From Seed to Scope: Reasoning to Identify Change Impact Sets

Aashish Yadavally
University of Texas at Dallas
Dallas, USA
aashish.yadavally@utdallas.edu

Tien N. Nguyen
University of Texas at Dallas
Dallas, USA
tien.n.nguyen@utdallas.edu

Abstract

Change impact analysis (IA), which identifies the set of co-changed program elements (*i.e.*, impact set), is critical for several software engineering tasks. However, existing IA approaches struggle with a trade-off between precision (correctly detecting impacted elements) and recall (detecting all relevant ones). More importantly, they are limited in intent-aware settings, where co-changing elements are determined by a given change intent. In this work, we propose Ripple, an intent-aware IA approach that leverages large language models (LLMs) to capture change dependencies by linking intent to program elements and estimating their co-change relationships. To address the *precision-recall tradeoff*, we adopt a two-phase design: (1) a *seed-to-scope* expansion strategy that expands the impact set using evolutionary and dependence coupling to improve recall, and (2) a *plan-then-predict* strategy where an LLM-generated change plan refines impact estimation for higher precision. We evaluate Ripple on real-world commits from Apache projects, achieving a 39.7%–380.8% improvement in F1-score over existing IA approaches. In addition, Ripple introduces flexibility, allowing users to prioritize higher precision or recall based on their preferences.

CCS Concepts

• **Computing methodologies** → **Neural networks**; • **Software and its engineering**;

Keywords

AI4SE, Change Impact Analysis, Large Language Models

ACM Reference Format:
Aashish Yadavally and Tien N. Nguyen. 2026. From Seed to Scope: Reasoning to Identify Change Impact Sets. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3773265

1 Introduction

Software undergoes continual changes for enhancement and maintenance throughout its lifecycle. However, every modification can have a *ripple effect* across the codebase, making it crucial to identify which parts will be affected. This process, known as *(change) impact analysis (IA)*, plays a vital role in regression testing [18, 44, 45], bug fixing [11, 44], and software maintainability [16].

To this end, researchers have proposed several automated approaches targeting the IA problem with two primary inputs or levels of abstraction: source code [43] and change requests [9, 46]. Those leveraging source code typically determine the initial location of the change (via feature location techniques [15]) and subsequently use it to identify *the impact set*. They can further be categorized into *top-down* strategies, that analyze the structural, semantic, conceptual, or execution information; and *bottom-up* strategies, which adopt mining from commit histories to infer co-changes.

Precision-Recall Tradeoff. Among the top-down solutions, *first*, there are structural approaches, which analyze relationships between program elements, e.g., class hierarchies, field accesses, *etc.* for impact analysis [38]. For example, modifications to a base class may require updating its derived classes. *Second*, the semantic IA approaches explore dependencies between program elements including data/control dependencies, and calling relations. Despite having a high recall by capturing structural/semantic relations, such analyses tend to generate large impact sets with a *low precision* [32].

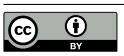
Third, conceptual IA techniques rely on shared semantic purposes between program elements (*e.g.*, methods or classes) as a proxy to generate impact sets. They formulate IA as an information retrieval (IR) task to model the conceptual relationships between program entities [42]. However, they struggle to detect changes lacking textual or semantic similarity, resulting in a *lower recall*.

Fourth, execution-based analyses collect information such as execution traces [40], code coverage [41], and execute-after relations between program elements [7] to compute the impact sets. Such dynamic IA approaches are *more precise* than the static ones. However, they are runtime-intensive while suffering from *low recall*, because they depend on the generalizability of the test suite.

In contrast, the bottom-up IA approaches mine evolutionary coupling among program elements in the commit history [65]. The idea is that the co-change elements in the past are likely to change together in the near future. While the resulting impact set from such evolutionary coupling might have high recall with large impact sets, they have many false positives (*i.e.*, *low precision*). Notably, they are useful to capture co-changing locations that are not captured by traditional program analyses (*i.e.*, detecting isolated changes not captured by explicit structural/semantic dependencies).

An ideal IA approach must strike balance between precision (to avoid overestimating the impact) and recall (to avoid missing critical locations). *Low precision* results in developers having to examine many irrelevant locations, increasing manual effort and potential cognitive overload. Conversely, *low recall* could lead to missing fixing locations, potentially leading to undetected errors. Some attempts have been made to combine different IA techniques with orthogonal information sources to gain this balance [25, 29]. Athena [59] advances this effort by integrating dependence-based and conceptual information using Transformer-based models [51].

What's missing? Despite their strengths, both top-down and bottom-up approaches only focus on varied information sources.



ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3773265

Broadly speaking, an IA solution can approach the IA problem in settings that require knowledge of the *change intent*, *i.e.*, the purpose behind the change, either *implicitly or explicitly*. Such an implicit intent can arise from program semantics, where the goal is to identify all code locations that need to be modified together [32]. For example, when renaming a method, all of its call sites need to be updated due to the implicit intent dictated by the semantics of renaming [43]. In contrast, explicit intent is typically expressed in natural language texts (*e.g.*, bug reports), where the goal is to *determine the code locations to be changed to fulfill the described intent*. Starting from the same initial code location, the set of impacted locations may differ depending on whether the intent is to fix different bugs from bug reports or introduce new features.

By design, those approaches focus on implicit intent IA settings. They rely on change propagation but do not account for how the underlying change intent (*e.g.*, as described in a bug report) influences how impact may be propagated. For instance, evolutionary approaches identify co-changing files based on historical patterns but do not distinguish between a bug fix and a refactoring change, even if the same code is modified for different reasons. Similarly, IR-based conceptual approaches may capture intent at the level of an existing code entity but fail to associate intent with the change itself. This omission leads to imprecise impact predictions, as different change intents require different forms of propagation analysis.

Our Approach. We develop an IA solution tailored to utilize explicit intent, which we term *intent-aware impact analysis*. We posit that understanding the intent/motivation behind a change allows us to better contextualize its propagation and disambiguate co-change relations based on their intended modifications.

We propose Ripple, an intent-aware IA approach that utilizes the reasoning capabilities of large language models (LLMs) *to connect natural language change intent with program elements and estimate their co-change dependencies*. To address the *precision-recall tradeoff*, Ripple adopts a two-phase design. First, we develop a *seed-to-scope* strategy that begins with a seed edit and progressively expands the impact set through a combination of evolutionary and dependence coupling. This *recall-focused* approach helps *cast a wider net*, *i.e.*, capture a full range of change semantics while eliminating irrelevant program elements. In the next *precision-focused* phase, we adopt a *plan-then-predict* strategy, where the Planner LLM first produces *a change plan* via Chain-of-Thought [56] that captures the change intent to derive the structured reasoning steps for impact analysis. To mitigate LLM hallucinations, we slice the repository based on dependence clusters extracted from the expanded impact set. Impact estimation is performed independently by the Reasoner LLM within each cluster, ensuring a localized reasoning. It utilizes the generated change plan for more precise impact set prediction.

We conducted several experiments to evaluate Ripple on a real-world IA benchmark covering Java projects from the Apache Software Foundation. As shown in Figure 1, Ripple-(1) (⋆) outperforms existing top-down (●) and bottom-up (●) IA approaches in both precision and recall, with an improvement of 39.7%–380.8% in F1-score. Ripple captures change dependencies beyond structural/semantic dependencies, which alone yield low precision. This enhances Ripple's performance for large single-file or multi-file commits, for which it improves over the state-of-the-art Athena by 59.7% in F1-score (Section 5.2.2). By default, Ripple adopts *self-consistency* [54]

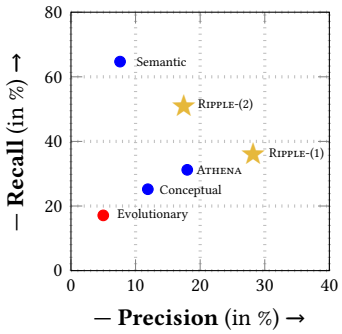


Figure 1: Precision-Recall plot comparing the performance of *top-down* and *bottom-up* IA approaches with Ripple

for precise impact estimation. However, as shown in Figure 1, by switching to an aggregation-based strategy *i.e.*, Ripple-(2) (⋆), we can get relatively less precise impact sets with significantly more coverage–introducing flexibility in users' preferences (Section 6.2).

In brief, this paper makes the following contributions:

(1) Ripple: an intent-aware, reasoning-driven, scalable IA approach with flexibility to balance high precision and high recall.

(2) Empirical evaluation: our extensive experiments demonstrate that Ripple is more effective than state-of-the-art IA approaches.

2 Motivation and Key Ideas

2.1 Motivating Example

Developers frequently modify source code and commit the changes to their code repositories. The term *"(change) commit"* is used to refer to such transactions. A commit includes a textual description and a corresponding *change set*, which comprises changes to one or multiple code locations. These modifications can serve various purposes. A study by Nguyen *et al.* [37] on a large number of commits in open-source projects identified six primary purposes of commits: 1) bug-fixing, 2) enhancement (adding functionality), 3) refactoring, 4) presentation (formatting, cosmetic), 5) code annotation, and 6) documentation enhancement [37]. *The goal of change impact analysis is to automatically identify all affected locations that belong to a change set* [60]. If an initial change location (or *seed edit location*) is identified, the resulting set of impacted locations is referred to as the *impact set* of the seed. In this paper, we use both terms *change set* and *impact set* interchangeably unless a distinction is necessary.

As explained in Section 1, for bug fixes or feature enhancements, the IA problem and solutions must account for *explicit change intents* (or simply *intents*), which are explicitly described in bug reports or change requests. Let us use an example to illustrate a fix in a dependency resolution tool, *Apache Ivy*. From the bug report `IVY-644` [2], as modules in different configurations evict the same dependency, a `NullPointerException` could occur. This issue was addressed in commit `283f77d` [1], where the eviction logic was refined to handle such cases robustly. The changes span four files: those in the `IvyNode` class and the `IvyNodeEviction` class ensure that the eviction scenarios are processed safely without causing `null` dereferences, while updates in the `ResolveEngine` class enforce consistent dependency resolution. Moreover, the `ResolveTest` class was extended to cover this usage in testing, preventing regressions in future updates.

