GRACE: Language Models Meet Code Edits

Priyanshu Gupta†
priyansgupta@microsoft.com
Microsoft
India

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

Yasharath Bajpai
ybajpai@microsoft.com
Microsoft
India

Saikat Chakraborty
saikate@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, CoDaX 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

¹Both authors contributed equally to this work.
†Work done while at Microsoft

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ESEC/FSE ’23, December 3–9, 2023, San Francisco, CA, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9327-0/23/12...
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, 39]. Due to the high cost of software maintenance [31, 34], popular Integrated Development Environments (IDEs) have tooling to support developers as they refactor code [25, 26, 44], fix defects, adapt code to changes in the environment, or add support for new or changed requirements [33, 44]. 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 [43, 67].

To automate code edit suggestions, researchers have proposed several approaches to learn edit patterns from edits in source code repositories [7, 19, 35, 55]. 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 [36]. 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 [67]. 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 [62] 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, 54, 67]. Overwatch [67] 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 [36]. 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 [54] 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 the `System.Runtime.SerializationException` provided by the `System.Runtime.Serialization` namespace to get to Version $v_2$ 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 $Δ_{1,2}$. 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 $Δ_{2,3}$.

Conditioning on prior edits: The task of predicting the edit $Δ_{2,3}$ 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.SerializationException` 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
The developer performs edit $\delta_{1,2}$ to go from Version $v_1$ 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

<table>
<thead>
<tr>
<th>Technique</th>
<th>Prediction</th>
<th>Correct?</th>
</tr>
</thead>
<tbody>
<tr>
<td>c3po</td>
<td>No Response (alien insertion)</td>
<td>✗</td>
</tr>
<tr>
<td>overwatch</td>
<td>No Response (no matching pattern)</td>
<td>✗</td>
</tr>
<tr>
<td>code-davinci-edit-1</td>
<td>Empty Response</td>
<td>✗</td>
</tr>
<tr>
<td>code-davinci without associated edits</td>
<td>using System.Net.Http;</td>
<td>✗</td>
</tr>
<tr>
<td>CodeT5 without associated edits</td>
<td>using System.Threading;</td>
<td>✗</td>
</tr>
<tr>
<td>code-davinci with associated edits (Our approach)</td>
<td>using System.Runtime.Serialization;</td>
<td>✓</td>
</tr>
<tr>
<td>CodeT5 with associated edits (Our approach)</td>
<td>using System.Runtime.Serialization;</td>
<td>✓</td>
</tr>
</tbody>
</table>

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, 67]. 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 is 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 method 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 [67]. 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 code-davinci-edit-1, 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 code-davinci-edit-1, 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 code-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 code-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 [54].

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 \( A^1 \) is \( \delta_{j,k} \), for some \( 0 \leq j < k < n \). Given the \( m \) associated edits \( A^1, A^2, \ldots, A^m \) 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

\[
P(v_n \mid L, v_{n-1}, A^1, A^2, \ldots, A^m)
\]

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 \( A^1 \) 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, \( A^1 \) 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 \( A^1 \) that modifies locations close to the target locations \( L \) is likely to be useful to include in the set of \( m \) edits, edits farther away from \( L \) may also be relevant. We make 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 [67], 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 [45]. 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 [67]. 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 [67]. 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 \( P(\delta_{n,2} \mid L, v_0, v_1, \delta_{0,1}, \delta_{1,2}) \) 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 [54] explicitly considers the study of \( P(\delta_{n} \mid v_0, v_1, \ldots, v_{n-1}) \), but it uses \( P(\delta_{n-1,n-2} \mid v_0, v_1, \ldots, v_{n-1}) \) 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, CODEX-T5 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 [49, 51, 52]. 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, 62] 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 model performs the infilling task - predict the code that should come after the prompt but before the suffix. Finally, the editing variant of the codeex 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

Figure 2: GRACE Prompt for the associated code update task. The model is better suited only for instructions given in natural language¹.

