CodePlan: Repository-level Coding using LLMs and Planning

RAMAKRISHNA BAIRI, Microsoft Research, India
ATHARV SONWANE, Microsoft Research, India
ADITYA KANADE, Microsoft Research, India
VAGEESH D C, Microsoft Research, India
ARUN IYER, Microsoft Research, India
SURESH PARTHASARATHY, Microsoft Research, India
SRIRAM RAJAMANI, Microsoft Research, India
B. ASHOK, Microsoft Research, India
SHASHANK SHET, Microsoft Research, India

Software engineering activities such as package migration, fixing errors reports from static analysis or testing, and adding type annotations or other specifications to a codebase, involve pervasively editing the entire repository of code. We formulate these activities as *repository-level coding* tasks.

Recent tools like GitHub Copilot, which are powered by Large Language Models (LLMs), have succeeded in offering high-quality solutions to localized coding problems. Repository-level coding tasks are more involved and cannot be solved directly using LLMs, since code within a repository is inter-dependent and the entire repository may be too large to fit into the prompt. We frame repository-level coding as a planning problem and present a task-agnostic framework, called CodePlan to solve it. CodePlan synthesizes a multi-step *chain of edits* (plan), where each step results in a call to an LLM on a code location with context derived from the entire repository, previous code changes and task-specific instructions. CodePlan is based on a novel combination of an incremental dependency analysis, a change may-impact analysis and an adaptive planning algorithm.

We evaluate the effectiveness of CodePlan on two repository-level tasks: package migration (C#) and temporal code edits (Python). Each task is evaluated on multiple code repositories, each of which requires inter-dependent changes to many files (between 2–97 files). Coding tasks of this level of complexity have not been automated using LLMs before. Our results show that CodePlan has better match with the ground truth compared to baselines. CodePlan is able to get 5/6 repositories to pass the validity checks (e.g., to build without errors and make correct code edits) whereas the baselines (without planning but with the same type of contextual information as CodePlan) cannot get any of the repositories to pass them. We will release our data and evaluation scripts at https://aka.ms/CodePlan.

1 INTRODUCTION

The remarkable generative abilities of Large Language Models (LLMs) [24, 28, 30, 35, 57, 73] have opened new ways to automate coding tasks. Tools built on LLMs, such as Amazon Code Whisperer [14], GitHub Copilot [38] and Replit [66], 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 [78]. 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 coding in the editor and 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 our code repository uses a library L, and the API of library L changes from version v_n to version v_{n+1}, we need to migrate our code repository to correctly invoke the revised version. Such a migration task involves making edits not only to all the regions of repository that make calls to the relevant APIs in library L, but

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet

We use a Complex Numbers library that had the following edit -

```
+ 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)
```

Modify the code repository in accordance with this change.

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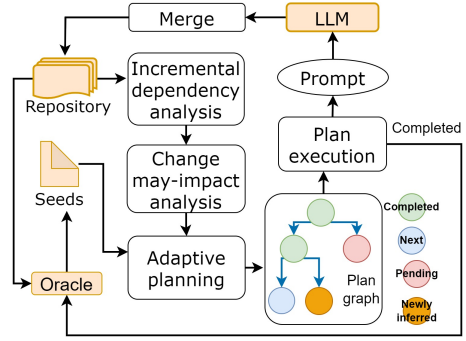

Fig. 2. Overview of CodePlan.

```
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)

Fig. 3. Relevant code snippets from our repository.

also to regions of the repository (across file boundaries) having transitive syntactic and semantic dependencies on the updated code.

This is illustrated in Figure 1, which shows a change in the API for a Complex Numbers library. Our task is to migrate our code repository in accordance with this change. The left side of Figure 3 shows relevant parts of our code repository that use the Complex Numbers library. Specifically, the file `Create.cs` has the method `func`, which invokes the `create_complex` method from the library, and `Process.cs` has the method `process` which invokes `func`.

We can pass the task description from Figure 1 and the body of `func` to an LLM to generate the revised code for `func` as shown in the right side of Figure 3. As seen, the LLM has correctly edited the invocation to the `create_complex` API so that it returns an object of type `Complex` instead of a tuple of two floating point values. Note that this edit has resulted in a change to the signature of the method `func` – it now returns an object of type `Complex`. This necessitates changes to callers of method `func` such as the `process` method in file `Process.cs`, shown in the left-bottom of Figure 3. Without a suitable change to the body of the `process` method, our code does not build! A suitable

change to the `process` method which gets the repository to a consistent state, so that it builds without errors, is shown in the bottom-right of Figure 3.

Problem Formulation. The migration task above 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. Each of these tasks involves a set of *seed specifications* such as the one shown in Figure 1, which are starting points for the code editing task. These seed specifications typically trigger other editing requirements on code, and such requirements need to be propagated across dependencies in the code repository to perform other edits across the repository to complete the coding task. Typically, such propagation of edits across dependencies is done manually.

Our goal is to construct a repository-level coding system, which automatically generates *derived specifications* for edits such as one required for the `process` method in Figure 3, in order to get the repository to a *valid* state. Here, validity is defined with respect to an oracle, which can be instantiated to various ways of enforcing repository-level correctness conditions such as building without errors, passing static analysis, passing a type system or a set of tests, or passing a verification tool. 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 $\Theta(R_{start})$ = 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 $\Theta(R_{target})$ = True.

Proposed Solution. In this paper, we propose a method to compute derived specifications by framing (LLM-driven) repository-level coding as a *planning problem*. Automated planning [37, 67] aims to solve multi-step problems, where each step executes one action among many alternatives towards reaching a target state. It is used in a wide range of areas such as motion planning [47], autonomous driving [39], robotics [44] and theorem proving [26].

We present a task-agnostic framework, called CodePlan, which synthesizes a multi-step plan to solve the repository-level coding task. As shown in Figure 2, the input to CodePlan is a repository, a task with seed specifications expressed through a natural language instruction or a set of initial code edits, a correctness oracle and an LLM. CodePlan constructs a *plan graph* where each node in the graph identifies a code edit obligation that the LLM needs to discharge and an edge indicates that the target node needs to be discharged consequent to the source node. CodePlan monitors the code edits and adaptively extends the plan graph. The edits Δ_{seeds} follow from the task description, whereas the edits $\Delta_{derived}$ are identified and contextualized based on a novel combination of an incremental dependency analysis, a change may-impact analysis and an adaptive planning algorithm. The merge block merges the code generated by the LLM into the repository. Once all the steps in a plan are completed, the repository is analyzed by the oracle. The task is completed if the oracle validates the repository. If it finds errors, the error reports are used as seed specifications for the next round of plan generation and execution.

Consider again, the example API migration task specified in Figure 1 on code in Figure 3. CodePlan performs the edit of the method `func` using the instruction in Figure 1 as a seed specification. By analyzing the code change between Figure 3(a)–(b), it classifies the change as an *escaping change* as 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

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet

dependencies to infer a derived specification to edit the method `process`, which invokes `func`. Both the seed and 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 themselves necessitate other changes transitively and CodePlan handles such cases.

A simpler alternative to our planning is to use the oracle to infer derived specifications. For example, the build system can find the error in the `process` method after the seed change is made in Figure 3. This has important limitations. First, not all changes induce build errors even though they result in behavioral changes, e.g., changing the return value from `True` to `False` without changing the return type. Second, the build system is agnostic to cause-effect relationship when code breaks. For example, if the signature of an overriding method is changed as per the seed specification then a similar change is needed in the corresponding virtual method. However, the build system (when run on the intermediate, inconsistent snapshot of the repository) blames the overriding method for not conforming to the virtual method. Naïvely trying to fix the build error would end up reverting the seed change. The static analysis and planning components of CodePlan overcome these limitations. We experimentally compare CodePlan against a baseline that uses a build system to iteratively identify breaking changes and uses an LLM to fix them. Our quantitative and qualitative results show that CodePlan is superior to this kind of oracle-guided repair technique.

Contributions. To the best of our knowledge, the problem of monitoring the effects of code edits made by an LLM to a repository and systematically planning *a chain of inter-dependent edits* has not been identified and solved before.

In the space of repository-level coding tasks, two types of contexts have been found to be useful for prompting LLMs: (1) *spatial context* to provide cross-file information to the model using static analysis [9, 34, 51, 59, 61, 70, 71, 77] or retrieval [81, 85], and (2) *temporal context* to condition the predictions on the history of edits to the repository [23, 40, 64, 76]. Since CodePlan monitors the code changes and maintains a repository-wide dependency graph, we provide both these forms of contexts in a unified framework. The existing techniques assume that the next edit location is provided by the developer and do not account for the effect of an edit on the dependent code. In contrast, by inferring the impact of each change, CodePlan propagates the changes to dependent code, paving a way to automate repository-level coding tasks through chain of edits.

In summary, we make the following contributions in this paper:

(1) We are the first to formalize the problem of automating repository-level coding tasks using LLMs, which requires analyzing the effects of code changes and propagating them across the repository. There are currently no systematic and scalable solutions to this problem.

(2) We frame repository-level coding as a planning problem and design a task-agnostic 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: package migration for C# repositories and temporal code edits for Python repositories. We compare against baselines that use the oracles (a build system for C# and a static type checker for Python) for identifying derived edit specifications (in contrast to planning used in CodePlan). We use the same contextualization method as CodePlan in the baselines.

(4) Our results show that CodePlan has better match with the ground truth compared to baselines. CodePlan is able to get 5/6 repositories to pass the validity checks, whereas the baselines cannot get any of the repositories to pass them. Except for the 2 proprietary repositories, we will release our data and evaluation scripts at https://aka.ms/CodePlan.

2 DESIGN

In this section, we first give an overview of the CodePlan algorithm for automating repository-level coding tasks (Section 2.1). We then present the static analysis (Section 2.2) and the adaptive planning and plan execution (Section 2.3) components of CodePlan.

2.1 The CodePlan Algorithm

