Enhancing Code Understanding for Impact Analysis by Combining Transformers and Program Dependence Graphs

YANFU YAN, William & Mary, USA
NATHAN COOPER, William & Mary, USA
KEVIN MORAN, University of Central Florida, USA
GABRIELE BAVOTA, USI Lugano, Switzerland
DENYS POSHYVANYK, William & Mary, USA
STEVE RICH, Cisco Systems, USA

Impact analysis (IA) is a critical software maintenance task that identifies the effects of a given set of code changes on a larger software project with the intention of avoiding potential adverse effects. IA is a cognitively challenging task that involves reasoning about the abstract relationships between various code constructs. Given its difficulty, researchers have worked to automate IA with approaches that primarily use coupling metrics as a measure of the "connectedness" of different parts of a software project. Many of these coupling metrics rely on static, dynamic, or evolutionary information and are based on heuristics that tend to be brittle, require expensive execution analysis, or large histories of co-changes to accurately estimate impact sets.

In this paper, we introduce a novel IA approach, called Athena, that combines a software system's dependence graph information with a conceptual coupling approach that uses advances in deep representation learning for code without the need for change histories and execution information. Previous IA benchmarks are small, containing less than ten software projects, and suffer from tangled commits, making it difficult to measure accurate results. Therefore, we constructed a large-scale IA benchmark, from 25 open-source software projects, that utilizes fine-grained commit information from bug fixes. On this new benchmark, our best performing approach configuration achieves an mRR, mAP, and HIT@10 score of 60.32%, 35.19%, and 81.48%, respectively. Through various ablations and qualitative analyses, we show that Athena's novel combination of program dependence graphs and conceptual coupling information leads it to outperform a simpler baseline by 10.34%, 9.55%, and 11.68% with statistical significance.

CCS Concepts: • **Software and its engineering → Software evolution**.

Additional Key Words and Phrases: Impact Analysis, Program Comprehension, Conceptual Coupling

ACM Reference Format:
Yanfu Yan, Nathan Cooper, Kevin Moran, Gabriele Bavota, Denys Poshyvanyk, and Steve Rich. 2024. Enhancing Code Understanding for Impact Analysis by Combining Transformers and Program Dependence Graphs. *Proc. ACM Softw. Eng.* 1, FSE, Article 44 (July 2024), 24 pages. https://doi.org/10.1145/3643770

1 INTRODUCTION

Modern software systems are long-lived, with extensive development and maintenance histories. Many projects experience churn in the developers or teams working on them, and can consist of millions of lines of code [Shin et al. 2011]. As such, understanding the potential cascading impacts

Authors' addresses: Yanfu Yan, William & Mary, Williamsburg, USA, yyan09@wm.edu; Nathan Cooper, William & Mary, Williamsburg, USA, nacooper01@wm.edu; Kevin Moran, University of Central Florida, Orlando, USA, kpmoran@ucf.edu; Gabriele Bavota, USI Lugano, Lugano, Switzerland, gabriele.bavota@usi.ch; Denys Poshyvanyk, William & Mary, Williamsburg, USA, denys@cs.wm.edu; Steve Rich, Cisco Systems, Maryville, USA, srich@cisco.com.

ACM 2994-970X/2024/7-ART44
https://doi.org/10.1145/3643770

of seemingly simple code changes can be a difficult proposition. This comprehension task forms the premise of *impact analysis (IA)* in which a given code change may result in *undesirable side effects*, such as a fault that leads to an erroneous program state, caused by unintended interactions between the changes and other parts of a software system [Kagdi et al. 2012; Kuang et al. 2012]. Thus, the task of IA involves estimating an impact set of entities, usually classes or methods of a software system, from a given change to an entity, also usually a class or a method [Arnold 1996] in the hopes of preventing unintended changes. This process can be cognitively challenging for developers, as reasoning about complex interactions of a software system requires careful comprehension of large volumes of code. Given that many important engineering and maintenance tasks – such as bug fixing and refactoring – require code change comprehension, they necessarily require IA as well. This process is typically performed *manually* by developers, but given its complexity, researchers have proposed a range of approaches for automating it.

Past techniques for automated IA have explored using four major types of information: (i) *structural information* (*i.e.,* from program dependence graphs), (ii) *semantic or conceptual information* (*i.e.,* code similarity), (iii) *evolutionary information* (*i.e.,* commit histories), and (iv) *execution information*. Conventional automatic IA techniques [Badri et al. 2005; Breech et al. 2006] have focused on analyzing structural dependencies (*e.g.,* control flow dependence) between different code entities to predict change impacts, but they tend to generate large impact sets with lower precision [Li et al. 2013]. As a result, other IA techniques have chosen to leverage additional information gathered via mining change histories from software repositories [Canfora et al. 2010; Gethers et al. 2012] or program executions [Kuang et al. 2012] to generate more accurate impact sets. However, these techniques rely on certain assumptions (*e.g.,* sufficient historical data, comprehensive execution profiles), require brittle heuristics, or significantly increase the computational overhead – making them less practical. These techniques may also ignore the conceptual/semantic information that naturally occurs in code (*e.g.,* identifiers) and is key in expressing the underlying intent of code entities. Given that code entities with similar intent likely contribute to similar problem domains, there are another set of IA techniques (*i.e.,* conceptual or semantic IA) [Gethers et al. 2012; Kagdi et al. 2012; Wang et al. 2018] which extract vectorized code semantics and compute a similarity-based ranked list of code entities that are potentially impacted by a change. Existing conceptual techniques formulate IA as an information retrieval (IR) task, and typically apply IR-based (*e.g.,* latent semantic indexing (LSI)) or machine learning-based (*e.g.,* doc2vec [Le and Mikolov 2014]) approaches to obtain code representations that capture the semantic relationships between code entities.

The possibility of combining *semantic* and *structural* information specifically for the task of impact analysis has not been well explored [Gyori et al. 2017]. Such a combination could prove beneficial due to the orthogonal nature of these information sources, and the practicality of forgoing the collection and sanitation of evolutionary or execution information. For instance, semantic coupling can help to relate methods or classes that share similar semantic purposes and hence may impact one another, whereas structural information can help deduce logical relationships between code entities which may appear to be unrelated based upon modeled semantics.

While there is promise in combining semantic and structural information for IA, there is also an opportunity to leverage recent advances in robust semantic models of code. Transformer-based [Vaswani et al. 2017] neural architectures [Feng et al. 2020; Guo et al. 2022; Wang et al. 2021b,a] have achieved great success in learning rich representations for a variety of code understanding and generation tasks, *e.g.,* code search, clone detection, program repair, *etc.*. These models are typically first pre-trained on large-scale datasets containing unimodal (code-only) and/or bimodal (comment, code) data to learn *generalized* code representations. The models are then fine-tuned

on task-specific datasets for downstream code-related tasks. However, despite their demonstrated benefits, none of these models have been applied to IA.

However, adapting transformer-based models of code to the task of IA, and integrating these models with structural information presents at least two major challenges. First, we currently lack large-scale vetted datasets that would allow a neural model to be fine-tuned on *IA-specific* code representations. This is due to the fact that deriving an IA dataset is labor intensive, as impact sets cannot be easily mined from software repositories without manual validation. Second, while the general code representations produced by pre-trained models could be directly used for similarity calculation for conceptual IA, they still ignore the global context the code finds itself in, *i.e.,* the structural dependencies that illustrate how the code is used within a software system. Unlike other code understanding tasks (*i.e.,* code search) that can rely solely on isolated code snippets to extract semantics, structural dependencies between code entities also play an important role in IA since the mutually dependent entities are likely to be impacted by each other.

To overcome these limitations, and advance the task of automated IA, we introduce Athena, which enhances code understanding with Transformer-based neural models [Vaswani et al. 2017] and structural dependence graphs for capturing relationships among code entities. We perform IA at method-level granularity for code entities in the Java programming language (PL). Specifically, Athena begins by constructing a software system's dependence graph, where nodes represent methods and edges represent the dependence relationship (*i.e.,* call dependence and class member dependence) between methods. We then leverage neural code models including CodeBERT [Feng et al. 2020], UniXcoder [Guo et al. 2022], and GraphCodeBERT [Guo et al. 2020], prominent Transformer-based code models, for initial method embedding extraction. These pre-trained neural code models are fine-tuned on a code understanding task, namely code search, to learn richer representations that are aware of underlying code intent and potentially transferring the additional knowledge learnt from code search to IA. To integrate the global dependence information into local code semantics, the initial method embeddings are further enhanced using an embedding propagation strategy inspired by graph convolutional networks (GCN) [Kipf and Welling 2017] based on the constructed dependence graphs.

Evaluating our proposed approach effectively also presents challenges. Existing IA benchmarks tend to be outdated and are constructed from original/unvetted commits, but as highlighted in multiple prior studies [Kirinuki et al. 2016; Kochhar et al. 2014; Mills et al. 2020; Wang et al. 2019], *tangling* has a high prevalence in these commits which is likely to affect the reliability of evaluation results of previous IA techniques on these benchmarks. Therefore, to evaluate Athena for the task of IA, we created a large-scale IA benchmark, called Alexandria, that leverages an existing dataset of fine-grained, manually untangled commit information from bug-fixes [Herbold et al. 2020]. The benchmark consists of 910 commits across 25 open-source Java projects, which we use to construct 4,405 IA tasks – where each task consists of a query method and a set of impacted methods. Using the standard information retrieval metrics of mRR, mAP, and HIT@10, we find Athena significantly (based on statistical tests) improve over the best performed conceptual IA baseline by 10.34%, 9.55%, and 11.68% respectively. In aggregate, we make the following contributions:

- A new large-scale evaluation benchmark for impact analysis, called Alexandria, composed of 4,405 IA tasks from 910 commits of 25 open source software systems;
- The first application of Transformer-based neural models to impact analysis for semantically-rich code representations;
- Athena, a novel approach that first integrates global dependence information into local code semantics to advance automated impact analysis;

- A comprehensive empirical evaluation that demonstrates ATHENA achieves state-of-the-art improvements compared to the conceptual IA baseline;
- A thorough set of ablations showing the improvements are attributable to the application of the Transformer-based neural model and the integration of structural dependence information;
- A comprehensive online appendix [Yan et al. 2024] that contains the code for ATHENA, our IA benchmark ALEXANDRIA and our experimental infrastructure to allow for the replication.