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 edited 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 <CurrentEdit> and </CurrentEdit> 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 edits 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...

¹We do not include CODE-DAVINCI-EDIT-1 in our experiments.
We now discuss how we create fine-tuned models for predicting code updates with and without associated edits. We started with the CodeT5-base model [38, 62], a pre-trained encoder-decoder Transformer model. This base model was trained on CodeSearchNet [32] 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>, <CurrentEdit>, <Prefix>, <Suffix>, <Edit>, <Prefix>, <Suffix>, </Prefix>, </Suffix>, </CurrentEdit>, </Edit>, <Prefix>, <Suffix>. 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.

### Table 2: The models used in our experiments.

<table>
<thead>
<tr>
<th>Name</th>
<th>Base Model</th>
<th>Fine-tuned on</th>
</tr>
</thead>
<tbody>
<tr>
<td>codex-davinci</td>
<td>CodeDavinci-002</td>
<td>c3po-edited</td>
</tr>
<tr>
<td>CodeT5-U</td>
<td>CodeT5-base</td>
<td>unfiltered c3po train</td>
</tr>
<tr>
<td>CodeT5-UF</td>
<td>CodeT5-U</td>
<td>filtered c3po train</td>
</tr>
<tr>
<td>CodeT5-UO</td>
<td>CodeT5-U</td>
<td>overwatch train</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#training</th>
<th>#eval</th>
<th>#test</th>
</tr>
</thead>
<tbody>
<tr>
<td>c3po filtered</td>
<td>39.5K</td>
<td>4.4K</td>
<td>5.9K</td>
</tr>
<tr>
<td>c3po unfiltered</td>
<td>1.67M</td>
<td>180K</td>
<td>210K</td>
</tr>
<tr>
<td>overwatch</td>
<td>9K</td>
<td>1k</td>
<td>1K</td>
</tr>
</tbody>
</table>

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 8 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 [67]; 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 [67] 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 [67] 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-003 (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 [38, 62], 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-U-O, 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 \langle\text{<stop>}\rangle 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 64GB 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[38] 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 \(1e^{-2}\), 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?

### 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>
<thead>
<tr>
<th>Model</th>
<th>Without assoc. edits</th>
<th>With assoc. edits (GRACE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C3PO test set</td>
<td></td>
<td></td>
</tr>
<tr>
<td>codeT5-U</td>
<td>64.52</td>
<td>74.16</td>
</tr>
<tr>
<td>codeT5-U-O</td>
<td>59.64</td>
<td>72.25</td>
</tr>
<tr>
<td>codeT5-UF</td>
<td>73.46</td>
<td>81.83</td>
</tr>
<tr>
<td>Overwatch test set</td>
<td></td>
<td></td>
</tr>
<tr>
<td>codeT5-U</td>
<td>74.09</td>
<td>87.92</td>
</tr>
<tr>
<td>codeT5-U-O</td>
<td>70.11</td>
<td>82.25</td>
</tr>
<tr>
<td>codeT5-UF</td>
<td>74.61</td>
<td>85.78</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Model</th>
<th>Without assoc. edits</th>
<th>With assoc. edits (GRACE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C3PO test set</td>
<td></td>
<td></td>
</tr>
<tr>
<td>codeT5-U</td>
<td>64.52</td>
<td>74.16</td>
</tr>
<tr>
<td>codeT5-U-O</td>
<td>59.64</td>
<td>72.25</td>
</tr>
<tr>
<td>codeT5-UF</td>
<td>73.46</td>
<td>81.83</td>
</tr>
<tr>
<td>Overwatch test set</td>
<td></td>
<td></td>
</tr>
<tr>
<td>codeT5-U</td>
<td>74.09</td>
<td>87.92</td>
</tr>
<tr>
<td>codeT5-U-O</td>
<td>70.11</td>
<td>82.25</td>
</tr>
<tr>
<td>codeT5-UF</td>
<td>74.61</td>
<td>85.78</td>
</tr>
</tbody>
</table>

Table 4: Associated edits improve code prediction.
Table 5: Less relevant edits degrade prediction: CODEX-DAVINCI on filtered c3po test set for different associated edits.