```
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, I) 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)
```

Algorithm 1: The CodePlan algorithm to automate repository-level coding tasks. The data structures and functions in Cyan and Orchid are explained in Section 2.2– 2.3 respectively.

The CodePlan algorithm (Algorithm 1) takes four inputs: (1) the source code of a repository R, (2) a set of seed edit specifications for the task in hand, Δ_{seeds}, (3) an oracle, Θ, and (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 4 illustrates the dependency graph structure. We will discuss it in details in Section 2.2.1. 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 round. 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 with an edit specification for a block B with edit instruction I involves the following five steps:

(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** for the edit 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).
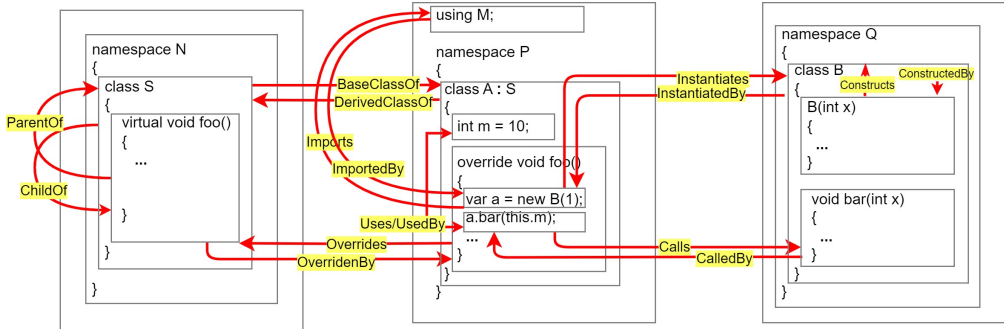
Fig. 4. Illustration of the dependency graph annotated with relations as the edge labels.

2.2 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 1.

2.2.1 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 [12] 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 4 illustrates all the relations we track as labeled edges. 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 [32] for deriving the semantic relations.

`ClassifyChanges`. As discussed in Section 2.1, 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 1 (the first and second columns) gives the types of atomic changes and

Atomic Change	Label	Dependency Graph Update	Change May-Impact Analysis
Modification Changes			
Body of method M	MMB	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	MMS	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	MF	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	MC	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	MCC	No change.	Rel(D, C, InstantiatedBy), Rel(D, C, BaseClassOf), Rel(D, C, DerivedClassOf)
Import/Using statement I	MI	Recompute the edges incident on the import statement.	Rel(D, I, ImportedBy)
Addition Changes			
Method M in class C	AM	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	AF	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	AC	Add new node and edges by analyzing the class declaration.	Nil
Constructor of class C	ACC	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	AI	Add new node and edges by analyzing the import statement.	Nil
Deletion Changes			
Method M in class C	DM	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	DF	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	DC	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	DCC	Remove the edges incident on the class due to object instatiations using the constructor.	Rel(D, C, InstantiatedBy), Rel(D, C, BaseClassOf), Rel(D, C, DerivedClassOf)

their labels. 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 1 (the third 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 1. All other deletion changes require removing nodes and edges as stated in the table.

2.2.2 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 1.

`GetAffectedBlocks`. The last column in Table 1 tells us how to identify blocks affected by a code change for each type of change. When the body of a method M is edited, we perform escape analysis [22, 29] 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 there are no affected blocks. If the signature of a method is edited, the callers and methods related to it through method-override relation in the inheritance hierarchy are affected. The signature change itself can affect the Overrides and OverridenBy relations, 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 1 (the row with MMS label). When a field F of a class C is modified, the 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 1 to be adequate. However, CodePlan can be easily configured to accommodate variations of the rules in Table 1 if necessary.

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet

2.3 Adaptive Planning and Plan Execution

We now discuss the data structures and functions from Algorithm 1 in the Orchid background.

2.3.1 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 2.1, 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 1 results in creation of the target obligation.

`ExtractCodeFragment`. As discussed in the first step in Section 2.1, 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. As stated in Section 1, 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, representing code as nodes and their relationships as edges. This graph enables CodePlan to navigate codebases, identify relevant code blocks, and maintain awareness of their spatial context. As a result, when generating code edits, the dependency graph empowers 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.

`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).

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

`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 **p₁** followed by listing the edits made in the repository so far **p₂** that are relevant to the fragment being edited. The next section **p₃** notes how each of the

fragments present in (p₂) are related to the fragment to be edited. This is followed by the spatial context (p₄) and the fragment to the edited (p₅).

Prompt Template

(p₁) Task Instructions: *Your task is to . . .*

(p₂) Earlier Code Changes (Temporal Context): *These are edits that have been made in the code-base previously -*

```
Edit 1:
  Before: «code_before»
  After: «code_after»
...
```

(p₃) Causes for Change: *The change is required due to -*

```
«code_to_be_edited» is related to «code_changed_earlier» by «cause»
 ...
```

(p₄) Related Code (Spatial Context): *The following code maybe related -*

```
«related_code_block-1»
 ...
```

(p₅) Code to be Changed Next: *The existing code is given below -*

```
«code_to_be_edited»
```

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 and no new nodes are added, an *iteration* of repository-level code edits is completed. As shown in Figure 2, the oracle is invoked on the repository. If the oracle flags any errors (e.g., build errors), the error locations and diagnostic messages are added as seed changes for the next iteration and the adaptive planning resumes once again. If the oracle does not flag any errors, CodePlan terminates.

3 IMPLEMENTATION

In this section, we provide a detailed overview of the implementation components that constitute the core of our method.

Dependency Graph Construction. At the core of the CodePlan methodology lies the Dependency Graph, which is instrumental in representing the intricate relationships between code blocks. To build this Dependency Graph from a code repository, we adopt a systematic approach. Initially, we

Code Snippet

```
1  public class SyncSubscriberTest
2  {
3      public SyncSubscriberTest(ITestOutputHelper output) : base(new TestEnvironment(output))
4      {
5          base.Initialize();
6      }
7  }
```

AST from tree-sitter