2.2 Observations and Key Ideas

The above example illustrates that changes in these files are specific to this bug whose description is in the bug report IVY-644. Analyzing the report's contents can help determine the intent behind the fix.

Observation 1 (Assessing Impact Through Change Intent). *For certain types of changes (e.g., bug-fixing), assessing the change impact requires explicit knowledge of the change intent. We refer to this problem setting of impact assessment as "intent-aware".*

Key Idea 1 (Intent-aware Impact Analysis). *To address the intent-aware problem setting, one needs impact analysis approaches that consider the change intent that is explicitly expressed in natural-language texts. We refer to it as "intent-aware impact analysis".*

As listed in Section 1, existing IA approaches can be broadly classified into 1) top-down, including structural, semantic, conceptual, or execution-based approaches, and 2) bottom-up, which mine commit histories to infer co-changed files.

Unfortunately, these approaches do not work well for the above example. First, no *structural relations* among methods or classes in the codebase can help identify the impact set comprising the changed classes. Second, semantic IA approaches leverage *calling relations* or *program dependencies* from a seed location, which is the initial code location in the impact set. Identifying a seed location in IA often involves bug localization techniques based on bug reports [36, 63], feature and concept location [15], *etc.* For this example, following from the developer discussions, one way to fix this bug is that every root configuration should be resolved separately prior to computing the conflicts. To this end, the developers decided that the first change, *i.e.*, the seed edit needed to be made in the `ResolveEngine.computeConflicts()` method [1]. However, from this seed edit location, through both direct and indirect semantic relations, no other methods in the impact set can be identified. In addition, these approaches wrongly identify several methods that have structural or semantic dependencies on `ResolveEngine.computeConflicts()` as being potentially affected. Third, the existing conceptual IA approaches are restricted to conceptual similarity, which does not work for this case since the changed elements in that set are not similar. Thus, the combination of semantic dependence and conceptual similarity in the state-of-the-art Athena [59] is not effective in this case either. Finally, execution-based IA approaches require dynamic information, which is not easily available.

The bottom-up IA approaches examine commit history to derive the co-changes via evolutionary coupling. For this example, the files in the impact set for this bug-fixing commit, *i.e.*, `IvyNode.java`, `IvyNodeEviction.java` and `ResolveEngine.java` were also co-changed, all together, on 8 previous instances in the commit history. However, in only one of the commits, they were the only co-changed files, and in the remaining, the total number of co-changed files ranged from 3–282. This shows their unreliability in identifying the impact sets, stemming from: (i) tangled commits, which include multiple unrelated code changes for different purposes [24]; (ii) not considering the intent presented in the bug report. As a result, despite a high recall, the evolutionary coupling approaches are less precise.

In general, the aforementioned IA solutions are effective to some extent. However, as discussed in Section 1, the approaches always have a trade-off between precision and recall. Neither top-down or bottom-up IA approaches alone can fully and accurately capture the range of co-change dependencies introduced by software evolution. Evolutionary coupling through mining provides insights into historical co-changes, but without accounting for semantic and conceptual dependencies, it may miss implicit relationships. Conversely, semantic dependence analysis captures semantic links but lacks historical context, leading to lower recall.

Observation 2 (Precision-Recall Trade-Off). *Existing IA solutions suffer from a trade-off between precision and recall.*

To address this, we hypothesize that *advanced LLMs possess the capability to understand and reason from the change intent to the relevant program elements (i.e., methods or classes) within the corresponding impact set.* We are motivated by their remarkable proficiency in mapping textual descriptions to code in tasks such as code generation/synthesis [10, 27, 39, 55]. Our empirical evaluation (Section 4) validates our hypothesis on this capability of LLMs.

Key Idea 2 (Reasoning with LLMs for Intent-Aware Impact Analysis). *LLMs can reason and infer the relations between change intent and the affected program elements, discovering the co-change dependencies within the corresponding impact set.*

To complement to the use of LLMs in addressing the precision-recall trade-off, we adopt a *seed-to-scope* approach–starting from a seed edit and progressively expanding the impact set through both evolutionary and dependency-based relationships. We posit that commit history reinforces logical dependencies, while structural and semantic dependencies are accounted for in the next step. This ensures a *recall-focused impact estimation*, casting a wider net over potential change propagation. We refer to this as the *dependence-enhanced impact set* ($\mathcal{I}_D$). With $\mathcal{I}_D$, the LLM is guided toward a focused subset of the codebase, eliminating irrelevant files.

Key Idea 3 (From Seed to Scope: Casting a Wider Net). *By integrating evolutionary coupling with dependence-enhanced impact sets, we expand from a seed edit location to a potentially imprecise, yet broader impact scope–capturing a full range of change semantics.*

It is well-known that LLMs struggle to process long, context-rich sequences. In the IA task, the LLMs must analyze entire repositories or their structures and, based on the change intent, reason about change propagation. However, a project might have a large number of classes and methods. As a result, they can encounter difficulties in identifying relevant dependencies, leading to incorrect analysis.

Observation 3 (Diminishing Returns of Large Contexts). *Increasing the context window results in degradation of IA as LLMs do not robustly make use of relevant information.*

To limit the context window and identify semantically-related change modifications, we extract connected components in $\mathcal{I}_D$ over call and class-member dependencies to form dependence clusters. We posit that each dependence cluster represents a logically cohesive unit of change propagation encapsulated in the change intent. By clustering $\mathcal{I}_D$ in this manner, we refine the impact set while reducing the unnecessary reasoning overhead for the LLM.

Key Idea 4 (Dependence Coupling-Based Clustering). *We extract connected components from $\mathcal{I}_D$ over call and class-member dependencies to form dependence clusters, which represent logically cohesive units of change propagation.*

3 Intent-Aware Change Impact Analysis

Impact analysis can be performed at the coarser-grained file-level [8], or fine-grained method [59] and statement-level [22]. Since the latter is cost-prohibitive, we focus on method-level IA. Given the change intent ψ and a seed edit location M_0 for the initial modification, Ripple predicts a set of methods $\mathcal{I}_f = \{M_1, M_2, ..., M_k\}$ within a code repository $\mathcal{R}$ that may potentially be impacted.

3.1 Overview

In Figure 2, we illustrate an overview of our proposed framework for intent-aware impact analysis. Ripple mainly operates in two phases: (1) *recall-focused* impact set generation, and (2) *precision-focused* impact set refinement. In the first phase, we expand from *seed to scope* by leveraging evolutionary coupling to construct an impact set from commit history ($\mathcal{I}_H$). We then further expand this set using dependence coupling, incorporating call and class-member dependencies to capture structural and semantic relationships. This combined approach casts a wider net–ensuring a **high recall** (*i.e.*, minimizing the number of missed impacted locations). Then, we cluster the resulting dependence-enhanced impact set ($\mathcal{I}_D$) into *dependence clusters*, where each represents a connected component of inter-dependent locations in the repository.

The recall-focused phase helps Ripple reduce the context size consisting of a much smaller number of program elements that the LLM must consider in the second phase. In our experiment (Section 4), we show that Ripple is able to reduce an average of 86.4% methods per repository, while maintaining a recall of 76.8%.

In the second phase, we refine the impact set by prompting the LLM to develop a *change plan*, which provides a structured summary of the required modifications as a sequence of change steps. Next, we process each dependence cluster independently, representing it as a repository-aware structure that includes all methods in the cluster along with their summaries. This structured representation provides the LLM with contextual hints about the conceptual meaning of relevant methods, enabling Ripple to **improve precision** by filtering out unnecessary locations while preserving key dependencies. We merge the refined impact subsets corresponding to all dependence clusters to construct $\mathcal{I}_f$, *i.e.*, the final impact set.

3.2 Recall-Focused Impact Set Generation

3.2.1 Impact Set Initialization. The selection of a seed edit location serves as the starting point for impact analyis. Various approaches exist for identifying such locations, including bug/fault localization techniques that analyze bug reports [36, 63] and feature/concept localization methods that map change requests to relevant code [15]. In Figure 2, we denote the seed edit location as M_0, highlighting it in a blue cell. We define the initial impact set I_0 as:

$$I_0 = \{M_0\} \tag{1}$$

3.2.2 Evolutionary Coupling-Based Impact Set Expansion. Previous research has demonstrated the advantages of using commit history to understand programs and their evolution, revealing logical relationships between co-changed locations in the repository [65]. Accordingly, we leverage historical commit data to expand the initial impact set I_0 by identifying additional methods that have historically co-evolved with the seed edit location M_0.

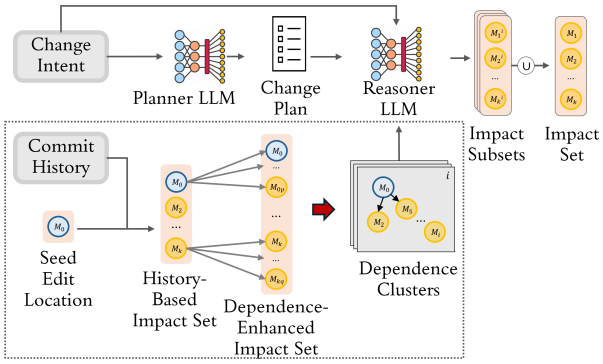


Figure 2: Ripple: Intent-aware change impact analysis

Given the seed edit location M_0, we mine the previous N commits where F_0 (*i.e.*, the file containing M_0) was also modified. We parse each modified file in these commits to extract the modified methods in each file based on the line ranges of the change hunks.

