GRACE: Language Models Meet Code Edits

Priyanshu Gupta*
priyansgupta@microsoft.com
Microsoft
India

Avishree Khare*†
akhare@seas.upenn.edu
University of Pennsylvania
USA

Yasharth Bajpai
ybajpai@microsoft.com
Microsoft
India

Saikat Chakraborty
saikatc@microsoft.com
Microsoft Research
USA

Sumit Gulwani
sumitg@microsoft.com
Microsoft
USA

Aditya Kanade
kanadeaditya@microsoft.com
Microsoft Research
India

Arjun Radhakrishna
arradha@microsoft.com
Microsoft
USA

Gustavo Soares
gsoares@microsoft.com
Microsoft
USA

Ashish Tiwari
astiwar@microsoft.com
Microsoft
USA

ABSTRACT

Developers spend a significant amount of time in editing code for a variety of reasons such as bug fixing or adding new features. Designing effective methods to predict code edits has been an active yet challenging area of research due to the diversity of code edits and the difficulty of capturing the developer intent. In this work, we address these challenges by endowing pre-trained large language models (LLMs) with the knowledge of relevant prior associated edits, which we call the GRACE (Generation conditioned on Associated Code Edits) method. The generative capability of the LLMs helps address the diversity in code changes and conditioning code generation on prior edits helps capture the latent developer intent. We evaluate two well-known LLMs, CODEX and CODET5, in zero-shot and fine-tuning settings respectively. In our experiments with two datasets, GRACE boosts the performance of the LLMs significantly, enabling them to generate 29% and 54% more correctly-edited code in top-1 suggestions relative to the current state-of-the-art symbolic and neural approaches, respectively.

CCS CONCEPTS

• **Software and its engineering** → **Software evolution**; **Automatic programming**; • **Computing methodologies** → *Artificial intelligence.*

KEYWORDS

Code editing, Associated edits, Large language models, Pre-trained model, Programming language processing

*Both authors contributed equally to this work.
†Work done while at Microsoft

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616253

ACM Reference Format:
Priyanshu Gupta, Avishree Khare, Yasharth Bajpai, Saikat Chakraborty, Sumit Gulwani, Aditya Kanade, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. 2023. GRACE: Language Models Meet Code Edits. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616253

1 INTRODUCTION

Maintaining and modifying existing code takes up a considerable portion of a developer's time compared to writing new code [9, 40]. Due to the high cost of software maintenance [32, 35], popular Integrated Development Environments (IDEs) have tooling to support developers as they refactor code [25, 26, 45], fix defects, adapt code to changes in the environment, or add support for new or changed requirements [34, 45]. One desirable feature is *code edit suggestions* wherein the tools use the location where the developer is editing code, and the surrounding code context, to generate candidate edits to recommend [44, 68].

To automate code edit suggestions, researchers have proposed several approaches to learn *edit patterns* from edits in source code repositories [7, 19, 36, 56]. However, these approaches suffer from two key limitations: (1) They focus on *individual edits* and learn program transformation rules for them. We note that edits are not performed in isolation. Developers make changes at one location, then jump to another, and then maybe back to the first location to make further changes [37]. The edits that developers make to the code at different locations may not be identical, but they are often interrelated. In fact, the next edit often depends on the previously performed edits [68]. Learning one-step edit patterns limits the ability of these approaches to accurately predict the most likely next edit. (2) The symbolic program transformation rules can only slice and dice the existing code and compose its pieces to create code – they *cannot generate new code* whose pieces do not already occur in the existing version. This limits the expressiveness of these approaches in terms of the types of edits that they can predict.

Unlike symbolic program transformation rules, neural language models have the capability to generate new code that does not

necessarily occur in the surrounding code context. The pre-trained large language models (LLMs) like CODEX [16] and CODET5 [63] have been shown to be highly proficient at generating code. In fact, they are already impacting software engineering in significant ways, e.g., through popular code completion tools like GitHub Copilot [28]. However, when it comes to editing code, without the knowledge of previous edits, these models are unable to infer the developers' intent and fail to generate code that should be used to replace existing code in the next edit. In this work, we explore ways to predict code edits using LLMs by conditioning code generation on prior, relevant edits. We call such prior edits *associated edits* and this methodology as *Generation conditioned on Associated Code Edits* (GRACE).

In recent times, there have been a few attempts to leverage past history of code evolution to learn to edit code [10, 55, 68]. OVERWATCH [68] is a symbolic technique that mines "edit sequence patterns". Such a pattern is essentially a program transformation rule whose application is conditioned on the prior application of some other program transformation rules. Being a pattern-based technique, OVERWATCH suffers from the inability to generate new code, and it also requires significant engineering effort to build the underlying symbolic pattern-learning engine. C3PO [10] is a neural model to predict the next edit at a location, given the edits only in the spatial vicinity of that location. While reliance on such spatially related edits shows initial promise towards automation in code editing, the hypothesis may not always hold true—developers may edit two locations simultaneously that are far away from each other spatially [37]. In this work, we attempt to relax this reliance, and do not restrict associated edits to be the ones that occur in the spatial vicinity of the location under consideration. We show that associated edits obtained from temporal history can also be useful. Further, the C3PO model is a custom model that generates the edited code by copy-pasting existing code fragments and is therefore unable to generate new code (similar to the symbolic techniques including OVERWATCH).

EditPro [55] is a recent neural model that aims to learn the edit process for natural language documents and code files. It proposes a special multi-step procedure where the model first predicts token-wise edit actions (insert, delete, etc.), which are then subsequently applied to the code. The edit actions requiring code generation, such as insert and replace, require a separate decoding step. EditPro experiments with single-line edits, whereas our datasets contain multi-line edits. Instead of training a new type of model from scratch, which can be expensive and requires a significant amount of data, GRACE allows us to repurpose the already powerful LLMs to generate edited code.

We demonstrate the benefits of our approach in two settings: (1) *zero-shot setting* in which the LLM is used out-of-the-box without additional training but with an informative prompt about associated edits and (2) *fine-tuning setting* in which the LLM is fine-tuned on data annotated with associated edits. In both cases, our results show significant benefits of conditioning existing LLMs on associated edits without having to pay the price of designing and training specialized models from scratch.

In our experiments, we use the code-editing benchmarks from OVERWATCH and C3PO. As the baseline LLMs, we use the CODEX-DAVINCI model in the zero-shot setting and the CODET5 model (220M params) in the fine-tuning setting. We show that the use of associated edits helps boost the ability of these models to predict the next edit compared to the pre-trained models used without associated edits. In the case of CODEX-DAVINCI, we get improvements of 17% and 30% (in absolute terms) for the Overwatch and C3PO datasets, and improvement of 7.45% and 9.64% in the case of CODET5. We also compare CODEX-DAVINCI with associated-edit prompting and the fine-tuned CODET5 model with the OVERWATCH and C3PO methods on the respective datasets. All our models substantially outperform these methods on their own datasets by a significant margin. Our best models outperform OVERWATCH on its dataset by 10.92% and C3PO on its dataset by 28.63% (absolute): this is 28.61% and 53.82% relative improvement respectively. EditPro dataset and model have not been released by the authors yet, therefore, we were unable to compare against it. Both OVERWATCH and C3PO construct edited code from *existing* or past code, whereas we use LLMs that are capable of generating *new* code. We show that this makes our approach more general, and we can predict code edits that are often out-of-scope for these approaches.

In summary, we make the following contributions:

(1) We consider a practically important software-engineering problem of predicting code edits and propose GRACE, a novel method of leveraging powerful LLMs to predict code edits by conditioning them on prior edits.
(2) Through experimentation on two datasets, we show that using GRACE we can substantially improve performance of LLMs in zero-shot or fine-tuning settings.
(3) GRACE is superior to the state-of-the-art symbolic or neural methods designed specifically to handle code edits.
(4) We conduct experiments to thoroughly evaluate GRACE and report insights gleaned from them.

2 MOTIVATING EXAMPLE

In this section, we motivate GRACE by using a concrete code development scenario. We further discuss how this approach differs from existing approaches.

Illustrative Example: Consider a developer refactoring code shown in Figure 1a as Version v_1. The goal of the developer is to use `SerializationException` provided by the `System.Runtime.Serialization` namespace to get to Version v_3 shown in Figure 1c. Let us say that the developer first replaces `Exception` on line 250 in Version v_1 with `SerializationException` to create Version v_2 shown in Figure 1b. This edit required to go from Version v_1 to Version v_2 is denoted as $\delta_{1,2}$. The developer's cursor then moves to Line 3 of Version v_2 and our goal is to predict the next edit the developer will perform to reach Version v_3, namely the edit $\delta_{2,3}$.