<table>
<thead>
<tr>
<th>Choice of Associated Edits</th>
<th>Exact Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Association</td>
<td>Dataset</td>
</tr>
<tr>
<td>Spatial</td>
<td>Filtered</td>
</tr>
<tr>
<td>Random</td>
<td>Filtered</td>
</tr>
<tr>
<td>Random</td>
<td>Filtered</td>
</tr>
<tr>
<td>Random</td>
<td>Unfiltered</td>
</tr>
<tr>
<td>Random</td>
<td>Unfiltered</td>
</tr>
<tr>
<td>No Associated Edits</td>
<td></td>
</tr>
</tbody>
</table>

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 code (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 repository drops performance only slightly. However, randomly picking filtered edits from other repositories drops performance more significantly to 55.82%. When sampling from unfiltered edits irrespective of whether edits are from the same or different repositories, 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 edits 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 partially 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 code 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.

**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 10.53 Exact Match [67], whereas all of CODEX-DAVINCI (49.09%), CODET5-U (34%), and CODET5-UF (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)). CodeT5 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 6: Comparison with c3po.

<table>
<thead>
<tr>
<th>Model</th>
<th>Exact Match on c3po</th>
<th>Exact Match on overwatch</th>
</tr>
</thead>
<tbody>
<tr>
<td>c3po</td>
<td>53.20</td>
<td>10.50</td>
</tr>
<tr>
<td>CODEX-DAVINCI</td>
<td>67.92</td>
<td>49.09</td>
</tr>
<tr>
<td>CodeT5-U</td>
<td>74.16</td>
<td>34.00</td>
</tr>
<tr>
<td>CodeT5-UF/ CodeT5-Uo</td>
<td>81.83</td>
<td>48.23</td>
</tr>
</tbody>
</table>
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 [67]. 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 level. For example, if a developer types a variable name `predicate`, each intermediate file version containing the prefixes `pr`, `pred`, `pre`… 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 history codes in the form

\[
\delta_{i,0}, \delta_{i,1}, \ldots, \delta_{i,k} \quad \text{and} \quad \nu_{i,0}, \nu_{i,1}, \ldots, \nu_{i,k}
\]

where each \( \delta_{i,j} \) is the edit between the potentially non-consecutive versions \( \nu_{i,j} \) and \( \nu_{i,j+1} \), and each \( \nu_{i,j} \) is the edit between the potentially non-consecutive versions \( \delta_{i,j} \) and \( \delta_{i,j+1} \). In short, the ESPs are doing two tasks: (a) identifying “associated edits” from fine-grained version histories, and (b) using these edits to predict the next edit to \( \nu_{i,k} \). In particular, this approach is not limited in scope; as we have discussed above.

Table 7: Comparison with Overwatch.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Overwatch</th>
<th>CodeX-Davinci</th>
<th>CodeT5-u</th>
<th>CodeT5-uo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact Match</td>
<td>38.17</td>
<td>49.09</td>
<td>34.90</td>
<td>48.23</td>
</tr>
</tbody>
</table>

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.