```
1   compilation_unit [0, 0] - [7, 0]
2     class_declaration [0, 0] - [6, 1]
3       modifier [0, 0] - [0, 6]
4       name: identifier [0, 13] - [0, 31]
5       body: declaration_list [1, 0] - [6, 1]
6         constructor_declaration [2, 1] - [5, 2]
7           modifier [2, 1] - [2, 7]
8           name: identifier [2, 8] - [2, 26]
9           parameters: parameter_list [2, 26] - [2, 52]
10            parameter [2, 27] - [2, 51]
11              type: identifier [2, 27] - [2, 44]
12              name: identifier [2, 45] - [2, 51]
13          constructor_initializer [2, 53] - [2, 88]
14            argument_list [2, 59] - [2, 88]
15              argument [2, 60] - [2, 87]
16                object_creation_expression [2, 60] - [2, 87]
17                  type: identifier [2, 64] - [2, 79]
18                  arguments: argument_list [2, 79] - [2, 87]
19                    argument [2, 80] - [2, 86]
20                      identifier [2, 80] - [2, 86]
21          body: block [3, 1] - [5, 2]
22            expression_statement [4, 2] - [4, 20]
23              invocation_expression [4, 2] - [4, 19]
24                function: member_access_expression [4, 2] - [4, 17]
25                  expression: base_expression [4, 2] - [4, 6]
26                  name: identifier [4, 7] - [4, 17]
27                arguments: argument_list [4, 17] - [4, 19]
```

Fig. 5. AST structure for a C# code snippet produced by tree-sitter.

parse all the code files within the repository, utilizing the tree-sitter library [25] to generate an AST-like structure. This structured representation simplifies the identification of various fundamental code blocks within the codebase. For instance, Figure 5 exemplifies an AST structure for a C# code snippet produced by tree-sitter. Code blocks are identified at different levels, including Classes, Methods, import statements, and non-class expressions. For instance, in Figure 5, the subtree rooted at the class_declaration node corresponds to the SyncSubscriberTest class.

Relation Identification in C#. In the context of C# repositories, the establishment of edges within the Dependency Graph involves the careful tracing of relationships within the AST. We have devised custom logic for each type of relationship outlined in Figure 4, encompassing vital connections such as caller-callee, overrides-overridden, base class-derived class, and others. To illustrate, for the Caller/Callee relationship, we search for invocation_expression nodes within the AST. Subsequently, we process the sub-tree beneath these nodes to resolve essential details such as the target class and the invoked method's name. Armed with this information, we create Calls/CalledBy relation links between the code block initiating the method call and the corresponding method block within the target class. While we have implemented custom logic for these relations, it's important to note that alternative dependency analysis tools for C# such as Language Servers for C# (LSP) [5], CodeQL [2], or similar solutions can also be integrated into our system, owing to its inherent flexibility.

Relation Identification in Python. For Python repositories, we use Jedi [4] - a static analysis tool which discovers references and declarations of symbols throughout the codebase. These capabilities

are harnessed to identify edges in the Dependency Graph for relationships such caller-callee, overrides-overridden, and base class-derived class.

Integration of GPT-4 for Code Edits. CodePlan leverages the remarkable capabilities of GPT-4 [57] to perform code edits effectively. During the construction of input data for the edit model, we meticulously provide temporal context, spatial context, and the actual code to be edited in the form of code snippets. These code snippets represent classes or methods that contain the edit site and are meticulously structured in a sketched representation, as stated in Section 2.1. This sketched representation ensures that the model is enriched with a substantial context for each edit site, significantly enhancing the quality and accuracy of the edits it generates.

Language Extensibility. While our current implementation proficiently supports C# and Python repositories, extending support to repositories in other programming languages is a straightforward endeavor. It primarily entails creating a dependency graph with the relations identified in Figure 4 and incorporating it into the CodePlan framework, thereby allowing for seamless adaptation to a diverse array of programming languages.

4 EXPERIMENTAL DESIGN

In this section, we'll explain how we conducted our experiments to test CodePlan. We'll start by talking about the different sets of data we used. Then, we'll discuss the methods we compared CodePlan to, which are like our reference points. Finally, we'll explain how we measured the results to see how well CodePlan performed compared to the other methods.

4.1 Datasets

In our experiments, we utilized diverse datasets representing a wide spectrum of code repositories with varying complexities and sizes. These datasets allowed us to thoroughly evaluate the performance of CodePlan in different real-world scenarios.

Internal Repositories (Int-1 and Int-2). These repositories are proprietary and belong to a large product company. They are characterized by their large size, complex patterns, and are typical of production-level codebases. The primary task we focused on here was the migration of these repositories from a legacy logging framework to a modern logging framework. This migration involved non-trivial changes, including creating service-specific loggers using a logging factory, passing the logger through call chains, managing class hierarchies, storing logger references at different scopes (class, method, etc.), and handling loggers at both static and non-static classes/methods. These two production repositories, Int-1 and Int-2, were chosen for their distinct coding styles and design patterns, providing a comprehensive internal dataset for our evaluation.

External Repositories (Public GitHub). We also considered external repositories from GitHub to diversify our dataset. These repositories were chosen to represent two different coding tasks: Migration and Temporal Edits (discussed next).

Migration Task. This task involves migrating APIs or addressing breaking changes within a codebase. Examples include updating dependencies, adapting to changes in external libraries, or aligning code with new coding standards. It is characterized by its complexity, often requiring consistent updates across numerous code files and dependencies. To select these repositories, we looked for those containing commits and pull requests related to various kinds of migrations (API, frameworks, etc.). We filtered for repositories with at least 50 files.

Temporal Edits Task. This task involves orchestrating a sequence of code changes given some initial code-change. Many code-changes can be characterised as temporal edits including refactoring

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet

Repositories	Migration (C#)			Temporal Edits (Python)		
	Int-1	Int-2	Ext-1	T-1	T-2	T-3
Number of files	91	168	55	21	137	4
Lines of code	8853	16476	8868	3883	20413	1874
Number of files changed	47	97	21	2	2	3
Number of seed changes	41	63	42	2	1	1
Number of derived changes	110	375	16 0	8	3	10
Size of diff (lines)	1744	4902	1024	104	15	39
Size of seed edits (lines)	242	242	379	76	4	1
Prompt template size (lines)	81	81	81	75	75	75
URL	-	-	[7]	[8]	[1]	[3]

Table 2. Dataset statistics. Int-1,2 are internal (proprietary) repositories, Ext-1 and T-1,2,3 are external (public GitHub) repositories.

or addition/removal of functionality. A temporal edit task is specified by a set of initial edits (usually made by the user) along with derived edits precipitated by the seed edits. The task for the tool is to infer the derived edits from the set of initial edits. An example temporal edit task can be where the initial edit is adding an argument to a method and the derived edits are changes to all the places where this method is called. We identify temporal edit tasks from commits made in public repositories on GitHub. We consider python repositories with permissive licenses, sort by stars, filter out documentation/tutorial related repositories and from amongst candidates with at least 10,000 stars, we select commits made after 1st November 2021 (after the cut-off date for training data of GPT-4) containing multiple related changes.

Source/Target/Predicted Repositories. To collect the code changes for the migration and temporal edit tasks, we obtained files from GitHub before and after the commits. We refer to these as the Source repository (before commit) and Target repository (after commit). Analyzing these changes allowed us to identify seed changes through manual inspection. For instance, in the NUnit to XUnit migration, one of the seed edits involved replacing `Console.WriteLine` with writing to an `ITestOutputHelper` object. With the Source repository and seed change instructions, CodePlan was tasked with making the necessary changes to the Source repository, resulting in what we call the Predicted repository. If CodePlan successfully executed all changes, the Predicted repository should match the Target repository, providing a robust evaluation of its capabilities.

Preprocessing Source and Target Repositories. When dealing with large repositories, it's common for multiple developers to contribute code, resulting in various coding styles driven by individual preferences. For instance, one developer might consistently use the `this` qualifier to reference class members, while another may not. When CodePlan executes changes through LLM prompting, it tends to establish a uniform style throughout the codebase, which may involve enforcing the consistent use of the `this` qualifier, among other things. While these changes ensure functional equivalence, they can impact evaluation metrics when comparing the Predicted repository with the Target repository. To address these issues, we employ a manual preprocessing step on both the Source and Target repositories. This preprocessing aims to establish uniformity across various files within the repository. By doing so, we provide a fair basis for comparing CodePlan's performance against the ground truth in these diverse codebases.

Table 2 summarizes the dataset statistics for both Migration (C#) and Temporal Edits (Python) repositories, including the names of repositories (both internal and external) and various key statistics including their size, code changes, and other relevant metrics:

- *Number of Files*: The total count of files in each repository.
- *Lines of Code*: The cumulative number of lines of code across all files.