Conditioning on Prior Edits: The task of predicting the edit $\delta_{2,3}$ is non-trivial. The code in Version v_2 has some useful information; for example, the code indicates that `SerializationException` is defined on Line 250 of Version v_2 but the required `System.Runtime.Serialization` namespace hasn't been imported anywhere. This signal, however, is faintly present within 250 lines of additional spatial context and the relationship between the added Exception and the required import is lost. This relationship is an important piece of information that is required to insert the `using` statement

```
1   using System;
2   using System.Linq;
3   using System.Text;                    ...
250 catch (Exception)
251       ...
```

(a) Version v_1

```
1   using System;
2   using System.Linq;
3   using System.Text;                    ...
250 - catch (Exception)
250 + catch (SerializationException)
```

(b) Version v_2 with associated edit $\delta_{1,2}$

```
1   using System;
2   using System.Linq;
3 + using System.Runtime.Serialization;
4   using System.Text;                    ...
251 catch (SerializationException)
```

(c) Version v_3 with target edit $\delta_{2,3}$

Figure 1: The developer performs edit $\delta_{1,2}$ to go from Version v_1 of their code to Version v_2. This edit serves as an associated edit that helps with predicting the edit $\delta_{2,3}$ needed to go from Version v_2 to v_3.

Table 1: Comparison of different approaches on the example from Figure 1

Technique	Prediction	Correct?
C3PO	NO RESPONSE (alien insertion)	✗
OVERWATCH	NO RESPONSE (no matching pattern)	✗
CODE-DAVINCI-EDIT-1	EMPTY RESPONSE	✗
CODEX-DAVINCI without associated edits	`using System.Net.Http;`	✗
CODET5 without associated edits	`using System.Threading;`	✗
CODEX-DAVINCI with associated edits (Our approach)	`using System.Runtime.Serialization;`	✓
CODET5 with associated edits (Our approach)	`using System.Runtime.Serialization;`	✓

on Line 3 of Version v_3. *Our first key observation for improving prediction of code updates is that it should be conditioned on related edits from the past.* In the above scenario, we want to predict the update to Version v_2 by also looking at the how Version v_2 was created from Version v_1. The edit $\delta_{1,2}$ is an *associated edit*. In this example, there is just one associated edit, but in general there can be multiple previous edits picked as associated edits.

There has been some recent work on predicting code changes conditioned on previous changes [10, 68]. We now discuss how these approaches work on the illustrative example. Table 1 shows the predictions of various techniques on the target.

C3PO: C3PO is a path-based edit prediction method that generates an edit script to predict subsequent edits. It uses a pointer network to pick valid target edits at v_2 by attending to $\delta_{1,2}$, represented as an edit path in the AST. As these target edits can only refer to nodes in the ASTs at v_2 and $\delta_{1,2}$, the pointer network does not have access to the `Serialization` token needed to be inserted on Line 3. Therefore, C3PO would filter out above-mentioned example in its training and testing pipelines categorizing it as an 'alien insertion'. When C3PO finetuned on the OVERWATCH train set is used to predict $\delta_{2,3}$, it *incorrectly* suggests picking an existing `using` statement.

OVERWATCH: OVERWATCH is a symbolic procedure that learns (abstract syntax) tree transformation rules from example edit sequences in the training data, and then makes predictions by applying those rewrite rules. The above-mentioned example does not match any of the ∼ 50 patterns that the authors released in [68]. Thus, out of the box, OVERWATCH would not be able to provide any suggestion because of unavailability of a matching pattern for the target edit in the example. *If* we provide enough edit sequences similar to "$\delta_{1,2}$ followed by $\delta_{2,3}$" as training data to OVERWATCH, then it might learn a few edit patterns depending on the examples it gets and the order in which they are generalized. The only two useful patterns that could be learned would be either (1) "the substitution of `Exception` by `SerializedException` is followed by importing the `System.Runtime.Serialization` namespace", or (2) "the substitution of `Exception` by a *placeholder Type* is followed by importing a *placeholder namespace*." While pattern (1) would return the correct response, it is an "overfit pattern" that does not generalize to other changes in the substituted type. Pattern (2) is too general and cannot generate a concrete suggestion due to the unbound placeholder.

Using LLMs: The approaches discussed above cannot generate the right predictions either when the target requires a new token (C3PO) or when it cannot match an existing learned pattern (OVERWATCH). LLMs of Code have emerged as competitive code completion tools that offer generative capabilities. This leads to our second key observation: *LLMs can handle diverse editing scenarios including those that involve generation of new tokens.* We now discuss how these models work on the illustrative example.

LLMs without Associated Edits: First, let us consider how a modern code completion tool (based on powerful LLMs) will attempt to predict the new code at Line 3 of Version v_3. Code completion tools, like CODEX-DAVINCI, look at the current snapshot of the code to make predictions. In other words, the tool will look at Version v_2 to predict Version v_3. When we provide code from version v_2 to CODEX-DAVINCI, it correctly predicts that something should be imported, but it predicts an incorrect namespace. If we use CODE-DAVINCI-EDIT-1, the editing variant of CODEX-DAVINCI that allows you to provide instructions for editing, the prediction continues to remain incorrect.

LLMs with GRACE: Following our two key observations, we present the edit $\delta_{1,2}$ to CODEX-DAVINCI, along with Line 3 of Version v_2 that needs to be updated. Now, the model successfully predicts that the updated code would be Line 3 of Version v_3. We discuss the prompt design in detail in Section 4.2.

We found that this utility of associated edits for edit prediction also extends to other models: a base CODET5 model fine-tuned to predict $\delta_{2,3}$ using v_2 incorrectly predicts `System.Threading` while the same model fine-tuned to additionally use $\delta_{1,2}$ to make the prediction gets the import right.

By building a code change prediction model over a code generation model, we are able to extend the scope of edit predictions. Moreover, we are able to also perform better than the existing works

on the subset of the benchmarks that are in their scope. We discuss our quantitative performance on these benchmarks compared to C3PO and OVERWATCH in Sections 5.4 and 5.5 respectively. We further present a qualitative analysis of the results in Section 5.6.

3 ASSOCIATED CODE UPDATES

We define the associated code update task in this section. The associated code update task is inspired from the EditCompletion task [10] and the edit likelihood prediction task [55].

Let $v_0, \ldots, v_n$ be a sequence of versions of a source code file. An edit $\delta_{i,j}$ is the difference between two versions, v_i and v_j. We view $\delta_{i,j}$ as a function that returns v_j on the input v_i, i.e., $\delta_{i,j}(v_i) = v_j$. Furthermore, an *associated edit* Δ^i is $\delta_{j,k}$, for some $0 \leq j < k < n$. Given the m associated edits $\Delta^1, \Delta^2, \ldots, \Delta^m$ and the version v_{n-1} along with locations L in v_{n-1}, the *associated code update* task is to predict version v_n assuming only the locations L in v_{n-1} are updated. We thus want to model the probability

$$P(v_n \mid L, v_{n-1}, \Delta^1, \Delta^2, \ldots, \Delta^m)$$

We next make a few remarks about the problem formulation above. First, the $n-1$ versions $v_0, v_1, \ldots, v_{n-2}$ need not necessarily match the history of the underlying source code file. The actual historical versions can be different, and in fact, in the formulation above, it is not the complete versions themselves, but the edits Δ^i that are used in the prediction task. The only version that is important here is the current version v_{n-1}. Furthermore, the set of m past edits need not even be the exhaustive set of *all* temporally consecutive edits; they could be a subset of the edits that have been performed so far. Hence, Δ^i doesn't necessarily have to be $\delta_{i-1,i}$.

Second, the edits are allowed to be spatially far away from each other and from the target locations L in Version v_{n-1}. While an edit Δ^i that modifies locations close to the target locations L is likely to be useful to include in the set m of edits, edits farther away from L may also be relevant. We makes no assumption on spatial locality of edits in contrast to the EditCompletion task in [10].

3.1 Assumptions about Sub-Problems

Our problem formulation above abstracts away three important and challenging related sub-problems that are crucial to build an end-to-end tool. These three sub-problems are: (1) edit localization, (2) edit granularity, and (3) associated edits identification. The associated code update problem formulation assumes that we have some solution for these three related problems.

