CodePlan: Repository-level Coding using LLMs and Planning

Anonymous Author(s)
Affiliation
Address
email

Abstract

Software engineering activities such as package migration, fixing error reports from static analysis or testing, and adding type annotations or other specifications to a codebase, involve pervasively editing the entire repository of code. While Large Language Models (LLMs) have shown impressive abilities in localized coding tasks, performing interdependent edits across a repository requires multi-step reasoning and planning abilities. We frame repository-level coding as a planning problem and present a task-agnostic, neuro-symbolic framework called CodePlan . Our framework leverages static analysis techniques to discover dependencies throughout the repository, which are utilised in providing sufficient context to the LLM along with determining the sequence of edits required to solve the repository-level task. We evaluate the effectiveness of CodePlan on two repository-level tasks: package migration (C#) and temporal code edits (Python) across multiple repositories. Our results demonstrate CodePlan consistently beats baselines across tasks. Further qualitative analysis is performed to highlight how different components of the approach contribute in guiding the LLM towards the correct edits as well as maintaining the consistency of the repository.

1 Introduction

The remarkable generative abilities of Large Language Models (LLMs) Brown et al. (2020); Chen et al. (2021); Chowdhery et al. (2022); Fried et al. (2022); OpenAI (2023); Touvron et al. (2023) have opened new ways to automate coding tasks. Tools built on LLMs, such as Amazon Code Whisperer Cod (2023), GitHub Copilot Gih (2023) and Replit Rep (2023), are now widely used to complete code given a natural language intent and context of surrounding code, and also to perform code edits based on natural language instructions Cop (2023). Such edits are typically done for small regions of code such as completing or editing the current line, or the body of the entire method.

While these tools help with the "inner loop" of software engineering where the developer is editing a small region of code, there are several tasks in the "outer loop" of software engineering that involve the entire code repository For example, if a repository uses a library L, and its API changes from version v_n to version v_{n+1}, we need to migrate the whole repository to correctly invoke the revised version. A simplified example is given in Figure 1. Such a migration task involves making edits not only to all the regions of code that make calls to the APIs from the library, but also to regions (across file boundaries) having transitive syntactic and semantic dependencies on the updated code.

We present a task-agnostic neuro-symbolic framework, called CodePlan that utilises the local code editing abilities of LLMs along with various static analysis techniques to solve such *repository-level* coding tasks. CodePlan keeps track of relations across the repository and monitors local code changes made by the LLM in order to plan how these changes should be propagated. Our evaluations

```
+ class Complex {
+   float real;
+   float imag;
+   dict<string, string> metadata;
+ }

- tuple<float, float> create_complex(float a, float
    b)
+ Complex create_complex(float a, float b, dict
    metadata)
```

Figure 1: Task instruction to migrate a code repository due to an API change in the Complex Numbers library.

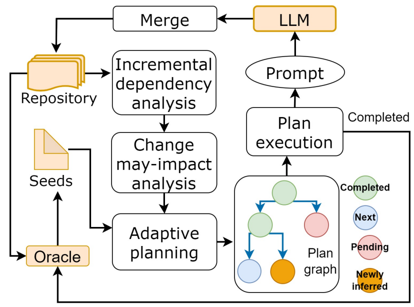


Figure 2: Overview of CodePlan.

```
tuple<tuple<float, float>, dict> func(float a, float
      b) {
  string timestamp = GetTimestamp(DateTime.Now);
  var c = (create_complex(a,b), new
       Dictionary<string, string>()"time",
       timestamp);
  return c;
}
```

(a) `Create.cs` - Original

```
Complex func(float a, float b) {
  String timestamp = GetTimestamp(DataTime
       .Now);
  dict_metadata = new Dictionary<string,
       string>(){"time", timestamp};
  Complex c = create_complex(a, b,
       metadata);
  return c;
}
```

(b) `Create.cs` - Modified (seed edit)

```
void process(float a, float b, float k) {
  var c = func(a, b);
  Console.WriteLine(c[0][0], c[0][1]);
  float norm = compute_norm(c[0][0], c[0][1]);
  Console.WriteLine(norm * k);
}
```

(c) `Process.cs` - Original

```
void process(float a, float b, float k) {
  Complex c = func(a, b);
  Console.WriteLine(c.real, c.imag);
  float norm = compute_norm(c.real, c.imag
       );
  Console.WriteLine(norm * k);
}
```

(d) `Process.cs` - Modified (derived edit)

Figure 3: Relevant code snippets from our repository.

against baselines across a benchmark of repository edits demonstrate the advantages of CodePlan for repository level code tasks. In summary, we make the following contributions:

1. We formalize the novel problem of automating repository-level coding tasks using LLMs, which requires analyzing the effects of code changes and propagating them across the repository.
2. We frame repository-level coding as a planning problem and design a task-agnostic, neuro-symbolic framework called CodePlan, based on a novel combination of an incremental dependency analysis, a change may-impact analysis and an adaptive planning algorithm. CodePlan synthesizes a multi-step chain-of-edits (plan) to be actuated by an LLM.
3. We experiment with two repository-level coding tasks using the `gpt-4-32k` model[1]: package migration for C# repositories and temporal code edits for Python repositories. We compare against baselines that use build system or type checker for guiding repository-wide edits.
4. Our results show that CodePlan has better match with the ground truth compared to baselines. CodePlan is able to get 5/7 repositories to pass the validity checks (i.e., to build without errors and make correct code edits), whereas the baselines cannot get any of the repositories to pass them.

2 Motivation

Consider the example API migration task specified in Figure 1 on code in Figure 3. Here we have an external library which provides an interface for creating complex numbers which is being used in two files within our repository. In this scenario, the external library modifies its interface by introducing a `Complex` number class and modifying the signature of the `create_complex` method accordingly. At this stage, our repository is in an inconsistent state according to the oracle – it will not build. To resolve this inconsistency and complete the migration, we first need to modify `func` to accomodate

[1] `https://platform.openai.com/docs/models/gpt-4`

the updated `create_complex`. As show in Fig 3b, this involves updating the signature of `func` to return an object of the new `Complex` type instead of a tuple. After this edit, our repository will still fail to build since now the use of the return object from `func` is incorrect inside the body of `process`. The edit required to `process` to resolve this is shown in Fig 3d and results in a repository that is consistent – it builds. We can think of the initial changes to the complex library as *seed changes* which trigger a set of *derived changes* across our repository.

CodePlan determines from the seed change that `func` needs to be modified, It analyses the code change between Figure 3(a)–(b) and classifies it as an *escaping change* since it affects signature of method `func`. The change may-impact analysis identifies that the caller(s) of `func` may be affected and hence, the adaptive planning algorithm uses caller-callee dependencies to infer a derived specification to edit the method `process`, which invokes `func`. The derived changes are executed by creating suitable prompts for an LLM and the resulting code repository passes the oracle, i.e., builds without errors.

Note that this is a simple example with only one-hop change propagation. In practice, the derived changes can necessitate many other changes transitively. Such a migration task is representative of a family of tasks that involve editing an entire code repository for various purposes such as fixing error reports from static analysis or testing, fixing a buggy coding pattern, refactoring, or adding type annotations or other specifications. We define an LLM-driven repository-level coding task as follows:

LLM-driven Repository-level Coding Task

Given a start state of a repository R_{start}, a set of seed edit specifications Δ_{seeds}, an oracle Θ such that $\Theta(R_{start}) = \mathsf{True}$, and an LLM L, the goal of an **LLM-driven repository-level coding task** is to reach a repository state $R_{target} = ExecuteEdits(L, R_{start}, P)$ where P is a chain of edit specifications from $\Delta_{seeds} \cup \Delta_{derived}$ where $\Delta_{derived}$ is a set of derived edit specifications so that $\Theta(R_{target}) = \mathsf{True}$.

3 Design

As described in Figure 2 CodePlan aims to solve repository-level coding tasks through an adaptive planning algorithm that iteratively combines (1) dependency analysis to keep track of the relationships within the repository and (2) change may-impact analysis to determine what other parts of the repository are effected by an edit. CodePlan maintains two key data structures -

Dependency Graph. We utilise dependency analysis Aho et al. (2007) to track syntactic and semantic relations between code elements and build a graph where nodes are code blocks (e.g. method, classes, imports) and edges are relationships (e.g. calls, overrides, inherits)

Plan Graph. P $= (O, C)$ is a directed acyclic graph with a set of code edit *obligations* O and edges C that record the *cause* from one obligation to the next. Each obligation O is characterised by a block to edit B, edit instruction I and the status indicating whether it have been discharged yet.

Given a repository and initial set of seed edit Δ_{seeds} based on the task description, CodePlan first instantiates a dependency graph G (from the initial state of the repository) and plan graph P (with obligations corresponding to Δ_{seeds}). It then infers the derived edits $\Delta_{derived}$ required to solve the task by iteratively editing the repository as described in Alg 2. At each stage it fetches an obligation from the plan graph P, uses the LLM to generate the local edit and analyses the change to update the dependency graph G and the plan graph P. The key components in Alg 2 are discussed briefly below. A detailed description is provided in the appendix.

Algorithm 1: Core algorithm

```
while do
    O ← GetNextPending(P);
    Q ← PrepareQuery(O, G);
    F ← InvokeLLM(Q);
    L ← ClassifyChange(Q, F);
    UpdateRepo(R, O, F);
    UpdateDepGraph(G, O, F);
    UpdatePlanGraph(P, G, L);
end
```

GetNextPending. Selects the next obligation to discharge from among the un-fulfilled obligations in the plan graph.

PrepareQuery. Given an edit obligation, constructs a query to the LLM to obtain an edit for the local code block specified by the obligation. The query aims to be as comprehensive as possible, consisting of - (1) task specific instructions (2) temporal context: previous edits that *caused* the need to edit the current block (extracted from the plan graph and presented as before and after code snippets), (3) spatial context: all related code for the current block such as methods being called or overridden and (4) the code block to be edited.

ClassifyChanges. Classifies the change made by the LLM to the code block by type (modification, addition and deletion changes) and further by which construct is changed (method body, method signature, class declaration etc...).

UpdateRepo. Stitches the modified code block back into the appropriate file in the repository. Also adds any new code blocks and deletes any code blocks that were removed in the LLMs response.

UpdateDepGraph. Updates the dependency relations associated with the code at the change site. For example if a method call to B is added in A, then an edge is added between A and B.

UpdatePlanGraph. Determines how the edit made may affect other parts of the repository and updates the plan graph accordingly with appropriate edit obligations. Uses a set of rules to identify blocks affected by the code change depending on the labels from `ClassifyChange`, constructs an obligation from each affected block, adds them to the plan graph and constructs an edge from the current obligation to each of the affected obligations, with the label being the relationship between the blocks. Finally marks the current obligation discharged.