- *Number of Files Changed*: The count of files that have undergone changes between the source and target repositories.
- *Number of Seed Changes*: The number of initial edits, often considered as the starting point of code changes.
- *Number of Derived Changes*: The count of subsequent edits that follow the initial seed changes.
- *Size of Diff*: The number of lines that differ between the source and target repositories.
- *Size of Seed Edits*: When the seed edits are made directly on the code, it represents the number of lines of initial edits. When the seed edits are made through LLM instruction, it denotes the size of the instruction text.
- *Prompt Template Size*: This number represents the size of the LLM prompt template used by CodePlan. The same template is employed for all the migration repository tasks, and another similar template is utilized for all the Temporal Edit repository tasks.

These metrics not only offer a comprehensive overview of the dataset characteristics but also highlight the considerable advantages of using CodePlan over manual processes, especially for large repositories. In manual scenarios, human effort is required to painstakingly identify dependent changes and implement each modification. Notably, metrics such as "Size of Diff" and "Size of Seed Edits" provide insights into the developer effort involved. CodePlan, on the other hand, automates these changes, effectively reducing the burden on developers. Furthermore, it's worth noting that the effort required to craft LLM instructions for CodePlan is significantly less than the extensive manual labor required for making all the code changes. These metrics collectively demonstrate the efficiency and effectiveness of CodePlan across diverse codebases, emphasizing its potential to streamline development workflows and save valuable developer time.

4.2 Oracles and Baselines

Oracles. Recall that our definition of repository-level coding tasks is centered around satisfying an oracle that can determine the validity of our solution. In our experiments, we consider two specific instatiations of such an oracle. For the C# migration tasks, we define the oracle as passing the C# build tools without any errors. For temporal edits scenario, we use Pyright [6], a static checker for python as the oracle.

Oracle-Guided Repair. Both of these oracles take a codebase as input and can output a list of errors in that codebase. This leads naturally to the formulation of baseline approaches to our tasks which we term as Oracle-Guided Repair. These are simple reactive approaches where at each step we attempt to rectify the errors flagged by the oracle. For the C# migration scenario, the baseline is Build-Repair and for temporal edits it is Pyright-Repair according to the oracles used in both.

The oracle-guide repair works in the following steps:

(1) *Initial Edit*: The process begins with application of the initial seed edit to the codebase.
(2) *Build and Error Detection*: After the seed edits, we invoke the oracle which detects errors in the codebase arising from the seed edits.
(3) *Error Message Analysis*: The error messages generated by the oracle is then parsed to precisely identify the location of the error within the code.

(4) *LLM Patching*: Subsequently, the error message, along with the code fragment from the flagged location, is passed to an LLM. The LLM leverages its code generation capabilities to generate a patch or fix for the identified error. For fair comparison with CodePlan, we use our implementation for spatial and temporal context extraction in oracle-guided repair. That is, the key difference between CodePlan and oracle-guided repair is that CodePlan uses adaptive planning whereas oracle-guided repair uses the diagnostic information generated by the oracle, but they use the same contextualization machinery.

Note that Oracle-Guided Repair approaches are reactive and lack a comprehensive "change may-impact analysis." This means that they may not thoroughly assess how the proposed code changes could affect other parts of the codebase. As a result, the fixes generated by such approaches may be incomplete or incorrect, especially when dealing with complex coding tasks.

Alternate Edit Model: Coeditor [76]. CodePlan by default leverages the text and code processing abilities to LLMs to make local edits to code fragments given proper context. However, in theory, the incremental dependancy analysis, change may-impact analysis and adaptive planning components of CodePlan can be used in conjunction with any tool or model that can make localized edits to code following some provided intent. Coeditor [76] is a transformer based model fine-tuned to edit code snippets while taking into account prior edits made in the repository. Such a model is good fit for the Temporal Edits task, where we need to make a sequence of edits from a set of seed edits, where each edit is dependant on some subset of the previous edits. Indeed, Coeditor is evaluated on the Temporal Edits task in [76]. In order to demonstrate the generality of our analysis and planning, we evaluate how our approach performs in the temporal edits scenario, when replacing `gpt-4-32k` with Coeditor as the edit model.

4.3   Evaluation Metrics

The evaluation metrics employed in our study are aimed at assessing how effectively CodePlan (or a baseline) propagates changes across the entire code repository and the correctness of each of these changes. To achieve this, we rely on two key metrics: Block Metrics and Edit Metrics.

Block Metrics. Block Metrics help us understand CodePlan's ability to accurately identify code blocks in need of modification. These metrics include:

- *Matched Blocks*: These are code blocks that exist in the Source Repository, have been edited in the Target Repository, and have also been edited in the Predicted Repository. Essentially, these are blocks that CodePlan successfully identifies for change.
- *Missed Blocks*: Missed Blocks refer to code blocks present in the Source Repository that have been edited in the Target Repository but were not edited in the Predicted Repository. In other words, these are blocks that CodePlan failed to modify when it should have.
- *Spurious Blocks*: Spurious Blocks are code blocks found in the Source Repository that were not edited in the Target Repository but were incorrectly edited by CodePlan in the Predicted Repository. These represent edits that CodePlan made unnecessarily.

The ideal outcome is to have a high number of Matched Blocks and low numbers of Missed and Spurious Blocks.

Edit Metrics. While Block Metrics assess the identification of code blocks, Edit Metrics delve into the correctness of the modifications made by CodePlan. These metrics include:

- *Levenshtein Distance*: Levenshtein Distance measures the edit distance at the file level between the Predicted Repository and the Target Repository. It calculates how many changes

were made to transform one file into the other. A higher Levenshtein Distance indicates that CodePlan did not make the correct changes to the repository.

- *Diff BLEU*: Typically, we use BLEU [58], a common metric in Natural Language Processing, to measure text similarity. However, when applied in our tasks, BLEU can produce overly high similarity scores because code edits for specific tasks often involve only a small portion of the file. To address this issue, we compute Diff BLEU: a modified version of the BLUE score, denoted as `BLUE(DIFF(Source repo file, Target repo file), DIFF(Source repo file, Predicted repo file))`. Here, `DIFF` calculates the differences (diff hunks) between two files. What sets Diff BLUE apart is its focus on comparing the modified sections of code between the Predicted and Target Repositories while disregarding common code. When the modifications in Predicted and Target Repositories match precisely, Diff BLUE yields a score of 1.0, indicating a high level of correctness and alignment in handling code modifications.

In summary, these evaluation metrics provide a comprehensive assessment of CodePlan's performance, both in terms of identifying code blocks for modification and ensuring that the modifications made are correct.

5 RESULTS AND ANALYSIS

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

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

RQ2: How important are the temporal and spatial contexts for 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 CodePlan is able to localize and make the required changes to automate repository-level coding tasks?

Motivation. The research question addressing the effectiveness of the CodePlan framework in automating repository-level coding tasks is of paramount importance in the context of modern software engineering. Several key motivations drive the significance of this inquiry:

- **Complexity of Repository-Level Tasks**: Software engineering activities, such as package migration and temporal code edits, often transcend the scope of local code changes. Repository-level coding tasks involve pervasive modifications across the entire codebase. This level of complexity necessitates novel approaches to ensure efficiency and correctness.
- **Real-world Relevance**: In practice, software repositories frequently encounter the need for large-scale changes. For instance, package migration involves updating dependencies across multiple files and dependencies, while temporal code edits require tracking and managing evolving codebase. These tasks are not only time-consuming but also error-prone when done manually.
- **Evaluation against Baselines**: The assessment of CodePlan against baseline methods is crucial. Baseline methods, such as "Oracle-Guided Repair", are common in software development but may lack efficiency when dealing with repository-level tasks. Evaluating CodePlan against baselines provides a benchmark for measuring its effectiveness and highlights areas where it excels. We also study how our system behaves when using a different edit model by evaluating a combination of Coeditor and CodePlan on the Temporal Edits Task.

- **Large-scale Repositories**: The research considers not only isolated coding problems but evaluates CodePlan's performance on large internal and external repositories. This broad scope ensures that the framework's effectiveness is tested under diverse and challenging real-world scenarios.

Experimental Setup.