Edit Localization: The edit localization problem seeks to find the locations L where the developer should make edits. How we get these locations is dependent on the application. For example, in an IDE, cursor location is a good indicator of where the developer wants to make changes. Another option is to build a model that predicts the next edit location given prior associated edits. In OVERWATCH [68], locations were picked based on whether certain learned patterns matched the code at those locations. The patterns that were matched against were selected conditioned on the past applications of associated edit patterns.

Edit Granularity: The edit granularity problem refers to the issue of defining what constitutes an "edit". We assume that we have heuristics to define when a local code change qualifies as a single edit. All changes between two versions that successfully parse can be used as a definition of a single edit, as in the work [46]. Another heuristic could be to combine all changes that occur within a small spatial vicinity of each other (in a commit) as a single edit [10].

Associated Edits: The associated edits problem seeks to find edits from the past history that would be most useful in predicting changes at the given locations L in the current version v_n of the source code file. Edits that are spatially close to the target locations L are likely relevant [10]. Similarly, edits that are temporally close – that is, edits that happened in the recent past – are also likely candidates for being relevant. We can use some combination of temporal and spatial proximity to obtain a candidate set of relevant edits [68]. For predicting updates on a target location, the temporally-proximal edits can indicate the developer's editing intent and the spatially-proximal edits can assist in providing meaning to the target snippet. We can even further selectively choose from the edits in the spatio-temporal vicinity of the target locations using the approach in a recent work that mines relevant edits based on their syntactic structure and their likelihood of occurring together [68]. We can use any or all of these approaches to construct the set of relevant edits. Our goal is to show that even when the relevant edits are *heuristically* generated, using them for associated code update predictions can be very beneficial.

3.2 Related Problem Formulations

Existing auto-regressive LLMs, such as, GPT3 and CODEX, predict completions for a given prompt. If the prompt contains the current version v_{n-1} of the artifact, then these LLMs predict text that is meant to be appended to v_{n-1} to generate the new version v_n. These models rely on the text in the spatial vicinity of the change-locations L to make predictions. In our terminology, these models are modeling the probability $P(v_n \mid v_{n-1})$. This is clearly different from the problem we are considering. We demonstrate that the associated code update formulation yields a simple yet effective way of improving LLM performance on software development tasks.

The EditCompletion task in [10] is formalized as a study of $P(\delta_{2,3} \mid L, v_2, \delta_{0,1}, \delta_{1,2})$ where the two given edits are edits performed in the spatial vicinity of the current location, one before and one after the current location. The EditCompletion task does not consider relevant edits that may be spatially distant. Our problem formulation is a generalization of EditCompletion problem, and in fact, we use the benchmarks from [10] for evaluation. As discussed in the introduction, our approaches are different too.

The edit likelihood prediction problem [55] explicitly considers the study of $P(v_n \mid v_0, v_1, \ldots, v_{n-1})$, but it uses $P(\delta_{n-1,n} \mid v_0, v_1, \ldots, v_{n-1})$ as a way to estimate the former. This problem differs from the associated code update problem in two ways: first, it includes the sub-problem of finding the locations L that need to be edited as part of the larger problem, and second, it considers the entire edit history as an ordered sequence (in an auto-regressive way) whereas we focus on a small set of associated edits.

4 EXPLOITING ASSOCIATED EDITS

We propose GRACE, a technique to use pre-trained language models for solving the associated code update problem. There are two possible ways of using these models to perform the associated

code update task. One approach is based on using the models as a black-box, but with carefully designed prompts (Section 4.2). This prompting strategy works for big LLMs, such as GPT3 and CODEX. The second approach is based on fine-tuning a pretrained language model, CODET5 in our case, for our specific associated code update task (Section 4.3).

4.1 Pre-trained Language Models

There is now a large collection of pre-trained language models. These models are pre-trained on data collected from millions of webpages, and treat all data as a sequence of tokens, which is the natural choice for representing natural language text [50, 52, 53]. The reason for the popularity of this class of models is that they exhibit ability to perform multiple different tasks with just some instructions and zero examples (zero-shot task transfer) even though they are not explicitly trained for these tasks.

It was observed that pre-trained LLMs are not as effective when working with code because code has strict syntactic and semantic correctness requirements. Different representations for code and code edits have been developed and models have been trained to work with those representations [5, 10]. However, as the size of pretrained language models has grown, their zero-shot performance across tasks has improved. Moreover, these models have also shown the ability to perform a new task given just a few demonstrations in the prompt (few-shot learning) [11]. Using their zero-shot and few-shot learning capabilities, these models are now being used successfully on tasks that involve understanding, manipulating, or generating code [16, 28, 63] while still viewing code just as text (and not as an abstract syntax tree, for example).

4.2 Prompting LLMs

We experimented with a few different prompt designs and then fixed one for our experiments. (The results were not significantly different for other reasonable prompt designs.) Before we describe the GRACE prompt, we first describe the completion, insertion, and editing variants of the CODEX family of models [8, 16].

The CODEX family of models is available in the "completion", "insertion" and "editing" variants. The completion model takes a *prompt*, which usually contains code before a cursor location, and predicts the code that will follow that prompt. Apart from the prompt, the insertion model also takes a *suffix* prompt, which usually contains the part of code that should come *after* the code the model predicts. Thus, the insertion models perform the infilling task - predict the code that should come after the prompt but before the suffix. Finally, the editing variant of the codex models has two different input prompts: an *input* that is the string that needs to be edited, and an *instruction* that tells the model how to edit the input.

We treat the associated code update task as an infilling problem and hence use the CODEX INSERT family of models for our experiments. The reasons for this choice are as follows:

(1) The insertion model allows us to include code that is spatially after the target location in the suffix.
(2) The editing variant (CODE-DAVINCI-EDIT-1) requires instruction on how to edit the given piece of code. Our experiments with providing the associated edits in this instruction prompt failed to generate good results. This is possible because the editing model is better suited only for instructions given in natural language[1].

```
<CurrentEdit>
    <Prefix> . . . </Prefix>
    <Before> . . . </Before>
    <After>  . . . </After>
    <Suffix> . . . </Suffix>
</CurrentEdit>
<CtxEdits>
    <Edit>
        <Prefix> . . . </Prefix>
        <Before> . . . </Before>
        <After>  . . . </After>
        <Suffix> . . . </Suffix>
    </Edit>
    <Edit>...</Edit>
    . . .
</CtxEdits>
```

Figure 2: GRACE Prompt for the associated code update task.

Figure 2 shows the GRACE prompt we provided CODEX models for the associated code update task. Let v_{n-1} be the current version of the file, L be the locations where code needs to be updated, and $\delta_{0,1}, \delta_{1,2}, \ldots, \delta_{n-2,n-1}$ be the $n-1$ associated edits. We assume that each edit $\delta_{i-1,i}$ can be partitioned in four parts: (1) `<Prefix>` , which contains the fragment of code in version v_{i-1} that is untouched by the edit, but occurs before the edited code, (2) `<Before>` , which contains the fragment of code in version v_{i-1} at locations L that is replaced by the edit, (3) `<After>` , which contains the fragment of code in version v_i at locations L in place of *before* in v_{i-1}, (4) `<Suffix>` , which contains the fragment of code in version v_{i_1} that is untouched by the edit, but occurs after the edited code. These four parts are included in the prompt for each edit as shown in Figure 2. The associated edits are all included within the `<CtxEdits>` tag. The edit to be predicted is included inside the `<CurrentEdit>` tag.

In this prompt format, the current edit is written out first followed by the associated edits. This style ensures that if the prompt gets bigger than what can fit in the input to the model, the tokens from the associated edits are pruned. We also experimented with variants where certain associated edits were placed before the current edit and some after depending on where they occurred spatially. Most such changes did not cause any significant change in our experimental observations.

The insertion model is expected to predict the string that should occur between `<After>` and `</After>` that occurs under `<CurrentEdit>` . The prefix of the prompt string up until `<After>` goes in the prompt, and the suffix of the prompt string starting from `</After>` is included in the suffix prompt of the insertion model.

The prompt design above is reminiscent of few-shot learning prompts where the prompt contains a few examples of the task to be performed. Technically speaking, the above prompt is not a few-shot prompt since we are not providing one or more examples of the "associated code update task". However, if we view the associated code update problem as a *means of providing few-shot examples for the "code update task"*, then a natural question is whether associated edit update task can just be viewed as a few-shot prompting for code update task. We answer this question in Section 5.