Let $\mathcal{C} = \{C_1, C_2, ..., C_N\}$ denote the set of previous N commits in which F_0 was modified, and $\mathcal{M}^i = \{M_0, M_1^{(i)}, ..., M_j^{(i)}\}$, the set of methods modified in C_i, obtained by parsing the changes. Since this step in Ripple is recall-focused, we do not apply additional precision-enhancing techniques (*e.g.*, association rule mining [47]), prioritizing efficiency and minimizing computational overhead. Accordingly, we define our history-based impact set ($\mathcal{I}_H$) as follows:

$$\mathcal{I}_H = \bigcup_{i=1}^{N} \mathcal{M}^i = \bigcup_{i=1}^{N} \{M_0, M_1^{(i)}, \ldots, M_j^{(i)}\} \tag{2}$$

Note that commit history can be noisy (*e.g.*, due to tangled commits that address multiple concerns at once [24]), incomplete, or even obsolete (*i.e.*, when the software has undergone significant refactoring or deprecated features remain in history). To mitigate these issues, following from our preliminary experiments, we truncate the commit history by considering only the most recent 100 relevant commits (*i.e.*, those where F_0 was modified) for each transaction. Moreover, since we prioritize coverage in this *recall-focused* phase and evolutionary coupling is more effective at capturing co-changing files than co-changing methods (Table 1, row 1), we select all commits where F_0 was modified, regardless of whether M_0 was changed. This strategy also conforms to the principle of the locality of code changes: "files that have been changed recently are likely to be changed in a near future" [21, 30]. Because newly added files have no commit history, we retain the initial impact set as is.

3.2.3 Dependence Coupling-Based Impact Set Expansion. To capture the structural relationships beyond historical co-evolution, we further expand the impact set via dependence coupling. Given the history-based impact set $\mathcal{I}_H$, we construct a dependence coupling-based impact set $\mathcal{I}_D$ by expanding over each method $M_i \in \mathcal{I}_H$ with its direct and indirect calling dependencies, where we limit the latter to at most L calling hops. That is, for all $M_i \in \mathcal{I}_H$, we include M_j in the dependence-expanded set $\mathcal{I}_D$ if:

(i) *direct call dependence*: $M_j \in Call(M_i)$, *i.e.*, M_i calls M_j
(ii) *indirect call dependence*: $\exists\{M_{I_1}, ..., M_{I_L}\}$ s.t. $M_{I_1} \in Call(M_i)$ and $M_{I_2} \in Call(M_{I_1})$,..., $M_j \in Call(M_{I_L})$, *i.e.*, M_i calls M_{I_1}, which in turn, calls M_{I_2},..., and M_{I_L} calls M_j.

(iii) *class-member dependence*: $M_j \in ClassMember(Class(M_i))$, *i.e.*, M_j belongs to the same class as M_i, ensuring that changes to class-wide members are captured.

Therefore, we formulate dependence-enhanced impact set as:

$$\mathcal{I}_D = \bigcup_{M_i \in \mathcal{I}_H} \begin{matrix} \{M_j \in Call^h(M_i) \mid 1 \le h \le L\} \\ \cup \{M_j \in ClassMembers(Class(M_i))\} \end{matrix} \quad (3)$$

3.3 Precision-Focused Impact Set Refinement

Recently, LLMs have exhibited emergent reasoning behaviors in various domains, e.g., arithmetic [31], formal logic [35], and source code [12, 13, 50, 58]. These successes can be attributed to effective *planning* capabilities, which include a structured decomposition and analysis of complex tasks through *multi-step inference* [13, 52, 56].

As illustrated in Figure 2, we leverage "Reasoner LLM" ($\mathcal{M}_R$) in Ripple for the task of intent-aware impact analysis. To reduce the cognitive load of the LLM, we adopt a *plan-then-predict* strategy which improves its ability to precisely predict the potentially impacted locations in the repository. To this end, first, we introduce a "Planner LLM" ($\mathcal{M}_P$) that constructs a *Change Plan* $\mathcal{P}$, encapsulating a high-level overview of the change intent as a structured sequence of reasoning and change actions. This explicit decomposition allows ($\mathcal{M}_R$) to reason over a structured step-by-step representation of changes toward predicting change impact.

Instead of exposing the entire code repository $\mathcal{R}$ to $\mathcal{M}_R$, we *localize impact reasoning* using *dependence-based clustering*. To this end, we leverage the dependence-enhanced impact set $\mathcal{I}_D$ as determined in the previous phase (Section 3.2.3) to construct *dependence clusters* ($\mathcal{D}$). This ensures that Reasoner LLM ($\mathcal{M}_R$) operates only on relevant repository slices, limited to structurally/semantically-dependent components. We then aggregate the impact set predictions corresponding to individual dependence clusters to construct the *final impact set* $\mathcal{I}_f$, thus enabling a *more precise* impact analysis.

Overall, the *plan-then-predict* approach to precision-focused impact set refinement phase in Ripple can be formulated as follows:

3.3.1 Change Plan Generation. In the *plan* phase of our *plan-then-predict* strategy, we adopt Planner LLM $\mathcal{M}_P$ which generates a structured *Change Plan* $\mathcal{P}$ to guide the downstream reasoning in the *predict* phase. Instead of allowing $\mathcal{M}_P$ to reason about change propagation in an unstructured manner, we adopt Chain-of-Thought (CoT) [56] prompting to generate a sequence of *change steps* that encapsulate a structured representation of the change intent ψ. By explicitly decomposing reasoning into intermediate steps, CoT enables $\mathcal{M}_P$ to produce logically consistent reasoning on changes.

In Figure 3 (*left*), we present the prompt template to $\mathcal{M}_P$, which takes as input the change intent ψ and the seed edit location M_0. Mathematically, the generated change plan $\mathcal{P}$ can be defined as:

$$\mathcal{P} = \mathcal{M}_P(\psi, M_0) \quad (4)$$

where $\mathcal{P} = \{P_i | P_i \text{ is a change step}\}$. Here, each change step P_i is a natural language instruction that describes a set of required changes. In addition, note that change steps within $\mathcal{P}$ often exhibit *change dependencies*, *where code modifications pertaining to one influence those in another*. For example, if one change step indicates a modification to the signature of the method `getListedMethods`, another could denote modifications to all methods invoking `getListedMethods`.

Planner LLM Prompt Template for Generating Change Plans

Task. [*Chain-of-Thought* Prompt]
Input Format. = ⟨Issue Summary, Issue Description, Filename, Seed Location Code⟩
Output Format. = [Plan with Change Steps]

Reasoner LLM Prompt Template for Intent-Aware Impact Analysis

Role. Instructions...
Input Format. ⟨ Issue Summary, [Plan with Change Steps], Repository Structure⟩
Task. [*Chain-of-Thought* Prompt]
Output Format. (ClassName1, methodName1), ... [Justification]

Figure 3: Prompt Templates to LLMs in Ripple for (*top*) generating change plans, and (*bottom*) performing impact analysis.

We expect the effectiveness of this step due to the pre-training data of LLMs, which includes large-scale corpora of natural language (NL)-programming language (PL) pairs from sources such as code documentation, developers' online forums, and commit messages. These diverse examples provide LLMs with the contextual knowledge necessary to infer structured modifications from high-level intent, allowing $\mathcal{M}_P$ to *capture the dependencies among change steps* and eventually generate semantically meaningful change plans.

3.3.2 Dependence-Aware Impact Set Clustering. Given $\mathcal{I}_D$, *i.e.*, the dependence-enhanced impact set, we identify *dependence clusters* that capture structurally/semantically interdependent groups of methods and provide a localized view of impact propagation. To this end, we model $\mathcal{I}_D$ as a dependence graph $G = (\mathcal{I}_D, E)$ where $E = \{(M_i, M_j) \mid M_i, M_j \in \mathcal{I}_D,$ and (M_i, M_j) satisfies a dependence relation as defined in Section 3.2.3$\}$

Accordingly, we define dependence clusters D_i as the *connected components* of G, ensuring that each cluster consists of a maximal set of methods that are transitively connected when edge direction is ignored. Formally, the set of dependence clusters is given by:

$$\mathcal{D} = \{D_i \mid D_i \subseteq \mathcal{I}_D, \text{ and } D_i \text{ is a connected component of } G\} \quad (5)$$

Such a partitioning of $\mathcal{I}_D$ into cohesive subsets of interdependent methods enables a localized and more precise impact reasoning. *Those subsets potentially have change dependencies, yet do not explicitly exhibit structural/semantic relations in a program.*

3.3.3 Reasoning to Identify Impact Subsets. In the *predict* phase of our *plan-then-predict* strategy, the objective of the Reasoner LLM $\mathcal{M}_R$ is to identify the set of all potentially impacted locations within the repository $\mathcal{R}$. To facilitate structured reasoning, we leverage the Change Plan $\mathcal{P}$ generated by the Planner LLM $\mathcal{M}_P$, which breaks down the change intent ψ into a sequence of change steps P_i describing changes. However, these modifications often exhibit *structural and semantic interdependencies* (*e.g.*, function calls, shared variables), making it insufficient to reason about each change step in isolation. Thus, we perform repository slicing via dependence clusters.

Repository Slicing with Dependence Clusters. Repository-wide analysis with LLMs is infeasible due to the large-scale nature of modern codebases. To address this limitation, we *localize reasoning* by leveraging dependence clusters $\mathcal{D}$, which capture groups of

structurally/semantically-dependent methods (Section 3.3.2). Specifically, we slice $\mathcal{R}$ to retain only methods

$$\{M_j \mid M_j \in D_i\}, \quad \text{where } D_i \in \mathcal{D} \subseteq \mathcal{I}_D \subseteq \mathcal{R} \tag{6}$$

Accordingly, $\mathcal{M}_R$ operates independently within each dependence cluster D_i, ensuring that *impact reasoning remains both contextually relevant and computationally feasible*. This structured decomposition allows $\mathcal{M}_R$ to consider interactions between interdependent changes over localized repository contexts–improving precision.