To study how well CodePlan is able to localize and make the required changes for repository-level tasks, we evaluate it in the context of the tasks described in 4.1. For both the C# Migration and Temporal Edits tasks, we start with the repository in its *Source* state without any edits having been made. We apply the seed edits on top of the *Source* state at which point CodePlan (or the baseline being evaluated) takes over to complete the the edit across the repository. In the case of C# Migrations, the seed edits themselves are performed using the LLM with a suitable prompt, while for Temporal Edits, we the seed edits for each repository task are stored a-priory and applied as patches. CodePlan performs incremental dependency analysis on the repository after the seed edit, identifies code that may be impacted by it, plans which edit to be made next, queries the LLM with a suitable prompt and merges the result back into the repository. CodePlan does this iteratively until it finds that there are no more sites to be edited.

At times, multiple iterations become necessary due to the inherent variability in Large Language Model (LLM) responses. We initiate the first iteration, known as "Iter 1," to kickstart the code editing process using LLMs. However, occasional inaccuracies in LLM responses may introduce erroneous code changes and subsequent build errors. To address these challenges, the second iteration, referred to as "Iter 2," becomes important. During this phase, CodePlan actively identifies and acknowledges any build errors from the previous iteration, and re-engagement with the LLM to obtain more precise responses for correcting the initial errors.

Alongside CodePlan, we also evaluate a series of baselines on the same repositories. For C# migration we evaluate Build-Repair and for Temporal Edits we evaluate Pyright-Repair, the setup for both of which is presented in 4.2. Pyright-Strict-Repair is a variant in which we use the Pyright tool with strict mode enabled. In all the Repair baselines, we provide the same context (temporal and spatial) as in CodePlan, the only difference being that the localisation of sites to edit is using the oracle. We also evaluate using Coeditor instead of gpt-4-32k as the edit model as described in 4.2. In all Coeditor baselines, contextualisation is performed as in [76], with the localisation of next edit site being done through either CodePlan or using the oracle.

We evaluate all of these approaches for how well they localise the site to be edited using the *Matched*, *Missed* and *Spurious* Blocks metrics and the correctness of the overall modification using *Levenshtein Distance* and *Diff BLEU* as described in 4.3. We also determine whether the state of the repository after the approach is finished executing passes the validity check – that is whether it satisfies the oracle and makes the correct edit according to the ground truth.

Results Discussion. The experimental results in Table 3 demonstrate the effectiveness of the CodePlan framework in automating repository-level coding tasks.

In the context of the C# Migration Task on Internal (Proprietary) Repositories, the results table provides a comprehensive view of the performance of two approaches: CodePlan and Build-Repair. Notably, CodePlan demonstrates superior capabilities in several key aspects. It excels in "Matched Blocks", achieving perfect results with 151 matched blocks for both "Int-1 (Logging)" and 438 matched blocks for "Int-2 (Logging)" datasets. This indicates CodePlan's exceptional precision in accurately identifying and addressing intended code changes. Moreover, CodePlan impressively exhibits zero "Missed Blocks" ensuring that no crucial code modifications are overlooked, thus minimizing the risk of functional issues. Equally noteworthy is the absence of "Spurious Blocks"

Dataset	Approach	Matched Blocks	Missed Blocks	Spurious Blocks	Diff BLEU	Levenshtein Distance	Validity Check
C# Migration Task on Internal (Proprietery) Repositories							
Int-1 (Logging)	CodePlan (Iter 1)	**151**	**0**	0	0.99	60	✗ (4)
	CodePlan (Iter 2)	**151**	**0**	0	1.00	0	✓ (0)
	Build-Repair	82	69	13	0.81	6465	✗ (46)
Int-2 (Logging)	CodePlan (Iter 1)	**438**	**0**	0	0.99	90	✗ (6)
	CodePlan (Iter 2)	**438**	**0**	0	1.00	0	✓ (0)
	Build-Repair	337	101	25	0.66	7496	✗ (68)
C# Migration Task on External (Public) Repositories							
Ext-1 (NUnit-XUnit)	CodePlan (Iter 1)	**58**	**0**	0	1.00	0	✓ (0)
	Build-Repair	52	6	1	0.94	530	✗ (8)
Python Temporal Edit Task on External (Public) Repositories							
T-1	CodePlan (Iter 1)	**8**	**2**	0	**0.90**	**1044**	✗
	Pyright-Repair	5	5	0	0.76	1090	✗
	Pyright-Strict-Repair	**8**	**2**	0	**0.90**	1045	✗
	Coeditor-CodePlan	**8**	**2**	0	**0.90**	1160	✗
	Coeditor-Pyright-Repair	5	5	0	0.66	1205	✗
	Coeditor-Pyright-Strict-Repair	5	5	0	0.65	1139	✗
T-2	CodePlan (Iter 1)	**4**	**0**	0	**0.86**	**248**	✓
	Pyright-Repair	1	3	0	0.00	344	✗
	Pyright-Strict-Repair	1	3	0	0.00	344	✗
	Coeditor-CodePlan (Iter 1)	3	1	0	0.82	254	✗
	Coeditor-Pyright-Repair	1	3	0	0.00	344	✗
	Coeditor-Pyright-Strict-Repair	1	3	0	0.00	344	✗
T-3	CodePlan (Iter 1)	**11**	**0**	0	**0.92**	**358**	✓
	Pyright-Repair	1	10	0	0.00	798	✗
	Pyright-Strict-Repair	1	10	0	0.00	798	✗
	Coeditor-CodePlan (Iter 1)	10	1	0	0.78	717	✗
	Coeditor-Pyright-Repair	1	10	0	0.00	798	✗
	Coeditor-Pyright-Strict-Repair	1	10	0	0.00	798	✗

Table 3. Comparison of CodePlan's repository edit metrics with Build-Repair baseline. Higher values of Matched Blocks, Diff BLEU, and lower values of Missed Blocks, Spurious Blocks, Levenshtein Distances are better.

introduced by CodePlan, signifying its ability to maintain codebase cleanliness and integrity. In contrast, Build-Repair falls behind with 82 matched blocks for "Int-1 (Logging)" and 437 for "Int-2 (Logging)", while also missing a substantial number of blocks (69 and 101, respectively). Additionally, Build-Repair introduces 13 spurious blocks for "Int-1 (Logging)" and 25 for "Int-2 (Logging)", which can increase code complexity and error potential. These findings underscore CodePlan's superiority over Build-Repair, not only in terms of matched blocks but also in its ability to avoid missed and spurious code changes, ultimately offering a more precise and reliable solution for the C# Migration Task on internal repositories.

CodePlan ***vs Build-Repair***. The comparative analysis reveals why Build-Repair falls behind CodePlan. One crucial factor contributing to its performance gap is its reliance on "build error location" as an indicator for code correction. Build errors often pinpoint the location where an error is detected, but they may not necessarily coincide with the actual required fix location. For instance, an error may manifest in a derived class's overridden function signature mismatch, but the fix is necessitated in the base class's virtual function signature, causing Build-Repair to misinterpret the correction site. Additionally, the build process in the context of compiler optimizations may

obscure subsequent errors, only displaying selected errors at a given time. This can lead to an incorrect identification of the correction location and hinder the proper propagation of changes, further exacerbating the performance gap between CodePlan and Build-Repair. These limitations underscore the challenges faced by Build-Repair in accurately pinpointing and addressing code modifications in complex migration tasks.

Multiple Iterations. We can see the necessity for multiple iterations to handle the inherent variability in Large Language Model (LLM) responses as illustrated with the 4 build errors after "Iter 1" of CodePlan in the "Int-1" dataset. To rectify this, "Iter 2" plays a vital role. In this phase, CodePlan identifies and acknowledges the build errors from the previous iteration and re-engages with the LLM to obtain more accurate responses for correcting the initial errors. It is noteworthy that there are no changes to the block metrics between the two iterations since CodePlan correctly identifies the blocks requiring correction and engages the LLM for modification. However, the LLM corrections in "Iter 1" were erroneous, leading to a lower Levenshtein distance metric. This iterative refinement process significantly contributes in mitigating the impact of occasional inaccuracies in LLM outputs during the code editing phases.

Performance on Ext-1. In the comparison between CodePlan and the Build-Repair baseline method on the "Ext-1" dataset, we observe a significant difference in their performance. CodePlan successfully identified and updated 58 code blocks, achieving a perfect DiffBLEU score of 1.00, indicating that it made changes identical to the target repository. In contrast, Build-Repair, the baseline method, fell short by missing six blocks and generating eight build errors. This discrepancy highlights a critical limitation of Build-Repair—the lack of comprehensive change may-impact analysis capabilities. Specifically, Build-Repair failed to update constructor blocks required for initializing the newly added `ITestOutputHelper _output` class member. This omission went unnoticed because the absence of initialization didn't trigger build errors, causing a cascading effect on the callers of this constructor. In contrast, CodePlan successfully handled the initialization of the newly added `ITestOutputHelper _output` class member. This achievement can be attributed to its robust change-may impact analysis, which accurately identified the necessary modifications to the constructor block upon the addition of a new field. As a result, CodePlan seamlessly updated the constructor and all its callers, preventing any missed blocks or build errors. This finding underscores the importance of CodePlan's advanced planning abilities, which ensure a more thorough and accurate approach to repository-level code edits. Further details and qualitative analysis are provided in RQ3 (Figure 11) of our study.

CodePlan ***vs Pyright-Repair***. In context of the Temporal Edits task, we can see that CodePlan is able to successfully identify all the derived edit location in 2 repos (T-2, T-3) and is almost successuful in the third (T-1). This is in contrast to the baseline Pyright-Repair approach which fails to identify any of the derived edits in two repos (T-2, T-3). We find that using Pyright on a strict checking mode yields improved results, but only on one repo (T-1) where it peforms just as well as CodePlan. Overall we observe that the Pyright-Repair baseline falls short in identifying location to edit. This is also reflected in the DiffBLEU score which is consistently higher for CodePlan and Levenshtein Distance (L.D.) which is consistently lower. Note that since the LLM does not necessarily make the exact edit present in the ground truth, both DiffBLEU and L.D. may not have perfect 1.0 and 0 values respectively.

For both T-2 and T-3 we see that the Pyright-Repair baseline is unable to identify any derived edits. In both of these repos Pyright does not flag any errors in the codebase once the seed-edit has been made. In the case of T-2, the seed edit involves adding an argument to a method as shown in Figure 6 However this new parameter is also assigned a default value. Due to the presence of a