One of the central goals of the paper is to find how using associated edits compares with not using it when predicting code updates. To enable this comparison, we need a prompt for the case when

[1]We do not include CODE-DAVINCI-EDIT-1 in our experiments

```
<CurrentEdit>
        <Prefix> . . . </Prefix>
        <Before> . . . </Before>
        <After> . . . </After>
        <Suffix> . . . </Suffix>
</CurrentEdit>
```

Figure 3: Prompt when associated edits are not used.

Table 2: The models used in our experiments.

Name	Base Model	Fine-tuned on
CODEX-DAVINCI	CODE-DAVINCI-002	-
CODET5-U	CODET5-BASE	unfiltered C3PO train
CODET5-UF	CODET5-U	filtered C3PO train
CODET5-UO	CODET5-U	OVERWATCH train

associated edits are unavailable. Here, we use the prompt shown in Figure 3. Specifically, we remove the <CtxEdits> section in Figure 2. Note that the current code context is still available to the model in the <Prefix> and <Suffix> tags within <CurrentEdit> .

4.3 Fine-tuning LLMs

We now describe how we create fine-tuned models for predicting code updates *with and without* associated edits. We started with the CODET5-BASE model [39, 63], a pre-trained encoder-decoder Transformer model. This base model was trained on CodeSearchNet [33] that contains source code in 6 common programming languages, extended with two additional C/C# datasets from BigQuery [29]. We further fine-tuned several variants of this model on the task of predicting code edits (see Table 2). There are two versions of each variant – one that is fine-tuned using the given associated edits and one that only uses the current version of the code. The two types of fine-tuning use the same dataset and base model weights, the only difference being how the data was prepared. The variants are discussed in detail in Section 5.1.

We prepare data for fine-tuning by turning each training example into the GRACE prompt, as shown in Figure 2. We adapt the CODET5 tokenizer by adding special tokens: <Prefix>, </Prefix>, <Suffix>, </Suffix>, <CurrentEdit>, </CurrentEdit>, <CtxEdit>, </CtxEdit>, <Edit> </Edit>, <After>, </After>, <Before>, </Before> . We formulate the training as a masked span prediction task where we replace the contents between <After> and </After> under <CurrentEdit> with a sentinel token and ask the model to predict the masked span.

When fine-tuning CODET5 to predict code update *without using associated edits* , we use the prompt shown in Figure 3. Again, we formulate the training as a masked span prediction task replacing the contents between <After> and </After> under <CurrentEdit> with a sentinel token and asking the model to predict the masked span.

4.4 Deployment

We now discuss how the sub-problems discussed in Section 3.1 can potentially be solved and integrated with our approach to create an IDE-based edit prediction tool:

Setup: As discussed in Section 3.1, the editing target could be the line corresponding to the user's cursor location. OVERWATCH can be used to extract temporal edits from patterns that match the target location and these edits can serve as our associated edits.

Table 3: The datasets used for fine-tuning and testing.

Dataset	#training	#eval	#test
C3PO filtered	39.5K	4.4K	5.9K
C3PO unfiltered	1.67M	180K	210K
OVERWATCH	9K	1k	1K

Worklow: Consider a user editing code in an IDE. The tool will get triggered on the line where the user's cursor resides and the associated edits would be retrieved using OVERWATCH. Our edit prediction prompt will be generated as discussed in Section 4.2. The prompt will then be sent as an input to an LLM (say, CODEX-DAVINCI) and the predicted edit (or top-k predicted edits) will be suggested to the user. We have designed an interactive tutorial to walk readers through this workflow using the various examples discussed in Section 2 and Section 5.6 (see Section 9 for instructions).

5 EXPERIMENTS & RESULTS

5.1 Experimental Setup

We use two datasets from prior work for our experiments, the C3PO dataset [10] and the OVERWATCH dataset [68]; see Table 3.

C3PO Dataset: The C3PO dataset [10] was created by scraping all *commits* in 53 most popular C# GitHub repositories. Each edit in a commit would create a single example, and the edits, if any, on the 10 lines above and 10 lines below the edit would make up the associated edits. The task is to predict the code after an edit is performed, given the code before the edit and the associated edits. Thus, the C3PO dataset is an instance of the associated code update task, where *spatial locality* is used to define associations between edits. Note that the C3PO paper refers to these edits as *contextual edits* which translate to *edits with spatial associations* in our work.

The C3PO dataset was further filtered by its creators into a *filtered C3PO dataset* by removing "simple" benchmarks (e.g. those containing only deletion or renaming). Further, they removed all benchmarks where the target edit involved insertion of *new code* as their approach cannot handle those. The filtered set was further partitioned into train, validation, and test benchmarks, containing respectively 39.5K, 4.4K, and 5.9K benchmarks; see Table 3. We used the same partitions in our evaluation.

OVERWATCH Dataset: The dataset described in [68] was gathered from versions of source code files taken as they were being edited in an IDE session over two separate periods. In the first period, 134.5K versions were collected over 682 sessions. In the second period, 201.1K versions were collected over 399 sessions. The versions in the first period were mined in [68] to get a set of *9.9K* edit sequences which are further used to learn a collection of symbolic rules representing commonly occurring Edit Sequence Patterns. These learned rules are used to generate code suggestions in the second period, and they are found to be capable of producing suggestions at *1048* file versions. For the purpose of this work, we are considering 90% of the *9.9K* edit sequences from the first period as the *OVERWATCH training* set, keeping other 10% as the *OVERWATCH evaluation* set; and the points of applications as the *OVERWATCH test* set. We discuss these datasets further in Section 5.5.

Models: We used two models as starting points. The first is CODE-DAVINCI-002 (referred to as CODEX-DAVINCI in this text), a

decoder-only transformer model which is a part of the OpenAI GPT-3.5 series [8, 16]. This model is presented with the prompts based on either Figure 2 or Figure 3 depending if we want to use use associated edits for edit prediction. The second model is CODET5 [39, 63], an encoder-decoder model introduced by Salesforce in the BASE and LARGE Variants. We use the CODET5-BASE variant which has 220M parameters with 12 transformer blocks in the encoder and decoder each. This model is fine-tuned on the unfiltered C3PO training dataset to create the model CODET5-U. The model CODET5-U is further fine-tuned on the C3PO filtered train set and OVERWATCH train set to create CODET5-UF and CODET5-UO, respectively; see Table 2. We used this two step fine-tuning process since the OVERWATCH training data was limited. The fine-tuning and inference setups for these models are described below.

Setup for CODEX-DAVINCI Experiments: We used the OpenAI public API to perform the inference experiments with the CODEX-DAVINCI model. The `insert` mode of the model was used and the input was divided into *prompt* and *suffix* following our prompting strategy discussed in Section 4.2. Temperature sampling was used to generate n=5 predictions, and the temperature was set to 0.1 after evaluating multiple candidate values. The maximum length (maximum number of tokens to generate) was set to 256, stop token to `</After>` and default values were used for all other parameters.

Setup for CODET5 Experiments: For our fine-tuning experiments, we use a virtual machine with 16 AMD MI200 GPUs (each with 64GiB of vRAM), 92 CPU cores and 1594 GB of RAM. We set the input token length to 1024 tokens and truncate any longer inputs from the end. There are two steps in our fine-tuning process: fine-tuning on the unfiltered C3PO dataset followed by dataset-specific fine-tuning on the OVERWATCH training and C3PO filtered datasets. For the initial fine-tuning with the unfiltered C3PO dataset, we initialize the model with the publicly released CODET5-BASE weights and train it for 8 epochs with a batch size of 8 per device. The optimization is done using the Adafactor[59] optimizer with learning rate initially set to $3e^{-4}$ and gradually updated using a linear scheduler after a warmup of 500 steps. The best model weights are determined using the perplexity score by evaluating on the C3PO validation dataset at every 1000 steps. For further fine-tuning on the OVERWATCH training and C3PO filtered datasets, we set the initial learning rate to $1e^{-4}$, the number of warmup steps to 50 and train the model for 10 epochs while evaluating it every 50 steps. During inference, we use beam search with a beam width of 5.

Metric: In order to stay consistent with the metrics used by papers that curated the target datasets (namely the C3PO and OVERWATCH datasets), we define a metric called the *exact match*. In the experiments with the C3PO dataset, a prediction is said to be an *exact match* if it syntactically matches the ground truth modulo whitespaces. We use *Exact Match* to also denote the percentage of cases where a prediction was an exact match. More details on the OVERWATCH dataset evaluation can be found in Section 5.5. In all our results, we report *Exact Match* for Top-1 predictions.