In Figure 3 (*bottom*), we present the template prompt for $\mathcal{M}_R$ to predict the set of impacted methods $\mathcal{I}_i$ within a given dependence cluster D_i. We structure D_i and augment each method entity $M_j^{(i)}$ with a *textual summary* $S_j^{(i)}$ (M_j and S_j for short), which helps to abstract the details of the implementation, yet provide contextual hints about the method. Mathematically:

$$\begin{aligned}\mathcal{I}_i &= \mathcal{M}_R(\psi, \mathcal{P}, D_i) \\ &= \mathcal{M}_R(\psi, \{P_1, P_2, ..., P_{|\mathcal{P}|}\}, \{(M_1, S_1), \ldots, (M_{|D_i|}, S_{|D_i|})\}) \\ &= \prod_j P\Big(\tilde{M}_j \in \mathcal{D}_i \mid \{(\tilde{M}_1, A_1), \ldots, (\tilde{M}_{j-1}, A_{j-1})\}, A_j, \psi, \\ &\qquad \{P_1, P_2, ..., P_{|\mathcal{P}|}\}, \{(M_1, S_1), \ldots, (M_{|D_i|}, S_{|D_i|})\}\Big)\end{aligned} \tag{7}$$

Here, A_j represents the analysis over change dependencies which results in $\mathcal{M}_R$ selecting $\tilde{M}_j \in D_i$ as being potentially impacted, *i.e.*, to belong to $\mathcal{I}_i$. Moreover, the subset of change steps $\mathcal{P}_i \subseteq \mathcal{P}$ corresponds to the modifications required in the interdependent group of methods in D_i. These steps encapsulate the reasons for changes to address the intent behind the impact set $\mathcal{I}_i \subseteq D_i$.

Sample-and-Marginalize Inference. Standard CoT prompting relies on greedy decoding inference, where a single reasoning path decides the final answer. In contrast, Wang *et al.* [54] introduced *self-consistency*, which explores multiple reasoning paths and selects the most optimal answer through majority voting. This strategy is grounded in the intuition that *the correct answer to a complex problem can be reached via multiple independent reasoning paths.*

In Ripple, $\mathcal{M}_R$-generated impact sets exhibit inherent variability due to the diverse reasoning trajectories explored while analyzing change impact propagation. To mitigate this variability, we enforce self-consistency by sampling multiple candidate impact sets for the dependence cluster D_i, and marginalize by computing their intersection. This can formally be defined as:

$$\mathcal{I}_i = \bigcap_{j=1}^{K} \mathcal{I}_i^{(j)} = \bigcap_{j=1}^{K} \mathcal{M}_R^{(j)}(\psi, \mathcal{P}, D_i) \tag{8}$$

where $\mathcal{I}_i^{(j)}$ denotes the j-th sampled impact set for D_i, among a total of K candidates. In other words, we leverage self-consistency to *refine* the Reasoner LLM's impact set predictions, yielding a *more precise and reliable* intent-aware impact analysis.

3.3.4 Impact Subset Aggregation. Finally, we construct the final impact set I_f denoting impacted locations by aggregating the impact subset predictions $\mathcal{I}_i$ for each dependence cluster D_i as follows:

$$\mathcal{I}_f = \bigcup_{i=1}^{|\mathcal{D}|} \mathcal{I}_i = \bigcup_{i=1}^{|\mathcal{D}|} \left(\bigcap_{j=1}^{K} \mathcal{I}_i^{(j)} \right) = \bigcup_{i=1}^{|\mathcal{D}|} \left(\bigcap_{j=1}^{K} \mathcal{M}_R^{(j)}(\psi, \mathcal{P}, D_i) \right) \tag{9}$$

4 Empirical Evaluation

We seek to answer the following research questions:

(RQ_1) Effectiveness in Change Impact Analysis
Can Ripple accurately predict the impact of the modifications to a specific seed location within Java code repositories?

(RQ_2) Qualitative Evaluation

- **2.1** Do generated *change plans* capture change dependencies to help predict isolated changes in the impact sets?
- **2.2** *Turning the Precision–Recall knobs.* How are the precision and recall for impact analysis affected by different aggregation strategies for multiple candidate impact sets in Ripple?
- **2.3** *Granularity*. How do the impact sets generated by Ripple scale to different granularities of program elements?

(RQ_3) Ablation Study
How does the inclusion of different information sources, such as commit history, structural/semantic dependencies, change plans, contribute to the effectiveness of Ripple in impact analysis?

5 Effectiveness in Impact Analysis (RQ_1)

5.1 Experiment Setup

5.1.1 Dataset. Commits in projects often include unrelated changes for different purposes [24], making the direct use of tangled commits less reliable for IA. Herbold *et al.* [23] manually untangled the commits to introduce a dataset with 3,498 commits from 28 Java projects, selected from the Apache Software Foundation. From this dataset, Yan *et al.* [59] build Alexandria, an IA benchmark containing 910 commits from 25 open-source projects where *every co-changing code entity only contributes to the bug fix.* To establish the intent-aware IA benchmark, we first extract issue IDs from the commit logs for all commits in Alexandria. Based on these, we link the commits to their respective bug reports in JIRA issue trackers. We retrieve metadata representing the intent, *i.e.*, the issue summary and description. This alignment ensures our benchmark having only the bug-fixing commits that are explicitly tied to their reports.

Seed Edit Locations. If Ripple is used in practice, a developer could start at an initial editing location, or one could use automated approaches in bug/fault localization from bug reports [36, 63], or feature/concept localization from change requests [15]. In our experiments, because the benchmark does not contain seed edit locations, we use a data-driven proxy to recover the seeds. Specifically, given the co-changing methods in an untangled commit, we identify the method with the highest number of direct or indirect call/class-member dependencies to other co-changing methods. This serves as the seed edit location as its structural position suggests a central role in propagating change effects across the impacted codebase.

Ground-Truth Impact Sets. For each untangled fixing commit in our benchmark, if M_0 denotes the seed edit location and $\mathcal{M} = \{M_0, M_1, ..., M_n\}$ denotes the set of co-changed methods in the commit, the corresponding ground-truth impact set is $\mathcal{M} \setminus \{M_0\}$.

Data Statistics and Selection. Our benchmark consists of 866 untangled bug-fixing commits in 25 projects, as explained. These projects contain from 24–1,202 Java files (mean: 302.9) and 232–12,690 methods (mean: 3,111.9). The commits themselves include 1–34 co-changed files and 2–121 co-changed methods. Among them, 737 commits involve ≤ 5 co-changed methods and 129 involve > 5 co-changed ones. To ensure a comprehensive yet computationally

feasible evaluation, we randomly select 50 instances from each category, yielding a total of 100 untangled commits for our experiments.

5.1.2 Methodology. In the *Recall-Focused Impact Set Generation* phase, we first expand the seed edit location using commit history (Section 3.2.2). To construct the dependence-enhanced impact set, we leverage the static dependence graph generator in Athena [59] to model calling and class-member dependencies within a project. Unlike tools such as WALA [5] and Soot [48], which require compilable Java bytecode, this approach works directly on source code using Tree-sitter [4]. This enables an efficient and scalable extraction of inter-method relationships without the overhead of bytecode analysis. For indirect calling dependencies, we limit the dependence depth L to 1 hop–thus ensuring that the most relevant inter-method dependencies are captured while avoiding combinatorial explosion.

Finally, in representing the code repository, we augmented methods with their natural language summaries to abstract implementation details (Section 3.3.3). To this end, we first extract method summaries from existing code documentation. In cases where documentation is missing or contains only external references without descriptions, we employ the StarCoder-1B [33] model in a few-shot setting to generate a concise (1–2 lines) summary directly from the method's implementation. By combining human-written documentation with LLM-generated summaries, we ensure that every method in the repository has a *conceptual* summary/representation.

5.1.3 Baselines. We compare Ripple with multiple *top-down* and *bottom-up* approaches, each leveraging different information sources. First, we establish an evolutionary coupling baseline based on commit history. Since commit history can be noisy [24], we limit this baseline to the most recent 100 relevant commits, *i.e.*, those where the seed edit location was modified. The impact set is then formed by merging the co-changed methods from these relevant commits. Second, we compare against a dependence coupling-based baseline, which, like Ripple, considers direct and indirect (up to L=1 hop) calling dependencies, along with class-member dependencies to determine the impact set. Third, for the conceptual baseline, we chose a traditional information retrieval (IR)-based TF-IDF approach as in Athena [59]. Each method in the corpus is tokenized and represented as a TF-IDF vector, with cosine similarity scores computed to establish a rank order. Fourth, we chose the state-of-the-art IA tool, Athena [59], which integrates dependence-based and conceptual information using Transformer-based neural models [51]. For both TF-IDF and Athena, we use the results on Athena's website for our selected instances [3]. Finally, we compared against multiple LLMs, including Gemini-2.0 Flash, Claude-3.5 Sonnet, and GPT-4o.

5.1.4 Evaluation Metrics. For a fair comparison across baselines that produce *ranked lists* (*i.e.*, TF-IDF and Athena), and *unordered impact sets* (*i.e.*, evolutionary and dependence coupling, Ripple), we evaluate using both *set-based* and *rank-based* metrics. Given the actual impact set (*i.e.*, *AIS*), let the estimated impact set be *EIS*. Precision = $\frac{|\text{EIS}\cap\text{AIS}|}{|\text{EIS}|}$, Recall = $\frac{|\text{EIS}\cap\text{AIS}|}{|\text{AIS}|}$, and F1-Score = $\frac{2*\text{Precision}*\text{Recall}}{\text{Precision}+\text{Recall}}$ are for the *set-based evaluation*, where Precision measures the proportion of predicted impacted entities that are actually impacted, Recall measures the proportion of actually impacted entities that are predicted, and F1-score balances the correctness and completeness.