```
def load_mbd_ckpt(
    file_or_url_or_id: Union[Path, str],
+   filename: tp.Optional[str] = None,
    cache_dir: tp.Optional[str] = None
):
    return _get_state_dict(
        file_or_url_or_id,
-       filename="all_in_one.pt",
+       filename=filename,
        cache_dir=cache_dir
    )
```

Fig. 6. Seed edit for T-2.

```
def send_request(data):
    api_key = data.pop("api_key")
    api_type = data.pop("api_type")
+   api_endpoint = data.pop("api_endpoint")
    ...
```

Fig. 7. Seed edit for T-3.

default value for this argument, Pyright does not flag any of the call sites of this method as errors. However in the ground truth, the developer does follow up and edit the call sites. CodePlan's change may-impact analysis identifies these call-sites as "might requiring" edit and so we pass them to the LLM to edit.

In the case of T-3, the seed edit involves modifying the body of the function as describe in Figure 6. After the seed edit, the method now expects the dictionary being passed as argument to contain a new key "api_endpoint". Pyright will not flag any errors in the repo at this stage however it is clear that this may require modification to the callers of send_request. CodePlan does detect this and is able to identify and make edits at 10 derived sites present in the ground truth.

In both of these scenarios, we do not know a-priory whether the call-sites need editing. Thus CodePlan leaves this decision upto the LLM to decide given the context. In contrast, the Oracle-Guided baseline does not even detect that a change maybe needed. This can be attributed to the fact that Oracles such as Pyright, or Build aim to detect errors and hence will only flag code that violates certain rules. This is not aligned with the task of propagating changes across a repo, as there may be many cases where a change may need to be propagated but the repo with the change applied does violate any of the oracle's rules.

Coeditor Evaluation. When comparing CodePlan with Coeditor-CodePlan, we can see that it performs on par on T-1 but lags slightly behind on T-2 and T-3 missing one edit site in each. While both approaches are using the same analysis and planning, the local edits made in CodePlan are more in-line with the ground-truth compared to those by Coeditor, reflected in the the lower DiffBLEU scores and higher L.D. exhibited by Coeditor-CodePlan in T-2 and T-3. This can be due to the difference in context-understanding abilities between gpt-4-32k and Coeditor. For example, opting to instantiate an argument to a method instead of adding a parameter to the caller can mean missing out of edits resulting from the signature change to the caller. 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. This is also observed when comparing the Pyright-Strict-Repair and Coeditor-Pyright-Strict-Repair on T-2, where incorrect local edits by Coeditor lead to missing of edit sites and worse DiffBLEU and L.D.

In conclusion, the experimental results substantiate that CodePlan's planning-based approach is highly effective in automating repository-level coding tasks, offering superior matching, completeness, and precision compared to traditional baseline methods. Its ability to handle complex coding tasks within large-scale repositories signifies a significant advancement in automating software engineering activities.

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet

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

Motivation.: The motivation behind RQ2 lies in recognizing that Large Language Models (LLMs) require both temporal and spatial contexts to provide accurate and contextually relevant code update suggestions. Temporal context is vital for understanding the sequence and timing of code changes, allowing LLMs to make suggestions that align with the code's evolution and maintain consistency. Without this temporal awareness, LLMs might offer solutions that conflict with earlier or subsequent modifications, leading to code errors. Spatial context, on the other hand, enables LLMs to comprehend the intricate relationships and dependencies between different parts of the codebase. This understanding is crucial for pinpointing where updates are needed and ensuring that they are applied in a way that preserves code functionality. Therefore, investigating the importance of these contexts in CodePlan's planning is fundamental to harnessing the full potential of LLMs in automating repository-level coding tasks effectively.

Experimental Setup.: To assess the importance of temporal and spatial contexts in CodePlan's planning (RQ2), a specific experimental setup is employed. Recall that temporal contexts are tracked by CodePlan through the maintenance of a list of previous related changes, and these contexts are subsequently incorporated into the Large Language Model (LLM) prompt.

To measure the significance of these contexts, a controlled experiment is carried out. In this experiment, the tracking of temporal changes is intentionally halted, and no temporal or spatial contexts are provided to the LLM during code modification tasks. This enables a detailed examination of how LLM responses are impacted when these essential contexts are omitted. The experiment also encompasses a comprehensive assessment of various metrics, including code consistency (block metrics) and code correctness (Diff BLEU and Levenshtein distance). By comparing the outcomes between CodePlan's setup with temporal and spatial contexts and the experimental setup without them, the research aims to precisely quantify the importance of these contexts in CodePlan's planning process and their influence on the quality of automated code modifications.

Results Discussion.:
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 8 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.

Furthermore, Figure 9 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.

Dataset	Approach	Matched Blocks	Missed Blocks	Spurious Blocks	Diff BLEU	Levenshtein Distance	Oracle Verdict
C# Migration Task on Int-1 with Temporal and Spatial Contexts							
Int-1	CodePlan (Iter 1 + Iter 2)	151	0	0	1.00	0	✓ (0)
C# Migration Task on Int-1 **without** Temporal and Spatial Contexts							
Int-1	CodePlan (Iter 1)	112	39	4	0.73	3674	✗ (34)
	CodePlan (Iter 2)	121	30	52	0.53	4522	✗ (68)
	CodePlan (Iter 3)	121	30	54	0.51	4524	✗ (69)

Table 4. Ablation study with and without temporal/spatial blocks. Without temporal/spatial contexts, CodePlan is unable to make correct edits.

Fig. 8. Illustration of importance of temporal context. Failure to update LogacyLogger to ModernLogger in Initialize() method is the results of missing missing temporal context.

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 10 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

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet
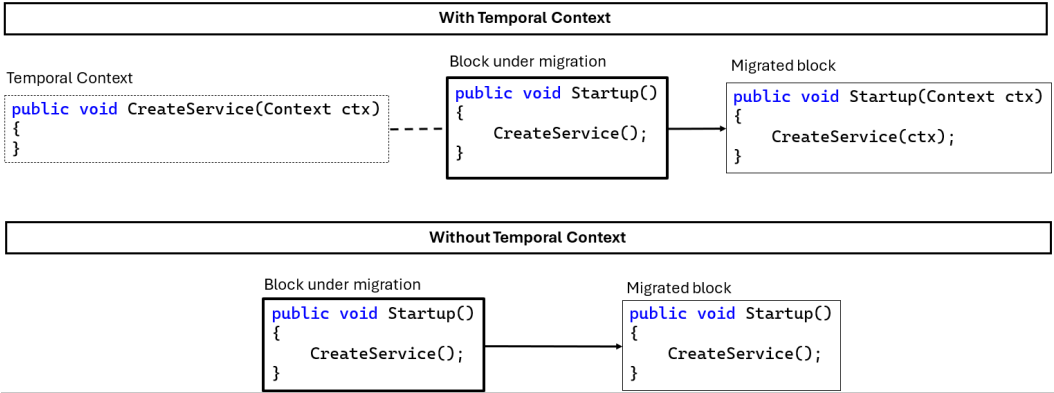

Fig. 9. Illustration of importance of temporal context. Failed update to Startup() method is the results of missing missing temporal context.
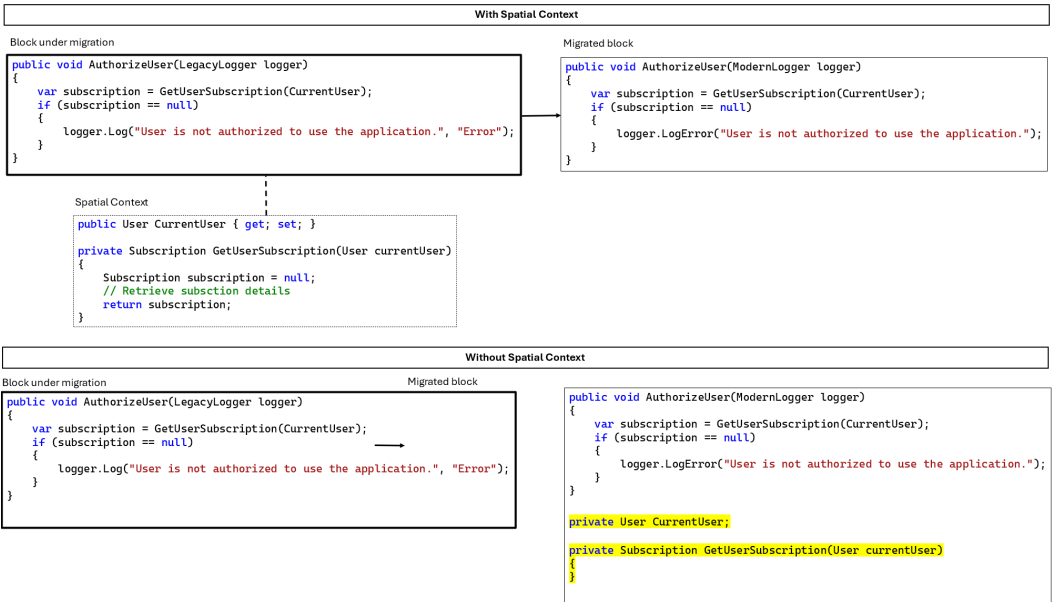

Fig. 10. Illustration of importance of spatial context. Spurious blocks, highlighted in yellow are the results of missing missing spatial context.

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.

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.

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

Motivation.: RQ3 seeks to uncover the key differentiators that empower CodePlan to excel in tackling intricate coding tasks compared to baseline approaches. This research question is motivated by the need to identify and understand the factors that contribute to CodePlan's superior performance. Given the increasing complexity of coding tasks, especially those at the repository level, it becomes crucial to pinpoint the specific aspects that set CodePlan apart from traditional methods. By discerning these differentiators, the research aims to shed light on the strengths and advantages of CodePlan, offering valuable insights into how it effectively addresses the challenges posed by complex coding tasks.

Experimental Setup.: The primary focus here is on qualitative analysis. After conducting code modifications with both CodePlan and the baseline methods, the results are meticulously examined through manual inspection. This entails a detailed examination of the changes made by each approach, with the aim of gaining qualitative insights into the decision-making processes and the nuances of code modifications. By manually eyeballing and comparing the alterations, the research seeks to uncover subtle yet crucial distinctions that illuminate the strengths and underlying mechanisms that give CodePlan its edge in addressing the challenges posed by intricate coding tasks. This qualitative approach provides a comprehensive understanding of how CodePlan excels in this context and what sets it apart from conventional methods.

Results Discussion.:

CodePlan***'s Strategic Planning and Context Awareness:***.
    CodePlan's exceptional performance in handling complex coding tasks can be attributed to its powerful features, notably 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 Ext-1 illustrated in Figure 11, 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 11. These cascading changes start from introducing `ITestOutputHelper _output` as a class-level member, accomplished via LLM updates.
    CodePlan's change-may-impact analysis proves invaluable 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 11.
    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

Fig. 11. Illustration of the CodePlan's plan execution.

in Figure 4. Instead, it solely executes the modification outlined in step 1, resulting in incomplete code updates.

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

Incremental Analysis: Maintaining Relationships with Dependency Graph:.