5.2 GRACE Improves Prediction

A key question we set out to answer was whether associated edits help predict future code changes. In other words:

RQ1. Does availability of associated edits improve code update predictions? Does the answer depend on the prediction approach?

Table 4: Associated edits improve code prediction.

Model	C3PO test set		OVERWATCH test set	
	Without assoc. edits	With assoc. edits (GRACE)	Without assoc. edits	With assoc. edits (GRACE)
CODEX-DAVINCI	37.09	**67.92**	31.81	**49.09**
CODET5-U	64.52	**74.16**	22.25	**34.00**
CODET5-UF	73.46	**81.83**	40.78	**48.23**

To answer this question, we tested both CODEX-DAVINCI and CODET5 on both the C3PO and OVERWATCH test sets, once with associated edits in the prompt and once without them.

Results: Table 4 shows the Exact Match obtained when we use the different models on the different datasets with and without associated edits. We see that CODEX-DAVINCI shows a 30% absolute increase in Exact Match when provided associated edits than when not on the C3PO dataset, and about 17% absolute increase on the OVERWATCH dataset. The fine-tuned CODET5 models showed about a 10% absolute increase in Exact Match on both datasets. Finally, although Table 4 reports the trend for 2 models and one prompting style, we tried other models (including other OpenAI models from GPT3 and GPT-3.5 series) and different styling of the prompts (for example, using C# comments, rather than tags, to delineate the "before" and "after" versions), and in every case, there was at least a 10% absolute increase in Exact Match – often it was much higher.

Result 1: *Conditioning code prediction on associated edits helps, across models and test datasets.*

5.3 Relevance of Edits Matters

The associated code update problem conditions code prediction on some *associated* edits. We have informally mentioned that the associated edits should be picked based on their relevance to the code that is being updated. Our next research question is concerned with how relevance impacts prediction.

To motivate this research question, we first make the connection to "few-shot prompting". Consider just the *code update task* – predict the new version of the code given its old version. The difference between the "code update task" and "associated code update task" are the associated edits. Now, a prompt containing an instance of the "associated code update" task begins to look a lot similar to a *few-shot prompt for a code update task* where the *associated edits* serve the purpose of few-shot examples of code update.

It may be tempting to say that the "associated code update" task just combines some few-shot examples with a code update task. However, this view is not beneficial since associated edits are more than just *any examples of code updates*. As discussed in Section 2, the associated edits contain crucial information for performing the given code update. To validate that associated edits are more than just code update examples, we turn to our next research question:

RQ2. Are associated edits important for code update prediction, or simply serve as few-shot examples for the code update task?

In other words, is there something to be gained by using associated edits beyond what we gain by just adding some few-shot examples of code updates (that are not necessarily associated)?

Results: Table 5 shows the *Exact Match* we get using the CODEX-DAVINCI model using different sets of edits as the "associated edits".

Table 5: Less relevant edits degrade prediction: CODEX-DAVINCI on filtered C3PO test set for different associated edits.

Choice of Associated Edits			Exact Match
Association	Dataset	Repository	
Spatial	Filtered	Same	**67.92**
Random	Filtered	Same	64.90
Random	Filtered	Other	55.82
Random	Unfiltered	Same	43.23
Random	Unfiltered	Other	43.64
No Associated Edits			37.09

We use the C3PO filtered test set again for evaluation. We saw before that we get a 67.92% Exact Match on this test set (Table 4). This case corresponds to when the prompt includes spatially close filtered edits from the same file (this is a property of the C3PO benchmarks). Let us now randomly sample edits to include in the prompt. There are two dimensions and two buckets in each dimension to use for sampling: the filtered dataset versus the unfiltered dataset, and edits from the same repository versus edits from different repositories. Randomly picking filtered edits from the same repo drops performance only slightly. However, randomly picking filtered edits from other repos drops performance more significantly to 55.82%. When sampling from unfiltered edits - irrespective of whether edits are from the same or different repos - the Exact Match remains consistently around 43%. We recall that when we provide no associated edits in the prompt, we had 37.09% Exact Match (Table 4).

The results show that going from filtered to unfiltered edits reduces relevance of edits to the target *filtered edit*. This is because the filtering step in [10] actually removes certain kinds of edits; for example, edits that are pure insertions or deletions, or edits that result in unparseable code. Hence, a randomly picked unfiltered edit is more likely to be structurally different from our target edit (which was picked from the filtered test set.)

The results also show that picking edits from repositories other than the repository of the target edit reduces relevance of the edit to the target edit. This is because edits from the same repository could potentially be using common concepts, classes, methods, programming practices, and even contain similar changes.

Finally, we note that using unfiltered edits from other repositories (43.64%) is still better than not using them (37.09%). This is possibly due to the LLM leveraging its *few-shot* learning capabilities in that case. The gain from around 43% to around 68% can thus be attributed to the associated edits. We can, therefore, conclude that:

Result 2: *Associated edits play a crucial role in predicting a target edit, and the Exact Match metric drops as the relevance of the edits to the target edit drops.*

5.4 Pre-trained Outperforms Custom

When working with code and code edits, LLMs (such as CODEX and GPT3) and other pre-trained models (such as CODET5) use byte-pair encodings (BPE) to tokenize code and then represent code as a sequence of tokens – in the same way as Natural Language is represented. In contrast, some works have argued for the use of custom representations for code and code edits that partly capture the parse structure and/or the programming language semantics. The paper that introduced the C3PO dataset [10] also used the spatial edits used by our pre-trained LLMs but they learned a custom model employing code-centric representations for code edits. Our next research question concerns comparing our approach based on pre-trained models with prior work on custom neural approaches. While both the approaches have access to the associated spatial edits, we want to understand how pre-trained models with their text-based prompts compare against models with custom code representations.

Table 6: Comparison with C3PO.

Model	Exact Match on C3PO	Exact Match on OVERWATCH
C3PO	53.20	10.50
CODEX-DAVINCI	67.92	**49.09**
CODET5-U	74.16	34.00
CODET5-UF/ CODET5-UO	**81.83**	48.23

RQ3. How does our LLM-based approach compare with the C3PO approach based on a custom neural model on the associated code update task?

Let us compare how the C3PO custom neural model performs in comparison to CODEX-DAVINCI and fine-tuned CODET5. We first compare these models on the filtered C3PO test set and the OVERWATCH test set. Table 6 shows that both CODEX-DAVINCI and fine-tuned CODET5 significantly outperform the custom C3PO model on both test datasets. The C3PO model was reported to give a 53.2% accuracy [10] on the C3PO test set, whereas both CODEX-DAVINCI and fine-tuned CODET5 give better results. The CODET5-UF model gives 81.83% Exact Match, which is significantly higher than 53.2% achieved by the C3PO model. Similarly, on the OVERWATCH dataset, the best possible configuration of C3PO was reported to give 10.5% Exact Match [68], whereas all of CODEX-DAVINCI (49.09%), CODET5-U (34%), and CODET5-UO (48.23%) perform significantly better.

Comparison on Unfiltered C3PO Test Set: The C3PO model does not report results on the unfiltered C3PO dataset. This is partly because it contains benchmarks that are out of scope for their technique. Two such notable benchmarks are: (a) benchmarks that contain *alien insertions* where the inserted code contains tokens that do not occur in either the associated edits or the current version of the target code snippet, and (b) benchmarks that contain code snippets that cannot be parsed by an underlying parser (this step is important for C3PO to generate the Abstract Syntax Tree (AST)). GRACE can handle both these classes of benchmarks. We evaluated CODEX-DAVINCI on a 5.9K random sample from this test set and obtained a 43.47% Exact Match. These 5.9K samples did not contain any benchmarks from the filtered set. (We used a sample because of the cost of doing inferences using an LLM.) On the *full* unfiltered C3PO test set, CODET5-U has 57.3% Exact Match using GRACE and 45.30% without. These numbers are lower than those for the filtered C3PO test set. This indicates that the unfiltered benchmarks are more challenging than the filtered benchmarks, which is at odds with the informal assertions to the contrary in [10].

Alien Insertion Benchmarks: We extracted the samples from unfiltered C3PO test set that involved alien insertions. On that set, CODET5-U with GRACE achieved 17.6% exact match, but only 10.28% without it. The CODEX-DAVINCI model achieved 17.37% exact match with GRACE and 10.67% without it. This indicates that conditioning on associated edits can help with hard benchmarks.

Table 7: Comparison with OVERWATCH.