Table 1: Effectiveness of Ripple in impact analysis (RQ1)

Approach	Evaluation Metrics (in %)			
	Hit@k	*Prec.*	*Recall*	*F1*
Evolutionary Coupling	32.0	4.9	17.1	5.2
Dependence Coupling	**86.0**	7.6	**64.7**	11.1
Conceptual (TF-IDF)	44.0	11.9	25.2	13.1
Athena	62.0	18.0	31.2	17.9
Ripple *w/* Gemini-2.0 Flash	60.0	25.8	32.2	23.3
w/ Claude-3.5 Sonnet	70.0	25.3	38.3	24.5
***w/* GPT-4o**	69.0	**28.2**	36.3	**25.0**

Let the ranked list of predicted impact locations from rank-based baselines be denoted as *EIR*. In *rank-based evaluation*, we compute: Hit@K = $1(|\text{EIR}_K \cap \text{AIS}| > 0)$, Precision@K = $\frac{|\text{EIR}_K\cap\text{AIS}|}{K}$, and Recall@K = $\frac{|\text{EIR}_K\cap\text{AIS}|}{|AIS|}$. Here EIR_K denotes the top-K ranked predictions, Hit@K measures whether at least one relevant entity in the impact set appears in EIR_K, precision@K measures the proportion of relevant entities among EIR_K, and recall@K measures the proportion of AIS retrieved within EIR_K.

Standardization. Since set-based baselines produce *unordered sets*, while rank-based baselines return *ranked lists*, we standardize K across all metrics to ensure a fair evaluation across approaches:

- in ranked baselines, K is set to the size of EIS produced by Ripple.
- in Ripple where no ranking exists, since K is set to |EIS|, we measure whether at least one relevant entity appears in EIS.

By default, we report *macro-averaged* scores, which treat metrics for each instance equally. In other words, for N instances, *macro*-Precision = $\sum_{i=1}^{N}\frac{\text{Precision}_i}{N}$ and *macro*-Recall = $\sum_{i=1}^{N}\frac{\text{Recall}_i}{N}$. For some in-depth evaluations (as in Sections 5.2.2 and 6.3), we also report *micro-averaged* scores which can be computed as *micro*-Precision = $\frac{\sum_{i=1}^{N}|\text{EIS}_i\cap\text{AIS}_i|}{\sum_{i=1}^{N}|EIS|}$ and *micro*-Recall = $\frac{\sum_{i=1}^{N}|\text{EIS}_i\cap\text{AIS}_i|}{\sum_{i=1}^{N}|AIS|}$.

5.2 Empirical Results

5.2.1 Comparison with Traditional IA Approaches. In Table 1, we compare with traditional IA approaches that leverage different information sources. Ripple outperforms all baselines in F1-score by 39.7%–380.8%. Compared to the evolutionary coupling baseline, which relies on commit history, we observe a 475.5% increase in precision and a 112.3% increase in recall. This improvement can be attributed to the fact that inherent noise in commit history is filtered in the *Precision-Focused Impact Set Refinement* phase in Ripple.

Compared to the dependence coupling-based approach, Ripple achieves a 125.2% improvement in F1-score. While this baseline has a 78.2% higher recall than Ripple, it is at the cost of *significantly lower precision* (by 271.1%). Notably, the 86% Hit@K indicates that in most commits, at least one co-changing location is structurally or semantically dependent on the seed edit location–reinforcing the importance of dependence coupling in estimating impact sets.

As discussed in Section 3.2.2, commit history captures change dependencies beyond structural/semantic relationships. In fact, *in 21% of the commits in our benchmark, evolutionary coupling identifies impact locations that dependence coupling alone misses.* Thus, despite its lower precision and recall, it provides complementary insights that enhance impact set estimation. By design, we incorporate both

Table 2: Stratified evaluation of impact analysis approaches by the *number of co-changed methods* in a commit (N) (RQ1)

Approach	N	Evaluation Metrics (in %)					
		Micro			Macro		
		Prec.	*Recall*	*F1*	*Prec.*	*Recall*	*F1*
Athena	≤ 5	7.8	36.8	12.9	12.4	41.7	17.4
	> 5	17.7	13.6	15.4	23.6	20.7	18.4
Ripple *w/* GPT-4o	≤ 5	**7.8**	**36.8**	**12.9**	16.0	42.7	20.7
	> 5	25.5	19.6	22.2	**40.3**	**29.8**	**29.4**

information sources in the *Recall-Focused Impact Set Generation*–thus attaining both higher precision and recall than other baselines.

Ripple improves over the TF-IDF conceptual coupling baseline by 90.8% in F1-score, with a 137% increase in precision and a 44% increase in recall. While this baseline is useful to identify impact locations that frequently refer to similar domain-specific concepts or share variable names, *relying only on conceptual coupling results in missing an average of 74.2% impact locations per commit*, that are otherwise captured by dependencies. Moreover, *in 56% of the commits in our benchmark, it fails to identify even a single impacted location.*

The best baseline Athena improves over conceptual coupling in F1-score by 36.6%. However, *Ripple outperforms Athena in F1-score by 39.7%, with 56.7% higher precision and 16.3% higher recall.* In fact, *Athena misses an average of 70.4% impact locations per commit that can otherwise be captured by dependencies.* This highlights its limitations in sufficiently capturing dependence coupling. In contrast, Ripple misses 49.9% of impact locations on average per commit while successfully identifying *at least one location that is dependent on the seed location in 77.9% of the relevant instances.*

As noted in Section 3.2, co-changed locations often exhibit logical relationships that are *not captured by dependence coupling*. 54% of the commits in our benchmark have at least one such isolated impact location. Ripple predicts 16.7% of these due to LLMs' ability to capture change dependencies. This can be attributed to: (i) incorporation of dependence clusters built from commit history and dependence-enhanced impact sets, which expands its scope; and (ii) change plans, which help identify relationships between clusters.

Finally, we compare the performance of three LLMs, *i.e.*, Gemini-2.0 Flash, Claude-3.5 Sonnet, and GPT-4o within Ripple. Notably, Gemini failed to adhere to the prescribed output format for 11% of the commits in our benchmark, whereas Claude and GPT-4o had significantly lower error rates at 1% and 2%, respectively. Among the three, GPT-4o demonstrates the best overall performance. In precision, it outperforms Claude by 11.5% and Gemini by 9.3%, while in recall, it improves over Gemini by 12.7%, although Claude surpasses it by 5.5%. Overall, these results indicate that Ripple with any LLM outperforms all baselines in both precision and recall.

5.2.2 Stratified Evaluation based on the Number of Co-Changed Methods. In our benchmark, we consider two categories, with ≤ 5 and > 5 co-changed methods–each indicative of the complexity of the change. Let us refer to them as *Lite* and *Complex*, respectively (Table 2). Due to the smaller size of the ground-truth impact sets in Lite, *micro* metrics are sensitive to variations in small sets, thus, more suitable for evaluation. As for Complex, *macro* metrics are more suitable due to robustness to varying impact set sizes.

Table 3: Sensitivity of Ripple to seed edit location (RQ1)

Seed Localization	Approach	Evaluation Metrics (in %)			
		Hit@k	*Prec.*	*Recall*	*F1*
Ranking-Based	Athena	59.0	15.0	28.6	14.4
	Ripple	64.0	19.1	32.8	19.5
LLM-Based	Athena	71.0	17.7	38.4	16.4
	Ripple	73.0	24.9	36.7	20.6
Entity-Matching	Athena	72.0	16.5	37.7	16.8
	Ripple	81.0	21.5	46.2	22.0
Dependence-Based	Athena	62.0	18.0	31.2	17.9
	Ripple	69.0	**28.2**	36.3	**25.0**

Table 2 shows that for the Lite benchmark, the performance of Ripple and Athena are similar. This can be due to the size of the dependence-enhanced impact sets $\mathcal{I}_D$s generated in the first phase, due to which we observe a drop in overall precision. However, in the Complex dataset, Ripple outperforms Athena by 59.7% in F1-score, with a precision of 40.3% and a recall of 29.8%. Notably, in 46% of these commits, the $\mathcal{I}_D$ sets predict all impact locations. In 12% of the instances, these are covered by multiple dependence clusters.

A fine-grained analysis showed that 18% of the commits in the Complex benchmark involve only one file, where Ripple achieved 47.4% precision and 52.1% recall in predicting impact locations. For the remaining 82% (*i.e.*, *multi-file commits*), the precision and recall were 38.8% and 25.0%, respectively. These show that leveraging both commit history and dependencies enables Ripple to better capture change dependencies, leading to better performance on large (multiple co-changed methods) single-file and multi-file commits.

RA$_1$. (1) Ripple excels in capturing change dependencies, outperforming all IA baselines by 39.7%–380.8% in F1-score. (2) In particular, Ripple enhances both precision and recall in complex, *i.e.*, large single-file or multi-file commits.