CodePlan's exceptional 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 prowess 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 remarkable success in complex coding tasks can be attributed to its proficiency in extracting 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 superior performance in complex coding tasks is its adeptness at detecting 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 excels in identifying such subtle 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.

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet

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.

Incremental static analysis is lightweight and easy to deploy.. One of the key distinguishing features that enables CodePlan to outperform baselines in tackling complex coding tasks is its utilization of lightweight and readily deployable incremental static analysis. In real-world software development, build systems often prove to be complex and resource-intensive. Traditional build processes can be quite expensive in terms of computational resources and time, especially when dealing with large and intricate codebases.

In contrast, CodePlan's incremental static analysis offers a more efficient and practical alternative. By focusing on analyzing only the code portions that have been modified, CodePlan significantly reduces the computational overhead associated with build processes. This approach not only accelerates the repository-wide editing but also minimizes the resources required for each editing task.

In summary, CodePlan's exceptional performance in complex coding tasks can be attributed to several key differentiators. Firstly, its strategic planning, coupled with its deep understanding of temporal and spatial contexts, enables CodePlan to make methodical and context-aware code modifications. Additionally, the system's ability to maintain relationships within the dependency graph through incremental analysis ensures the preservation of code correctness and functionality. CodePlan's enhanced spatial and temporal context extraction further aids in accurately updating code elements. Notably, the change-may-impact analysis not only identifies subtle behavioral changes but also maintains a clear cause-effect relationship between code modifications. Finally, CodePlan's lightweight and easily deployable incremental static analysis offer practical advantages, reducing resource overhead in complex software projects. These combined differentiators empower CodePlan to excel in solving intricate coding challenges.

6 LIMITATIONS AND THREATS TO VALIDITY

While CodePlan demonstrates significant capabilities, there are certain limitations and potential threats to the validity of its results. One crucial factor for CodePlan's success lies in the quality of the dependency analysis. In statically typed languages like C# and Java, rich code dependency information can be extracted effectively. However, in dynamically typed languages such as Python (without type hints) or JavaScript, establishing semantically rich relationships between code blocks can be more challenging due to the dynamic nature of the code.

Our current implementation of CodePlan primarily handles relations between code blocks through static analysis. In real-world software systems, numerous dynamic dependencies exist, including data flow dependencies, complex dynamic dispatching (via virtual functions and dynamic castings), algorithmic dependencies (e.g., when input lists are expected to be sorted), and various execution dependencies (such as multi-threading and distributed processing). Addressing these dynamic dependencies will be a crucial focus of our future work.

We selected multiple repositories across two challenging tasks (migration and temporal edits) and two languages (C# and Python) to evaluate the generality of CodePlan. These repositories and tasks are representative of real-world usecases. However, given the complexity of setting up

each experiment, our evaluation is restricted to a few repositories. More experimentation will be required to probe the strengths and weaknesses of CodePlan.

While CodePlan excels in planning and editing for programming language repositories, enterprise software systems often comprise various other artifacts like configuration files, metadata files, project setting files, external dependencies, and more. A comprehensive repository-level editing approach, particularly for tasks like software migrations, necessitates the ability to edit these additional artifacts. This calls for an evolution of dependency graphs to include nodes and relations encompassing all these diverse artifacts, and change-may-impact analysis that spans across them.

CodePlan currently relies on Large Language Models (LLMs) to utilize the necessary context information for making code edits. It trusts the LLM's response for change-may-impact analysis. In cases where the LLM response is incorrect or spurious, it may lead to erroneous updates in the repository. While running CodePlan in multiple iterations in our experiments helped rectify such issues, there may be instances where this approach may not suffice.

In terms of CodePlan's update strategy, it currently focuses on updating one block at a time. While this approach aligns with our current design choices, there may be scenarios where it would be more efficient or necessary to implement multiple simultaneous changes. More sophisticated techniques for handling multiple block updates (within the same file or across different files) and change propagation will be a key area of exploration in our future work.

Although our current methodology employs zero-shot prompting, there exists potential for extension to include few-shot examples, Chain of Thought (CoT), and other techniques. These extensions represent a promising avenue for future research.

7 RELATED WORK

LLMs for Coding Tasks. A multitude of LLMs [10, 19, 21, 24, 28, 30, 35, 57, 73–75, 80] 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 [50, 56], program repair [11, 43, 79], vulnerability patching [60], inferring program invariants [62], test generation [69] and multi-task evaluation [72]. However, these investigations are performed on curated examples that are extracted from their repositories and are meant to be accomplished with independent invocations of the LLM. We consider a different class of tasks posed at the scale of code repositories, where an LLM is called multiple times on different examples which are inter-dependent. We monitor the results of each LLM invocation within the repository-wide context to identify future code change obligations to get the repository to a consistent state, e.g., where the repository is free of build or runtime errors.

Automated Planning. Automated planning [37, 67] is a well-studied topic in AI. Online planning [67] 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. In orthogonal directions, [42] uses an LLM to derive a plan given a natural language intent before generating code to solve complex coding problems and [86] performs lookahead planning (tree search) to guide token-level decoding of code LMs. Planning in our work is based on analyzing dependency relations and changes to them as an LLM makes changes to a code repository.

Analysis of Code Changes. Static analysis is used for ensuring software quality. It is expensive to recompute the analysis results every time the code undergoes changes. The field of incremental

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet

program analysis offers techniques to recompute only the analysis results impacted by the change. Specialized algorithms have been developed for dataflow analysis [18, 68], pointer analysis [84], symbolic execution [63], bug detection [52] and type analysis [27]. Program differencing [16, 46, 48] and change impact analysis [17, 41] determine the differences in two program versions and the effect of a change on the rest of the program. The impact of changes has been studied for regression testing [65], analyzing refactorings [33] and assisting in code review [13, 36]. 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.

Spatial and Temporal Contextualization. As discussed in the Introduction, LLMs benefit from relevant context derived from other files in the repository and from past edits. We provide both these pieces of information to the LLM by tracking the code changes and dependency relations.

Learning Edit Patterns. Many approaches have been developed to learn edit patterns from past edits or commits in the form of rewrite rules [31], bug fixes [15, 20], type changes [45], API migrations [49, 82] and neural representations of edits [83]. Approaches such as [53] and [54] 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 [55] and temporally-related edit sequences [87]. 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.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we introduced CodePlan, a novel framework designed to tackle the challenges of repository-level coding tasks, which involve pervasive code modifications across large and inter-dependent codebases. CodePlan leverages incremental dependency analysis, change may-impact analysis, and adaptive planning to orchestrate multi-step edits guided by Large Language Models. We evaluated CodePlan on diverse code repositories with varying complexities and sizes, including both internal proprietary repositories and public GitHub repositories in C# and Python for migration and temporal edit tasks. Our results demonstrated that CodePlan outperforms baseline methods, achieving better alignment with the ground truth. In conclusion, CodePlan presents a promising approach to automating complex repository-level coding tasks, offering both productivity benefits and accuracy improvements. Its success in addressing these challenges opens up new possibilities for efficient and reliable software engineering practices.