Technique	OVERWATCH	CODEX-DAVINCI	CODET5-U	CODET5-UO
Exact Match	38.17	**49.09**	34.00	48.23

Admittedly, the C3PO model is much smaller (750K parameters) compared to both CODEX-DAVINCI (175B) and CODET5 (220M). However, these large models are pre-trained and hence they can be quickly fine-tuned or prompt engineered for downstream tasks without need for excessive training data. Furthermore, the pre-trained models are not limited in scope, as we have discussed above.

Result 3: *Pre-trained language models can be tuned to yield higher Exact Match compared to the custom C3PO model for associated code update prediction.*

5.5 Temporal Edit Prediction

The temporal edit prediction problem is an application that is well-suited for using GRACE. The state of the art in this application domain is OVERWATCH [68]. Fundamentally, OVERWATCH is solving a different problem from the associated edits prediction problem: the input to OVERWATCH are *fine-grained IDE version histories* of the form $v_0, \ldots, v_n$ where each v_i is a version of the source code file. The edit histories are extremely fine-grained, at the keystroke unlike source control histories. For example, if a developer types a variable name `predicate`, each intermediate file version containing the prefixes `p`, `pr`, ... is present in the edit history.

The OVERWATCH technique takes the set of such IDE version histories as a training set, and produces a ranked sequence of edit sequence patterns (ESPs). At inference or run-time in an IDE, each ESP examines the current version history $v_0, \ldots, v_n$ and (a) identifies a sequence of transitive coarse-grained edits $\delta_{i_0,i_1}, \delta_{i_1,i_2}, \ldots, \delta_{i_{k-1},i_k}$ (i.e., each δ_{i_0,i_k} is the edit between the potentially non-consecutive versions v_{i_j} and $v_{i_{j+1}}$), and (b) uses these edits to predict the next edit to v_{i_k}. In short, the ESPs are doing two tasks: (a) identifying "associated edits" from fine-grained version histories, and (b) using these associated edits to predict the next edit.

RQ4. Can our LLM based approach be used in conjunction with OVERWATCH's temporal associated edit identification? How does it compare with OVERWATCH's symbolic edit prediction component?

The second task above is exactly the prediction from associated edits problem we are tackling in this paper. Hence, we run OVERWATCH on its test data of 399 version histories with over 200,000 versions, and gather the associated edits wherever the ESPs are able to identify them. This results in a dataset of 1048 cases as mentioned in Section 5.1. At training time, OVERWATCH identifies a set of 9.9K edit sequences from the older data of 682 version histories, however using different techniques. The edit sequences are such that each of them belong to some commonly occurring edit sequence pattern across version histories – they are the supports for ESPs – and thus each of them can be treated as a set of associated edits (all edits in the sequence but the last), and expected edit prediction (last edit in the sequence). We use this set of 9.9K instances to further fine-tune CODET5-U to obtain CODET5-UO; see Table 2.

Table 7 summarizes the different models' performance on the 1048 test cases, along with OVERWATCH's predictions as a baseline.

```
1  catch (Exception ex)
2  {
3  -  info.ReportClientError('Scheme is missing');
4  +  info.ReportClientError('Scheme is missing',System.Net.
       HttpStatusCode.BadRequest);
5  }
6  default:
7  -  info.ReportClientError('No such action');
8  +  info.ReportClientError('No such action',System.Net.
       HttpStatusCode.NotFound);
```

Figure 4: User adds `BadRequest` error code on Line 3 and moves to Line 5 where we should predict inserting `NotFound`.

Except CODET5-U, all of our models beat the prediction component of OVERWATCH by a considerable margin of roughly 10%.

Result 4: *Our LLM-based techniques, in conjunction with systems like OVERWATCH creates neuro-symbolic solutions that are better at predicting next edit compared to purely symbolic techniques.*

5.6 Qualitative Analysis

Our experiments support two major observations: (a) LLMs can predict edits that existing techniques fundamentally cannot support, and (b) the addition of associated edits improves the performance of LLMs on the task of predicting code edits. Next, we provide insights into why these observations hold true.

5.6.1 Comparison with existing techniques. In the following few paragraphs, we discuss the salient features of LLMs and the GRACE prompt design that help our approach outperform existing techniques, i.e., C3PO and OVERWATCH on certain kinds of edits.

Generative Capabilities of LLMs are Useful in Predicting Alien Insertions: As discussed in Section 2, the LLMs we discuss in this paper can support most forms of insertions as they have access to a wide number of tokens through their pre-training and our prompting setup doesn't restrict the tokens that the models can generate. Existing techniques are restricted in this aspect by design: *C3PO cannot insert tokens other than those found in the contextual edits and Overwatch may learn patterns where the prediction template is incomplete due to unavailable mappings for holes in the Temporal Edit Pattern.* For instance, consider the scenario in Figure 4 where a developer is trying to add HTTP error codes to error reporting calls. Here, the developer first edits Line 3 by adding a `BadRequest` error code to the reporting call. They then move to Line 5 to make a similar edit. Note that the expected error code on Line 5 is different from the one on Line 3 as it corresponds to a "No such action" error message. Moreover, the expected error code has a token 'NotFound' which is not present anywhere in the existing context. As C3PO's pointer network can only pick paths to/from existing nodes, it cannot generate this new token. CODEX-DAVINCI with GRACE can correctly predict this edit.

Access to Local Spatial Context in the Prompt is Useful: OVERWATCH learns patterns and templates from observed edit sequences and strictly relies on these patterns to make predictions. There are cases, however, where the pattern learnt by OVERWATCH is too general and is applicable irrespective of what is in the spatial vicinity of the target edit. To better understand this limitation, consider the scenario in Figure 5 from an active IDE editing session. The user first replaces `ex` on Line 5 with `ex.Output` and then moves to Line 3 to make the next edit. Overwatch gets triggered on Line 3

```
1  foreach (var ex in currentExamples)
2  {
3  -  Console.WriteLine(GetText(ex, diff.BeforeFile));
4  +  Console.WriteLine(GetText(ex.Input, diff.BeforeFile));
5  -  Console.WriteLine(GetText(ex, diff.AfterFile));
6  +  Console.WriteLine(GetText(ex.Output, diff.AfterFile));
7  }
8  var output = Run(currentExamples.First().Input);
9  AssertEqual(currentExamples.First().Output, output);
```

Figure 5: User replaces `ex` on Line 5 by `ex.Output`, moves to Line 3 where we should predict replacing `ex` by `ex.Input`.

and predicts that `ex` should be replaced by `ex.Output` since it learns the pattern "repeat the same replacement", which is an instance of a common pattern. However, this is incorrect since `ex` should be replaced by `ex.Input` here. CODEX-DAVINCI, with GRACE, correctly predicts this edit because `ex` on Line 3 is followed by `diff.BeforeFile` and Line 8 has additional information about the property `Input` that is associated with each entry in 'currentExamples'. Our prompt design allows flexible addition of this additional spatial context through the `<Prefix>` and `<Suffix>` tags. OVERWATCH, on the other hand, cannot access spatial context that is not already present in the learned template.

Language Based Pre-training is Useful in Identifying Semantic Editing Patterns: Scenarios in Figures 4 and 5 also highlight the ability of LLMs to predict patterns based on semantics of identifiers in the context. In Figure 5, CODEX-DAVINCI seems to understand that the relationship between `diff.AfterFile` and `diff.BeforeFile` would also reflect in the preceding argument (`ex.Output` and `ex.Input`, respectively). In Figure 4, CODEX-DAVINCI uses the signal from the `'No such action'` error message to correctly predict that the error code should be `System.Net.HttpStatusCode.NotFound`. Existing techniques such as C3PO and OVERWATCH rely on edit path analogies and symbolic editing patterns respectively to understand the editing intent. Without the use of a language-based pre-training component, it may be difficult to obtain the semantic understanding needed to perform the edits in Figures 4 and 5.

5.6.2 Benefits of using associated edits. We observed three key benefits of providing associated edits to LLMs:

Associated Edits Help in Clarifying the Editing Intent of the Developer: The illustrative example in Section 2 (Figure 1) showed that associated edits provide strong signals about the next edit that the developer intends to perform. In fact, without associated edits, CODEX-DAVINCI doesn't predict the right edit even in the top-5 results. With associated edits, the correct prediction is ranked at the top suggesting that associated edits help improve the top-1 performance of the model.