4 Experimental Setup

4.1 Tasks

Migration. Given client repository being migrated from one framework to another, infer the code edits required to account for differences in APIs between the older and newer frameworks. We evaluate on examples from two specific migration scenarios - (1) migration from legacy logging framework to a more modern logging framework where the repositories considered are two large production-level proprietary codebases (I1, I2) and (2) modifying repos to use the newer `System.Text.Json` serialization framework instead of the older `NewtonSoft.Json` framework for which we use two open-source repositories (E1, E2). Further details in the appendix.

Temporal edits. Given a set of repository-local seed edits (e.g. adding an argument to a method), infer the derived code edits throughout the repository. This task aims to model the process a developer may follow when making a repository-level edits – making an initial edit followed by related edits to make the repository consistent. We evaluate on three open source repository changes. (T1, T2, T3) Further details in the appendix.

4.2 Oracles and Baselines

Oracles. In our experiments, we rely on two specific oracles to evaluate the validity of our solutions. For C# migration tasks, passing C# Build tools msb ([n. d.]) without errors serves as the oracle. In temporal edits scenarios, we use Pyright pyr ([n. d.]), a Python static checker, as the oracle.

Oracle-Guided Repair Baselines. An alternative to planning is to use the oracle to detect errors with each change. These approaches are reactive and involve attempting to fix errors identified by the oracles. We refer to them as *oracle-guided repair baselines*. For C# migration, we use Build-Repair, while for temporal edits, it's Pyright-Repair. The process includes applying an initial seed edit, detecting errors, analyzing error messages, and using an LLM for patching. However, oracle-guided repair may lack comprehensive change impact analysis, leading to potentially incomplete or incorrect fixes, especially in complex coding tasks. For fair comparison, we use the same contextualization method as CodePlan for the baselines.

Alternate Edit Model: Coeditor Wei et al. (2023). While CodePlan primarily leverages LLMs for localized code edits, it can also work with custom models like Coeditor Wei et al. (2023). Coeditor is designed for making an edit conditioned on prior temporal edits for Python code. We use Coeditor to evaluate whether CodePlan can work with different models and to perform a model ablation study.

4.3 Evaluation

We use two key metrics, Block Metrics and Edit Metrics, to assess how effectively CodePlan propagates changes throughout the code repository and the correctness of these changes.

Block Metrics. Block Metrics evaluate CodePlan's ability to identify code blocks in need of modification, including: *Matched Blocks:* Code blocks successfully identified for change; *Missed Blocks:* Code blocks that should have been modified but weren't; *Spurious Blocks:* Incorrectly edited blocks.

Edit Metrics:. Edit Metrics assess the correctness of CodePlan's modifications, including: *Levenshtein Distance:*, which measures edit distance between the Predicted and Target Repositories at the file level; and, *DiffBLEU:*, a modified BLEU Papineni et al. (2002) score focusing on comparing modified code sections while disregarding common code. Let Δ_{gt} and Δ_p respectively be diffs between the Source and Target repositories (ground truth), and the Source and Predicted repositories. The BLEU score between Δ_{gt} and Δ_p gives us the DiffBLEU score.

Validity Check. We say that a Predicted repository passes the *validity check* if the oracle (the build system for C# and Pyright for Python) does not detect any errors in it and we have a perfect match (modulo whitespace and formatting differences) with the ground truth Target repository.

Data Pre-processing. We pre-process the data to reduce noise during evaluation (details in the appendix). For each repository, we collect the before (*Source*) and after (*Target*) snapshots of the code from the pull requests and apply changes unrelated to the task either to both Source and Target, or remove them from the Target. To prepare the Source, we patch in the seed changes or prepare instructions for the LLM to carry them out. We also pre-process the Target repositories to ensure uniform coding practices. Note that all methods are evaluated on the same Source repositories (after the pre-processing).

5 Results and Analysis

In this section, we present empirical results to answer the following research questions:

RQ1: How well is CodePlan able to localize and make the required changes to automate repository-level coding tasks compared to baselines?

RQ2: How important are temporal and spatial contexts to CodePlan's performance?

RQ3: What are the key differentiators that allow CodePlan to outperform baselines in solving complex coding tasks?

5.1 RQ1: How well is CodePlan able to localize and make the required changes to automate repository-level coding tasks compared to baselines?

CodePlan ***outperforms baselines***. As shown in Table 1, CodePlan consistently does better at identifying the correct edit sites as it matches on more blocks and misses fewer blocks. The edits it makes are more closely aligned to the ground truth edits as seen with higher DiffBLEU score and lower Levenshtein Distance. Most notably CodePlan is able to successfully bring 5/7 repositories to a consistent state. We discuss these results in detail below.

C# Migration. Alongside the fact that CodePlan achieves better blocks and edit metrics on both I1 and I2, 3/4 C# repositories migrated using CodePlan pass the build check. Build-Repair on the other hand is not able to complete any of the tasks, in each case getting stuck on a particular set of errors which it is unable to fix even after multiple retries. Note that the non-perfect DiffBlue and Levenstein distances for E1 and E2 are due to differences in code formatting and the order of method declarations in the predicted file. In E2, where CodePlan is unable to reach a valid state, we observe that the LLM did not perform a necessary type cast when using a library API, which was uncaught by CodePlan, resulting in missed blocks. Some of the resulting errors are fixed in "Iter-2".

<u>CodePlan versus Build-Repair</u> We observe that a significant factor contributing to this performance difference is Build-Repair's reliance on "build error location" to indicate where code corrections are needed. Build errors may not always align with the actual correction site, leading to misinterpretation. For instance, an error may manifest as a derived class's overridden function signature mismatch, but

Dataset	Approach	Matched Blocks	Missed Blocks	Spurious Blocks	Diff BLEU	Levenshtein Distance	Validity Check
C# Migration Task on Internal (Proprietary) Repositories							
I1 (Logging)	CodePlan (Iter 1)	**151**	**0**	**0**	0.99	60	✗ (4) ≠
	CodePlan (Iter 2)	4	**0**	**0**	**1.00**	**0**	✓
	Build-Repair	82	69	13	0.81	6465	✗ (46) ≠
I2 (Logging)	CodePlan (Iter 1)	**438**	**0**	**0**	0.99	90	✗ (6) ≠
	CodePlan (Iter 2)	6	**0**	**0**	**1.00**	**0**	✓
	Build-Repair	337	101	25	0.66	7496	✗ (68) ≠
C# Migration Task on External (Public) Repositories							
E1	CodePlan (Iter 1)	**64**	**0**	**0**	**0.86**	**2931**	✓
	Build-Repair	34	30	27	0.65	9145	✗ (40) ≠
E2	CodePlan (Iter 1)	**38**	**8**	**0**	0.61	**1121**	✗ (13) ≠
	CodePlan (Iter 2)	2	0	6	**0.62**	1261	✗ (7) ≠
	Build-Repair	19	27	5	0.49	1379	✗ (11) ≠
Python Temporal Edit Task on External (Public) Repositories							
T1	CodePlan (Iter 1)	**8**	**2**	0	**0.90**	**1044**	✗ (0) ≠
	Pyright-Repair	5	5	0	0.76	1089	✗ (0) ≠
	Pyright-Strict-Repair	**8**	**2**	0	**0.90**	**1045**	✗ (0) ≠
	Coeditor-CodePlan	**8**	**2**	0	**0.90**	1160	✗ (0) ≠
	Coeditor-Pyright-Repair	5	5	0	0.66	1206	✗ (0) ≠
	Coeditor-Pyright-Strict-Repair	**8**	**2**	0	0.83	1106	✗ (6) ≠
T2	CodePlan (Iter 1)	**4**	**0**	0	**0.86**	**147**	✓
	Pyright-Repair	1	3	0	0.58	344	✗ (0) ≠
	Pyright-Strict-Repair	1	3	0	0.58	344	✗ (0) ≠
	Coeditor-CodePlan (Iter 1)	2	2	0	0.82	254	✗ (0) ≠
	Coeditor-Pyright-Repair	1	3	0	0.58	344	✗ (0) ≠
	Coeditor-Pyright-Strict-Repair	1	3	0	0.58	344	✗ (0) ≠
T3	CodePlan (Iter 1)	**11**	**0**	0	**0.94**	**288**	✓
	Pyright-Repair	1	10	0	0.53	840	✗ (0) ≠
	Pyright-Strict-Repair	1	10	0	0.53	840	✗ (0) ≠
	Coeditor-CodePlan (Iter 1)	10	1	0	0.76	759	✗ (0) ≠
	Coeditor-Pyright-Repair	1	10	0	0.53	840	✗ (0) ≠
	Coeditor-Pyright-Strict-Repair	1	10	0	0.53	840	✗ (0) ≠

Table 1: Comparison of CodePlan with baselines. Higher values of Matched Blocks and DiffBLEU, and lower values of Missed Blocks, Spurious Blocks, Levenshtein Distances are better. For each repository, different approaches are separately by a dashed line and the respective best values are highlighted in the bold font (except when all approaches have the same value). ✓ and ✗ respectively indicate if the Validity Check (Section 4.3) passes or fails, respectively. Against ✗, we also give the number of errors detected by the oracle in parentheses and indicate via ≠ that the output from the approach does not match the ground truth. In several cases in Python, even though the oracle (Pyright) does not flag any errors, the generated code does not match ground truth as indicated by "✗ (0) ≠" entries in the last column. This is because of the lack of sufficient type hints in the Python repositories to catch correctness requirements. In contrast, for the statically typed language C#, mismatch with ground truth is also reflected in non-zero build errors.

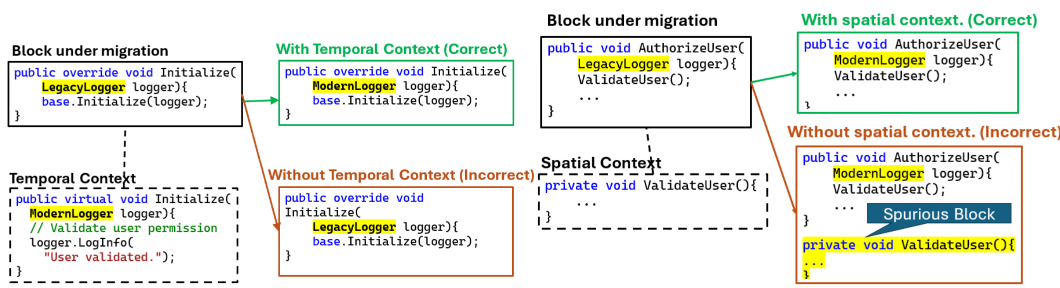
the fix is required in the base class's virtual function signature, causing Build-Repair to misinterpret the correction site.

