333
Module:       dataset
File Path:    ./SwiftFashion/backend/dataset/downloadImages.py
File Name:    downloadImages.py
Snippet:      detect_encoding(csv_file_path)
Usage Count:  None
--------------------------------------------------
Name:         detect_encoding
Code Type:    Function
Signature:    def detect_encoding(file_path):
Docstring:    None
Lines:        11 - 14
Score:        0.8333
Module:       dataset
File Path:    ./SwiftFashion/backend/dataset/downloadImages.py
File Name:    downloadImages.py
Snippet:      def detect_encoding(file_path):
Usage Count:  1
```

Fig. 2. Query result for a function search

```
--------------------------------------------------
Name:         PineconeConnector
Code Type:    Class
Signature:    class PineconeConnector:
Docstring:    None
Lines:        12 - 23
Score:        1.0000
Module:       others
File Path:    ./SwiftFashion/backend/PineconeLocal/others/Modular.py
File Name:    Modular.py
Snippet:      class PineconeConnector:
Usage Count:  1
--------------------------------------------------
Name:         PineconeConnector
Code Type:    ClassInstantiation
Signature:    PineconeConnector(api_key, environment)
Docstring:    None
Lines:        54 - 54
Score:        0.5833
Module:       others
File Path:    ./SwiftFashion/backend/PineconeLocal/others/Modular.py
File Name:    Modular.py
Snippet:      PineconeConnector(api_key, environment)
Usage Count:  None
--------------------------------------------------
```

Fig. 3. Query result for a class search

6. Conclusion

The future scope of the proposed AI powered system lies in its potential to evolve and expand to meet the increasing complexity of software development. As the demands of modern software projects grow, the need for tools that can efficiently manage code changes, track dependencies, and predict impacts becomes even more critical. One of the key areas for improvement is the incorporation of advanced machine learning techniques, such as reinforcement learning, to further enhance the system's ability to predict the outcomes of code changes in dynamic and rapidly evolving codebases. By utilizing reinforcement learning, the system could not only understand the immediate impact of a change but also continuously learn from the ongoing modifications, making it more adaptive to changing codebases over time.

As software development practices shift toward continuous integration and delivery, the system could be integrated with real time build and deployment pipelines, allowing it to assess code changes as they are being written and tested. This integration would enable the system to detect potential issues and provide feedback almost immediately, reducing the time developers spend identifying and resolving problems later in the process. Additionally, incorporating feedback from real world deployments could enable the system to refine its predictions, improving its accuracy over time. This feedback loop could include userreported bugs, performance degradation, or even customer satisfaction metrics, giving the system the ability to adjust and improve its decision-making based on real-world results.

The use of more granular data, such as performance metrics or user feedback, could be integrated to provide a comprehensive risk and impact assessment, taking into account not just the code structure but also the user experience and system performance. By understanding the correlation between code changes and real-world user interactions, the system could offer a more nuanced prediction, factoring in aspects like load time, error rates, or user engagement. These additional insights could help development teams prioritize changes based on their potential business impact and not just technical feasibility, allowing for a more efficient allocation of resources.

In addition to these technical advancements, the system could be extended to support multiple programming languages and frameworks, allowing it to provide insights across a broader range of projects. This multi-language support would allow teams working in polyglot environments to leverage the system's capabilities, offering consistent analysis across various codebases and technologies. By incorporating support for domain-specific languages and emerging technologies, the system would be able to handle a variety of use cases across different industries, from web development to embedded systems. This flexibility would make the system indispensable in diverse software environments, including those dealing with specialized or niche languages, ensuring that it remains relevant as the software landscape continues to evolve.

Furthermore, as AI and natural language generation techniques continue to evolve, the system could improve its documentation capabilities, providing more contextually aware and user friendly reports. The ability to generate detailed yet easily understandable documentation is crucial for large development teams, and improved natural language generation could offer both technical and non-technical users clear, actionable insights. This would allow non-technical stakeholders to understand the impact of code changes more easily, enabling better collaboration between developers, project managers, and business leaders. The system could offer explanations that break down complex technical concepts into simpler terms, helping decision-makers make more informed choices.

Ultimately, the system's future lies in its ability to adapt to the fast-paced and ever changing nature of modern software development, providing an invaluable tool for managing complexity, enhancing decision-making, and driving efficiency across the entire software lifecycle. By continuously evolving to address new challenges, integrating with the latest technologies, and streamlining collaboration across teams, the system has the potential to become a central component of the software development ecosystem, supporting developers, project managers, and business leaders in navigating the complexities of modern software systems. Its ongoing evolution will ensure that it remains a vital asset in an increasingly dynamic development environment, ensuring that teams can keep pace with change while delivering high-quality software.

7.References

[1] S. Bohner and R. Arnold, Software Change Impact Analysis, CA, USA:IEEE Computer Society Press, 1996.

[2] Günther, M., Ong, J., Mohr, I., Abdessalem, A., Abel, T., Akram, M. K., Guzman, S., Mastrapas, G., Sturua, S., Wang, B., Werk, M., Wang, N., & Xiao, H. (2024). Jina Embeddings 2: 8192-Token General-Purpose Text Embeddings for Long Documents. arXiv. https://doi.org/10.48550/arXiv.2310.19923

[3] Qian, H., Liu, W., Ding, Z., Sun, W., & Fang, C. (2023). Abstract syntax tree for method name prediction: How far are we? In 2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS) (pp. 464–475). IEEE. https://doi.org/10.1109/QRS60937.2023.00052

[4] Kukreja, S., Kumar, T., Bharate, V., Purohit, A., Dasgupta, A., & Guha, D. (2023). Vector databases and vector embeddings—Review. In 2023 International Workshop on Artificial Intelligence and Image Processing (IWAIIP) (pp. 231–236).