5.2.3 Sensitivity to Seed Edit Locations. In this experiment, we evaluate the sensitivity of Ripple to the quality of initial seed edit locations. We first design a *ranking-based* seed localization method by adapting prior work on information retrieval (IR)-based bug localization and concept location [17, 64]. Instead of lexical similarity, we use embedding-based similarity by generating vector representations for the bug report and all methods in a code repository with UniXCoder [19]. We then rank the methods based on their semantic similarities to the bug report and select the top-ranked method as the seed location. Next, we design an *LLM-based* method to identify the seed edit location from a bug report by adapting our prompt for all impacted methods in Section 3.3.3. Both ranking-based and LLM-based methods serve as imperfect localization approaches (*i.e.*, noisy approximations) to *identify the seed edit location.*

Furthermore, we use two data-driven, pseudo-perfect proxies to *recover the seed method*, because our benchmark does not have the seed edit locations. In the first one, we use an *entity matching strategy* guided by bug reports [14]. In particular, we extract references to method and class names from the issue summaries and descriptions, and match them against entities (method and class names) in the ground-truth impact set. To resolve ambiguities: (*i*)

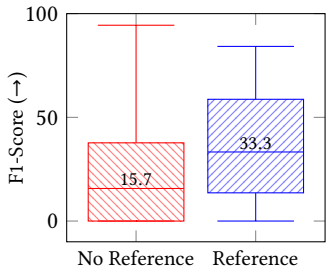


Figure 4: F1-score distributions for commits where the Change Plan in Ripple either references actual impacted locations (*i.e.*, Reference) or does not (*i.e.*, No Reference)

if a method name match is found, we use it as the seed; (*ii*) if only class name match is found, we randomly select a method from the ground-truth impact set that belongs to this class as the seed; (*iii*) if multiple matched methods or classes are found, we randomly select one among them; (*iv*) if no match is found, we randomly choose a method from the ground-truth impact set. Finally, the second proxy is the dependence-based one presented in Section 5.1.1. Note that to study Ripple' sensitivity, the first two seed edit localization methods are imperfect ones to locate the seed, while the last twos are pseudo-perfect in which the set of locations in the ground truth is used to recover the seed. For all seed edit localization methods, we compare with the corresponding Athena variants.

As seen in Table 3, Ripple consistently outperforms Athena across all seed edit localization methods by 25.6%–35.4%. Both ranking-based and LLM-based approaches are imperfect (no ground truth access), therefore, Ripple's performance using them are lower than when using the dependence-based proxy, by 28.2% and 21.4%, respectively. For the entity-based matching method, among the analyzed bug reports, 36% contained explicit references to methods names in the ground-truth impact set, 27% referenced only class names, and the remaining 37% had no identifiable entity matches, resulting in random seed selection. When method-level references were available (indicating near-perfect seed edit localization), Ripple achieved an F1-score of 33.8%. When only class-level references were found, performance remained high at 27.1% F1-score, highlighting that even coarser-grained context provides a meaningful signal. Its F1-score drops to 12.6% only in the most noisy setting, when the seed edit location was completely random. Overall, Ripple performs 13.6% better with dependence-based seed edit location than with entity-based matching. These results show that Ripple is generally robust to seed edit localization and is better than Athena.

6 Qualitative Evaluation (RQ_2)

6.1 Change Plans ⇝ Change Dependencies

In this experiment, we evaluate how our change plans contribute to impact prediction. To this end, we classify them based on whether they reference the class or method names in the ground-truth impact set. We use explicit references to actually impacted elements as a proxy for their potential influence on impact analysis reasoning. As shown in Figure 4, commits with such references achieve significantly higher F1-scores, suggesting the effectiveness of these change plans. Notably, 55.6% of the commits with these change plans have impact locations that are *not structurally or semantically dependent on the seed edit location*. Yet, Ripple successfully identified these methods in 33.3% of the instances. This suggests that the context reasoning in a change plan helps capture *indirect change dependencies that were not captured by program dependencies*.

Table 4: Effect of inference-time reasoning strategies in Ripple on precision and recall of impact analysis ($RA_{2.2}$)

Ripple *w/*	*N*	Evaluation Metrics (in %)			
		Micro		Macro	
		Prec.	*Recall*	*Prec.*	*Recall*
sample-and-marginalize (**intersection**)	≤ 5	7.8	36.8	16.0	42.7
	> 5	25.5	19.6	40.3	29.8
	All	18.3	21.3	28.2	36.3
sample-and-aggregate (**union**)	≤ 5	3.3	58.6	7.1	57.6
	> 5	13.9	31.9	27.8	45.0
	All	9.0	34.6	17.5	51.3

> **$RA_{2.1}$.** Change plans enhance Ripple's reasoning in impact prediction by providing contextual hints to capture *change dependencies beyond structural and semantic dependencies.*

6.2 Sample–and–Marginalize versus Aggregate

By default, Ripple employs the *sample-and-marginalize* strategy in Reasoner LLM to enforce self-consistency within each dependence cluster. This strategy takes the *intersection* of all impact subsets corresponding to the dependence clusters, ensuring a more conservative and *precise estimation*. Alternatively, Ripple can adopt the *sample-and-aggregate* strategy, which considers the *union* of all impact subsets, capturing a *broader range* of potential impacts.

Table 4 reports precision and recall across aggregation strategies. The intersection strategy results in a higher precision in both *micro* and *macro* settings, with 103.3% and 61.1% improvements, respectively. Union improves recall in both settings by 62.4% and 41.3%.

As noted in Section 5.2.2, *micro* evaluation metrics are more suitable for the Lite benchmark (*i.e.*, ≤ 5 co-changed methods), and *macro* for Complex (*i.e.*, > 5 co-changed methods). Accordingly, for commits in Lite, union yields a 59.2% improvement in recall as precision drops from 7.8% to 3.3%. In Complex, intersection yields a 45% improvement in precision as recall drops from 57.6% to 42.7%.

Choosing between marginalization and aggregation is akin to turning the precision and recall knobs, where each strategy prioritizes one over the other, allowing flexibility in users' preferences.

> **$RA_{2.2}$.** Sample-and-marginalization (*i.e.*, intersection) yields *more precise* impact sets with *relatively less coverage*, while sample-and-aggregation (*i.e.*, union) yields *relatively less precise* impact sets with *significantly more coverage.*

6.3 Method versus File-Level Impact Analysis

Traditional IA approaches focus on the method level [53, 59, 65], as file-level IA is often too coarse [49], and statement-level IA [20] is execution-dependent and costly. In this experiment, we extend baselines to the file level to assess whether coarser granularity

Table 5: Impact analysis at different granularities ($RA_{2.3}$)

Approach	Evaluation Metrics (in %)				
	Method-Level		File-Level		
	Prec.	*Recall*	*Prec.*	*Recall*	*F1*
Evolutionary Coupling	4.9	17.1	21.1	43.2	21.1
Dependence Coupling	7.6	**64.7**	37.7	61.2	39.7
Conceptual (TF-IDF)	11.9	25.2	44.0	29.2	32.2
Athena	18.0	31.2	**62.0**	40.9	45.3
Ripple *w/* GPT-4o	**28.2**	36.3	60.9	**62.4**	**54.6**

Table 6: Ablation study (RQ3)

Ablation Baseline	Evaluation Metrics (in %)			
	Hit@k	*Prec.*	*Recall*	*F1*
History-based (B_1)	32.0	4.9	17.1	5.2
Dependence-enhanced (B_2)	30.4	6.3	**76.8**	9.6
Ripple *w/o* Change Plan (B_3)	39.6	14.6	31.5	17.2
Ripple	69.0	**28.2**	36.3	**25.0**

improves precision and recall by capturing transitive dependencies and reducing noise. We do so by mapping method-level impact sets to files and comparing them to ground-truth impacted files.

Table 5 displays the comparative results against the baselines in both method-level and file-level IA. Ripple with GPT-4o outperforms all baselines by 20.5%–158.8% in F1-score. Notably, Athena's limitations in modeling dependencies (as discussed in Section 5.2.1) persist at the file-level, indicating that the missing dependencies are transitive and span multiple files. In contrast, the increase in Ripple's recall from method-level to file-level highlights its ability to capture such file-level dependencies, although it may not precisely predict all impacted methods. This can be attributed to its design, which incorporates *repository slicing with dependence clusters* and change plans to better model file-level change dependencies.

$RA_{2.3}$. Ripple scales effectively from method-level to file-level impact analysis by better modeling file-level change dependencies, improving over all baselines by 20.5%–158.8% in F1-score.

7 Ablation Study (RQ_3)

In this experiment, we quantify the contributions of different phases in Ripple to its overall performance. As the first baseline (B_1), we consider *commit history*-based impact set expansion (Figure 2). The second baseline (B_2) extends (B_1) by incorporating *program dependencies*. Both B_1 and B_2 give insights about the role of the recall-focused first phase in Ripple. Finally, we compare against a Ripple variant *without* the planning phase (B_3), *i.e.*, excluding the Change Plan input to the Reasoner LLM, to assess its contribution.

As seen in Table 6, we observe that both B_1 and B_2 perform poorly, with Ripple outperforming them by 380.8% and 160.4% in F1-score, respectively. Notably, B_1 *proves useful in capturing logical dependencies that are not explicitly linked to the seed location through structural or semantic relations.* In fact, we found that *commit history helps identify isolated impact locations in 45.3% of commits with multiple dependence clusters*, of which only one contains the seed edit location and related methods. In addition, the high recall of

Issue Summary. Some file descriptors are left open.

Issue Description. In some cases it seems that file descriptors are left opened by Ivy. It happens especially when loading a properties file from an `ivyconf.xml`, and also with ivy files used during the resolve process.

Change Plan Generated by GPT-4o in Ripple.
To address the issue of file descriptors being left open, these modifications are needed:
(1) Ensure that InputStream `is` is properly closed after its usage in the `toIvyFile` method.
(2) Implement a try-with-resources statement or explicitly close the `InputStream` in a `finally` block to guarantee that the resource is closed even if an exception occurs.
(3) Similar changes might be needed in:
  – Other methods within `XmlModuleDescriptorParser.java` that handle InputStreams or other I/O resources.
  – Any other classes or methods in the module that perform file operations, particularly those that load properties files or handle ivy files during the resolve process.
(4) Dependent updates potentially required:
  – Other methods within `XmlModuleDescriptorParser.java` that handle InputStreams or other I/O resources.
  – Any other classes or methods in the module that perform file operations, particularly those that load properties files or handle ivy files during the resolve process.
(5) Assumption: The issue primarily pertains to the handling of InputStreams in the specified method and similar methods in the codebase.