Multiple Iterations We see the importance of supporting multiple iteration in 3/4 C# migration cases where the first iteration of CodePlan still left some build errors. By requesting the LLM to fix the left-over build errors and seeding CodePlan with the resultant changes, we are able to reduce errors further in all 3 cases, completely eliminating them in 2. We observe that these iterations are especially useful in making the system more robust to inaccuracies in LLM outputs as they allow a pathway for these to be repaired.

Python Temporal Edit Task on External (Public) Repositories. In the Python Temporal Edits task, CodePlan identifies all edit locations across two repositories (T2, T3) and performs well in the third (T1) It also consistently has higher DiffBLEU score and lower Levenshtein Distance, although not always achieving perfect 1.0 and 0 values due to slight differences in LLM edits and ground truth. In contrast, the Pyright-Repair baseline fails to make any derived edits at all in two repositories (T2, T3). In T2, Pyright doesn't flag errors for method call sites due to presence of a default parameter while in T3, Pyright misses edits required by changes to method behavior that were not reflected in changes to type information. Pyright in strict checking mode (Pyright-Strict-Repair) improves results but matches CodePlan only in one repository (T1). CodePlan's change may-impact analysis handles these cases, whereas the oracle-guided repair baseline lacks such detection, focusing on fixing rule violations rather than propagating changes.

	Approach	Matched Blocks	Missed Blocks	Spurious Blocks	Diff BLEU	Levenshtein Distance	Validity Check
I1	CodePlan	151	0	0	1.00	0	✓
	– Temporal Context	135	16	32	0.63	3892	✗ (61) ≠
	– Spatial Context	134	17	51	0.61	4161	✗ (65) ≠
	– Temporal & Spatial	121	30	54	0.51	4524	✗ (69) ≠
E1	CodePlan	65	0	0	0.86	2931	✓
	– Temporal Context	62	3	2	0.74	1014	✗ (8) ≠
	– Spatial Context	62	3	2	0.74	1014	✗ (8) ≠
	– Temporal & Spatial	61	4	2	0.71	1036	✗ (9) ≠
T1	CodePlan	8	2	0	0.90	1044	✗ (0) ≠
	– Spatial Context	8	2	0	0.89	1266	✗ (0) ≠
T2	CodePlan	4	0	0	0.86	147	✓
	– Spatial Context	4	0	0	0.76	443	✓
T3	CodePlan	11	0	0	0.94	288	✓
	– Spatial Context	11	0	0	0.92	325	✓

Table 2: Ablation study with and without temporal/spatial context. For Temporal Edit task (T-1,2,3), temporal context is the necessary part of input and hence, only spatial context is ablated.



(a) Temporal (b) Spatial

Figure 4: Illustration of the importance of (a) temporal and (b) spatial context.

Coeditor Evaluation (Model Ablation). To study the behavior of CodePlan with a smaller model as well as to demonstrate the framework's flexibility, we experimented with using Coeditor in place of codegpt-4-32k. We see that Coeditor-CodePlan misses one edit site each in both T2 and T3 when compared to CodePlan (with the GPT model). In both cases, Coeditor misses adding an argument to a method being edited, thus missing out on editing the callers of that method. We also observe lower DiffBLEU scores and higher Levenshtein Distance (L.D.) in T2 and T3 for Coeditor-CodePlan compared to CodePlan. On T1, we further observe that Coeditor-Pyright-Strict-Repair incorrect local edits lead to 6 Pyright errors popping up. Since Coeditor was not trained with build errors as context, it was unable to fix these. Being a significantly more powerful model, gpt-4-32k is better at understanding the context of the temporal edits, hence the edits it makes are more aligned with the ground truth as compared to Coeditor. These observations indicate the importance of LLMs for tools such as CodePlan.

5.2 RQ2: How important are temporal and spatial contexts to CodePlan's performance?

The results of ablating on temporal and spatial context are reported in Table 2. We observe that both types of context are integral to CodePlan as removing them leads to failure in all the migration tasks as well as more missed and spurious blocks across tasks. We briefly discuss the importance of each aspect here. A detailed discussion is present in the appendix.

Temporal Context. Removing temporal contexts leads to a noticeable increase in *missed* blocks. Without the context of edits made in the past, the LLM is not able to comprehend the need for edits to certain blocks as illustrated in Figure 7 Here, changes to the virtual method in the base class necessitate an edit to the overriding method in the derived class. However, without temporal context, the LLM does not know about the base class's method, leading it to believe that no changes are necessary to the derived class method.

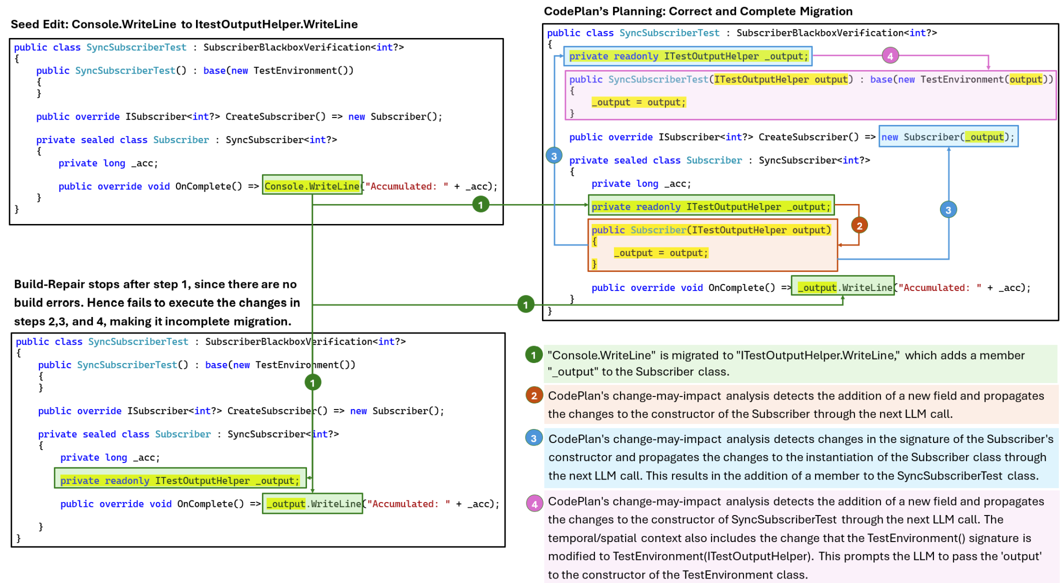


Figure 5: Example from E1 where CodePlan effectively executes a series of changes in steps 1-4 while Build-Repair fails to perform steps 2-4.

Importance of Spatial Context. We also observe an increase in spurious blocks when spatial context is insufficient. In the absence of adequate spatial context, the LLM incorrectly attempts to re-create blocks that exist in the code but are not supplied in the prompt, leading to the generation of spurious code blocks as illustrated in Figure 9. Here, the task is to modify the `AuthorizeUser` method by migrating the logging calls from an old logging framework to a new one. However, due to the lack of spatial context that would specify the existence of the `ValidateUser` method, the LLM attempts to unnecessarily create this method as well.

5.3 RQ3: What are the key differentiators that allow CodePlan to outperform baselines in solving complex coding tasks?

The core of repository-level coding problems is being able to do multi-step reasoning over repositories towards achieving a goal. LLMs have been shown to struggle with direct multi-step reasoning Creswell et al. (2022) and planning Valmeekam et al. (2023). CodePlan leverages the structure inherently present in source code via dependency and change may-impact analysis to provide robust planning. These features also distinguish it from baseline methods like Build-Repair, which prioritize syntactic correctness but overlook contextual details and change propagation as described in Fig 10. The key factors contributing to the success of CodePlan are -

- Dependency analysis provides a rich semantic view of the repository.
- Change may-impact analysis robustly propagates a variety of behavioral changes.
- Comprehensive spatial and temporal context guide the LLM to make the correct edits.
- Support for repairing errors makes it robust to incorrect outputs from the LLM.

Please refer to the supplementary material for detailed discussion of further differentiators.

6 Related Work

LLMs for Coding Tasks. A multitude of LLMs Ahmad et al. (2021); Wang et al. (2021); Austin et al. (2021); Chen et al. (2021); Black et al. (2022); Chowdhery et al. (2022); OpenAI (2023); Touvron et al. (2023) have been trained on large-scale corpora of source code and natural language text. These have been used to accomplish a variety of coding tasks. A few examples of their use include program synthesis Li et al. (2022); Nijkamp et al. (2023), program repair Xia et al. (2023); Jin et al. (2023); Ahmed and Devanbu (2023), vulnerability patching Pearce et al. (2022), inferring program invariants Pei et al. (2023), test generation Schäfer et al. (2023) and multi-task evaluation Tian et al. (2023). These investigations are performed on independent examples that are extracted isolated from their origin repositories and are meant to be accomplished with independent invocations of the LLM. In orthogonal directions, Jiang et al. (2023) uses an LLM to derive a plan given a natural language intent before generating code to solve complex coding problems and Zhang et al. (2023) performs lookahead planning (tree search) to guide token-level decoding of code LMs. In contrast, we consider tasks posed at the scale of code repositories, where an LLM needs to process multiple different interdependent examples across a repository.

Automated Planning and Reasoning with LLMs. Automated planning Ghallab et al. (2004); Russell (2010) is a well-studied topic in AI. Online planning Russell (2010) is used when the effect of actions is not known and the state-space cannot be enumerated *a priori*. It requires monitoring the actions and plan extension. In our case, the edit actions are carried out by an LLM whose results cannot be predicted before-hand and the state-space is unbounded. As a consequence, our adaptive planning is an online algorithm where we monitor the actions and extend the plan through static analysis. Many recent works also develop techniques to iteratively prompt the LLM in different ways to extract a plan to achieve a given goal – leveraging the the common sense knowledge of the LLM for decision making Raman et al. (2022); Huang et al. (2022); Ahn et al. (2022); Yao et al. (2023). In contrast we aim to solve a planning problem within the code domains where we leverage the highly structured nature of code to generate the plan, where each action is a combination of edit site (identified through static analysis and adaptive planning) along with local code edit (generated by the LLM).