2 BACKGROUND & RELATED WORK

2.1 IA techniques

Typical IA techniques require a seed/starting entity to perform the analysis. Some start with a change request [Gethers et al. 2012; Torchiano and Ricca 2010] in natural language form, while most start with code entities [Kagdi et al. 2012; Kuang et al. 2012; Poshyvanyk et al. 2009] at different levels of granularity (*e.g.,* classes, methods, statements) since developers can usually identify at least one code entity that needs to be changed by using feature location techniques [Dit et al. 2013] and their software development knowledge. The output of the IA (*i.e.,* estimated impact set) is usually at the same granularity-level as the seed entity. Given that the class/file-level IA [Torchiano and Ricca 2010] is too coarse and the statement-level IA [Gyori et al. 2017] is too costly, most existing techniques choose to conduct IA at the method level [Kuang et al. 2012; Wang et al. 2018]. Moreover, Java, as one of the most commonly used object-oriented programming languages (PLs), has been selected as the primary focus of IA more often than any other PL. (*e.g.,* C [Gyori et al. 2017]).

In general, IA comprises two branches of techniques. One is to predict/infer *potential* impact of all possible changes [Cai and Santelices 2015; Cai and Thain 2016; Gyori et al. 2017] (*i.e.,* dependence analysis); the other is to reason about the *actual* impact sets of code changes [Kagdi et al. 2012; Kuang et al. 2012; Wang et al. 2018]. Specifically, the first branch assesses the user-perceived accuracy by creating the ground-truth impact set based on the static program dependence analysis or dynamic execution differencing, since they regard the real ground-truth is unknown. However, identifying the full set of dependencies based on static analysis is uncertain, and execution differencing relies on certain test cases and executions which cannot cover all possible dependencies either. [Cai 2020] gives a comprehensive summary of the first branch of techniques, while our approach falls into the second category, and we will now introduce the related techniques within this category in detail.

Existing IA techniques in the second category can be further divided into four types based upon the information they analyze. *i.e.,* structural, conceptual/textual, evolutionary, or dynamic. Conventional IA approaches [Badri et al. 2005; Breech et al. 2006] that use program graphs or slicing tend to generate very large impact sets [Li et al. 2013], and most importantly, they ignore the conceptual information encoded in the code (*e.g.,* identifiers) which is also important for expressing the intent of code entities. Since code entities with similar intents likely contribute to similar problem/solution domains, conceptual IA techniques [Kagdi et al. 2012; Poshyvanyk et al. 2009; Torchiano and Ricca 2010; Wang et al. 2018] typically apply IR-based (*e.g.,* LSI) or machine learning-based (*e.g.,* doc2vec [Le and Mikolov 2014]) approaches on code to extract vectorized code semantics and estimate impact sets by computing a cosine similarity-based ranked list of code entities. [Poshyvanyk et al. 2009] quantitatively show that the conceptual coupling is superior to the structural coupling-based measures for IA. Moreover, some IA techniques analyze evolutionary couplings [Jashki et al. 2008; Sherriff and Williams 2008; Zimmermann et al. 2004] mined from multiple historical releases/commits of version control systems in order to discover frequent co-change patterns to predict current change impacts, but the sufficient historical data is not always available (*e.g.,* for new projects), and sometimes previous change patterns may be

outdated and misleading. In addition, dynamic IA [Breech et al. 2004; Kuang et al. 2012] utilizes execution information (*e.g.,* execution traces, relations) to compute more accurate impact set, but the computation overhead is much greater than static IA. The quality of dynamic techniques relies heavily on the representativeness of the test suites and/or profiles gathered during program execution. The industrial case studies [Acharya and Robinson 2011; Borg et al. 2017; de la Vara et al. 2016; Gyori et al. 2017; Tao et al. 2012] indicate preferences for static IA techniques over dynamic ones as there is a lack of published studies reporting the adoption of dynamic IA [Cai 2020].

To further improve the accuracy of impact set estimation, some research attempts to combine existing techniques. [Kagdi et al. 2010] blend conceptual and evolutionary analysis showing additional advantages over using either of them alone. [Gethers et al. 2012] further augment them with dynamic analysis to obtain more accurate impact sets. It is worth noting that these two hybrid techniques are only compared with their variants (*i.e.,* using only one of the components) to validate the effectiveness. A recent work [Kuang et al. 2012] combines dynamic analysis with structural analysis (*i.e.,* data and call dependencies) demonstrating that dynamic data sharing dependencies are complementary to dynamic call dependencies.

Our approach belongs to the set of hybrid analysis-based IA techniques as Athena extracts code semantics and dependencies and computes a ranked list for impact set estimation. Therefore, it avoids the associated limitations and drawbacks of other categories of techniques (*i.e.,* evolutionary and dynamic analysis) while retaining the benefits of multiple information sources. LSI is the most frequently used model to obtain code semantics for conceptual IA [Gethers et al. 2012; Kagdi et al. 2010; Poshyvanyk et al. 2009]. The latest and most closely related work to ours is [Wang et al. 2018] which integrates LSI with doc2vec to enhance code semantics by considering the context of each code token within the code entity. They quantitatively show the combined model outperforms using LSI only on IA.

Different from existing conceptual IA techniques, our approach (i) leverages advanced transformer-based code models to obtain more meaningful code representations (ii) further enhances code semantics by embedding propagation based on structural dependence graphs. To the best of our knowledge, our approach is the first IA technique which integrates the global structural information into local code semantics based on only a single release of the source code without any additional information (*e.g.,* previous releases and/or execution information). Given that [Wang et al. 2018] has not made their implementation publicly available, we directly use LSI and doc2vec independently as conceptual IA baselines for our work. This also allows us to compare the performance of different models for code semantics extraction when they are individually applied for IA.

2.2 IA benchmarks

Existing IA benchmarks [Cai and Santelices 2015; Gethers et al. 2012; Kuang et al. 2012] are typically constructed in two ways. The first type of construction considers ground-truth impact sets to be unknown and tries to create them using program dependence analysis [Cai and Santelices 2014; Cai et al. 2016a] or execution differencing [Cai and Santelices 2015; Cai et al. 2016b; Cai and Thain 2016; Gyori et al. 2017]. However, computing a full set of program dependencies [Cai 2020] is an undecidable problem. As such they are usually generated based on artificial changes and/or by sampling changes in real open-source projects. All possible changes to a code entity (only involving *one* certain release of code repository) are used as the seeding entities.

The other more popular way for constructing IA benchmarks involves building multiple co-changed sets of code entities, each of which are collected based on *two* consecutive commits [Kuang et al. 2012] or several grouped commits [Wang et al. 2018]. All entities within a co-changed set are assumed to be impacted by each other. To construct the ground-truth, one [Kagdi et al. 2012] or a few code entities [Kuang et al. 2012] in the co-changed set are selected as the seed entity, and

the remaining others are served as the real impact set. Existing benchmarks/case studies in this category usually consist of 3-6 open-source repositories and the commits used are either from bug fixing commits only [Jiang et al. 2019] or dominated by bug fixing commits [Gethers et al. 2012]. However, the prevalence of *tangling* [Herbold et al. 2020; Herzig and Zeller 2013; Kirinuki et al. 2014; Mills et al. 2020] existing in commits negatively affects the reliability of evaluation performance of techniques (*e.g.,* bug localization [Mills et al. 2020], defect detection [Herzig et al. 2015]) that rely on commit data for testing due to the presence of noise. Tangled commits refer to the changes to software which address multiple concerns at once. For example, a (original) commit which claims to be fixing a bug, may not only fix the bug but also include additional unrelated changes (*e.g.,* refactorings). While we have limited knowledge on the exact impact of tangled commits on the reliability IA technique evaluations, the potential for impact is clear — in tangled commits the co-changed code entities within a commit don't all contribute to a single concern (*e.g.,* bug fixing) and thus are not necessarily impacted by each other, leading to inaccurate ground-truth impact sets. Given that prior studies have confirmed the prevalence of tangled commits [Herbold et al. 2020], it is highly likely evaluations of past techniques were affected by this phenomenon.

Our Alexandria dataset falls into the second category of IA benchmark, but with a *notable key difference* — it is built from untangled bug fixing commits [Herbold et al. 2020]. Herbold *et al.*'s work quantitatively shows tangled commits have a high prevalence and the authors manually untangle them by annotating line-level change types. By utilizing only the co-changed code entities that have been manually verified to contribute to one concern (*i.e.,* bug fix), our benchmark contains more reliable ground-truth impact sets, and this favorable characteristic is demonstrated quantitatively through experiments. To the best of our knowledge, our Alexandria is the first IA benchmark whose ground-truth impact sets are built from manually-validated untangled commits. Moreover, Alexandria contains 910 commits from 25 systems, making it larger than past benchmarks.

2.3 Code Representation Learning

Traditional IR approaches (*e.g.,* LSI, Term Frequency - Inverse Document Frequency (TF-IDF), Latent Dirichlet Allocation (LDA)) were first used to generate vectorized code representations in order to support SE tasks. They typically require building a corpus from all documents (code artifacts) and then represent code by measuring the importance of each code token to a document in the corpus and/or exploiting co-occurrences of code tokens based on singular value decomposition (SVD) or Bayesian topic modelling. However, these IR approaches treat the code as *bag-of-words*, ignoring the order and semantics of code tokens. Thus, neural networks have been employed to obtain more meaningful code representations. For instance, word2vec [Mikolov et al. 2013] takes into account each individual token and its context tokens by using a sliding context window during training. Furthermore, doc2vec [Le and Mikolov 2014] could learn a paragraph vector for the code of variable length, instead of using an average representation of the code tokens as word2vec does. Subsequently, more and more end-to-end deep models (*e.g.,* Bi-RNN [Cho et al. 2014], TextCNN [Kim 2014], Self-Attention [Vaswani et al. 2017]) have been used to extract code embeddings.