While CodePlan has shown significant promise, there are several avenues for future research and enhancements. First, we aim to expand its applicability to a broader range of programming languages and code artifacts, including configuration files, metadata, and external dependencies, to provide a more holistic solution for repository-level editing. Additionally, we plan to explore further customization of CodePlan's change may-impact analysis. This could involve incorporating task-specific impact analysis rules, either through rule-based methods or more advanced machine learning techniques, to fine-tune its editing decisions for specific coding tasks. Furthermore, we will address the challenge of handling dynamic dependencies, such as data flow dependencies, complex dynamic dispatching (via virtual functions and dynamic castings), algorithmic dependencies (e.g., when input lists are expected to be sorted), and various execution dependencies (such as multi-threading and distributed processing), to make CodePlan even more versatile in addressing a wider range of software engineering tasks.

REFERENCES

[1] [n. d.]. audiocraft. https://github.com/facebookresearch/audiocraft.

[2] [n. d.]. CodeQL. https://github.com/github/codeql.

[3] [n. d.]. JARVIS. https://github.com/microsoft/JARVIS.

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

[5] [n. d.]. OmniSharp. https://github.com/OmniSharp/csharp-language-server-protocol.

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

[7] [n. d.]. Reactive Streams TCK. https://github.com/reactive-streams/reactive-streams-dotnet/tree/master/src/tck.

[8] [n. d.]. whisper. https://github.com/openai/whisper.

[9] Lakshya A Agrawal, Aditya Kanade, Navin Goyal, Shuvendu K. Lahiri, and Sriram K. Rajamani. 2023. Guiding Language Models of Code with Global Context using Monitors. arXiv:2306.10763 [cs.CL]

[10] 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]

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

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

[13] Everton L. G. Alves, Myoungkyu Song, and Miryung Kim. 2014. RefDistiller: A Refactoring Aware Code Review Tool for Inspecting Manual Refactoring Edits. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 751–754. https://doi.org/10.1145/2635868.2661674

[14] Amazon Web Services, Inc. 2023. *Amazon Code Whisperer - AI Code Generator*. https://aws.amazon.com/codewhisperer/ Accessed: July 25, 2023.

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

[16] 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.

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

[18] 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.

[19] 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].

[20] 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

[21] 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).

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

[23] Shaked Brody, Uri Alon, and Eran Yahav. 2020. A structural model for contextual code changes. 4, OOPSLA (Nov. 2020). https://doi.org/10.1145/3428283 Publisher Copyright: © 2020 Owner/Author..

[24] 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.

[25] Max Brunsfeld, Andrew Hlynskyi, Patrick Thomson, Josh Vera, Phil Turnbull, Timothy Clem, Douglas Creager, Andrew Helwer, Rob Rix, Hendrik Van Antwerpen, Michael Davis, , Ika, Tuấn-Anh Nguyễn, Stafford Brunk, Niranjan Hasabnis, Bfredl, Mingkai Dong, Matt Massicotte, Jonathan Arnett, Vladimir Panteleev, Steven Kalt, Kolja Lampe, Alex Pinkus, Mark Schmitz, Matthew Krupcale, Narpfel, Santos Gallegos, Vicent Martí, and , Edgar. 2023. tree-sitter/tree-sitter: v0.20.8. https://doi.org/10.5281/ZENODO.4619183

[26] Alan Bundy. 1988. The use of explicit plans to guide inductive proofs. In *9th International Conference on Automated Deduction: Argonne, Illinois, USA, May 23–26, 1988 Proceedings 9*. Springer, 111–120.

[27] 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.

[28] 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).

[29] 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.

[30] 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).

[31] 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

[32] 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.

[33] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. 2006. Automated detection of refactorings in evolving components. In *ECOOP 2006–Object-Oriented Programming: 20th European Conference, Nantes, France, July 3-7, 2006. Proceedings 20*. Springer, 404–428.

[34] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2022. CoCoMIC: Code Completion By Jointly Modeling In-file and Cross-file Context. http://arxiv.org/abs/2212.10007 arXiv:2212.10007 [cs].

[35] 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).

[36] Xi Ge, Saurabh Sarkar, Jim Witschey, and Emerson Murphy-Hill. 2017. Refactoring-aware code review. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 71–79.

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

[38] Github, Inc. 2023. *GitHub Copilot: Your AI pair programmer*. https://github.com/features/copilot Accessed: July 25, 2023.

[39] David González, Joshué Pérez, Vicente Milanés, and Fawzi Nashashibi. 2015. A review of motion planning techniques for automated vehicles. *IEEE Transactions on intelligent transportation systems* 17, 4 (2015), 1135–1145.

[40] Priyanshu Gupta, Avishree Khare, Yasharth Bajpai, Saikat Chakraborty, Sumit Gulwani, Aditya Kanade, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. 2023. GrACE: Generation using Associated Code Edits. arXiv:2305.14129 [cs.SE]

[41] 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.

[42] 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]

[43] 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).

[44] Erez Karpas and Daniele Magazzeni. 2020. Automated planning for robotics. *Annual Review of Control, Robotics, and Autonomous Systems* 3 (2020), 417–439.

[45] 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.

[46] 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.

[47] Steven M La Valle. 2011. Motion planning. *IEEE Robotics & Automation Magazine* 18, 2 (2011), 108–118.

[48] 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.

[49] 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.

[50] 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.

[51] Tianyang Liu, Canwen Xu, and Julian McAuley. 2023. RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems. arXiv:2306.03091 [cs.CL]

[52] Scott McPeak, Charles-Henri Gros, and Murali Krishna Ramanathan. 2013. Scalable and incremental software bug detection. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 554–564.

[53] 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.

[54] 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.

[55] 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.

[56] 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_

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

[58] 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.

[59] Pardis Pashakhanloo, Aaditya Naik, Yuepeng Wang, Hanjun Dai, Petros Maniatis, and Mayur Naik. 2022. Codetrek: Flexible modeling of code using an extensible relational representation. (2022).

[60] 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.

[61] Hengzhi Pei, Jinman Zhao, Leonard Lausen, Sheng Zha, and George Karypis. 2023. Better context makes better code language models: A case study on function call argument completion. In *AAAI*. https://www.amazon.science/publications/better-context-makes-better-code-language-models-a-case-study-on-function-call-argument-completion

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

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

[64] Machel Reid and Graham Neubig. 2022. Learning to Model Editing Processes. https://doi.org/10.48550/ARXIV.2205.12374

[65] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G Ryder, and Ophelia Chesley. 2004. Chianti: a tool for change impact analysis of java programs. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 432–448.

[66] Replit, Inc. 2023. *Replit*. https://replit.com/ Accessed: July 25, 2023.

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

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

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

[70] Disha Shrivastava, Denis Kocetkov, Harm de Vries, Dzmitry Bahdanau, and Torsten Scholak. 2023. RepoFusion: Training Code Models to Understand Your Repository. arXiv:2306.10998 [cs.LG]

[71] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2022. Repository-level prompt generation for large language models of code. *arXiv preprint arXiv:2206.12839* (2022).

[72] 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]

[73] 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]

[74] Ben Wang and Aran Komatsuzaki. 2021. GPT-J-6B: A 6 billion parameter autoregressive language model.

[75] 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).

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

[77] Jiayi Wei, Greg Durrett, and Isil Dillig. 2023. TypeT5: Seq2seq Type Inference using Static Analysis. arXiv:2303.09564 [cs.SE]

[78] Mark Wilson-Thomas. [n. d.]. *GitHub Copilot chat for Visual Studio 2022.* https://devblogs.microsoft.com/visualstudio/github-copilot-chat-for-visual-studio-2022/ Accessed: July 25, 2023.

[79] 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*.

[80] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A Systematic Evaluation of Large Language Models of Code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (MAPS 2022)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3520312.3534862 event-place: San Diego, CA, USA.

[81] Frank F Xu, Junxian He, Graham Neubig, and Vincent J Hellendoorn. 2021. Capturing structural locality in non-parametric language models. *arXiv preprint arXiv:2110.02870* (2021).

[82] 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.

[83] 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].

[84] 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.

[85] Fengji Zhang, Bei Chen, Yue Zhang, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. *arXiv preprint arXiv:2303.12570* (2023).

[86] 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]

[87] 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