5.6 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 Template. For instance, consider the scenario in Figure 4 where a developer is trying to add an HTTP error code to an existing code call. 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 prompt 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 with:

```java
1 catch (Exception ex)
2 {
3   - info.ReportClientError("Scheme is missing");
5     HttpStatusCode.BadRequest);
6 }
7  default:
8   - info.ReportClientError("No such action");
10     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.
foreach (var ex in currentExamples)
{
    // Console.WriteLine(GetText(ex, diff.BeforeFile));
+ Console.WriteLine(GetText(ex.Input, diff.AfterFile));
    // Console.WriteLine(GetText(ex, diff.AfterFile));
    // Console.WriteLine(GetText(ex.Output, diff.AfterFile));
}

var output = Run(currentExamples.First().Input);
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 the relationship between `diff.AfterFile` 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 [59]. 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, 44] support automated code changes like the addition of boilerplate code, developer-assisted refactoring, etc. While these developer-assisted approaches tremendously help boost productivity [46], 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, 47, 48, 53, 54, 60, 66, 67]. 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 [55] represents syntactic changes with Domain Specific language and uses a deductive inference algorithm to generalize and synthesize common edit patterns. More recently, Overwatch [67] 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, 33, 41, 42, 55] primarily focused on syntactic editing, i.e., refactoring, similar to Overwatch [67], 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, 61, 66] 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, 61], Tree to Tree translations approach [13], and Graph Neural Network based approach [21, 64]. 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, 66] generate a script of edit operations (i.e., insert, delete, update), others [13, 17, 61] 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 [62], code translation [3, 37], program repair [17, 61], 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, 64]. 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, 50]. There are a wide variety of pre-trained models for source code proposed over the years [2, 16, 23, 62], 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 [40, 56, 63, 65]. As such, these LLMs are leveraged in many industrial developer assistance tools such as GitHub Copilot [28], Amazon CodeWhisperer [57], Intellilcode Compose [18, 43], 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 uncompileable 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 [1]. 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 DATA AVAILABILITY
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/GraceCode. 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.

9 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.
REFERENCES


A MOTIVATING EXAMPLES
We present an illustrative example that has two associated edits which are both used to make a final prediction using a fine-tuned CodeT5 model.

Consider a developer refactoring code as shown in Figure 6. The developer first performs edits on Line 1 to turn the asynchronous function `AssertQueryAsync<Order>` to the function `AssertQuery<Order>` that takes one extra argument - a Boolean flag `isAsync` (Line 3). The second edit performed by the developer involves replacing the attribute `[ConditionalFact]` on Line 24 by new attributes `[Theory]` and `[InlineData(false)]` on Lines 25, 26. After performing these two edits, the developer now moves to Lines 9–18. Can we suggest the code changes to be performed on these lines and correctly predict Lines 9–18?