Recently, the finetuning-after-pretraining scheme [Brown et al. 2020; Devlin et al. 2019; Raffel et al. 2019; Yang et al. 2019] has achieved great success in NLP tasks wherein a model is pre-trained on large-scale text in a self-supervised manner to learn general representations, and then fine-tuned for specific downstream tasks on a more limited dataset. The Transformer [Vaswani et al. 2017] architecture stands out as the most representative encoder backbone for this scheme. With the advent of large-scale code datasets (*i.e.,* CodeSearchNet [Husain et al. 2019]), this scheme has also been increasingly applied to learn code representations and autoamte software engineering tasks [Ahmad et al. 2021; Guo et al. 2022; Wang et al. 2021b,a]. CodeBERT [Feng et al. 2020] was one of the first Transformer-based NL-PL pretrained model for supporting various code-related tasks. It

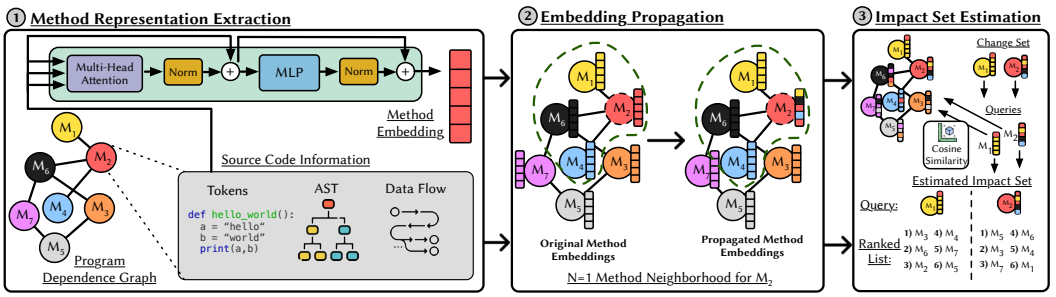


Fig. 1. Overview of the Workflow of the Athena Impact Analysis Approach

distinguishes between the PL and the NL modality and captures their semantic connection during pretraining. However, it only utilizes the sequential information of the bi-modal data while ignoring the inherent structure of code. Therefore, GraphCodeBERT [Guo et al. 2020] further incorporates data flow information within methods into sequenced code snippets during pretraining resulting in enhanced code embeddings. Other pretrained models like UniXcoder [Guo et al. 2022] that encode abstract syntax tree (AST) information to produce syntax-aware code embeddings. We explore the use of pre-trained CodeBERT, GraphCodeBERT, and UniXcoder representations for Athena which are then fine-tuned on the code search task to extract initial code embeddings for performing IA. However, any Transformer-based code models can serve as the encoder backbone of our approach.

3 ATHENA

In line with previous conceptual IA techniques [Gethers et al. 2012; Kagdi et al. 2012; Wang et al. 2018], we formulate impact analysis as an information retrieval task where if a developer intends to modify a method (*i.e.,* query/seed method) in a software system, Athena will return a ranked list of other methods being potentially impacted in descending order of likelihood. All methods but the query are used as the search corpus. Formally, for a software system S containing a set of methods $S = \{m_1, m_2, ..., m_n\}$, a potential change to one of the methods $m_i \in S$ triggers Athena to rank all other methods thus estimating the impact set.

Figure 1 provides an overview of Athena. Athena begins by building a dependence graph among all methods across an entire software system, where nodes represent methods and edges represent dependence relationships between methods. Each method is processed by a state-of-the-art Transformer-based code model (*e.g.,* GraphCodeBERT) to obtain an initial method representation by considering the context that exist within the method. These neural code models are then fine-tuned on the code search task to generate richer code representations and potentially transfer the additional knowledge learned from code search to IA. Next, Athena analyzes the global dependencies and propagates information from the "neighbor" method nodes in the dependence graph to a given target method. Specifically, each initial method embedding is updated/augmented based on a propagation strategy inspired by Graph Convolutional Networks (GCNs) [Kipf and Welling 2017] so that the information of global dependences is integrated into its local code semantics. To obtain a final ranked list, the cosine similarity between the augmented representations of a given query method and each method in the corpus is computed. We next discuss each step of Athena in detail.

3.1 Dependence Graph Generator

The initial step of Athena is to build a static dependence graph generator to capture method dependencies across a software system. Essentially, we identify two methods as having dependencies if there exists a caller-callee relationship between them (*i.e.,* call dependence) and/or if they belong

to the same class (*i.e.,* class member dependence). While certain existing tools like WALA [Fink and Dolby 2012] and Soot [Sable Research Group 2023] can produce static call graphs for Java, they require JVM bytecode as input, thus necessitating compliable source code. Although the latest version of Soot provides source code analysis, it limits the source code up to Java 7 and still requires internal compilation. These tools thus increase preprocessing time for IA and negatively affect their scalability. To better integrate the graph generator into ATHENA and capture both call and class member dependencies, we developed our own tool to generate static dependence graphs, which simply takes the source code of a software system as its input.

A dependence graph can be formally defined as $G = (V, E)$, where V denotes a set of method nodes and E denotes a set of edges representing the method dependence relationships. Since impact analysis is usually performed on the production entities (*i.e.,* excluding the entities for testing) [Kuang et al. 2012], we first collect all `.java` production source files in a software system and use the Tree-Sitter [Brunsfeld et al. 2022] library to identify all methods contained in these files. The library enables constructing a specific syntax tree for each file and supports searching for various patterns (*e.g.,* method calls, method declarations) in the tree. All identified methods then serve as the nodes of the dependence graph. To precisely locate each method and facilitate the process of method representation extraction, we attach to each method node the complete method content (*i.e.,* the method declaration with its body), the name of the class it belongs to, and the package path.

Next, we construct the edges for the dependence graph. To capture the class member dependencies, the edges are added between each pair of the methods in the same class. As for the call dependencies, we utilize the Tree-Sitter library to identify all the method invocation statements (*e.g.,* `receiver.method()`) within each method and resolve these statements by finding its callee methods. The edges are then added between each pair of caller-callee methods. In general, we traverse upwards from each invocation statement to find where the `receiver` is introduced by analyzing the declaration statements and the arguments of the caller method. It is then easy to obtain the class name of the callee method and its belonging package path. In order to locate the callee method based on the class name and the package path, we utilize both the method name and # arguments (rather than the complete signature) to ensure the efficiency and scalability of our generator. When the callee method is overloaded with the same number of arguments, we add the edges from the caller method to each of these overloaded callee methods. It is worth noting that combining the method name and # arguments helps filter quite a few overloaded methods than using the method name only.

Although we can add directed edges from caller to callee methods, their semantics are actually interrelated and mutually affect each other when performing IA. Thus, by using our tool, the dependence graph is constructed in an undirected manner. Moreover, edges representing class member dependencies are distinguished from those representing call dependencies by attaching each edge to its property (*i.e.,* call or class member dependence). If two methods have both types of dependencies, we add two edges with different properties between them.

3.2 Code Representation Extraction

We then use one of three Transformer-based code models (CodeBERT, UniXcoder, or GraphCodeBERT) to extract initial method embeddings for performing IA, as shown in Figure 1-(1). In the case of GraphCodeBERT, it goes beyond sequential code information by considering inherent structure of code (*i.e.,* data flow) to encode the relation "where-the-value-comes-from" between variables. In this model, the input is encoded by a multi-layer bidirectional Transformer containing a sequence of self-attention and feed-forward layers (*i.e.,* multi-layer perceptron (MLP)) with normalizations.

These pre-trained models can directly produce code embeddings, but the self-supervised objectives used during pretraining are quite different from IA, and most importantly, the representations are not specifically learned for Java but generally for multiple PLs. Although these neural models can be further fine-tuned for downstream tasks, neither GraphCodeBERT nor other Transformer-based code models has been fine-tuned or evaluated for IA due to the absence of large available IA training/fine-tuning datasets. IA belongs to a general family of code understanding-tasks (and hence is not generative), and there are two other downstream understanding tasks that have been extensively researched and evaluated — namely code search and clone detection. Code search aims to retrieve relevant code given a NL query, while clone detection aims to predict whether two code snippets can output similar results when given the same input. We leverage code search as a proxy to potentially transfer additional knowledge learned from code search during fine-tuning to enhance code semantics for IA. Although clone detection may initially seem more closely aligned with IA, we do not use it because (i) datasets such as BigCloneBench [Lu et al. 2021; Svajlenko et al. 2014] which could be used for fine-tuning does not include comments, which is likely to enhance code understanding; and (ii) instead of generating separate code embeddings, the fine-tuned neural model for clone detection concatenates two code snippets as a whole and only generate one embedding for them, thus making the following embedding propagation process more difficult. They typically add a classifier on top of the Transformer-based encoder to directly produce the probability of whether two code snippets can yield similar results.

To fine-tune our neural code models for code search, we follow the pipelines recommended in their corresponding papers. For example, for GraphCodeBERT, our best performing model, we follow the authors' recommendation [Guo et al. 2020] to use a Siamese framework on the CodeSearchNet [Husain et al. 2019] Java split dataset. CodeSearchNet consists of 2.3 million functions in six programming languages paired with NL descriptions (*i.e.,* comments). The CodeSearchNet Java split has been filtered by handcrafted rules by [Guo et al. 2020] to remove low-quality data, and contains 164,923 bimodal (comment, code) pairs. Each code snippet in the paired data is a method from a software GitHub repository with all comments removed, and the corresponding comment is extracted from the first line of the method's documentation comment. The objective of fine-tuning is to map the code and its comment onto the vectors close to each other in order to learn high-level intent-aware code semantics. During fine-tuning, the comment and code (with data flow extracted) are separately fed into a comment encoder and a code encoder. These two encoders have identical model architectures (*i.e.,* GraphCodeBERT) and are initialized from the pre-trained GraphCodeBERT parameters (*i.e.,* weights and biases). The parameter updating is synchronized across both encoders during fine-tuning based on the standard cross entropy loss. We use the AdamW [Kingma and Ba 2015] optimizer and the same hyperparameters (*e.g.,* # epochs, learning rate, batch size *etc.*) recommended by [Guo et al. 2020] for parameter updating, and the whole process was performed on an Ubuntu 20.04 server with an NVIDIA A100 40GB GPU. The finetuned GraphCodeBERT is expected to generate more meaningful representations of code that are aware of the underlying intent.

When performing IA, we need to first preprocess the method content attached to each method node in the generated dependence graph. Taking GraphCodeBERT as an example, we first follow the preprocessing procedure of CodeSearchNet [Husain et al. 2019] by extracting the initial line of the documentation comment and the code-only data. The code is further parsed into an abstract syntax tree (AST), the leaves of which are used to identify the variable sequence for the data flow construction. The input to the fine-tuned GraphCodeBERT for IA is the concatenation of comment, source code, the set of variables $X = ([CLS], A, [SEP], C, [SEP], V)$ or $X = ([CLS], C, [SEP], V)$. A, C and V stand for the comment token sequence, code token sequence, and variable sequence respectively. $[CLS]$ is a token for learning aggregated information from the entire sequence during

training, with its final representation is typically used for classification-related tasks. [*SEP*] is a separation token used to split two data types. Edges are added between variables in the variable sequence where a data flow relationship exists, and the variables are aligned across source code and data flow. The input is then processed by the fine-tuned encoder, and we take the average output of all the hidden states of the last layer as the method representation. The input sequence length is set to 256 and the output representation dimension is 768 to maintain consistency with GraphCodeBERT. Finally, the initial method embeddings are generated for all method nodes in the dependence graph of a given software system.