Analysis of Code Changes. Static analysis can be expensive to recompute the analysis results every time the code undergoes changes. Incremental program analysis offers techniques to recompute only the analysis results impacted by the change Ryder (1983); Arzt and Bodden (2014); Yur et al. (1999); Person et al. (2011); Busi et al. (2019). Program differencing Apiwattanapong et al. (2004); Lahiri et al. (2012); Kim et al. (2012) and change impact analysis Arnold and Bohner (1996); Jashki et al. (2008) determine the differences in two program versions and the effect of a change on the rest of the program. We analyze the code generated by an LLM and incrementally update the syntactic (e.g., parent-child) and dependency (e.g., caller-callee) relations. We further analyze the likely impact of those changes on related code blocks and create change obligations to be discharged by the LLM.

Learning Edit Patterns. Many approaches have been developed to learn edit patterns from past edits or commits in the form of rewrite rules de Sousa et al. (2021), bug fixes Andersen and Lawall (2010); Bader et al. (2019), type changes Ketkar et al. (2022), API migrations Lamothe et al. (2020); Xu et al. (2019) and neural representations of edits Yin et al. (2019). Approaches such as Meng et al. (2011) and Meng et al. (2013) synthesize context-aware edit scripts from user-provided examples and apply them in new contexts. Other approaches observe the user actions in an IDE to automate repetitive edits Miltner et al. (2019) and temporally-related edit sequences Zhang et al. (2022). We do not aim to learn edit patterns and we do not assume similarities between edits. Our focus is to identify effects of code changes made by an LLM and to guide the LLM towards additional changes that become necessary.

7 Conclusions and Future Work

In this paper, we introduced CodePlan, a neuro-symbolic framework for handling complex repository-level coding tasks involving extensive code changes across interdependent files in large codebases. CodePlan employs incremental dependency analysis, change may-impact analysis, and adaptive planning to coordinate multi-step code edits using large language models. Our evaluation on various code repositories in C# and Python demonstrated that CodePlan surpasses baseline methods in accuracy. It shows great promise for automating repository-level coding tasks, but there's room for future improvements. We plan to extend its applicability to more programming languages and explore enhancements to its editing strategy and analysis as well as conducting large-scale experiments to further refine CodePlan's effectiveness across diverse coding tasks. Additionally there are opportunities to explore the use of the LLM itself for planning within the dependency graph.

References

[n. d.]. Jedi. `https://github.com/davidhalter/jedi`.

[n. d.]. MS-Build. https://learn.microsoft.com/en-us/visualstudio/msbuild/msbuild.

[n. d.]. Pyright. https://github.com/microsoft/pyright.

2020. Reactive Streams TCK. https://github.com/reactive-streams/reactive-streams-dotnet/tree/master/src/tck.

2022. das-qna-api. https://github.com/SkillsFundingAgency/das-qna-api.

2023. Amazon Code Whisperer - AI Code Generator. https://aws.amazon.com/codewhisperer/.

2023. audiocraft. https://github.com/facebookresearch/audiocraft.

2023. GitHub Copilot chat for Visual Studio 2022. https://devblogs.microsoft.com/visualstudio/github-copilot-chat-for-visual-studio-2022/.

2023. GitHub Copilot: Your AI pair programmer. https://github.com/features/copilot.

2023. JARVIS. https://github.com/microsoft/JARVIS.

2023. Replit. https://replit.com/.

2023. whisper. https://github.com/openai/whisper.

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. arXiv:2103.06333 [cs.CL]

Toufique Ahmed and Premkumar Devanbu. 2023. Better patching using LLM prompting, via Self-Consistency. arXiv:2306.00108 [cs.SE]

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Keerthana Gopalakrishnan, Karol Hausman, Alexander Herzog, Daniel Ho, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Eric Jang, Rosario Jauregui Ruano, Kyle Jeffrey, Sally Jesmonth, Nikhil Jayant Joshi, Ryan C. Julian, Dmitry Kalashnikov, Yuheng Kuang, Kuang-Huei Lee, Sergey Levine, Yao Lu, Linda Luu, Carolina Parada, Peter Pastor, Jornell Quiambao, Kanishka Rao, Jarek Rettinghouse, Diego M Reyes, Pierre Sermanet, Nicolas Sievers, Clayton Tan, Alexander Toshev, Vincent Vanhoucke, F. Xia, Ted Xiao, Peng Xu, Sichun Xu, and Mengyuan Yan. 2022. Do As I Can, Not As I Say: Grounding Language in Robotic Affordances. In *Conference on Robot Learning*. https://api.semanticscholar.org/CorpusID:247939706

Alfred V Aho, Ravi Sethi, Jeffrey D Ullman, et al. 2007. *Compilers: principles, techniques, and tools*. Vol. 2. Addison-wesley Reading.

Jesper Andersen and Julia L Lawall. 2010. Generic patch inference. *Automated software engineering* 17 (2010), 119–148.

Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. 2004. A differencing algorithm for object-oriented programs. In *Proceedings. 19th International Conference on Automated Software Engineering, 2004.* IEEE, 2–13.

RS Arnold and SA Bohner. 1996. An introduction to software change impact analysis. *Software Change Impact Analysis* (1996), 1–26.

Steven Arzt and Eric Bodden. 2014. Reviser: efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes. In *Proceedings of the 36th International Conference on Software Engineering*. 288–298.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. http://arxiv.org/abs/2108.07732 arXiv:2108.07732 [cs].

Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to Fix Bugs Automatically. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 159 (Oct. 2019), 27 pages. https://doi.org/10.1145/3360585

Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, and others. 2022. Gpt-neox-20b: An open-source autoregressive language model. *arXiv preprint arXiv:2204.06745* (2022).

Bruno Blanchet. 2003. Escape analysis for JavaTM: Theory and practice. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 25, 6 (2003), 713–775.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

Matteo Busi, Pierpaolo Degano, and Letterio Galletta. 2019. Using standard typing algorithms incrementally. In *NASA Formal Methods: 11th International Symposium, NFM 2019, Houston, TX, USA, May 7–9, 2019, Proceedings 11*. Springer, 106–122.

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

Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C Sreedhar, and Sam Midkiff. 1999. Escape analysis for Java. *Acm Sigplan Notices* 34, 10 (1999), 1–19.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311* (2022).

Antonia Creswell, Murray Shanahan, and Irina Higgins. 2022. Selection-Inference: Exploiting Large Language Models for Interpretable Logical Reasoning. *ArXiv* abs/2205.09712 (2022). https://api.semanticscholar.org/CorpusID:248887351

Reudismam Rolim de Sousa, Gustavo Soares, Rohit Gheyi, Titus Barik, and Loris D'Antoni. 2021. Learning Quick Fixes from Code Repositories. In *SBES '21: 35th Brazilian Symposium on Software Engineering, Joinville, Santa Catarina, Brazil, 27 September 2021 - 1 October 2021*, Cristiano D. Vasconcellos, Karina Girardi Roggia, Vanessa Collere, and Paulo Bousfield (Eds.). ACM, 74–83. https://doi.org/10.1145/3474624.3474650

Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP'95—Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995 9*. Springer, 77–101.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).

Malik Ghallab, Dana Nau, and Paolo Traverso. 2004. *Automated Planning: theory and practice*. Elsevier.

Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Commun. ACM* 59, 5 (2016), 122–131.

Wenlong Huang, P. Abbeel, Deepak Pathak, and Igor Mordatch. 2022. Language Models as Zero-Shot Planners: Extracting Actionable Knowledge for Embodied Agents. *ArXiv* abs/2201.07207 (2022). https://api.semanticscholar.org/CorpusID:246035276

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*. 84–90.

Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li. 2023. Self-planning Code Generation with Large Language Model. arXiv:2303.06689 [cs.SE]

Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. InferFix: End-to-End Program Repair with LLMs. *arXiv preprint arXiv:2303.07263* (2023).

Ameya Ketkar, Oleg Smirnov, Nikolaos Tsantalis, Danny Dig, and Timofey Bryksin. 2022. Inferring and applying type changes. In *Proceedings of the 44th International Conference on Software Engineering*. 1206–1218.

Miryung Kim, David Notkin, Dan Grossman, and Gary Wilson. 2012. Identifying and summarizing systematic code changes via rule inference. *IEEE Transactions on Software Engineering* 39, 1 (2012), 45–62.

Shuvendu K Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *Computer Aided Verification: 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings 24*. Springer, 712–717.

Maxime Lamothe, Weiyi Shang, and Tse-Hsun Peter Chen. 2020. A3: Assisting android api migrations using code examples. *IEEE Transactions on Software Engineering* 48, 2 (2020), 417–431.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (2022), 1092–1097. https://doi.org/10.1126/science.abq1158 _eprint: https://www.science.org/doi/pdf/10.1126/science.abq1158.

Na Meng, Miryung Kim, and Kathryn S McKinley. 2011. Sydit: Creating and applying a program transformation from an example. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 440–443.

Na Meng, Miryung Kim, and Kathryn S McKinley. 2013. LASE: locating and applying systematic edits by learning from examples. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 502–511.

Anders Miltner, Sumit Gulwani, Vu Le, Alan Leung, Arjun Radhakrishna, Gustavo Soares, Ashish Tiwari, and Abhishek Udupa. 2019. On the fly synthesis of edit suggestions. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=iaYcJKpY2B_

OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2022. Examining Zero-Shot Vulnerability Repair with Large Language Models. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 1–18.

Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023. Can Large Language Models Reason about Program Invariants? (2023).

Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. 2011. Directed incremental symbolic execution. *Acm Sigplan Notices* 46, 6 (2011), 504–515.

S. Sundar Raman, Vanya Cohen, Eric Rosen, Ifrah Idrees, David Paulius, and Stefanie Tellex. 2022. Planning with Large Language Models via Corrective Re-prompting. *ArXiv* abs/2211.09935 (2022). https://api.semanticscholar.org/CorpusID:253707906

Stuart J Russell. 2010. *Artificial intelligence a modern approach*. Pearson Education, Inc.

Barbara G Ryder. 1983. Incremental data flow analysis. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 167–176.

Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. Adaptive test generation using a large language model. *arXiv preprint arXiv:2302.06527* (2023).

Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F. Bissyandé. 2023. Is ChatGPT the Ultimate Programming Assistant – How far is it? arXiv:2304.11938 [cs.SE]

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, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL]

Karthik Valmeekam, Alberto Olmo, Sarath Sreedharan, and Subbarao Kambhampati. 2023. Large Language Models Still Can't Plan (A Benchmark for LLMs on Planning and Reasoning about Change). arXiv:2206.10498 [cs.CL]

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. *ArXiv* abs/2109.00859 (2021).

Jiayi Wei, Greg Durrett, and Isil Dillig. 2023. Coeditor: Leveraging Contextual Changes for Multi-round Code Auto-editing. arXiv:2305.18584 [cs.SE]

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems* 35 (2022), 24824–24837.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery*.