First, let us consider how modern code completion tools based on LLMs will attempt to predict the new code. These code completion models look at the current snapshot of the code to make predictions. Note that the associated edits preserve the relationship between the `AssertQueryAsync` and `AssertQuery` and without the inclusion of these edits it would be difficult to know that the two in fact refer to the same method. Similarly, the relationship between `[ConditionalFact]` and the new attributes `[Theory]`, `[InlineData(false)]` is also lost. Without all that information, it will be a difficult to make the correct prediction. However, if we have information about the two associated edits, then it is possible to correctly predict the third edit that is needed on Lines 9–18. In fact, given the preceding two edits, our fine-tuned CodeT5 model correctly predicts the third edit above. If we instead fine-tune a CodeT5 model to predict code changes just from the code in the current version, then such a model fails to predict the correct change.

Figure 7 shows another scenario where the developer wants to migrate from the NUnit C# testing framework to XUnit. The developer first replaces `[Test]` on Line 10 by `[Fact]` and the Teardown method on Line 8 by `Dispose`. Using these associated edits, our task is to predict that (1) `NUnit.Framework` on Line 2 needs to be replaced by `XUnit` and (2) the `TimeConfigurationTests` class needs to import the `IDisposable` interface on Line 4. The associated edits provide strong signals for these edits: (1) the `[Fact]` attribute is used to denote unit tests in `XUnit` and (2) `XUnit` test classes need to implement a `Dispose` method for performing tasks at teardown (3) this `Dispose` method is available via the `IDisposable` interface. Our fine-tuned CodeT5 model is able to successfully predict this change when the associated edits are provided.

Note that existing techniques such as c3po and overwatch may not be able to generate the right predictions on the examples in Figure 6 and Figure 7. As c3po can only use tokens from the context edits and the target, it will not be able to generate the new tokens required for the two edits (the scenario in Figure 6 needs the new `true` token on Line 12 and the scenario in Figure 7 needs the `XUnit` and `IDisposable` tokens on Lines 3 and 5 respectively). Moreover, as these are not common editing patterns, overwatch may not be able to learn templates for them.

B ALTERNATE PROMPT
The alternate comment-style prompt is shown in Figure 8. The main difference between this prompt and the prompt described in the main text of the paper is that we use comments in place of tags to delineate the various parts of the prompt.

C DATASET STATISTICS
There are two datasets discussed in the experiments: c3po filtered test set and overwatch test set. In order to discuss the difficulty of these datasets, we need to define the notion on an `edit`. For simplicity, we consider an `edit` to be a line of code that is either replaced, inserted or deleted. Here, modification of an existing line is considered a replacement, addition of a new line is considered an insertion and removal of an existing line is considered a deletion.

**c3po filtered test set:** Figure 9 shows the distribution of examples from the c3po filtered test set in terms of the number of lines edited in the context and target. Note that the number of lines edited in the context (Figure 9a) includes all the associated edits provided to make prediction. In the context, the number of lines edited on average are 3.8 and most of the samples (90th percentile) have less than 8 line edits. The target edits are much smaller with an average of 1.8 line edits (Figure 9b). 90% of the targets contain less than 4 line edits.

**overwatch test set:** Figure 10 shows the edit length distribution of associated edits from the overwatch test set. 1.8 lines have been edited in the context on average and 90% of the samples have at most 2 edits. It is expected to see smaller edits in this dataset because it contains edits made by developers in an IDE (which are often small). Note that it is not possible to get the edit length distribution for the targets in this dataset as there is no single ground truth target edit and the predicted edit is compared with several future versions of the target (see Section 5.5 for more details on the evaluation).

D ADDITIONAL EXPERIMENTAL RESULTS