3.3 Embedding Propagation

While the initial embeddings effectively capture meaningful code semantics via the self-attention mechanism, they are limited to local context and lacking the global dependence of methods. To further improve code understanding, we utilize an embedding propagation strategy that updates each method embedding by propagating the embeddings of its neighbor methods based on the constructed dependence graph G, thus integrating the information of global structural dependence into local code semantics. We visualize this process in Figure 1-②. Formally, this is represented as $m_i' = f(m_i, m_1^{nebr}, m_2^{nebr}, ..., m_k^{nebr})$, where m_i is the method being updated through the embedding propagation strategy f with its neighbors $m_j^{nebr} (1 \leq j \leq k)$. In particular, our embedding propagation strategy is inspired by the Graph Convolutional Network [Kipf and Welling 2017] which adopts layer-wise propagation on the neural networks motivated by a localized first-order approximation of spectral graph convolutions:

$$M' = \sigma(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}MW), \tag{1}$$

where σ represents an activation function and W is a trainable weight matrix. $\tilde{A} = A + I_N$ denotes the adjacency matrix of a graph G with self-connections. I_N is the identity matrix and $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$. This propagation strategy has been modified using a renormalization method [Kipf and Welling 2017] in order to mitigate the effects of numerical instabilities and exploding/vanishing gradients when matrix multiplication operators are repeated during the training of the deep neural network. Since we do not train our dependence graph G in this phase, our embedding propagation strategy is directly derived from the first-order approximation of localized spectral filters on graphs [Defferrard et al. 2016; Hammond et al. 2011], which can be summarized as follows:

$$M' = (I_N + wD^{-\frac{1}{2}}(A^c + A^{cm})D^{-\frac{1}{2}})M. \tag{2}$$

$M \in \mathbb{R}^{N \times F}$ represents the matrix of all method embeddings with respect to the dependence graph G and $M' \in \mathbb{R}^{N \times F}$ stands for the matrix in which each method embedding is updated by its neighbor method embeddings. N denotes the number of method nodes and F denotes the dimension of each method embedding (*i.e.,* 768). A^c is the adjacency matrix based on call dependence edges of G, while A^{cm} is the one based on class dependence edges. Neither of them contains self-connections. D is the degree matrix of $(A^c + A^{cm})$ for normalization with respect to both rows and columns. w is a constant that is responsible for balancing the information between methods and its neighbor methods. According to this formula, if a method exhibits both call and class member dependencies with its neighbor method, the embedding of this neighbor method will be propagated/aggregated twice to the target method embedding. Intuitively, methods sharing multiple dependencies are inherently more closely related than those with just a single type of dependency. Moreover, in order to evaluate the effect of the distance of neighbor methods used for embedding propagation, neighbor methods in other orders(hops) are also utilized in addition to the direct neighbors:

$$M' = (I_N + w \sum_i D_i^{-\frac{1}{2}}(A_i^c + A_i^{cm})D_i^{-\frac{1}{2}})M, \quad (3)$$

where $1 \leq i \leq 3$ since we at most take into account the neighbor methods within three orders due to computational constraints. After the Embedding propagation strategy has completed, all of the identified methods in a given software system will have an augmented embedding calculated by propagating the *original* method embedding from neighbors to the target method, as illustrated at the top of Figure 1-③.

3.4 Impact Set Estimation

Finally, as illustrated in Figure 1-③, ATHENA computes the cosine similarity between the augmented embedding of a given query method and the augmented embeddings of each of the methods in the search corpus. Based on the cosine similarity scores, ATHENA returns a ranked list in descending order to help developers find other methods that are possibly affected and likely to be modified.

4 EXPERIMENTAL DESIGN

To evaluate ATHENA's effectiveness in IA, we investigate the following research questions (RQs):

RQ$_1$: *How effective is* ATHENA *with/without embedding propagation when compared with conceptual baselines on the task of impact analysis?*

RQ$_2$: *How do call and class member dependencies improve* ATHENA's *effectiveness in IA?*

RQ$_3$: *How well does* ATHENA *perform on IA based on different configurations (e.g., using other Transformer-based pre-trained code models)?*

RQ$_4$: *How does the tangled benchmark affect the reliability of IA evaluation results?*

RQ$_5$: *How do properties of different impact analysis tasks affect our studied techniques?*

4.1 Impact Analysis Benchmark: Alexandria

Our IA benchmark ALEXANDRIA is constructed from manually untangled bug fixing commits [Herbold et al. 2020] in order to generate more reliable ground-truth impact sets. Multiple prior studies [Kirinuki et al. 2016; Nguyen et al. 2013; Wang et al. 2019], supported by manual validation, have consistently shown that tangled commits naturally occur in codebases. However, all existing IA benchmarks [Kagdi et al. 2012; Kuang et al. 2012; Wang et al. 2018], built directly from these original/unvetted commits, inaccurately assume that all co-changed entities in a commit address one single concern, thus impacted by each other. The invalidated data (*i.e.,* (query, ground-truth impact set) pairs) is likely to be noisy which can affect the reliability of experimental results of previous IA techniques.

Recently, Herbold *et al.* [Herbold et al. 2020] introduced a large dataset covering 3,498 commits from 28 Java projects, with the purpose of studying the tangling that occurs in bug fixing commits. All selected projects are from the Apache Software Foundation and were developed by contributors from the open source community or industry. These projects cover diverse application domains, such as build systems (*e.g., ant-ivy*), web applications (*e.g., jspwiki*), general purpose libraries (*e.g., commons*), *etc.*. In this dataset, each changed line was annotated with its type of change, whether it was modified to fix a bug, or was a change to tests, whitespace, a documentation change, a refactoring, or unrelated feature improvement. The data were annotated by four participants, and consensus was obtained if at least three participants agreed on the annotation to ensure accuracy. While some existing datasets [Kirinuki et al. 2014; Kochhar et al. 2014; Mills et al. 2020] also manually untangle the commits, they either cover a limited sample of commits or typically perform

untangling at the commit or file level, which is relatively coarse-grained so that the validated co-changed entities cannot be identified at method-level. Therefore, we constructed our IA benchmark based on the fine-grained untangled dataset [Herbold et al. 2020] allowing us to know exactly which methods are changed for addressing one single concern, thereby generating reliable ground-truth for evaluation.

Co-Changed Set Construction. To create evaluation IA tasks, we systematically mined the dataset from [Herbold et al. 2020]. By utilizing only the co-changed code entities that have been rigorously manually verified to contribute to one concern, our benchmark Alexandria contains more reliable ground-truth impact sets. Specifically, for each changed line in production code files labeled as *"contributes to the bug fix"*, we added the corresponding method to our benchmark by recording the information of GitHub Diff URL, repository name, commit ID, parent commit ID, file path, method name, line numbers indicating where the method starts and ends. Since [Herbold et al. 2020] does not provide method-related information, such as method names and line numbers of method boundaries, we employed the srcML library [Collard et al. 2013] to locate each changed method based on labeled changed line numbers. We utilized the snapshot/release of a software system that corresponds to the parent commit ID, as that is the state in which the change would be applied. Then, for each parent commit, we formulate a co-changed method set based on concurrently changed methods. Since there is no clear indication of a query/seed method, *i.e.,* which method would be changed "first" in the commit, we treat each method in the co-changed method set as a potential query, whereas the remaining others constitutes the ground-truth impact set. From developers' point of view, they usually at least know where the change starts and intend to know which other methods need to be modified. We further post-process the dataset to exclude commits that contain only one changed method.

IA Task Definition and Settings. Formally, for each co-changed method set $M = \{m_1, m_2, ..., m_n\}$, $n \geq 2$, we perform IA with a query being $\forall\ m_i \in M$ and the corresponding ground-truth impact set being $M - m_i$. We consider three different settings wherein the search corpus differs. In the first setting (**Setting 1 -whole**), the search corpus includes all methods except the query in all production files from the corresponding snapshot of the software system. This setting provides a comprehensive evaluation scenario where all methods in the software system are taken into consideration. The similar process of formulating co-changed methods into IA tasks has been widely adopted by past work to assess IA approaches [Gethers et al. 2012; Kagdi et al. 2012; Kuang et al. 2012]. In practice, conceptual IA techniques will generate a ranked list of methods in the corpus and developers would determine whether a method should be modified by inspecting the corpus in the given order. After analyzing our benchmark, it was observed that methods in the same class are more likely to be changed together. To account for this and mitigate potential biases introduced by IA approaches that equally prioritize methods within the same class as the query, we formulate two more specific task settings. In our second setting, the methods in both the ground-truth impact set and the search corpus are from the same class as the query (**Setting 2 - inner**). In our third setting, the methods in both the ground-truth impact set and the search corpus are from different classes than the query (**Setting 3 - outer**).

Dataset Statistics. Two software projects (*i.e., santuario-java* and *wss4j*) in [Herbold et al. 2020] are no longer accessible and for the software project *eagle*, we were unable to build any valid co-changed method sets, *i.e.,* the size of the co-changed set less than two. As a result, our benchmark contains 25 Java software projects, and the lines of code (LOC), # commits, # tasks for each project is shown in Table 6. Moreover, for each of the three settings, Table 1 shows # tasks, # commits, the average number of methods in the ground-truth impact set and in the search corpus respectively. Compared to Setting 2 (inner) which requires retrieving four or five affected methods out of 31

Table 1. Dataset statistics of our evaluation benchmark

Settings	# queries	# commits	ground-truth set	corpus
1 - whole	4,405	910	15.14	3,346
2 - inner	3,379	734	4.47	30
3 - outer	2,999	444	17.21	3,440

methods, Setting 3 (outer) is far more challenging, requiring 17 or 18 methods to be retrieved from a larger corpus with an average of 3,440 methods.

Tangled Counterpart. To analyze the effect of tangling commits on the evaluation of IA techniques, we also construct a benchmark without manually untangling similar to what previous IA benchmarks did [Kagdi et al. 2012; Kuang et al. 2012; Wang et al. 2018]]. Specifically, we directly construct co-changed method sets from original/tangling commits, so the bug fix changes are likely to be tangled with refactoring and unrelated improvement changes. Then, we compare the Alexandria dataset with its tangled counterpart in terms of the tasks with inconsistent (query, impact set) pairs. We observe that 606 tasks from 50 commits (setting 1) in Alexandria could have brought inaccurate ground-truth impact sets if without untangling. Further, the tangled Alexandria dataset has 856 tasks (out of 4,655) from 81 commits that are inaccurate with respect to (query, ground-truth impact set) pairs. The increase in the number of tasks and commits is due to an increase in the size of co-changed method sets, *i.e.,* more changed methods (for refactoring/unrelated improvement) are used as queries and some previously filtered commits with co-changed set less than 2 are likely to be added again.