Shengzhe Xu, Ziqi Dong, and Na Meng. 2019. Meditor: inference and application of API migration edits. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 335–346.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:2210.03629 [cs.CL]

Pengcheng Yin, Graham Neubig, Miltiadis Allamanis, Marc Brockschmidt, and Alexander Gaunt. 2019. Learning to Represent Edits. In *ICLR 2019*. `https://www.microsoft.com/en-us/research/publication/learning-to-represent-edits/` arXiv:1810.13337 [cs.LG].

Jyh-shiarn Yur, Barbara G Ryder, and William A Landi. 1999. An incremental flow-and context-sensitive pointer aliasing analysis. In *Proceedings of the 21st International conference on Software Engineering*. 442–451.

Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B. Tenenbaum, and Chuang Gan. 2023. Planning with Large Language Models for Code Generation. arXiv:2303.05510 [cs.LG]

Yuhao Zhang, Yasharth Bajpai, Priyanshu Gupta, Ameya Ketkar, Miltiadis Allamanis, Titus Barik, Sumit Gulwani, Arjun Radhakrishna, Mohammad Raza, Gustavo Soares, and Ashish Tiwari. 2022. Overwatch: Learning patterns in code edit sequences. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 395–423. `https://doi.org/10.1145/3563302`

A Appendix A

A.1 Implementation

In our implementation of CodePlan, we construct the Dependency Graph, by parsing code files using the "tree-sitter" library Brunsfeld et al . (2023), which provides identification of code blocks such as classes, methods, import statements etc... as well as the AST. In C#, for relationships such as caller-callee, overrides-overridden, and more, we establish edges within the Dependency Graph by implementing custom logic that traces relationships within the AST. For Python, we utilize Jedi Jed ([n. d.]), a static analysis tool, to identify relationships. Our implementation integrates the gpt-4-32k LLM for code edits, providing it with structured input for enhanced quality and accuracy. We use `temperature = 0` and, `top_p = 1` and sample a single response for every call to the LLM. While our current implementation handles C# and Python repositories, it is extensible to other programming languages due to the various abstractions and layered architecture of CodePlan

A.2 Data

At present, there is no benchmark to evaluate repository-level coding tasks. We therefore construct a benchmark by selecting code repositories of varying complexities and sizes. This includes internal C# Repositories (I1, I2) that are large proprietary codebases requiring non-trivial migrations from legacy to modern logging frameworks. We also include External Repositories from Public GitHub, focusing on Migration and Temporal Edits Wei et al. (2023) tasks. For Migration, we selected C# repositories (E1 rep (2020), E2 rep (2022)) having API or framework migrations, while for Temporal Edits, which involves series of code changes following initial edits, we selected Python repositories (T1 whi (2023), T2 aud (2023), T3 JAR (2023)). We identified the GitHub repositories by searching for migration and multi-step temporal edit scenarios, and selected corresponding pull requests. As reported in Table 3, these repositories have between 4–168 files and 1.8K–20.4K lines of code while the *number of files changed* range from 2–97. *Seed changes* are the number of initial edits (1–63 changes), considered as the starting point, and *derived changes* (3–375 changes) are the subsequent edits that follow the initial seed changes, which CodePlan is expected to automate. *Diff size b/w source and target (lines)* is the total number of lines (15–4.9K) in the file-wise diff between the Source and Target versions of the repositories. This tells us the size of the required code changes. We used the same prompt template for C# migration across internal and public repositories (81 lines, as reported in *Prompt template size (lines)*) and another one (75 lines) for Python temporal edits.

A.3 Data Pre-Processing

For each repository, we collected the before (*Source*) and after (*Target*) snapshots of the code from the pull requests. The pull requests contained code changes unrelated to the task. We either 1) applied them to both Source and Target, or 2) removed them from the Target. From the remaining changes, *seed changes* were identified through manual inspection. To prepare the Source for evaluation with both CodePlan and the baselines, we patched in the seed changes or prepared instructions for the LLM to carry them out. We observed that in contrast to the internal repositories, the external repositories did not have uniformity in the coding styles. Our initial experimentation revealed that this resulted in even the correct edits being flagged as differing from the ground truth edits. To mitigate this, we pre-process the Target repositories to ensure uniform coding practices. This may involve formatting changes such as standardising whitespace, adding commas to lists or ordering imports as well as minor code changes such as enforcing common coding practices or removing code-edits unrelated to the task. Note that all methods are evaluated on the same Source repositories (after the pre-processing).

A.4 Benchmark Statistics

We now discuss statistics of our benchmark to understand its scale and complexity (Table 3). The *number of files changed* range from 2–97. *Seed changes* are the number of initial edits (1–63 changes), considered as the starting point, and *derived changes* (3–375 changes) are the subsequent edits that follow the initial seed changes, which CodePlan is expected to automate. *Diff size b/w source and target (lines)* is the total number of lines (15–4.9K) in the file-wise diff between the Source and Target versions of the repositories. This tells us the size of the required code changes. Similarly, we

Repositories	Migration				Temporal Edits		
	I1	I2	E1	E2	T1	T2	T3
Number of files	91	168	55	341	21	137	4
Lines of code	8853	16476	8868	1978	3883	20413	1874
Number of files changed	47	97	21	23	2	2	3
Number of seed changes	41	63	42	50	2	1	1
Number of derived changes	110	375	22	68	8	3	10
Diff size b/w Source & Target (lines)	1744	4902	1024	154	104	15	39
Size of seed edits (lines)	242	242	379	340	76	4	1
Prompt template size (lines)	81	81	81	110	75	75	75

Table 3: Benchmark statistics.

report the *size of seed edits*. We used the same prompt template for C# migration across internal and public repositories (81 lines, as reported in *Prompt template size (lines)*) and another one (75 lines) for Python temporal edits.

A.5 Limitations and Threats to Validity

CodePlan relies on high-quality dependency analysis, which works well in statically typed languages like C# and Java but can be challenging in dynamically typed languages like Python or JavaScript without type hints due to their dynamic nature.

Our current CodePlan implementation mainly deals with code block relations through static analysis. However, real-world software systems have dynamic dependencies, like data flows, complex dispatching, and execution dependencies, and include various artifacts beyond code files. Addressing these dynamic dependencies and software artifacts is a priority for our future work.

CodePlan edits one code block at a time, which might not be the most efficient approach in all cases. Also, LLMs can make errors while editing code. Our ablations show that CodePlan's spatial and temporal context helps avoid such errors considerably. Besides, instead of blindly trusting the changes made by the LLM, CodePlan employs an oracle to validate the changes and initiates further iterations if the changes are found unsatisfactory. This oracle-in-the-loop strategy helped us get to the desired, error-free edits in multiple C# migration cases. We want to explore techniques to exploit feedback from oracles to improve reliability of repository-wide changes.

We chose multiple repositories for two challenging tasks (migration and temporal edits) in two languages (C# and Python) to assess CodePlan's generality. These tasks and repositories represent real-world scenarios. However, due to limited access to the LLM, our evaluation is confined to the current experiments. There is a potential concern that our selected repositories might have been part of the LLM's training set. To address this, we conducted experiments on two proprietary internal C# repositories that the LLM didn't encounter during training. Moreover, except for E1, our tasks use GitHub pull requests created after September 2021, the LLM's training data cutoff date. We intentionally included E1 before this date to test if the model could perform better, but our baseline and ablation results indicate that it couldn't make the desired edits without appropriate context. We aim to expand our experimental results to include more repositories in the future.

Although our current methodology employs zero-shot prompting, there exists potential to include few-shot examples Brown et al. (2020), Chain of Thought (CoT) Wei et al. (2022), and other techniques, which can improve the performance of CodePlan further.

A.6 Design Details

The design section 3 and algorithm 2 provide a highly abstracted picture of CodePlan. Some terms have been renamed or combined to make the description less verbose. Complete details details of the CodePlan algorithm (Section A.6.1) and its core components: static analysis (Section A.6.2), adaptive planning and plan execution (Section A.6.3) are provided in this section.

A.6.1 The CodePlan Algorithm

The CodePlan algorithm (Algorithm 2) takes four inputs:

1. the source code of a repository, R

Algorithm 2: The CodePlan algorithm to automate repository-level coding tasks. The data structures and functions in Cyan and Orchid are explained in Section A.6.2– A.6.3 respectively.

```
1  /* Inputs: R is the source code of a repository, Delta_seeds is a set of seed edit
        specifications, Theta is an oracle and L is an LLM. */

3  CodePlan(R, Delta_seeds, Theta, L):
4    let mutable G: PlanGraph = null in
5    let mutable D: DependencyGraph = ConstructDependencyGraph(R) in
6      while Delta_seeds is not empty
7        IntializePlanGraph(G, Delta_seeds)
8        AdaptivePlanAndExecute(R, D, G)
9        Delta_seeds := Theta(R)

11 InitializePlanGraph(G, Delta_seeds):
12   for each ⟨B, I⟩ in Delta_seeds
13     AddRoot(G, ⟨B, I, Pending⟩)

15 AdaptivePlanAndExecute(R, D, G):
16   while G has Nodes with Pending status
17     let ⟨B, I, Pending⟩ = GetNextPending(G) in
18     // First step:  extract fragment of code
19     let Fragmemt = ExtractCodeFragment(B, R) in
20     // Second step:  gather context of the edit
21     let Context = GatherContext(B, R, D) in
22     // Third step:  use the LLM to get edited code fragment
23     let Prompt = MakePrompt(Fragment, I, Context) in
24     let NewFragment = InvokeLLM(L, Prompt) in
25     // Fourth step:  merge the updated code fragment into R
26     let R := Merge(NewFragment, B, R) in
27     let Labels = ClassifyChanges(Fragment, NewFragment) in
28     let D' = UpdateDependencyGraph(D, Labels, Fragment, NewFragment, B) in
29     // Fifth step:  adaptively plan and propogate the effect of the edit on dependant code
30     let BlockRelationPairs=GetAffectedBlocks(Labels, B, D, D') in
31       MarkCompleted(B, G)
32       for each ⟨B', rel⟩ in BlockRelationPairs
33         let N = GetNode(B) in
34         let M = SelectOrAddNode(B', Nil, Pending) in
35           AddEdge(G, M, N, rel)
36     D := D'

38 GatherContext(B, R, D):
39   let SC = GetSpatialContext(B, R) in
40   let TC = GetTemporalContext(G, B) in
41     (SC, TC)
```

2. a set of seed edit specifications for the task in hand, Δ_{seeds}
3. an oracle, Θ
4. an LLM, L

The core data structure maintained by the algorithm is a *plan graph* G, a directed acyclic graph with multiple root nodes (line 4). Each node in the plan graph is a tuple $\langle B, I, Status \rangle$, where B is a block of code (that is, a sequence of code locations) in the repository R, I is an edit instruction (along the lines of the example shown in Figure 1), and $Status$ is either $pending$ or $completed$.