Additional details about the metrics: In the case of overwatch dataset, a prediction is considered an `exact match` if the `version` produced by incorporating the suggestion syntactically matches any of the 50 `future versions`. The percentage of exact match predictions is also interchangeably called `precision`, as reported by the overwatch paper. In all our results, we report `Exact Match` for Top-1 predictions and explicitly call out whenever we report Top-5 numbers (any one of the 5 is an `exact match`).

Table 8 presents complete set of numbers from our various experiments on the filtered c3po test set that contained 5.9 benchmarks. Apart from the models discussed in the main text of the paper, we also have results for a smaller Codex model `code-cushman-1`.

Table 9 presents results on the 1K test benchmarks from the overwatch dataset.

Table 10 shows results on an edit sequence patterns test dataset that contains 13K instances of edit sequences learned by overwatch on the test IDE versions.

Table 11 shows results obtained by `code-davinci` on a random sample of 5.9K benchmarks taken from the unfiltered c3po test set.

Table 12 contains data from CodeT5 on filtered c3po test set using different associated edits. Since CodeT5 models were fine-tuned on certain prompts, presenting them with prompts that look different causes a uniform degradation of accuracy – unlike the case for `code-davinci` that showed a graceful degradation as the edits in the prompt became less relevant to the target edit.
- return AssertQueryAsync<Order>(
+ return AssertQueryAsync<Order>(
+ isAsync
os = o.Where( o => o.OrderDate != null & & o.EmployeeID.Contains( "10" )
Select( o => new Order { CustomerID = o.CustomerID } ),
e => e.CustomerID );

- [ ConditionalFact ]
+ [ Theory ]
+ [ InlineData( false )]
+ [ InlineData( true )]
- public virtual Task Select_expression_other_to_string( )
+ public virtual Task AssertQueryAsync_Order(Object)
{
- return AssertQueryAsync<Order>(
+ return AssertQueryAsync<Order>(
+ isAsync
os = o.Where( o => o.OrderDate != null )
Select( o => new Order { ShipName = o.OrderDate.Value } ),
e => e.ShipName );

- [ ConditionalFact ]
+ [ Theory ]
+ [ InlineData( false )]
...

Figure 6: User replaces AssertQueryAsync on Line 1 with AssertQuery and adds an argument isAsync (Line 3). The next edit replaces [ConditionalFact] on Line 24 with [Theory], [InlineData(false)]. The task is to predict edits on Lines 9-18.

Table 8: C3PO filtered test set (5.9K samples)

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Exact Match Accuracy</th>
<th>Top1</th>
<th>Top5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without Associated Edits</td>
<td>Cushman-002 (comment-style)</td>
<td>3.54</td>
<td>5.59</td>
</tr>
<tr>
<td>Davinci-002 (comment-style)</td>
<td>28.85</td>
<td>35.36</td>
<td></td>
</tr>
<tr>
<td>Davinci-002 (tag-style, temp=0.1)</td>
<td>37.09</td>
<td>40.94</td>
<td></td>
</tr>
<tr>
<td>Davinci-002 (tag-style, temp=0.5)</td>
<td>33.72</td>
<td>52.9</td>
<td></td>
</tr>
<tr>
<td>C3PO (finetuned on unfiltered C3PO train)</td>
<td>64.52</td>
<td>72.22</td>
<td></td>
</tr>
<tr>
<td>C3PO (finetuned on filtered C3PO train)</td>
<td>72.28</td>
<td>83.13</td>
<td></td>
</tr>
<tr>
<td>C3PO (trained on unfiltered C3PO train, finetuned on filtered C3PO train)</td>
<td>73.46</td>
<td>85.27</td>
<td></td>
</tr>
<tr>
<td>With Associated Edits (taken from the spatial context)</td>
<td>Cushman-002 (comment-style)</td>
<td>64.09</td>
<td>65.36</td>
</tr>
<tr>
<td>Davinci-002 (comment-style)</td>
<td>65.63</td>
<td>66.92</td>
<td></td>
</tr>
<tr>
<td>Davinci-002 (tag-style, temp=0.1)</td>
<td>67.92</td>
<td>69.77</td>
<td></td>
</tr>
<tr>
<td>Davinci-002 (tag-style, temp=0.5)</td>
<td>66.59</td>
<td>74.64</td>
<td></td>
</tr>
<tr>
<td>C3PO (finetuned on unfiltered C3PO train)</td>
<td>74.16</td>
<td>79.66</td>
<td></td>
</tr>
<tr>
<td>C3PO (finetuned on filtered C3PO train)</td>
<td>75.30</td>
<td>84.7</td>
<td></td>
</tr>
<tr>
<td>C3PO (trained on unfiltered C3PO train, finetuned on filtered C3PO train)</td>
<td>81.83</td>
<td>89.5</td>
<td></td>
</tr>
<tr>
<td>With Associated Edits (taken from other repos filtered)</td>
<td>Cushman-002</td>
<td>55.82</td>
<td>60.05</td>
</tr>
<tr>
<td>Davinci-002 (tag-style)</td>
<td>43.64</td>
<td>47.58</td>
<td></td>
</tr>
<tr>
<td>With Associated Edits (taken from same repo filtered)</td>
<td>Cushman-002</td>
<td>64.9</td>
<td>68.09</td>
</tr>
<tr>
<td>Davinci-002 (tag-style)</td>