4.2 Evaluation Metrics

We use standard information retrieval metrics to measure the effectiveness of Athena, namely mRR (mean Reciprocal Rank), mAP (mean Average Precision) and HIT@k. For each task, the ranked list generated by Athena is compared with the ground-truth impact set. Specifically, we computed the *rank* of the *first* truly affected method found in the ranked list, indicating the number of methods developers need to inspect before finding the first one that requires modification. The *reciprocal rank* is then calculated for each task, and these values are averaged across all tasks to derive the final mRR score. Furthermore, we compute the AP score for each task and average these scores across all tasks to obtain the final mAP score. AP is the average of precision values calculated after *each* method in the ground-truth impact set is retrieved, which approximates the area under the uninterpolated Precision-Recall curve. mAP scores measure the ability of the approach in helping developers identify all possibly affected methods. Moreover, we use HIT@k to measure the proportion of successful tasks for the cut point k. A successful task means that the approach has found at least one truly affected method among the top-k results it returns.

Many IA techniques [Kuang et al. 2012] rely on *Precision, Recall* and *F-measure* for evaluation since they consider IA as a binary classification task by finding possibly affected methods based on structural/evolutionary/dynamic dependencies. Therefore, what these techniques produce is not a ranked list, but an unordered estimated impact set, which is then directly compared with the ground truth impact set to compute an F-score (*i.e.,* the harmonic mean of the *Precision* and *Recall* values). However, conceptual IA techniques [Gethers et al. 2012; Kagdi et al. 2012; Wang et al. 2018], formulate IA as an information retrieval task but still adapted prior Recall/Precision/F-score metrics to the IR context. We argue that IR metrics provide a more realistic representation of the potential benefits that conceptual IA approaches may actually provide to a developer in a recommender system setting. Furthermore, mAP score is more accurate than F-measure because it analyzes *Precision-Recall* relationship globally rather than just based on the mean value calculation.

4.3 Baselines

We compare our approach, ATHENA, with three baseline approaches that extract code semantics for intent-aware IA. Specifically, two traditional IR-based approaches (*i.e.,* TF-IDF and LSI) and a deep learning-based model (*i.e.,* doc2vec [Le and Mikolov 2014]) are used as our conceptual IA baseline. To use IR for IA, we first build a corpus using all production methods from a specific snapshot/commit of a software system. For each code token in a method, we calculate its term frequency (TF) which represents the number of times the token appears in the method and the Inverse Document Frequency (IDF) which is the number of occurrences of the code token in all code tokens from the corpus. Each method in the corpus is then represented as a TF-IDF vector for the following cosine similarity computation. In line with previous conceptual IA techniques [Gethers et al. 2012; Wang et al. 2018], LSI further employs singular value decomposition (SVD) on the TF-IDF matrix consisting of TF-IDF representations of all methods in the corpus, and the cosine similarity is computed based on the new dimension-reduced method representations. As for doc2vec, we first train the model utilizing the *distributed memory* algorithm on the CodeSearchNet Java split dataset by concatenating comment tokens with code tokens to maintain consistency with the Transformer-based models (*e.g.,* GraphCodeBERT) training process. The doc2vec model can then generate paragraph-based method representations for the constructed IA tasks.

4.4 ATHENA Configurations

By using our approach ATHENA, we integrate the global dependence information into local code semantics to improve IA and we set $w = 0.5$ for information balancing. We use GraphCodeBERT as the encoder for the final version of ATHENA, and in **RQ**$_1$, given it achieves the best IA performance. We also validate the effectiveness of initial method representations (without embedding propagation) obtained by GraphCodeBERT for conceptual IA ($Athena_{ct}$) and conduct experiments by using either call ($Athena_{ct+cd}$) or class member dependencies ($Athena_{ct+cmd}$) with GraphCodeBERT in order to quantitatively show the contribution of each type of dependence from the dependence graphs.

We also experimented with different encoders (*i.e.,* CodeBERT [Feng et al. 2020] and UniXcoder [Guo et al. 2022]) that are also fine-tuned on the code search task following the similar procedure described in section 3 in order to demonstrate the effectiveness of our approach when using other Transformer-based code models. Moreover, we try neighbors of different orders/distances (1-3) when propagating the embeddings based on structural dependence graphs. Additionally, we conduct experiments based on different initial method representations obtained by GraphCodeBERT, including with/without comment, using outputs of [CLS] token to represent methods, and using the pretrained GraphCodeBert directly without finetuning it on code search. Last, we also fine-tune the pre-trained GraphCodeBERT on the BigCloneBench dataset [Lu et al. 2021] constructed for the clone detection task following the same procedure provided by [Guo et al. 2020], and employ this fine-tuned GraphCodeBERT to directly generate the probability of whether two methods are semantically similar for the IA task.

5 EVALUATION RESULTS & DISCUSSION

5.1 **RQ**$_1$: ATHENA Performance on IA

Table 3 presents ATHENA's performance (%) on our ALEXANDRIA benchmark whereas Table 2 reports results for three baseline models for IA. All of these models take code-only information (*i.e.,* without comment) as input except LSI (+comm.), and we will show the performance of Athena (+comm.) in the **RQ**$_3$ Ablation Study. The results in Table 2 reveal that the LSI model achieves the highest effectiveness in the baseline models across three settings. Given the effect of the number of related topics on LSI's performance, we experimented with varying numbers of related topics (for 0

Table 2. Effectiveness of Baseline Techniques

Baseline	Settings	mRR	mAP	Hit@10
TF-IDF	1-whole	49.57	25.38	70.35
	2-inner	73.86	64.69	94.61
	3-outer	34.50	16.50	49.35
LSI	whole	49.98	25.64	69.80
	inner	74.11	64.97	94.53
	outer	34.85	16.68	49.45
doc2vec	whole	43.62	19.97	58.59
	inner	68.93	59.05	90.97
	outer	29.63	12.35	40.25
LSI (+comm)	whole	50.28	26.16	70.94
	inner	73.83	64.69	94.61
	outer	34.60	19.93	49.91

Table 3. Effectiveness of Athena

Athena Config	Settings	mRR	mAP	Hit@10
ATHENA$_{ct}$	whole	52.38	28.86	73.87
	inner	**75.94**	**66.24**	95.44
	outer	40.39	21.43	58.19
ATHENA$_{ct+cd}$	whole	54.26	30.43	76.96
	inner	75.05	65.52	95.38
	outer	42.50	22.70	60.95
ATHENA$_{ct+cmd}$	whole	59.55	34.50	80.50
	inner	75.91	66.22	95.32
	outer	42.93	22.02	59.92
ATHENA	whole	**60.32**	**35.19**	**81.48**
	inner	75.59	65.94	**95.80**
	outer	**45.07**	**23.41**	**61.59**

to 2,000 in 100 increments) and selected the one with the best performance (1,300) for the final LSI configuration. Moreover, LSI only slightly outperforms TF-IDF on three metrics, indicating that the advantage is not significant if high-level code semantics is extracted through SVD. Surprisingly, the doc2vec model performs worse than LSI. This could be due to the fact that the IR-based approaches can directly build corpora and measure importance of code tokens on the evaluation dataset, and thus excel at keyword matching in favor of IA. However, for the deep learning-based model doc2vec, it is primarily trained for high-level semantics understanding rather than keyword matching with evaluation set unknown, but it struggles with understanding code intent compared to Transformer-based code models. In addition, we add comment information to the input for the best performing baseline LSI, but the with-comment version only performs slightly better than the one without comments in Setting 1 (whole), but not in Setting 2(inner) and 3(outer) on mRR and mAP, which does not result in the real improvement for IA. We provide detailed explanation of this in **RQ$_2$**.

As can be seen from Table 3, both ATHENA$_{ct}$ (without embedding propagation) and ATHENA outperforms LSI with statistical significance (Wilcoxon's paired test $p < 0.05$) on three metrics across all settings, and their improvements in Setting 1 (whole) can mainly be attributed to the improvements in Setting 3 (outer). Specifically, ATHENA$_{ct}$ improves LSI by 2.4%/3.22% mRR/mAP in Setting 1, and 5.54%/4.75% mRR/mAP in Setting 3. In fact, LSI performs quite well in Setting 2 (inner) because of its proficiency in keyword matching and the observation that keyword overlap is more common among methods within the same class as the query. Yet the Transformer-based model GraphCodeBERT excels in understanding the underlying code semantics, resulting in superior performance of ATHENA$_{ct}$ in both Setting 2 and 3. However, the improvements from Setting 2 and 3 do not all contribute to the performance gain for Setting 1. The reason behind this is that LSI tends to rank all methods in the same class as the query higher than those in other classes and methods in the same class are more likely to be actually affected as indicated by the ratio of ground-truth impact set size to the corpus size based on Table 1. Consequently, LSI results in better relative performance in Setting 1 (*i.e.,* smaller improvement margin got by ATHENA$_{ct}$) than in Setting 3, but this does not change the relative positions of methods within the same class (Setting 2) or methods in different classes (Setting 3). More evidence supporting this explanation is provided in **RQ$_2$**. In addition, when integrating global dependence information into local code semantics, ATHENA substantially outperforms LSI by 10.34%/9.55% and 10.22%/6.73% mRR/mAP in Setting 1 and 3 respectively. ATHENA considers neighbor methods within two orders (hops) in dependence graphs for embedding propagation.

Table 4. Ablation Study of ATHENA on mRR and mAP

Settings	Encoders				# neighbor orders				[CLS] token		pretrain-only		+comm.		clone detect.	
	CodeBERT		UniXcoder		1 order		3 orders									
	mRR	mAP	mRR	mAP	mRR	mAP	mRR	mAP	mRR	mAP	mRR	mAP	mRR	mAP	mRR	mAP
whole	58.40	33.37	60.19	34.61	59.42	34.33	59.90	34.73	56.36	32.10	59.92	32.86	59.92	34.96	47.26	22.72
inner	74.68	64.74	75.87	66.18	75.95	66.26	74.94	65.20	73.74	63.83	75.62	65.94	75.12	65.37	71.18	61.11
outer	43.09	22.08	43.93	22.64	43.80	22.56	44.66	23.12	42.67	22.12	41.48	19.99	45.11	23.54	32.42	14.43

5.2 RQ$_2$: The Impact of Call Dependence and Class Member Dependence

In Table 3, we also present the performance of ATHENA when utilizing either the call (*i.e.,*ATHENA$_{ct+cd}$) or the class member dependences (*i.e.,*ATHENA$_{ct+cmd}$) for embedding propagation based on dependence graphs, which allows us to investigate how each type of dependency contributes to the effectiveness of ATHENA in IA. By comparing both ATHENA$_{ct+cd}$ and ATHENA$_{ct+cmd}$ with ATHENA$_{ct}$, we observed that both of them outperform ATHENA$_{ct}$ and their improvements in Setting 1 (whole) are also attributed to the improvements in Setting 3 (outer). This confirms the accuracy of our dependence graph generator when capturing either the call or class member dependence.