The CodePlan algorithm also maintains a *dependency graph* D (line 5). Figure 6 illustrates the dependency graph structure. We will discuss it in details in Section A.6.2. For now, it suffices to know that the dependency graph D represents the syntactic and semantic dependency relations between code blocks in the repository R.

The loop at lines 6–9 is executed until Δ_{seeds} is non-empty. Line 7 calls the `InitializePlanGraph` function (lines 11–13) that adds all the changes in Δ_{seeds} as root nodes of the plan graph. Each edit specification comprises of a code block B and an edit instruction I. The status is set to pending for the root nodes (line 13). The function `AdaptivePlanAndExecute` is called at line 8 which executes the plan, updates the dependency graph with each code change and extends the plan as necessary. Once the plan graph is completely executed, the oracle Θ is run on the repository. It returns error locations and diagnostic messages which form Δ_{seeds} for the next iteration. If the repository passes the oracle's checks then it returns an empty set and the CodePlan algorithm terminates.

We now discuss `AdaptivePlanAndExecute`, which is the main work horse. It iteratively picks each pending node and processes it. Processing a pending node for a block B with edit instruction I involves the following five steps:

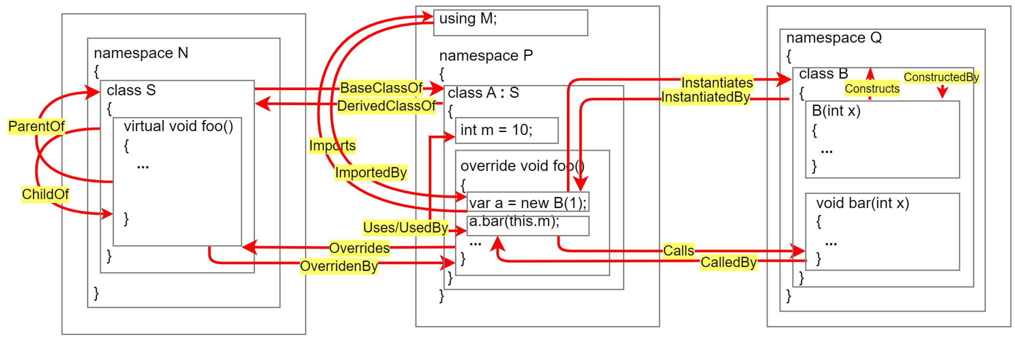


Figure 6: Illustration of the dependency graph annotated with relations as the edge labels.

1. **The *first step* (line 19) is to extract the fragment of code to edit.** Simply extracting code of the block B loses information about relationship of B with the surrounding code. Keeping the entire file on the other hand takes up prompt space and is often unnecessary. We found the surrounding context is most helpful when a block belongs to a class. For such blocks, we sketch the enclosing class. That is, in addition to the code of block B, we also keep declarations of the enclosing class and its members. As we discuss later, this sketched representation also helps us merge the LLM's output into a source code file more easily.
2. **The *second step* (line 21) is to gather the context of the edit.** The context of the edit (line 38–41) consists of (a) *spatial context*, which contains related code such as methods called from the block B, and (b) *temporal context*, which contains the previous edits that *caused* the need to edit the block B. The temporal context is formed by edits along the paths from the root nodes of the plan graph to B.
3. **The *third step* (lines 23–24) constructs a prompt** using the fragment extracted in the first step, the instruction I from the edit specification and the context extracted in the second step, and **invokes the LLM using the prompt** to get the edited code fragment.
4. **The *fourth step* (lines 26–28) merges the edited code back into the repository.** Since the code is updated, many dependency relationships such as caller-callee, class hierarchy, etc. may need to change, and hence, **this step also updates the dependency graph D.**
5. **The *fifth and final step* (lines 30–35) does adaptive planning to propagate the effects of the current edit on dependant code blocks.** This involves classifying the change in the edited block, and depending on the type of change, picking the right dependencies in the dependency graph to traverse and locate affected blocks. For instance, if the edit of a method m in the current block B involves update to the signature of the method, then all callers of m get affected (the scenario in Figure 3). For each affected block B' and the dependency relation `rel` connecting B to B' in the dependency graph, we get a pair $\langle B', \texttt{rel}\rangle$. If a node exists for B' in the plan graph and it is pending, then we add an edge from B to B' labeled with `rel` to the plan graph. Otherwise, the edge is added to a newly created node for B' (line 34). The block B is marked as completed (line 31).

A.6.2 Static Analysis Components

We now turn our attention to the static analysis components used in CodePlan. We will cover all the data structures and functions in Cyan background from Algorithm 2.

***Incremental Dependency Analysis*:**

An LLM can be provided a code fragment and an instruction to edit it in a prompt. While the LLM may perform the desired edit accurately, analyzing the impact of the edit on the rest of the repository is outside the scope of the LLM call. We believe static analysis is well-suited to do this and propose an incremental dependency analysis for the same.

`DependencyGraph`. Dependency analysis Aho et al. (2007) is used for tracking syntactic and semantic relations between code elements. In our case, we are interested in relations between import statements, methods, classes, field declarations and statements (excluding those that operate only on variables defined locally within the enclosing method). Formally, a *dependency graph* D

$= (N, E)$ where N is a set of nodes representing the code blocks mentioned above and E is a set of labeled edges where the edge label gives the relation between the source and target nodes of the edge. Figure 6 illustrates all the relations we track. The relations include (1) *syntactic relations* (ParentOf and ChildOf, Construct and ConstructedBy) between a block c and the block p that encloses c syntactically; a special case being a constructor and its enclosing class related by Construct and ConstructedBy, (2) *import relations* (Imports and ImportedBy) between an import statement and statements that use the imported modules, (3) *inheritance relations* (BaseClassOf and DerivedClassOf) between a class and its superclass, (4) *method override relations* (Overrides and OverridenBy) between an overriding method and the overriden method, (5) *method invocation relations* (Calls and CalledBy) between a statement and the method it calls, (6) *object instantiation relations* (Instantiates and InstantiatedBy) between a statement and the constructor of the object it creates, and (7) *field use relations* (Uses and UsedBy) between a statement and the declaration of a field it uses.

`ConstructDependencyGraph`. The dependency relations are derived across the source code spread over the repository through static analysis. We represent the source code of a repository as a forest of abstract syntax trees (ASTs) and add the dependency edges between AST sub-trees. A file-local analysis derives the syntactic and import relations. All other relations require an inter-class, inter-procedural analysis that can span file boundaries. In particular, we use the class hierarchy analysis Dean et al. (1995) for deriving the semantic relations.

`ClassifyChanges`. As discussed in Section A.6.1, in the fourth step, CodePlan merges the code generated by the LLM into the repository. By pattern-matching the code before and after, we classify the code changes. Table 4 (the first column) gives the type of atomic change. Broadly, the changes are organized as modification, addition and deletion changes, and further by which construct is changed. We distinguish between method body and method signature changes. Similarly, we distinguish between changes to a class declaration, to its constructor or to its fields. The changes to import statements or the statements that use imports are also identified. These are *atomic changes*. An LLM can make multiple simultaneous edits in the given code fragment, resulting in multiple atomic changes, all of which are identified by the `ClassifyChanges` function.

`UpdateDependencyGraph`. As code generated by the LLM is merged, the dependency relations associated with the code at the change site are re-analyzed. Table 4 (the second column) gives the rules to update the dependency graph D to D′ based on the labels inferred by `ClassifyChanges`. For modification changes, we recompute the relations of the changed code except for constructors. A constructor is related to its enclosing class by a syntactic relation which does not have to be recomputed. For addition changes, new nodes and edges are created for the added code. Edges corresponding to syntactic relations are created in a straightforward manner. If a change simultaneously adds an element (an import, a method, a field or a class) and its uses, we create a node for the added element before analyzing the statements that use it. Addition of a method needs special handling as shown in the table: if an overriding method C.M is added then the Calls/CalledBy edges incident on the matching overriden method B.M are redirected to C.M if the call is issued on a receiver object of type C. The deletion of an overriding method requires an analogous treatment as stated in Table 4. All other deletions require removing nodes and edges as stated in the table.

***Change May-Impact Analysis*:**

In the fifth step, CodePlan identifies the code blocks that may have been impacted by the code change by the LLM. Let Rel(D, B, rel) be the set of blocks that are connected to a block B via relation rel in the dependency graph D. Let D and D′ be the dependency graph before and after the updates in Table 4.

`GetAffectedBlocks`. The last column in Table 4 tells us how to identify blocks affected by a code change. When the body of a method M is edited, we perform escape analysis Choi et al. (1999); Blanchet (2003) to identify if any object accessible in the callers of M (an escaping object) has been affected by the change. If yes, the callers of M (identified through Rel(D, M, CalledBy)) are identified as affected blocks. Otherwise, the change is localized to the method and no blocks are affected. If the signature of a method is edited, the callers and methods related to it through method-override relations in the inheritance hierarchy are affected. The signature change can affect the Overrides and OverridenBy relations themselves, e.g., addition or deletion of the `@Override` access modifier. Therefore, the blocks related by these relations in the updated dependency graph D′ are also considered as affected as shown in Table 4. When a field F of a class C is modified, the