Figure 5: Change plan generated by Planner LLM in Ripple, for which the reference code change is shown in Figure 6.

B_2 highlights its advantage, as structurally broadening the scope effectively reduces the number of methods under consideration (by an average of *86.4%* per repository). Finally, the 45.3% drop in F1-score when omitting change plans underscores the importance of encapsulating change intent, as in our *plan-then-predict* approach.

RA_3. These findings reinforce the need for *change intent* and *LLM planning* that integrate the learning signals from different sources to achieve precise and complete impact analysis.

8 When Does Ripple Work—and When Not?

Qualitative Breakdown of Common Failures. In Ripple *w/* GPT-4o, the Reasoner LLM failed to predict any impacted location in 31% of the instances, whereas in 18% of them, it successfully predicted impact locations with an F1-score ≥ 50%. Among the failure cases, 22 involved ≤ 5 co-changed methods (*i.e.*, Lite) and 9 had > 5 co-changed methods (*i.e.*, Complex); 18 were single-file commits, while the remaining 13 spanned multiple files. In contrast, among the success cases (F1-score ≥ 50%), 7 were Lite and 11 were Complex; 11 were single-file and 7 were multi-file commits. That is, Ripple struggles with complex cases having more *multiple files*.

Manual inspection showed that 51.6% of the failure cases are *conceptual at the change level*, involving *modifications with common purposes* across impacted locations. Ripple employs evolutionary coupling-based impact set expansion to capture such changes but does not explicitly model them. In contrast, our conceptual coupling baselines, TF-IDF and Athena, use only *lexical similarity* and *code search-guided fine-tuning* to capture such changes, predicting the impacted sets in these instances with F1-scores of 11.4% and 12.7%.

An Example. Figure 5 illustrates a bug report, comprising the the summary of the Jira ticket and its description, along with *the change plan* generated by the Planner LLM in Ripple. We can see that it correctly identifies the underlying issue, *i.e.,* the file descriptors being left open, and recommends adding resource management

```
diff --git
↪ a/src/java/fr/jayasoft/ivy/external/m2/PomModuleDescriptorParser.java
↪ b/src/java/fr/jayasoft/ivy/external/m2/PomModuleDescriptorParser.java
@@ -252,7 +252 @@ public class PomModuleDescriptorParser extends
↪ AbstractModuleDescriptorParser {
-       try {
-               XmlModuleDescriptorWriter.write(md, destFile);
-       } finally {
-               if (is != null) {
-                       is.close();
-               }
+       XmlModuleDescriptorWriter.write(md, destFile);
diff --git a/src/java/fr/jayasoft/ivy/xml/XmlModuleDescriptorParser.java
↪ b/src/java/fr/jayasoft/ivy/xml/XmlModuleDescriptorParser.java
@@ -107,4 +106,0 @@ public class XmlModuleDescriptorParser extends
↪ AbstractModuleDescriptorParser {
-       } finally {
-               if (is != null) {
-                       is.close();
-               }
```

Figure 6: An example for a conceptual code change in Ivy project: removing stream cleanup logic across parser classes.

logic such as try-with-resources or explicit `finally` blocks. It also suggests reviewing similar locations across parser components where such logic may be duplicated. Figure 6 shows the actual code change as the redundant `finally` blocks handling `InputStream` cleanup in two parser classes were removed. These edits reflect a shared conceptual intent to simplify resource management across sibling components. However, they are not structurally connected via call or data dependencies, despite occurring in classes that share a superclass. As a result, while the Planner LLM correctly captured the high-level fix intent, Reasoner LLM in Ripple did not identify all impacted locations. This highlights a limitation of our *plan-then-predict* approach in bridging high-level intent with impact prediction, particularly for conceptually coupled yet structurally decoupled changes.

9 Discussion

Threats to Validity. *External Validity:* Our benchmark focuses on Apache projects which might not be representative. However, it was extracted from Alexandria, which was used in prior research [59]. We evaluated Ripple with three LLMs and only on Java. However, it is generic and the LLMs are programming language-agnostic.
Internal Validity: The third-party tools for program dependencies may introduce errors. However, we used well-established tools.
Construct Validity: Varied prompts may lead to varied input distributions, potentially affecting results. To address this, we define the prompts with well-specified structural templates.

Limitations. Ripple, like existing IA approaches [59, 65], assumes access to a seed edit location to predict the impact set. Recent work on program repair has focused on autonomously localizing and fixing bugs in both Python [28, 62] and Java [61], removing the need for a predetermined seed edit location. However, they *struggle to scale* beyond a single file and exhibit poor bug localization on multi-file commits (*e.g.*, SWE-Bench-Full [28]). Ripple struggles with *complex cases* having more *multiple files*. It also limited in bridging *conceptually* coupled yet structurally decoupled changes.

10 Related Work

Change Impact Analysis (IA). IA is crucial for software testing [18, 44, 45] and maintenance [11, 16, 44]. Researchers have proposed approaches that rely on source code [43] or change requests [9, 46]. The former include *top-down* and *bottom-up* IA approaches.

Top-down Approaches. Structural approaches propagate changes by analyzing relations between program elements [38]. Semantic approaches examine data and control flows, achieving high recall but often low precision due to large change sets [32]. Conceptual approaches use IR to detect shared intent between elements [42], improving precision but having lower recall when changes lack direct semantic overlap. Execution-based approaches analyze execution traces [40], code coverage [41], and execute-after relations [7], offering higher precision but lower recall due to test adequacy.

Bottom-up Approaches. These techniques mine evolutionary dependencies by analyzing past commits [65]. While these methods often achieve high recall, they suffer from low precision due to false positives. Other approaches combine different IA categories with orthogonal sources to gain the balance [25, 29]. Athena [59] integrates dependence-based and conceptual information.

Automated Program Repair (APR). Assuming access to buggy statements (*i.e.*, perfect fault localization), earlier APR approaches leveraged deep learning [34] or APR-specific LLM prompting [26]. Recently, APR has evolved into an end-to-end scheme with LLM agents, performing both fault localization and patch synthesis [57, 62]. However, they prioritize minimal test-passing edits, often under-fixing by generating localized patches for specific faults, leading to weaker performance on multi-file commits in SWE-Bench despite strong results on single-file commits [28]. Such limitations are problematic for IA, where full semantic coverage is essential.

Ripple is designed to generalize across diverse code changes (feature enhancements, refactoring, bug fixes) while offering intent-aware change impacts. While optimizing for bug-fixing, APR tools are not explicitly incentivized to identify all semantically impacted locations implied by change intent. Thus, Ripple exhibits broader applicability, complementing APR tools by enabling an execution-free assessment of code changes and providing a useful proxy for progress monotonicity, task advancement, and fixing completion.

11 Conclusion and Implications

Novelty. Ripple is an intent-aware IA approach that guides an LLM to devise a change plan to derive the impact set. Our results matched with the theoretical one in addressing the precision-recall tradeoff. Our finding is the LLM's ability to infer structured changes from high-level intent and identify change dependencies among the change steps. These dependencies may not always align with explicit relations explored in existing approaches. The Planner LLM effectively captures these dependencies and generates a meaningful change plan for Reasoner LLM to derive a precise final impact set.

Implications. First, Ripple can be complementary to APR tools in guiding complete fixing. Second, it can be extended to guide LLMs in identifying necessary co-changes to implement a new feature by maintaining consistency changes to multiple elements. Third, Ripple can support automated change documentation. Finally, in an IDE, Ripple can be used to build LLM-based assistants to offer context-aware editing suggestions based on change dependencies.

Data Availability. All data, code, and detailed prompts for LLMs in Ripple to replicate our experiments are available at [6].

Acknowledgments

This work was supported in part by the US National Science Foundation (NSF) grant CNS-2120386.

References

[1] 2007. *commit 283f77d.* https://github.com/apache/ant-ivy/commit/283f77d29ab46c5ae47d312455641a0317c58f58

[2] 2007. *NPE in case of eviction by 2 other modules on different confs.* https://issues.apache.org/jira/browse/IVY-644

[3] 2024. *Athena: Enhancing Code Understanding for Impact Analysis by Combining Transformers and Program Dependence Graphs.* https://github.com/yanyanfu/Athena/tree/main

[4] 2024. *Tree-sitter.* https://tree-sitter.github.io/tree-sitter/

[5] 2024. *WALA.* https://github.com/wala/WALA

[6] 2025. *Replication package for RIPPLE.* https://github.com/se-doubleblind/ripple

[7] T. Apiwattanapong, A. Orso, and M.J. Harrold. 2005. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* 432–441. doi:10.1109/ICSE.2005.1553586

[8] L.C. Briand, J. Wust, and H. Lounis. 1999. Using coupling measurement for impact analysis in object-oriented systems. In *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360).* 475–482. doi:10.1109/ICSM.1999.792645

[9] Lionel C. Briand, Yvan Labiche, and Leeshawn O'Sullivan. 2003. Impact Analysis and Change Management of UML Models. In *19th International Conference on Software Maintenance (ICSM 2003), The Architecture of Existing Systems, 22-26 September 2003, Amsterdam, The Netherlands.* 256–265. doi:10.1109/ICSM.2003.1235428

[10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[11] Ophelia C. Chesley, Xiaoxia Ren, and Barbara G. Ryder. 2005. Crisp: A Debugging Tool for Java Programs. In *21st IEEE International Conference on Software Maintenance (ICSM 2005), 25-30 September 2005, Budapest, Hungary.* 401–410. doi:10.1109/ICSM.2005.37

[12] Hridya Dhulipala, Aashish Yadavally, and Tien N. Nguyen. 2024. Planning to Guide LLM for Code Coverage Prediction. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering* (Lisbon, Portugal) *(FORGE '24).* Association for Computing Machinery, New York, NY, USA, 24–34. doi:10.1145/3650105.3652292

[13] Hridya Dhulipala, Aashish Yadavally, and Tien N. Nguyen. 2024. Planning to Guide LLM for Code Coverage Prediction. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering* (Lisbon, Portugal) *(FORGE '24).* Association for Computing Machinery, New York, NY, USA, 24–34. doi:10.1145/3650105.3652292

[14] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2025. Vulnerability Detection with Code Language Models: How Far are We? . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE).* IEEE Computer Society, Los Alamitos, CA, USA, 1729–1741. doi:10.1109/ICSE55347.2025.00038

[15] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process* 25, 1 (2013), 53–95. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.567 doi:10.1002/smr.567

[16] Fabrizio Fioravanti and Paolo Nesi. 2001. Estimation and Prediction Metrics for Adaptive Maintenance Effort of Object-Oriented Systems. *IEEE Trans. Software Eng.* 27, 12 (2001), 1062–1084. doi:10.1109/32.988708

[17] Gregory Gay, Sonia Haiduc, Andrian Marcus, and Tim Menzies. 2009. On the use of relevance feedback in IR-based concept location. In *ICSM.* IEEE Computer Society, 351–360.