Although ATHENA$_{ct+cd}$ and ATHENA$_{ct+cmd}$ obtain comparable results in Setting 2 and Setting 3, ATHENA$_{ct+cmd}$ outperforms ATHENA$_{ct+cd}$ in Setting 1 by 5.29%/4.07% on mRR/mAP. This is because in ATHENA$_{ct+cmd}$, the query method is integrated with the information from all the other methods in the same class. As such it ranks all these methods higher than those in other classes, as previously described in Section 5.1. To further support this explanation, we experimented with another strategy for considering only class member dependence. Instead of using embedding propagation, we directly reduce the cosine distance of the query method and each method within the same class as the query by 50% for IA. The results are quite good in Setting 1 (60.72%/37.23% mRR/mAP), but as expected it behaves exactly the same as ATHENA$_{ct}$ in Setting 2 and 3 because while all methods in the same class are drawn closer to the query, the relative positions of methods in the same class or those in other classes remain unchanged. In addition, when comparing both ATHENA$_{ct+cd}$ and ATHENA$_{ct+cmd}$ with ATHENA, both contribute to ATHENA's effectiveness particularly in Setting 1+3.

5.3 RQ$_3$: Ablation Study

Table 4 illustrates the various configurations of ATHENA for the ablation study. Specifically, we first conducted experiments using different pre-trained Transformer-based code models, namely CodeBERT and UniXcoder. Both of them were also fine-tuned on the code search task in order to transfer additional knowledge learned from code search to IA, similar to our approach with GraphCodeBERT. Also, we follow the procedures recommended in the corresponding papers for finetuning and IA evaluation (*e.g.,* AST is only used for UniXcoder pretraining, but not for finetuning and evaluation). Since CodeBERT only considers sequential code information during pretraining and finetuning, the method representations obtained by CodeBERT are not as meaningful as those obtained by GraphCodeBERT, which results in poorer performance than ATHENA on IA. On the other hand, UniXcoder's IA results are comparable to GraphCodeBERT for IA in Setting 1 (whole), but it does not perform as well as GraphCodeBERT in Setting 3 (outer). This may be due to the fact that UniXcoder only utilizes AST information in pretraining, but not in finetuning and evaluation, unlike GraphCodeBERT, which utilizes data flow in all these phases, thus benefiting the understanding of the underlying code intent. Moreover, we experimented with neighbor methods of different orders (1 and 3) for embedding propagation for IA, and the results showed that utilizing neighbor methods within two orders (ATHENA) is the optimal choice. Although considering the third order involves more dependent methods and requires more computational resources, it does not improve the IA performance.

Table 5. The evaluation results of LSI and ATHENA on the filtered ALEXANDRIA and its tangled counterpart.

Settings	LSI				ATHENA$_{ct}$				ATHENA			
	tangled		untangled		tangled		untangled		tangled		untangled	
	mRR	mAP	mRR	mAP	mRR	mAP	mRR	mAP	mRR	mAP	mRR	mAP
whole	52.94	17.51	**58.42**	**18.88**	54.93	19.56	**60.55**	**20.86**	64.56	23.71	**68.36**	**24.88**
inner	80.72	70.36	**82.17**	**71.50**	81.37	69.03	**82.67**	**70.69**	81.81	69.16	**83.65**	**71.05**
outer	37.72	11.45	**42.89**	**12.80**	41.31	15.10	**46.06**	**15.72**	47.53	16.41	**50.65**	**17.09**

Moreover, instead of taking the average output of all hidden states from the final layer, we experimented with using the output of the [CLS] token of the Transformer-based model (*i.e.,* GraphCodeBERT) as the initial method representation for ATHENA. While the output of the [CLS] token is widely used for code understanding-related tasks (*e.g.,* code search), taking the average output of all hidden states is more suitable for representing code semantics for IA, according to the results showed in Table 3 and Table 4. We also conducted experiments by removing the code search fine-tuning of ATHENA and using the pre-trained GraphCodeBERT directly for initial method embedding extraction, but the pretrained GraphCodeBERT is less effective than the fine-tuned one (ATHENA) for IA especially in Setting 3 (by 3.59%/3.42% mRR/mAP). The reason is that during the code search finetuning, the code is mapped closer to its corresponding NL description, further enhancing the model's ability of understanding the underlying code intent and thereby improving ATHENA's effectiveness. In addition, we add the comment information to the input of ATHENA, but the benefit isn't obvious, probably because our IA evaluation benchmark ALEXANDRIA directly collect developer-written methods from commit history, resulting in some methods having (documentation) comments while others do not (in a realistic setting for IA), which may negatively affect the similarity computation between methods. However, the CodeSearchNet dataset used for code search fine-tuning is well-curated to ensure each code snippet is paired with its corresponding NL description (*i.e.,* the first line of the documentation comment). Therefore, for the sake of efficiency, our final version of ATHENA takes code-only information as input with data flow extracted for IA.

In addition, we replace code search with clone detection to use it as a proxy for IA. Specifically, we finetuned the GraphCodeBERT for clone detection following the same pipeline recommended by [Guo et al. 2020]. Instead of generating separate code embeddings, the model directly produces the probability of whether two code snippets can yield similar results, and as a result, the embedding propagation strategy cannot be applied. Therefore, we utilize the generated probability scores to obtain a ranked list for IA and compare it with ATHENA$_{ct}$ (without embedding propagation). However, from Table 3 and Table 4, we observe that using clone detection as a proxy is less effective than ATHENA$_{ct}$ using code search.

5.4 RQ$_4$: ATHENA and Baseline Performance on the Tangled Benchmark

In Table 5, we present the evaluation results of the best performing baseline LSI, ATHENA$_{ct}$, and ATHENA on the filtered ALEXANDRIA and its corresponding tangled counterpart using the mRR and mAP metrics. Specifically, after comparing our IA benchmark ALEXANDRIA with its tangled counterpart, we extract the tasks with inconsistent (query, ground-truth impact set) pairs and conduct experiments on these filtered tasks from ALEXANDRIA (untangled) and its tangled counterpart respectively. The statistics of the filtered datasets are described in Section 4.1. As observed in Table 5, there is a significant performance difference between untangled ALEXANDRIA and its tangled counterpart across three settings when using any of the models, especially on mRR (ranging from 3.80% to 5.62% in Setting 1). However, existing IA benchmarks are typically built from tangled/original commits, which affects the reliability of evaluation results of previous IA techniques. Moreover, as expected, each of the three models perform better on untangled ALEXANDRIA than on the tangled version across 3 settings. The reason is that each co-changed set in

Table 6. Effectiveness of ATHENA for each software system

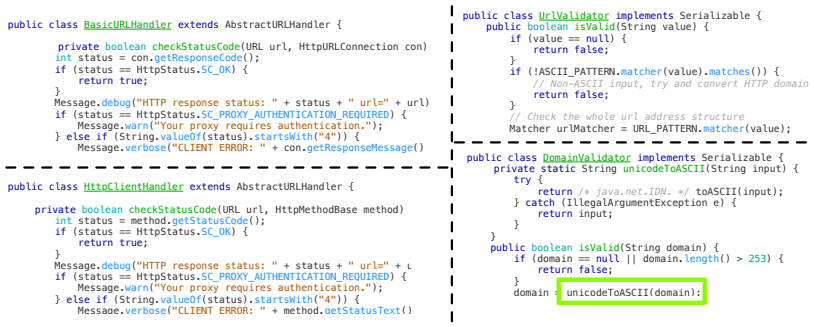
Repo Name	LOC(k)	# Commits	# queries	ATHENA			LSI		
				mRR	mAP	HIT@10	mRR	mAP	HIT@10
ant-ivy	412.3	176	785	**50.19**	**26.47**	**72.36**	39.79	18.48	60.64
archiva	361.2	2	43	**70.81**	**32.17**	**88.37**	69.39	10.17	**88.37**
commons-bcel	168.3	18	138	**66.07**	**30.79**	**87.68**	57.76	21.68	71.74
commons-beanutils	67.5	11	42	65.64	**44.58**	**95.24**	**67.67**	43.64	83.33
commons-codec	55.1	8	41	**67.78**	**52.79**	**90.24**	57.65	34.91	78.05
commons-collections	136.3	15	73	**47.84**	**24.80**	**84.93**	41.43	18.85	68.49
commons-compress	147.3	61	260	**51.67**	**32.99**	**68.85**	45.26	23.73	66.15
commons-configuration	72.9	65	253	**56.89**	**36.87**	**78.26**	41.04	24.75	58.10
commons-dbcp	55.6	21	91	**67.17**	**52.55**	**92.31**	61.73	46.65	84.62
commons-digester	89.7	8	22	**38.65**	**29.07**	**77.27**	28.05	23.86	45.46
commons-io	102.5	19	58	**64.34**	**49.13**	**91.38**	52.16	32.53	75.86
commons-jcs	164	26	221	**70.35**	**26.10**	**85.07**	61.02	18.86	76.92
commons-lang	192.5	36	115	**67.16**	**56.23**	**89.57**	58.38	46.66	80.87
commons-math	431.1	124	589	**65.93**	**42.02**	**87.44**	52.43	29.20	73.35
commons-net	58.2	44	171	**66.59**	**44.59**	**84.80**	51.02	26.35	70.18
commons-scxml	43.8	28	114	**50.32**	**34.62**	**75.44**	45.82	31.19	72.81
commons-validator	42.3	12	35	**62.74**	**56.51**	**85.71**	51.70	40.29	74.29
commons-vfs	91.2	40	166	**55.02**	**36.62**	**83.13**	51.30	35.71	74.10
deltaspike	174.2	2	5	**60.98**	**57.65**	**60.00**	35.04	27.25	**60.00**
giraph	200.6	68	527	**70.80**	**38.40**	**89.75**	59.01	26.78	81.59
gora	132.4	40	174	**49.31**	**26.93**	**68.97**	41.91	23.59	62.64
jspwiki	439.4	1	12	87.50	40.03	**100.00**	**100.00**	**70.28**	**100.00**
opennlp	293.5	33	141	**64.61**	**40.16**	**78.72**	52.13	28.94	69.50
parquet	177.6	50	324	**60.09**	**25.92**	**81.48**	48.27	17.65	67.28
systemml	4000	2	5	**47.15**	**41.60**	**80.00**	41.63	31.25	40.00

ALEXANDRIA was manually verified to address one single concern, ensuring that methods within it are truly impacted by each other. In contrast, the tangled counterpart is built from original/unvetted commits and the methods within each co-changed set may not all contribute to one concern, thus not necessarily be impacted by each other. Therefore, Identifying the methods that are necessarily impacted with respect to the query is harder for each of the representative models.