Associated Edits Emphasize Relevant Code Context: While LLMs like CODEX-DAVINCI can support a large number of tokens in their prompts (4K in CODEX-DAVINCI's case), it has been observed that irrelevant information in the prompt affects the model's ability to attend to the right set of tokens [60]. In the illustrative example in Figure 1, the target edit is 247 lines away from the required spatial context. CODEX-DAVINCI can predict the right import with only 4-5 lines in the spatial context and access to the associated edit. The scenario without associated edits, on the other hand, requires providing 250 lines of mostly irrelevant code to the model to include the Exception that the required import provides. CODEX-DAVINCI fails to generate the right prediction in the top-5 results even with all of this spatial context. On a simpler version of this example where the relevant code context is moved closer to the target edit (from Line 250 to Line 15), CODEX-DAVINCI without associated edits predicts the right import in top-10, but it is not the top-1 prediction.

Associated Edits Contain Information about Edited Code Elements: There may be key variables that are deleted or replaced by previous edits but referenced by the target code location. Without access to these associated edits, the model has no context about these variables, methods or other code elements. For example, if a variable `var1` is replaced by `var2` in a previous edit and the developer now moves to line `var1 = var1 / 2`, the model is expected to replace this line with `var2 = var2 / 2`. Without access to the previous edit, the model doesn't know the relationship between `var1` and `var2` and may consider them to be two distinct variables.

5.7 Additional Results & Discussion

We conducted additional experiments to understand how GRACE affects *robustness* and *entropy* during prediction. We also evaluated other prompting styles and model configurations. See the technical report for details and further discussion.

6 RELATED WORK

6.1 Automatic Code Editing

In recent years, there has been a significant boom in academic and industrial research for automating developers' code editing activities. Most modern IDEs [25, 45] support automated code changes like the addition of boilerplate code, developer-assisted refactoring, etc. While these developer-assisted approaches tremendously help boost productivity [47], a significant amount of further research exists in automated code editing aimed at learning code edit patterns from developer's previous edits [6, 10, 13, 17, 27, 48, 49, 54, 55, 61, 67, 68]. We divide these approaches into two orthogonal directions:

Symbolic Approaches: Symbolic approaches learn the code transformation patterns by representing the example edits with symbolic abstractions. Given a set of such symbolically represented abstract edits, these approaches generalize the edit patterns as a sequence of edit operations. For instance, Refazer [56] represents syntactic changes with Domain Specific language and uses a deductive inference algorithm to generalize and synthesize common edit patterns. More recently, Overwatch [68] learns to generalize developer code editing behavior from a sequence of code versions. Each edit is represented as *pre* and *post* program states, and generalized edit sequences are derived from an edit graph from these state pairs. While the earlier works in symbolic editing [24, 34, 42, 43, 56] primarily focused on syntactic editing, i.e., refactoring, similar to OVERWATCH [68], we also focus on semantic changes in code. Similar to OVERWATCH, we emphasize on conditioning future edit *w.r.t.* associated edits. However, unlike OVERWATCH, GRACE does not necessarily need demonstrations of the specific edit sequence pattern to learn to apply that pattern.

Neural Network Based Approaches: Recent advancements in machine learning and neural networks have catapulted the field of code editing with Neural Networks relying on their noise tolerance and generalization capabilities. As such, several approaches [10, 13, 17, 21, 62, 67] have been proposed over the years using different types of Neural Networks for automatically generating edits. Notable among these are Sequence to Sequence Neural

Machine Translation based approaches [15, 17, 62], Tree to Tree translations approach [13], and Graph Neural Network based approach [21, 65]. While most of these approaches learn to generalize code edit patterns from seemingly unrelated example edits, this work shows the importance of related associated edits. Nevertheless, the most notable feature of Neural Code Editing approaches is how the approach generates the edited code. While some approaches [10, 21, 67] generate a script of edit operations (*i.e.,* insert, delete, update), others [13, 17, 62] generate the edited code applying the edit pattern in the process of translation. Similar to the latter approach, we generate the edited code given the code before the edit.

6.2 Deep Learning for Source Code

Recent advancement in Deep Neural Networks (DNN) has drawn focus on the application of such in different source code understanding and generation tasks, including bug detection [14, 22], code comprehension [1], code search [12], code generation [63], code translation [3, 38], program repair [17, 62], etc. The vast plethora of DNN models in SE tasks ranges from general-purpose models [2] inspired by Natural Language Processing to custom-built models for source code modeling [21, 30, 65]. These models, however, require a large quantity of labeled data to optimize millions of parameters. To overcome this problem, researchers have proposed to *pre-train* models with a large quantity of unlabelled data and subsequently re-use such a pre-trained model across different tasks [20, 51]. There are a wide variety of pre-trained models for source code proposed over the years [2, 16, 23, 63], some containing hundreds of billions of parameters [16], colloquially known as large language models or LLMs. LLMs show excellent promise in autonomously learning programming language properties and additionally, have shown the ability to learn deductive reasoning inherent in programming and natural languages [41, 57, 64, 66]. As such, these LLMs are leveraged in many industrial developer assistance tools such as GitHub Copilot [28], Amazon CodeWhisperer [58], Intellicode Compose [18, 44], etc. In this work, we show an in-depth investigation of harnessing the power of these LLMs for automated code editing.

7 LIMITATIONS & THREATS TO VALIDITY

Limitations: There are certain limitations of our approach that we would like to address in future work. Firstly, as our approach depends on other edit mining techniques, it is restricted by the quality of the collected edits. On rare occasions, associated edits in the prompt can also mislead the model with some irrelevant information which in turn leads to incorrect predictions. Moreover, our approach can also fail when the ground truth requires knowledge of certain context (method signatures, for example) that does not appear in the associated edits. Secondly, the LLMs used are prone to known issues such as hallucinations, generation of uncompilable code, etc. Despite being generative, these models can still fail to predict edits that involve generating entirely new code.

Threats to Validity: When using a pre-trained model, there is always a threat of test data leaking to the train set [4]. It is possible that the data used for pre-training CODEX-DAVINCI contained some or all of the data in the C3PO test set since the C3PO dataset was created from GitHub repositories. One way to mitigate this threat is to perform evaluation on multiple test sets. Therefore, we also performed our evaluation on the OVERWATCH dataset. The OVERWATCH test set was not publicly available and we obtained it directly from the authors. Hence, we believe our results are not inflated because of the possibility of CODEX-DAVINCI having seen the C3PO test set. We mitigated the threat further by performing the same experiments on fine-tuned CODET5. All conclusions we make in this work are informed by results from both models on both datasets. Finally, this potential data leak would affect all our experiment settings with the CODEX-DAVINCI model equally and any benefit would also have been available to the model without associated edits. Our results suggest that the model clearly benefits from the addition of associated edits thus entailing a fair comparison.

The test sets are another source of possible gap between what we observe in our experiments and what we may see if the approach were deployed in real world. The C3PO dataset was created from commits. It defined an edit at a certain level of granularity. This definition may not match the notion of edits used in some target application (of our code prediction models). Again, we mitigate this threat by also testing on OVERWATCH dataset that uses a different level of granularity for defining an edit. Our results appear to hold across the different possible notions of an "edit". In fact, by presenting the associated edits to the model (in the prompt and during fine-tuning), we are able to teach the notion of an edit to it. Even with the notion of edit conveyed, the distribution of associated edits in our test sets may not reflect what we observe in practice. The approach based on CODEX-DAVINCI is not immune to this threat, but the fine-tuning approach can adapt if we have fine-tuning data.

8 CONCLUSIONS & FUTURE WORK

Predicting code edits is an important software-engineering problem. In this paper, we leverage the generative capability of LLMs to address this problem. Without the knowledge of prior edits, the LLMs fail to predict the required edits, but when we combine them with associated edits, their performance improves greatly. This simple strategy is quite effective, and as shown in the experiments, GRACE outperforms the current state-of-the-art specialized symbolic and neural methods on their respective datasets.

The generative capability of LLMs has opened up many opportunities for addressing software-engineering problems that have been hard to deal with. We believe that combining the LLMs with domain-specific insights, such as our use of associated edits, holds promise for hitherto challenging problems. In the future, we shall seek to exploit this strategy for other software engineering problems. On the problem of predicting code edits, we plan to explore the problem of discovering associated edits, and the application to large-scale migrations, refactorings, and maintenance activities.

9 DATA-AVAILABILITY STATEMENT

The C3PO dataset is made publicly available by the authors of [10]. We share the scripts, prompts, and instructions to access the fine-tuned models on C3PO at `https://aka.ms/GrACE-Code`[31]. Since the OVERWATCH dataset is private, we do not hold the authority to redistribute the dataset or any models learned from that dataset. Readers with access to OVERWATCH data can reproduce the experiments using the shared scripts. An interactive tutorial notebook discussing deployment of our approach in an IDE-based edit suggestions tool is also available at the same webpage.