Atomic Change	Dependency Graph Update	Change May-Impact Analysis
	Modification Changes	
Body of method M	Recompute the edges incident on the statements in the method body.	If an escaping object is modified then Rel(D, M, CalledBy) else Nil.
Signature of method M	Recompute the edges incident on the method.	Rel(D, M, CalledBy), Rel(D, M, Overrides), Rel(D, M, OverriddenBy), Rel(D′, M, Overrides), Rel(D′, M, OverriddenBy)
Field F in class C	Recompute the edges incident on the field.	Rel(D, F, UsedBy), Rel(D, C, ConstructedBy), Rel(D, C, BaseClassOf), Rel(D, C, DerivedClassOf)
Declaration of class C	Recompute the edges incident on the class.	Rel(D, C, InstantiatedBy), Rel(D, C, BaseClassOf), Rel(D, C, DerivedClassOf), Rel(D′, C, BaseClassOf), Rel(D′, C, DerivedClassOf)
Signature of constructor of class C	No change.	Rel(D, C, InstantiatedBy), Rel(D, C, BaseClassOf), Rel(D, C, DerivedClassOf)
Import/Using statement I	Recompute the edges incident on the import statement.	Rel(D, I, ImportedBy)
	Addition Changes	
Method M in class C	Add new node and edges by analyzing the method. If C.M overrides a base class method B.M then redirect the Calls/CalledBy edges from B.M to C.M if the receiver object is of type C.	Rel(D, C, BaseClassOf), Rel(D, C, DerivedClassOf), Rel(D′, M, CalledBy)
Field F in class C	Add new node and edges by analyzing the field declaration.	Rel(D, C, ConstructedBy), Rel(D, C, BaseClassOf), Rel(D, C, DerivedClassOf)
Declaration of class C	Add new node and edges by analyzing the class declaration.	Nil
Constructor of class C	Add new node and edges by analyzing the constructor.	Rel(D, C, InstantiatedBy), Rel(D, C, BaseClassOf), Rel(D, C, DerivedClassOf)
Import/Using statement I	Add new node and edges by analyzing the import statement.	Nil
	Deletion Changes	
Method M in class C	Remove the node for M and edges incident on M. If C.M overrides a base class method B.M then redirect the Calls/CalledBy edges from C.M to B.M if the receiver object is of type C.	Rel(D, M, CalledBy), Rel(D, M, Overrides), Rel(D, M, OverriddenBy)
Field F in class C	Remove the node of the field and edges incident on it.	Rel(D, F, UsedBy), Rel(D, C, ConstructedBy), Rel(D, C, BaseClassOf), Rel(D, C, DerivedClassOf)
Declaration of class C	Remove the node of the class and edges incident on it.	Rel(D, C, InstantiatedBy), Rel(D, C, BaseClassOf), Rel(D, C, DerivedClassOf)
Constructor of class C	Remove edges to the class due to object instatiations using the constructor.	Rel(D, C, InstantiatedBy), Rel(D, C, BaseClassOf), Rel(D, C, DerivedClassOf)
Import/Using statement I	Remove the node of the import statement and edges incident on it.	Rel(D, I, ImportedBy)

Table 4: Rules for updating the dependency graph and for change may-impact analysis for atomic changes. We refer to the dependency graphs before and after the updates by D and D′ respectively.

statements that use F, the constructors of C and sub/super-classes of C are affected. When a class is modified, the methods that instantiate it and its sub/super-classes as per D and D′ are affected. A modification to a constructor has a similar rule except that such a change does not change inheritance relations and hence, only D is required. When an import statement I is modified, the statements that use the imported module are affected.

The addition and deletion changes are less complex than the modification changes, and their rules are designed along the same lines as discussed above. In the interest of space, we do not explain each of them step-by-step. We assume that there is no use of a newly added class or an import in the code. Therefore, adding them does not result in any affected blocks. In our experiments, we have found the rules in Table 4 to be adequate. However, CodePlan can be easily configured to accommodate extensions of the rules in Table 4 if necessary.

A.6.3 Adaptive Planning and Plan Execution

We now discuss the data structures and functions from Algorithm 2 in the Orchid background.

***Adaptive Planning*:** Having identified the affected blocks (using `GetAffectedBlocks`), CodePlan creates change obligations that need to be discharged using an LLM to make the dependent code consistent with the change. As discussed in Section A.6.1, this is an iterative process.

`PlanGraph`. A *plan graph* P $= (O, C)$ is a directed acyclic graph with a set of *obligations* O, each of which is a triple $\langle B, I, status \rangle$ where B is a block, I is an instruction and status is either pending or completed. An edge in C records the *cause*, the dependency relation between the blocks in the source and target obligations. In other words, the edge label identifies which Rel clause in a change may-impact rule in Table 4 results in creation of the target obligation.

`ExtractCodeFragment`. As discussed in the first step in Section A.6.1, simply extracting code for a block B is sub-optimal as it loses context. The `ExtractCodeFragment` function takes the whole class the code block belongs to, keeps the complete code for B and retains only declarations of the class and other class members. We found this to be useful because the names and types of the class and other members provide additional context to the LLM. Often times the LLM needs to make multiple simultaneous changes. For example, in some of our case studies, the LLM has to add a field declaration, take an argument to a constructor and use it within the constructor to initialize the field. Providing the sketch of the surrounding code as a code fragment to the LLM allows the LLM to make these changes at the right places. The code fragment extraction logic is implemented by traversing the AST and "folding" away the subtrees (e.g., method bodies) that are sketched. This reduces the code size without sacrificing naturalness of code Hindle et al. (2016). As stated in Section 2, this sketched representation also allows us to place the LLM generated code back into the AST without ambiguity, even when there are multiple simultaneous changes.

`GetSpatialContext`. Spatial context in CodePlan refers to the arrangement and relationships of code blocks within a codebase, helping understand how classes, functions, variables, and modules are structured and interact. It's crucial for making accurate code changes. CodePlan utilizes the dependency graph to extract spatial context. This enables CodePlan to make context-aware code modifications that are consistent with the code's spatial organization, enhancing the accuracy and reliability of its code editing capabilities. In particular, when generating an edit to a method, CodePlan fetches all the methods called in the body of the method to be edited, class members accessed, along with methods that override or are overridden by the method to be edited. For constructors, we fetch the constructor of super-class if present.

`GetTemporalContext`. The plan graph records all change obligations and their inter-dependences. Extracting temporal context is accomplished by linearizing all paths from the root nodes of the plan graph to the target node. Each change is a pair of the code fragments before and after the change. The temporal context also states the "causes" (recorded as edge labels) that connect the target node with its predecessor nodes. For example, if a node A is connected to B with a CalledBy edge, then the temporal context for B is the before/after fragments for A and a statement that says that "B calls A", which helps the LLM understand the cause-effect relation between the latest temporal change (change to A) and the current obligation (to make a change to B).

***Plan Execution*:** CodePlan iteratively selects a pending node in the plan graph and invokes an LLM to discharge the change obligation.

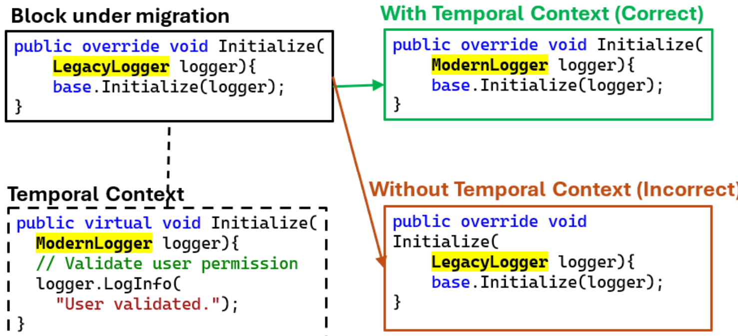


Figure 7: Illustration of importance of temporal context. Failure to update LogacyLogger to Modern-Logger in Initialize() method is the results of missing missing temporal context.

`MakePrompt`. Having extracted the code fragment to be edited along with the relevant spatial and temporal context, we construct a prompt to pass to the LLM with the structure given below. We open with the task specific instructions p_1 followed by listing the edits made in the repository so far p_2 that are relevant to the fragment being edited (temporal context). The next section p_3 notes how each of the fragments present in p_2 are related to the fragment to be edited. This is followed by the spatial context p_4 and the fragment to the edited p_5.

p_1 Task Instructions: *Your task is to ...*
p_2 Earlier Code Changes: *These are edits that have been made in the code-base previously -*
p_3 Causes for Change: *The change is required due to -*
p_4 Related Code: *The following code maybe related -*
p_5 Code to be Changed Next: *The existing code is given below -*

Edit the "Code to be Changed Next" and produce "Changed Code" below. Edit the "Code to be Changed Next" according to the "Task Instructions" to make it consistent with the "Earlier Code Changes", "Causes for Change" and "Related Code". If no changes are needed, output "No changes."

Oracle and Plan Iterations. Once all the nodes in the plan graph are marked as completed, an *iteration* of CodePlan is completed. As shown in Figure 2, the oracle is invoked on the repository. If it flags any errors, the error locations and messages are used for seed changes for the next iteration and the planning resumes once again. If the oracle does not flag any errors, CodePlan terminates.

B Appendix B

B.1 Results Discussion

B.1.1 RQ2: How important are the temporal and spatial contexts for CodePlan's performance?

The results regarding the importance of temporal and spatial contexts for CodePlan's planning (RQ2) reveal critical insights. As observed in Table 2, when temporal contexts are not considered, there is a noticeable increase in missed blocks during the code modification process. This increase is attributed to the Large Language Model (LLM) not making necessary changes to certain code blocks due to its inability to comprehend the need for those modifications in the absence of temporal context.

An illustrative example in Figure 7 exemplifies this issue. In this scenario, a correction is required in the base class's virtual method based on changes to the overridden method's signature in the derived class. However, the LLM, lacking temporal context, does not possess information about the derived class's method, leading it to believe that no changes are necessary to the base class method. This highlights the critical role that temporal context plays in understanding code dependencies and ensuring accurate updates.

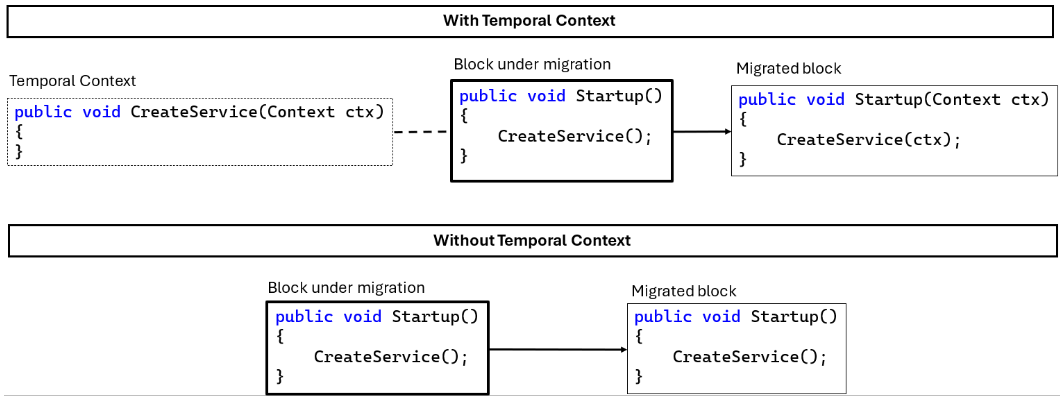


Figure 8: Illustration of importance of temporal context. Failed update to Startup() method is the results of missing missing temporal context.