5.5 RQ_5: Qualitative Analyses on IA Tasks

We begin our analysis of IA tasks by looking at the performance of our studied techniques across different studied software projects. Table 6 provides a finer-grained picture of the improvements per repository our ATHENA model achieves over the LSI baseline. As shown, ATHENA improves performance on 24 of 25 repositories in terms of mAP and 23 of 25 in terms of mRR in setting 1 (whole). For the failing repository *commons-beanutils*, we found that ATHENA substantially outperforms LSI in setting 3 (34.97%/30.58% vs.18.68%/12.37% mRR/mAP), but not in setting 2 (75.35%/64.92% vs. 88.37%/80.20% mRR/mAP). As for the repository *jspwiki*, it contains a single commit with 12 methods in the constructed co-changed set, which corresponds to 12 IA tasks. Among these 12 methods, 6 methods belong to one class, and the remainder are from another class. After investigating the failed tasks, we found that LSI was able to identify the affected methods quite well when the query and the ground truth methods had similar code lengths and a lot of keyword overlap, especially when they belonged to the same class. Now that we have examined the performance of ATHENA across IA tasks at a repository level, we will now discuss some exemplars from our benchmark that showcase how incorporating both structural information and semantic information can benefit IA.

Example 1: The Importance of Semantics. Figure 2 (a) shows two methods from different classes. The top method `checkStatusCode_URL_HttpURLConnection` from class `BasicURLHandler` is the query method and the bottom method `checkStatusCode_URL_HttpMethodBase` is in the corresponding ground-truth impact set. This is representative of conceptual coupling [Poshyvanyk et al. 2009], where the concepts of the two methods, *i.e.,* both performing a check on a status code, couples them together making it more likely that a change in one would result in a change in the other.



(a) **(b)**

Fig. 2. Two qualitative examples for illustrating the effectiveness of ATHENA.

Utilizing the semantic information between the methods, either through a traditional LSI or a Transformer-based neural model is necessary to determine that these two methods are highly related. Since they are not structurally dependent (via call or class member dependencies), structural dependence-only approach is likely to fail on this scenario.

Example 2: The Importance of Richer Semantics and Integration of Dependence Graphs. Figure 2 (a) illustrates a scenario with three methods from two different classes, where the method `isValid` from the class `UrlValidator` is the query, and the method `unicodeToASCII` and `isValid` from the class `DomainValidator` are in the ground-truth impact set. In this scenario, the baseline LSI ranks the `unicodeToASCII` method quite high at 589 due to the limited keyword overlap. When using ATHENA$_{ct}$ (without embedding propagation), which leverages GraphCodeBERT for better code understanding, the rank of the `unicodeToASCII` method improves to 137. However, it's still relatively high, which means developers might need substantial effort to locate this method. Remarkably, our ATHENA achieves a rank of 36, significantly outperforming the baseline. To understand why this occurred, we found that the method `isValid` in the `DomainValidator` class calls the `unicodeToASCII` method, which means these two methods have both call and class member dependencies. Through embedding propagation of ATHENA, the `unicodeToASCII` method is updated with information from the `isValid` method (in the `DomainValidator` class) that is more semantically similar to the query. This additional information helps improve the rank of the ground truth, even though there is no direct dependence relationship between the query and the `unicodeToASCII`.

As can be observed from these examples, there are clear benefits when code understanding is enhanced by the Transformer-based neural model and structural dependence graphs, and we saw this pattern hold after investigating additional cases where ATHENA outperforms the baseline LSI. The contextual information obtained from the global call/class member dependencies among methods enriches the original semantics of the methods, which indeed helps to identify the impact set associated with the given query.

6 THREATS TO VALIDITY

6.1 Internal Validity

To reduce potential issues from internal threats to validity, we experimented with three different DL models when validating our proposed approach of incorporating program dependence graph information into local code semantics to improve IA. Additionally, we constructed our benchmark from commits that have been manually annotated and had the changes made to fix bugs untangled from other changes such as ones to documentation to ensure our benchmark is more reliable.

6.2 External Validity

To lessen the potential for threats to external validity, we used a significantly larger set of projects, 25 compared to previous work that used around five, and tested our appraoch across different DL models to show generalizability. One potential issue with generality is that we only evaluated our approach on Java and Apache projects, therefore, our approach may not generalize to other programming languages such as Python or to different types of projects. However, the DL models we used have shown success across multiple programming languages and so most likely the same would apply to our approach.

7 CONCLUSION

In this paper, we introduce Athena, a novel technique for impact analysis that combines Transformer-based neural code semantics with structural dependence graphs. Additionally, we established a large benchmark for impact analysis, which has been manually verified for bug-fixing commits. On our new benchmark, Athena demonstrates significant improvements over the simple conceptual baseline (+10.34% mRR, +9.55% mAP, and +11.68% HIT@10) and exhibits robust performance across software systems, with 23 out of 25 systems showing improvement. Furthermore, our analysis reveals that Athena's performance boost lies in its ability to more effectively identify impacted methods when they are outside the query method's class.

ACKNOWLEDGMENTS

This research has been supported in part by the following NSF grants: CCF-2311469, CCF-2311468, CNS-2132281, CCF-2007246, and CCF-1955853. We also acknowledge support from Cisco Systems. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

REFERENCES

Mithun Acharya and Brian Robinson. 2011. Practical change impact analysis based on static program slicing for industrial software systems. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 746–755. https://doi.org/10.1145/1985793.1985898

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Online, 2655–2668. https://doi.org/10.18653/v1/2021.naacl-main.211

Robert S Arnold. 1996. *Software change impact analysis*. IEEE Computer Society Press.

Linda Badri, Mourad Badri, and Daniel St-Yves. 2005. Supporting predictive change impact analysis: a control call graph based technique. In *12th Asia-Pacific Software Engineering Conference (APSEC'05)*. 9 pp.–. https://doi.org/10.1109/APSEC.2005.100

Markus Borg, Krzysztof Wnuk, Björn Regnell, and Per Runeson. 2017. Supporting Change Impact Analysis Using a Recommendation System: An Industrial Case Study in a Safety-Critical Context. *IEEE Transactions on Software Engineering* 43, 07 (jul 2017), 675–700. https://doi.org/10.1109/TSE.2016.2620458

Ben Breech, Anthony Danalis, Stacey Shindo, and Lori Pollock. 2004. Online impact analysis via dynamic compilation technology. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* 453–457. https://doi.org/10.1109/ICSM.2004.1357834

Ben Breech, Mike Tegtmeyer, and Lori Pollock. 2006. Integrating Influence Mechanisms into Impact Analysis for Increased Precision. In *2006 22nd IEEE International Conference on Software Maintenance*. 55–65. https://doi.org/10.1109/ICSM.2006.33

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural*

Information Processing Systems (Vancouver, BC, Canada) *(NIPS'20)*. Curran Associates Inc., Red Hook, NY, USA, Article 159, 25 pages. https://doi.org/10.5555/3495724.3495883

Max Brunsfeld, Patrick Thomson, Andrew Hlynskyi, Josh Vera, Phil Turnbull, Timothy Clem, Douglas Creager, Andrew Helwer, Rob Rix, Hendrik van Antwerpen, Michael Davis, Ika, Tuan-Anh Nguyen, Stafford Brunk, Niranjan Hasabnis, bfredl, Mingkai Dong, Vladimir Panteleev, ikrima, Steven Kalt, Kolja Lampe, Alex Pinkus, Mark Schmitz, Matthew Krupcale, narpfel, Santos Gallegos, Vicent Martí, Edgar, and George Fraser. 2022. *tree-sitter/tree-sitter: v0.20.7*. https://doi.org/10.5281/zenodo.7045041

Haipeng Cai. 2020. A Reflection on the Predictive Accuracy of Dynamic Impact Analysis. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 562–566. https://doi.org/10.1109/SANER48275.2020.9054806

Haipeng Cai and Raul Santelices. 2014. Diver: precise dynamic impact analysis using dependence-based trace pruning. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. Association for Computing Machinery, New York, NY, USA, 343–348. https://doi.org/10.1145/2642937.2642950

Haipeng Cai and Raul Santelices. 2015. A comprehensive study of the predictive accuracy of dynamic change-impact analysis. *Journal of Systems and Software* 103 (2015), 248–265. https://doi.org/10.1016/j.jss.2015.02.018

Haipeng Cai, Raul Santelices, and Siyuan Jiang. 2016a. Prioritizing Change-Impact Analysis via Semantic Program-Dependence Quantification. *IEEE Transactions on Reliability* 65, 3 (2016), 1114–1132. https://doi.org/10.1109/TR.2015.2481000

Haipeng Cai, Raúl A. Santelices, and Douglas Thain. 2016b. DiaPro: Unifying Dynamic Impact Analyses for Improved and Variable Cost-Effectiveness. *ACM Trans. Softw. Eng. Methodol.* 25, 2 (2016), 18:1–18:50. https://doi.org/10.1145/2894751

Haipeng Cai and Douglas Thain. 2016. DistIA: a cost-effective dynamic impact analysis for distributed programs. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE '16)*. Association for Computing Machinery, New York, NY, USA, 344–355. https://doi.org/10.1145/2970276.2970352

Gerardo Canfora, Michele Ceccarelli, Luigi Cerulo, and Massimiliano Di Penta. 2010. Using multivariate time series and association rules to detect logical change coupling: An empirical study. In *2010 IEEE International Conference on Software Maintenance*. 1–10. https://doi.org/10.1109/ICSM.2010.5609732

Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the Properties of Neural Machine Translation: Encoder–Decoder Approaches. In *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, Dekai Wu, Marine Carpuat, Xavier Carreras, and Eva Maria Vecchi (Eds.). Association for Computational Linguistics, Doha, Qatar, 103–111. https://doi.org/10.3115/v1/W14-4012

Michael L. Collard, Michael John Decker, and Jonathan I. Maletic. 2013. srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration. In *2013 IEEE International Conference on Software Maintenance*. 516–519. https://doi.org/10.1109/ICSM.2013.85

Jose Luis de la Vara, Markus Borg, Krzysztof Wnuk, and Leon Moonen. 2016. An Industrial Survey of Safety Evidence Change Impact Analysis Practice. *IEEE Transactions on Software Engineering* 42, 12 (2016), 1095–1117. https://doi.org/10.1109/TSE.2016.2553032

Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. 2016. Convolutional neural networks on graphs with fast localized spectral filtering. In *Proceedings of the 30th International Conference on Neural Information Processing Systems* (Barcelona, Spain) *(NIPS'16)*. Curran Associates Inc., Red Hook, NY, USA, 3844–3852. https://doi.org/10.5555/3157382.3157527

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/V1/N19-1423

Bogdan Dit, Meghan Revelle, and Denys Poshyvanyk. 2013. Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. *Empirical Softw. Engg.* 18, 2 (apr 2013), 277–309. https://doi.org/10.1007/s10664-011-9194-4

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

Stephen Fink and Julian Dolby. 2012. WALA–The TJ Watson Libraries for Analysis.