REFERENCES

[1] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics* (2020), 4998–5007.

[2] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* (2021), 2655–2668.

[3] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2023. Summarize and Generate to Back-translate: Unsupervised Translation of Programming Languages. *The 17th Conference of the European Chapter of the Association for Computational Linguistics (EACL 2023)* (2023).

[4] Miltiadis Allamanis. 2019. The Adverse Effects of Code Duplication in Machine Learning Models of Code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Athens, Greece) *(Onward! 2019)*. Association for Computing Machinery, New York, NY, USA, 143–153. https://doi.org/10.1145/3359591.3359735

[5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.* 3, POPL (2019), 40:1–40:29. https://doi.org/10.1145/3290353

[6] Jesper Andersen, Anh Cuong Nguyen, David Lo, Julia L Lawall, and Siau-Cheng Khoo. 2012. Semantic patch inference. In *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*. IEEE, 382–385.

[7] 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

[8] Mohammad Bavarian, Angela Jiang, Heewoo Jun, and Henrique Pondé. 2022. New GPT-3 Capabilities: Edit & Insert. (2022). At https://openai.com/blog/gpt-3-edit-insert.

[9] B.W. Boehm. 1976. Software Engineering. *IEEE Trans. Computers* 25, 12 (1976).

[10] 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..

[11] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. *CoRR* abs/2005.14165 (2020). arXiv:2005.14165 https://arxiv.org/abs/2005.14165

[12] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 964–974.

[13] Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. 2020. Codit: Code editing with tree-based neural models. *IEEE Transactions on Software Engineering* 48, 4 (2020), 1385–1399.

[14] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering* (2021).

[15] Saikat Chakraborty and Baishakhi Ray. 2021. On multi-modal learning of editing source code. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 443–455.

[16] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. https://doi.org/10.48550/ARXIV.2107.03374

[17] Z. Chen, S. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus. 2021. SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Transactions on Software Engineering* 47, 09 (sep 2021), 1943–1959. https://doi.org/10.1109/TSE.2019.2940179

[18] Microsoft Corp. 2022. *Overview of IntelliCode.* https://learn.microsoft.com/en-us/visualstudio/intellicode/overview

[19] 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

[20] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[21] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *International Conference on Learning Representations (ICLR)*.

[22] Yangruibo Ding, Luca Buratti, Saurabh Pujar, Alessandro Morari, Baishakhi Ray, and Saikat Chakraborty. 2022. Towards Learning (Dis)-Similarity of Source Code from Program Contrasts. In *Annual Meeting of the Association for Computational Linguistics*.

[23] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[24] Stephen R. Foster, William G. Griswold, and Sorin Lerner. 2012. WitchDoctor: IDE support for real-time auto-completion of refactorings. In *2012 34th International Conference on Software Engineering (ICSE)*. 222–232. https://doi.org/10.1109/ICSE.2012.6227191

[25] Eclipse Foundation. 2018. Eclipse IDE (https://www.eclipse.org). https://www.eclipse.org

[26] Martin Fowler. 2018. *Refactoring.* Addison-Wesley Professional.

[27] Xi Ge, Quinton L DuBose, and Emerson Murphy-Hill. 2012. Reconciling manual and automatic refactoring. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 211–221.

[28] github.com. 2022. *GitHub Copilot: Your AI pair programmer.* github.com. https://github.com/features/copilot

[29] google.com. 2022. *GitHub Acitvity Data.* google.com. https://console.cloud.google.com/marketplace/details/github/github-repos

[30] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).

[31] Priyanshu Gupta, Avishree Khare, Yasharth Bajpai, Saikat Chakraborty, Sumit Gulwani, Aditya Kanade, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. 2023. Reproduction Package for Grace: Language Models Meet Code Edits. https://doi.org/10.1145/3580411

[32] Anandi Hira and Barry Boehm. 2016. Function Point Analysis for Software Maintenance. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Ciudad Real, Spain) *(ESEM '16)*. Association for Computing Machinery, New York, NY, USA, Article 48, 6 pages. https://doi.org/10.1145/2961111.2962613

[33] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. https://doi.org/10.48550/ARXIV.1909.09436

[34] JetBrains. 2021. ReSharper. (2021). At https://www.jetbrains.com/resharper/.

[35] Capers Jones. 1998. *Estimating Software Costs.* McGraw-Hill.

[36] M. Kim, D. Notkin, D. Grossman, and G. Wilson. 2013. Identifying and Summarizing Systematic Code Changes via Rule Inference. *IEEE Transactions on Software Engineering* 39, 1 (2013), 45–62. https://doi.org/10.1109/TSE.2012.16

[37] Amy J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering* 32, 12 (2006), 971–987. https://doi.org/10.1109/TSE.2006.116

[38] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511* (2020).

[39] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. 2022. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. *arXiv preprint arXiv:2207.01780* (2022).

[40] M. M. Lehman and L. Belady. 1985. *Software Evolution–Processes of Software Change.* Academic.

[41] Christopher D Manning. 2022. Human language understanding & reasoning. *Daedalus* 151, 2 (2022), 127–138.

[42] 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.

[43] 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.

[44] Microsoft. 2021. IntelliCode suggestions. (2021). At https://devblogs.microsoft.com/visualstudio/intellicode-suggestion-apply-all/.

[45] Microsoft. 2021. Visual Studio. (2021). At https://www.visualstudio.com.

[46] 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. 3, OOPSLA, Article 143 (oct 2019), 29 pages. https://doi.org/10.

1145/3360569

[47] Raimund Moser, Pekka Abrahamsson, Witold Pedrycz, Alberto Sillitti, and Giancarlo Succi. 2008. A case study on the impact of refactoring on quality and productivity in an agile team. In *Balancing Agility and Formalism in Software Engineering: Second IFIP TC 2 Central and East European Conference on Software Engineering Techniques, CEE-SET 2007, Poznan, Poland, October 10-12, 2007, Revised Selected Papers*. Springer, 252–266.

[48] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N Nguyen, and Danny Dig. 2016. API code recommendation using statistical learning from fine-grained changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 511–522.

[49] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2014. Statistical learning approach for mining API usage mappings for code migration. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 457–468.

[50] Alec Radford and Karthik Narasimhan. 2018. Improving Language Understanding by Generative Pre-Training.

[51] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. (2018).

[52] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners.

[53] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *J. Mach. Learn. Res.* 21 (2020), 140:1–140:67. http://jmlr.org/papers/v21/20-074.html

[54] Veselin Raychev, Max Schäfer, Manu Sridharan, and Martin Vechev. 2013. Refactoring with synthesis. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 339–354.

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

[56] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 404–415.

[57] Christopher Rytting and David Wingate. 2021. Leveraging the inductive bias of large language models for abstract textual reasoning. *Advances in Neural Information Processing Systems* 34 (2021), 17111–17122.

[58] Amazon Web Services. 2022. *ML-powered coding companion - Amazon CodeWhisperer*. Amazon Web Services. https://aws.amazon.com/codewhisperer/

[59] Noam Shazeer and Mitchell Stern. 2018. Adafactor: Adaptive learning rates with sublinear memory cost. In *International Conference on Machine Learning*. PMLR, 4596–4604.

[60] Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed Huai hsin Chi, Nathanael Scharli, and Denny Zhou. 2023. Large Language Models Can Be Easily Distracted by Irrelevant Context. *ArXiv* abs/2302.00093 (2023).

[61] Wesley Tansey and Eli Tilevich. 2008. Annotation refactoring: inferring upgrade transformations for legacy applications. In *ACM Sigplan Notices*, Vol. 43. ACM, 295–312.

[62] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On learning meaningful code changes via neural machine translation. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 25–36.

[63] 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. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*. Association for Computational Linguistics, 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685 See also https://arxiv.org/abs/2109.00859.

[64] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903* (2022).

[65] 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].

[66] Eric Zelikman, Yuhuai Wu, and Noah D Goodman. 2022. Star: Bootstrapping reasoning with reasoning. *arXiv preprint arXiv:2203.14465* (2022).

[67] Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2023. CoditT5: Pretraining for Source Code and Natural Language Editing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 22, 12 pages. https://doi.org/10.1145/3551349.3556955

[68] 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

Received 2023-02-02; accepted 2023-07-27