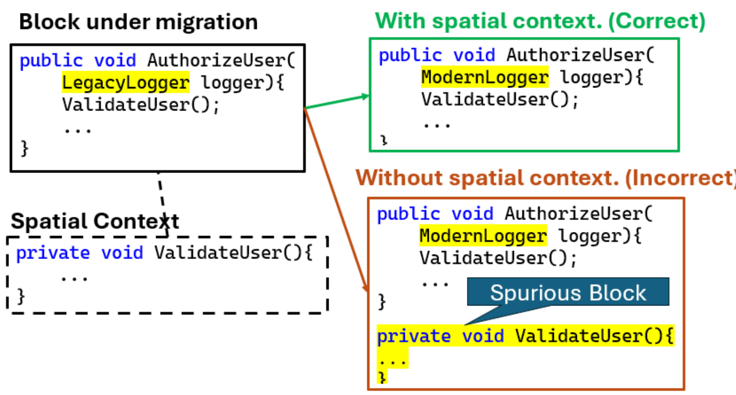


Figure 9: Illustration of importance of spatial context. Spurious blocks, highlighted in yellow are the results of missing missing spatial context.

Furthermore, Figure 8 provides another instance where the absence of temporal context impacts the code modification process. In this case, a "Context" parameter needs to be added to the "CreateService()" call within the "Startup()" method. However, since the LLM lacks temporal context, it is unaware of the signature change to "CreateService()" and, consequently, fails to recognize the need for updates to all the callers. This omission results in numerous missed updates throughout the codebase.

It's crucial to highlight another significant observation: the increase in the count of spurious blocks when spatial context is insufficient. This phenomenon occurs because, in the absence of adequate spatial context, the Large Language Model (LLM) may incorrectly perceive missing code elements and attempt to create them, leading to the generation of spurious code blocks.

An illustrative example in Figure 9 demonstrates this issue. In this scenario, the task is to modify the "AuthorizeUser()" method by migrating the logging calls from an old logging framework to a new one. However, due to the lack of spatial context that would specify the existence of the "GetUserSubscription()" method and the "CurrentUser" property, the LLM attempts to create these elements as well. Consequently, not only is the logging migration addressed, but the LLM also introduces unnecessary code blocks, such as the creation of the "GetUserSubscription()" method and the addition of "CurrentUser" as a class-level object.

This observation underscores the critical role of spatial context in guiding the LLM's understanding of code structure and relationships. Providing comprehensive spatial context can help prevent the generation of superfluous code blocks and ensure that code modifications are precise and aligned with the intended changes.

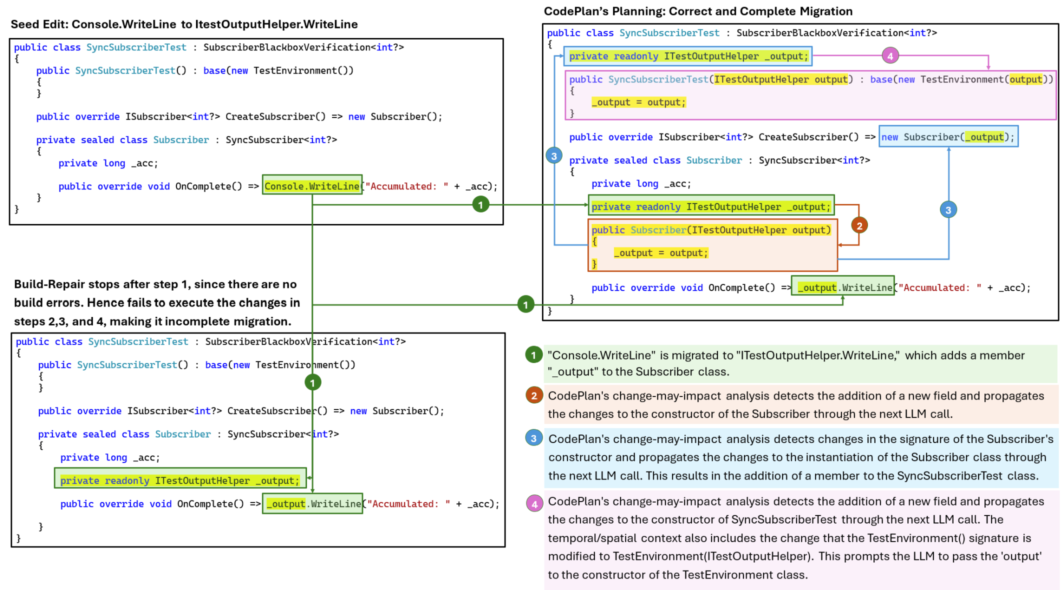


Figure 10: Illustration of the CodePlan's plan execution.

In summary, the experimental results emphasize the essential nature of temporal and spatial contexts in CodePlan's planning. The increase in missed and spurious updates due to the absence of temporal and spatial contexts underscores the significance of providing the LLM with a comprehensive understanding of code evolution and dependencies through these contexts to ensure accurate and effective code modifications.

B.1.2 RQ3: What are the key differentiators that allow CodePlan to outperform baselines in solving complex coding tasks?

CodePlan*'s Strategic Planning and Context Awareness:*.

CodePlan's performance in handling complex coding tasks can be attributed to its its incremental analysis and change-may-impact analysis. These capabilities set it apart from baseline methods like Build-Repair, which primarily focus on maintaining syntactic correctness while overlooking critical contextual details. To illustrate this, let's delve into an example from repository E1 illustrated in Figure 10, where CodePlan is tasked with migrating the `Console.WriteLine` method to `ITestOutputHelper.WriteLine`. This migration involves a series of changes 1 to 4 as described in the Figure 10. These cascading changes start from introducing `ITestOutputHelper _output` as a class-level member, accomplished via LLM updates.

CodePlan's change-may-impact analysis proves useful in this scenario. It recognizes that the addition of a new field necessitates modifications to the constructor to ensure proper initialization. As a result, CodePlan schedules the necessary constructor modification. Consequently, the constructor `Subscriber(...)` is correctly updated to accept `ITestOutputHelper` as a parameter and initialize the class member `_output`. This in turn results in a series of changes through the repository as explained in steps 1 to 4 in the Figure 10.

This example demonstrates how CodePlan makes methodical and contextually-aware changes to the repository, thanks to its ability to do change impact analysis and incorporate temporal contexts. In contrast, Build-Repair, reliant solely on syntactic correctness, fails to even detect the need for modification in the Subscriber's constructor. Given that all syntactic rules are adhered to, it does not prompt a build error and consequently fails to implement changes in steps 2 to 4, as illustrated in Figure 4. Instead, it solely executes the modification outlined in step 1, resulting in incomplete code updates.

CodePlan's advantage lies in its holistic understanding of code relationships and its planning, which ensures the integrity and functionality of the codebase are maintained throughout complex coding tasks. This qualitative analysis highlights how CodePlan's approach outperforms baselines in handling intricate coding challenges.

Incremental Analysis: Maintaining Relationships with Dependency Graph:.

CodePlan's performance in tackling complex coding tasks is attributed to its incremental analysis, which effectively links edits with the underlying dependency graph. Unlike a static snapshot of code, which may result in an incomplete representation of dependencies, our incremental analysis method ensures that relationships within the dependency graph are maintained until the affected blocks are modified.

Consider a scenario where a caller function undergoes a renaming process. Traditional static snapshots would struggle to preserve the caller-callee relationship because, in their view, the caller has already been renamed. However, CodePlan's incremental analysis steps in, preserving the caller-callee relation until the caller function itself undergoes an update. This dynamic approach ensures that critical relationships aren't prematurely severed, allowing for more accurate and context-aware code modifications.

Another instance of CodePlan's lies in handling modifications to import statements. Suppose an import statement originally reads as `import numpy`, and it's modified to `import numpy as np`. In a static snapshot, this alteration could result in the loss of the "ImportedBy" relationship. However, CodePlan's incremental analysis ensures that such vital relationships are maintained, facilitating precise and comprehensive code updates.

Incremental Analysis: Enhanced Spatial and Temporal Context Extraction:.

CodePlan's success in complex coding tasks can be attributed to its abiltity to extract spatial context more accurately, thanks to incremental analysis. Attempting to extract spatial context without the support of incremental analysis often leads to a loss of accuracy and completeness.

Consider a scenario where a method within the codebase constructs an object of a class, let's say "A." However, at some point in the code's history, "A" was renamed to "B." Traditional methods that lack incremental analysis may struggle with this situation. When attempting to extract the class definition, they may encounter a roadblock because, in the current static snapshot, "A" no longer exists.

However, CodePlan's incremental analysis comes to the rescue by establishing the crucial link between the historical context and the present state. It accurately extracts the class definition, recognizing that the object is now of class "B" due to the earlier temporal edit (the renaming of "A" to "B"). This holistic approach ensures that spatial context extraction is both precise and comprehensive, allowing CodePlan to make informed and context-aware code modifications.

Change-may-impact analysis propagates subtle behavioral changes..

One of the key factors differentiating CodePlan's performance in complex coding tasks is its ability to detect subtle behavioral changes through extensive change-may-impact analysis. While certain code edits, like modifying method signatures, result in obvious breaking changes that can be detected by build tools, others induce more nuanced behavioral shifts without directly breaking the build. These subtle alterations, often overlooked, can significantly affect code correctness and functionality. For instance, a seemingly minor change in a method's return value, from True to False, may invalidate assertions in unit tests.

CodePlan is able to identify such behavioral transformations that may elude oracles such as build or static checking tools. Its thorough change-may-impact analysis delves beyond surface-level modifications, proactively recognizing these inconspicuous shifts. This capability sets CodePlan apart from baseline methods, which primarily focus on changes related to build success. Consequently, CodePlan emerges as a powerful solution for addressing complex coding tasks, ensuring that even the most subtle alterations are meticulously considered, ultimately enhancing code quality.

Change may-impact analysis maintains cause-effect relationship.. One of CodePlan's differentiators lies in its proficiency in preserving the cause-effect relationship when handling complex coding tasks. Traditional build tools are effective at pinpointing breaking changes but often fall short in identifying the underlying causes and their corresponding effects. For instance, if a method signature is altered within an overridden method, a typical build tool would flag the issue at the overridden method's location, where the error is observed. However, this approach fails to recognize the underlying cause—the change in the method signature, which should ideally lead to an update in the corresponding virtual method in the base class.

In contrast, CodePlan's change-may-impact analysis excels in maintaining the causal link between code modifications. When a breaking change is introduced, CodePlan not only identifies the error but also traces it back to the root cause, establishing the need for subsequent changes. In the aforementioned example, CodePlan recognizes that the change in the overridden method's signature necessitates an update to the corresponding virtual method in the base class. This meticulous preservation of cause and effect sets CodePlan apart from baseline methods, which often treat issues in isolation without considering the broader context.