Malcom Gethers, Bogdan Dit, Huzefa Kagdi, and Denys Poshyvanyk. 2012. Integrated impact analysis for managing software changes. In *2012 34th International Conference on Software Engineering (ICSE)*. 430–440. https://doi.org/10.1109/ICSE.2012.6227172

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, Dublin, Ireland, 7212–7225. https://doi.org/10.18653/v1/2022.acl-long.499

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2020. GraphCodeBERT: Pre-training Code Representations with Data Flow. *CoRR* abs/2009.08366 (2020). arXiv:2009.08366 https://arxiv.org/abs/2009.08366

Alex Gyori, Shuvendu K. Lahiri, and Nimrod Partush. 2017. Refining interprocedural change-impact analysis using equivalence relations. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 318–328. https://doi.org/10.1145/3092703.3092719

David K. Hammond, Pierre Vandergheynst, and Rémi Gribonval. 2011. Wavelets on graphs via spectral graph theory. *Applied and Computational Harmonic Analysis* 30, 2 (2011), 129–150. https://doi.org/10.1016/j.acha.2010.04.005

Steffen Herbold, Alexander Trautsch, Benjamin Ledel, Alireza Aghamohammadi, Taher Ahmed Ghaleb, Kuljit Kaur Chahal, Tim Bossenmaier, Bhaveet Nagaria, Philip Makedonski, Matin Nili Ahmadabadi, Kristóf Szabados, Helge Spieker, Matej Madeja, Nathaniel Hoy, Valentina Lenarduzzi, Shangwen Wang, Gema Rodríguez-Pérez, Ricardo Colomo Palacios, Roberto Verdecchia, Paramvir Singh, Yihao Qin, Debasish Chakroborti, Willard Davis, Vijay Walunj, Hongjun Wu, Diego Marcilio, Omar Alam, Abdullah Aldaeej, Idan Amit, Burak Turhan, Simon Eismann, Anna-Katharina Wickert, Ivano Malavolta, Matús Sulír, Fatemeh H. Fard, Austin Z. Henley, Stratos Kourtzanidis, Eray Tuzun, Christoph Treude, Simin Maleki Shamasbi, Ivan Pashchenko, Marvin Wyrich, James Davis, Alexander Serebrenik, Ella Albrecht, Ethem Utku Aktas, Daniel Strüber, and Johannes Erbel. 2020. Large-Scale Manual Validation of Bug Fixing Commits: A Fine-grained Analysis of Tangling. *CoRR* abs/2011.06244 (2020). arXiv:2011.06244 https://arxiv.org/abs/2011.06244

Kim Herzig, Sascha Just, and Andreas Zeller. 2015. The impact of tangled code changes on defect prediction models. *Empirical Software Engineering* 21 (04 2015). https://doi.org/10.1007/s10664-015-9376-6

Kim Herzig and Andreas Zeller. 2013. The impact of tangled code changes. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. 121–130. https://doi.org/10.1109/MSR.2013.6624018

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *CoRR* abs/1909.09436 (2019). arXiv:1909.09436 http://arxiv.org/abs/1909.09436

Mohammad-Amin Jashki, Reza Zafarani, and Ebrahim Bagheri. 2008. Towards a more efficient static software change impact analysis method. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Atlanta, Georgia) *(PASTE '08)*. Association for Computing Machinery, New York, NY, USA, 84–90. https://doi.org/10.1145/1512475.1512493

Zijian Jiang, Ye Wang, Hao Zhong, and Na Meng. 2019. Automatic Method Change Suggestion to Complement Multi-Entity Edits. *Journal of Systems and Software* 159 (10 2019), 110441. https://doi.org/10.1016/j.jss.2019.110441

Huzefa Kagdi, Malcom Gethers, and Denys Poshyvanyk. 2012. Integrating conceptual and logical couplings for change impact analysis in software. *Empirical Software Engineering* 18 (10 2012). https://doi.org/10.1007/s10664-012-9233-9

Huzefa Kagdi, Malcom Gethers, Denys Poshyvanyk, and Michael L. Collard. 2010. Blending Conceptual and Evolutionary Couplings to Support Change Impact Analysis in Source Code. In *2010 17th Working Conference on Reverse Engineering*. 119–128. https://doi.org/10.1109/WCRE.2010.21

Yoon Kim. 2014. Convolutional Neural Networks for Sentence Classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Alessandro Moschitti, Bo Pang, and Walter Daelemans (Eds.). Association for Computational Linguistics, Doha, Qatar, 1746–1751. https://doi.org/10.3115/v1/D14-1181

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1412.6980

Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=SJU4ayYgl

Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. 2014. Hey! are you committing tangled changes?. In *Proceedings of the 22nd International Conference on Program Comprehension* (Hyderabad, India) *(ICPC 2014)*. Association for Computing Machinery, New York, NY, USA, 262–265. https://doi.org/10.1145/2597008.2597798

Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. 2016. Splitting Commits via Past Code Changes. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*. 129–136. https://doi.org/10.1109/APSEC.2016.028

Pavneet Singh Kochhar, Yuan Tian, and David Lo. 2014. Potential biases in bug localization: do they matter?. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. Association

for Computing Machinery, New York, NY, USA, 803–814. https://doi.org/10.1145/2642937.2642997

Hongyu Kuang, Patrick Mäder, Hao Hu, Achraf Ghabi, Liguo Huang, Lv Jian, and Alexander Egyed. 2012. Do data dependencies in source code complement call dependencies for understanding requirements traceability? 181–190. https://doi.org/10.1109/ICSM.2012.6405270

Quoc V. Le and Tomás Mikolov. 2014. Distributed Representations of Sentences and Documents. *CoRR* abs/1405.4053 (2014). arXiv:1405.4053 http://arxiv.org/abs/1405.4053

Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. 2013. A survey of code-based change impact analysis techniques. *Software Testing, Verification and Reliability* 23 (12 2013). https://doi.org/10.1002/stvr.1475

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *CoRR* abs/2102.04664 (2021). arXiv:2102.04664 https://arxiv.org/abs/2102.04664

Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1301.3781

Chris Mills, Esteban Parra, Jevgenija Pantiuchina, Gabriele Bavota, and Sonia Haiduc. 2020. On the relationship between bug reports and queries for text retrieval-based bug localization. *Empirical Software Engineering* 25 (09 2020). https://doi.org/10.1007/s10664-020-09823-w

Hoan Anh Nguyen, Anh Tuan Nguyen, and Tien N. Nguyen. 2013. Filtering noise in mixed-purpose fixing commits to improve defect prediction and localization. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. 138–147. https://doi.org/10.1109/ISSRE.2013.6698913

Denys Poshyvanyk, Andrian Marcus, Rudolf Ferenc, and Tibor Gyimóthy. 2009. Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering* 14 (02 2009), 5–32. https://doi.org/10.1007/s10664-008-9088-2

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2019. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *CoRR* abs/1910.10683 (2019). arXiv:1910.10683 http://arxiv.org/abs/1910.10683

Sable Research Group. 2023. Soot: A Java Bytecode Optimization Framework. https://soot-oss.github.io/soot/.

Mark Sherriff and Laurie Williams. 2008. Empirical Software Change Impact Analysis using Singular Value Decomposition. In *2008 1st International Conference on Software Testing, Verification, and Validation*. 268–277. https://doi.org/10.1109/ICST.2008.25

Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A. Osborne. 2011. Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities. *IEEE Transactions on Software Engineering* 37, 6 (2011), 772–787. https://doi.org/10.1109/TSE.2010.81

Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, and Mohammad Mamun Mia. 2014. Towards a Big Data Curated Benchmark of Inter-project Code Clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 476–480. https://doi.org/10.1109/ICSME.2014.77

Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. 2012. How do software engineers understand code changes? an exploratory study in industry. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina) *(FSE '12)*. Association for Computing Machinery, New York, NY, USA, Article 51, 11 pages. https://doi.org/10.1145/2393596.2393656

Marco Torchiano and Filippo Ricca. 2010. Impact analysis by means of unstructured knowledge in the context of bug repositories. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (Bolzano-Bozen, Italy) *(ESEM '10)*. Association for Computing Machinery, New York, NY, USA, Article 47, 4 pages. https://doi.org/10.1145/1852786.1852847

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

Min Wang, Zeqi Lin, Yanzhen Zou, and Bing Xie. 2019. CoRA: Decomposing and Describing Tangled Code Changes for Reviewer. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1050–1061. https://doi.org/10.1109/ASE.2019.00101

Wei Wang, Yun He, Tong Li, Jiajun Zhu, and Jinzhuo Liu. 2018. An Integrated Model for Information Retrieval Based Change Impact Analysis. *Scientific Programming* 2018 (03 2018), 1–13. https://doi.org/10.1155/2018/5913634

Xin Wang, Yasheng Wang, Pingyi Zhou, Fei Mi, Meng Xiao, Yadao Wang, Li Li, Xiao Liu, Hao Wu, Jin Liu, and Xin Jiang. 2021b. CLSEBERT: Contrastive Learning for Syntax Enhanced Code Pre-Trained Model. *CoRR* abs/2108.04556 (2021). arXiv:2108.04556 https://arxiv.org/abs/2108.04556

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021a. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685

Yanfu Yan, Nathan Cooper, Kevin Moran, Gabriele Bavota, Deny Poshyvanyk, and Steve Rich. 2024. Athena's Online Appendix https://github.com/yanyanfu/Athena.

Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. 2019. *XLNet: generalized autoregressive pretraining for language understanding*. Curran Associates Inc., Red Hook, NY, USA.

T. Zimmermann, P. Weibgerber, S. Diehl, and A. Zeller. 2004. Mining version histories to guide software changes. In *Proceedings. 26th International Conference on Software Engineering*. 563–572. https://doi.org/10.1109/ICSE.2004.1317478

Received 2023-09-29; accepted 2024-01-23