[18] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015.* 211–222. doi:10.1145/2771783.2771784

[19] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850* (2022).

[20] 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, July 10 - 14, 2017,* Tevfik Bultan and Koushik Sen (Eds.). ACM, 318–328. doi:10.1145/3092703.3092719

[21] Ahmed E. Hassan. 2009. Predicting faults using the complexity of code changes. In *2009 IEEE 31st International Conference on Software Engineering.* 78–88. doi:10.1109/ICSE.2009.5070510

[22] Lile Hattori, Dalton Guerrero, Jorge Figueiredo, João Brunet, and Jemerson Damásio. 2008. On the Precision and Accuracy of Impact Analysis Techniques. In *Seventh IEEE/ACIS International Conference on Computer and Information Science (icis 2008).* 513–518. doi:10.1109/ICIS.2008.104

[23] Steffen Herbold, Alexander Trautsch, Benjamin Ledel, Alireza Aghamohammadi, Taher A Ghaleb, Kuljit Kaur Chahal, Tim Bossenmaier, Bhaveet Nagaria, Philip Makedonski, Matin Nili Ahmadabadi, et al. 2022. A fine-grained data set and analysis of tangling in bug fixing commits. *Empirical Software Engineering* 27, 6 (2022), 125.

[24] Kim Herzig, Sascha Just, and Andreas Zeller. 2013. It's not a bug, it's a feature: how misclassification impacts bug prediction. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13).* IEEE Press, 392–401.

[25] Lulu Huang and Yeong-Tae Song. 2007. Precise Dynamic Impact Analysis with Dependency Analysis for Object-oriented Programs. In *Proceedings of the 5th ACIS International Conference on Software Engineering Research, Management & Applications (SERA '07).* IEEE Computer Society, USA, 374–384. doi:10.1109/SERA.2007.109

[26] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. In *ICSE.* IEEE, 1430–1442.

[27] Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2024. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology* 33, 7 (2024), 1–30.

[28] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues?, 2024. *URL https://arxiv. org/abs/2310.06770* (2023).

[29] 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 *Proceedings of the 2010 17th Working Conference on Reverse Engineering (WCRE '10).* IEEE Computer Society, USA, 119–128. doi:10.1109/WCRE.2010.21

[30] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. 2007. Predicting Faults from Cached History. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07).* IEEE Computer Society, USA, 489–498. doi:10.1109/ICSE.2007.66

[31] Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay V. Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. 2022. Solving Quantitative Reasoning Problems with Language Models. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022,* Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (Eds.).

[32] Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. 2013. A survey of code-based change impact analysis techniques. *Softw. Test. Verification Reliab.* 23, 8 (2013), 613–646. doi:10.1002/STVR.1475

[33] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[34] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: context-based code transformation learning for automated program repair. In *ICSE.* ACM, 602–614.

[35] Terufumi Morishita, Gaku Morio, Atsuki Yamaguchi, and Yasuhiro Sogawa. 2023. Learning Deductive Reasoning from Synthetic Corpus based on Formal Logic. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA (Proceedings of Machine Learning Research, Vol. 202),* Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 25254–25274. https://proceedings.mlr.press/v202/morishita23a.html

[36] Anh Tuan Nguyen, Tung Thanh Nguyen, Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N. Nguyen. 2011. A topic-based approach for narrowing the search space of buggy files from a bug report. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011).* 263–272. doi:10.1109/ASE.2011.6100062

[37] 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. doi:10.1109/ISSRE.2013.6698913

[38] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar Al-Kofahi, and Tien N. Nguyen. 2010. Recurring bug fixes in object-oriented programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) *(ICSE '10).* Association for Computing Machinery, New York, NY, USA, 315–324. doi:10.1145/1806799.1806847

[39] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. *arXiv preprint* (2022).

[40] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. 2003. Leveraging field data for impact analysis and regression testing. *SIGSOFT Softw. Eng. Notes* 28, 5 (Sept. 2003), 128–137. doi:10.1145/949952.940089

[41] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. 2003. Leveraging field data for impact analysis and regression testing. In *Proceedings of*

the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (Helsinki, Finland) *(ESEC/FSE-11)*. Association for Computing Machinery, New York, NY, USA, 128–137. doi:10.1145/940071.940089

[42] Denys Poshyvanyk, Andrian Marcus, Rudolf Ferenc, and Tibor Gyimóthy. 2009. Using information retrieval based coupling measures for impact analysis. *Empirical Softw. Engg.* 14, 1 (Feb. 2009), 5–32. doi:10.1007/s10664-008-9088-2

[43] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. 2004. Chianti: a tool for change impact analysis of java programs. *SIGPLAN Not.* 39, 10 (Oct. 2004), 432–448. doi:10.1145/1035292.1029012

[44] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia C. Chesley. 2004. Chianti: a tool for change impact analysis of java programs. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*. 432–448. doi:10.1145/1028976.1029012

[45] Gregg Rothermel and Mary Jean Harrold. 1997. A Safe, Efficient Regression Test Selection Technique. *ACM Trans. Softw. Eng. Methodol.* 6, 2 (1997), 173–210. doi:10.1145/248233.248262

[46] Maryam Shiri, Jameleddine Hassine, and Juergen Rilling. 2007. Modification analysis support at the requirements level. In *9th International Workshop on Principles of Software Evolution (IWPSE 2007), in conjunction with the 6th ESEC/FSE joint meeting, Dubrovnik, Croatia, September 3-4, 2007*. 43–50. doi:10.1145/1294948.1294961

[47] Surbhi K. Solanki and Jalpa T. Patel. 2015. A Survey on Association Rule Mining. In *2015 Fifth International Conference on Advanced Computing & Communication Technologies*. 212–216. doi:10.1109/ACCT.2015.69

[48] Soot. [n. d.]. Soot Introduction. https://sable.github.io/soot/. Last Accessed July 11, 2019.

[49] Marco Torchiano and Filippo Ricca. 2010. Impact analysis by means of unstructured knowledge in the context of bug repositories. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement, ESEM 2010, 16-17 September 2010, Bolzano/Bozen, Italy*, Giancarlo Succi, Maurizio Morisio, and Nachiappan Nagappan (Eds.). ACM. doi:10.1145/1852786.1852847

[50] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *CoRR* abs/2307.09288 (2023). arXiv:2307.09288 doi:10.48550/ARXIV.2307.09288

[51] A Vaswani. 2017. Attention is all you need. *Advances in Neural Information Processing Systems* (2017).

[52] Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. 2023. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. *arXiv preprint arXiv:2305.04091* (2023).

[53] Wei Wang, Yun He, Tong Li, Jiajun Zhu, and Jinzhuo Liu. 2018. An Integrated Model for Information Retrieval Based Change Impact Analysis. *Sci. Program.* 2018 (2018), 5913634:1–5913634:13. doi:10.1155/2018/5913634

[54] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/forum?id=1PL1NIMMrw

[55] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922* (2023).

[56] Jasion Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. 2022. Chain of Thought Prompting Elicits in Large Language Models. In *Proceedings of Advances in Neural Information Processing Systems*. 24824–24837.

[57] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying LLM-based Software Engineering Agents. *CoRR* abs/2407.01489 (2024).

[58] Aashish Yadavally, Xiaokai Rong, Phat Nguyen, and Tien N. Nguyen. 2025. Large Language Models for Safe Minimization. (2025), To Appear.

[59] 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 (2024), 972–995. doi:10.1145/3643770

[60] 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. doi:10.1145/3643770

[61] Daoguang Zan, Zhirong Huang, Ailun Yu, Shaoxin Lin, Yifan Shi, Wei Liu, Dong Chen, Zongshuai Qi, Hao Yu, Lei Yu, Dezhi Ran, Muhan Zeng, Bo Shen, Pan Bian, Guangtai Liang, Bei Guan, Pengjie Huang, Tao Xie, Yongji Wang, and Qianxiang Wang. 2024. SWE-bench-java: A GitHub Issue Resolving Benchmark for Java. *CoRR* abs/2408.14354 (2024). arXiv:2408.14354 doi:10.48550/ARXIV.2408.14354

[62] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, Maria Christakis and Michael Pradel (Eds.). ACM, 1592–1604. doi:10.1145/3650212.3680384

[63] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In *2012 34th International Conference on Software Engineering (ICSE)*. 14–24. doi:10.1109/ICSE.2012.6227210

[64] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In *ICSE*. IEEE Computer Society, 14–24.

[65] Thomas Zimmermann, Sunghun Kim, Andreas Zeller, and E. James Whitehead. 2006. Mining version archives for co-changed lines. In *Proceedings of the 2006 International Workshop on Mining Software Repositories* (Shanghai, China) *(MSR '06)*. Association for Computing Machinery, New York, NY, USA, 72–75. doi:10.1145/1137983.1138001