<td>43.23</td>
<td>47.11</td>
<td></td>
</tr>
<tr>
<td>With Associated Edits (2 associated edits + 1 unrelated)</td>
<td>Cushman-002</td>
<td>67.4</td>
<td>74.91</td>
</tr>
<tr>
<td>Davinci-002 (tag-style, temp=0.5)</td>
<td>68.51</td>
<td>70.21</td>
<td></td>
</tr>
</tbody>
</table>
Table 9: Overwatch applications test set (1K samples)

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Exact Match Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Top1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Overwatch</td>
<td></td>
</tr>
<tr>
<td>Without Associated Edits</td>
<td>Davinci-002</td>
</tr>
<tr>
<td></td>
<td>CODET5-base (trained on unfiltered C3PO train)</td>
</tr>
<tr>
<td></td>
<td>CODET5-base (trained on unfiltered C3PO train fine tuned on overwatch train)</td>
</tr>
<tr>
<td>With Associated Edits</td>
<td>Davinci-002</td>
</tr>
<tr>
<td></td>
<td>CODET5-base (trained on unfiltered C3PO train)</td>
</tr>
<tr>
<td></td>
<td>CODET5-base (trained on unfiltered C3PO train fine tuned on overwatch train)</td>
</tr>
</tbody>
</table>

Table 10: Overwatch complete test set (13K samples)

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Exact Match Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Top1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Without Associated Edits</td>
<td>Davinci-002 (tag-style)</td>
</tr>
<tr>
<td></td>
<td>CODET5-base (trained on unfiltered C3PO train fine tuned on overwatch train)</td>
</tr>
<tr>
<td>With Associated Edits</td>
<td>Davinci-002 (tag-style)</td>
</tr>
<tr>
<td></td>
<td>CODET5-base (trained on unfiltered C3PO train fine tuned on overwatch train)</td>
</tr>
</tbody>
</table>

Table 11: C3PO unfiltered sampled test set (5.9K samples)

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Exact Match Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Top1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>With Associated Edits</td>
<td>Davinci-002 (tag-style)</td>
</tr>
<tr>
<td>Without Associated Edits</td>
<td>Davinci-002 (tag-style)</td>
</tr>
</tbody>
</table>

Figure 8: Comment-style prompt for the associated code update task.

Table 12: Less relevant edits degrade prediction: CODET5 on filtered C3PO test set using different associated edits

<table>
<thead>
<tr>
<th>Source of associated edits</th>
<th>Exact Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spatial</td>
<td>74.16</td>
</tr>
<tr>
<td>Any unfiltered edit from same repo</td>
<td>63.25</td>
</tr>
<tr>
<td>Any unfiltered edit from other repo</td>
<td>63.32</td>
</tr>
<tr>
<td>No associated edits</td>
<td>64.52</td>
</tr>
</tbody>
</table>

D.1 Comparison with few-shot prompting without associated edits

In Section 5.2 of the main text, we empirically show that predictions using associated edits are significantly better than those made using randomly sampled edits. These results are with the prompt design we discuss in Figure 2 (main text). While these results signify the importance of associated edits, there is another important baseline to consider: few-shot prompting without associated edits. In this case, we create a few-shot prompt using the design mentioned in Figure 3.
(main text). Our preliminary analysis on the c3po test set suggests that this approach is better than zero-shot prompting without associated edits but not as good as our approach. When CODEX-DAVINCI is prompted with 2 random examples from the same repository (2-shot prompting), it reports an Exact Match of 45.53%. Compared with the results in Table 5 (main text), this is expectedly better than an Exact Match of 37.07% from the CODEX-DAVINCI model without associated edits. It is however much lower than 67.92% reported by CODEX-DAVINCI with associated edits. Interestingly, this is also lower than the case where randomly sampled edits are added to the prompt in Figure 2 from the main text (64.9%). As these two cases only differ in the way the edits are presented in the prompt, this
may suggest that our prompt design actually helps the model use the provided examples better. More experiments on the unfiltered c3po and overwatch test sets would be needed to confirm this hypothesis.

D.2 Associated Edits Improve Robustness

A central thesis of our work is that code updates should be conditioned on associated edits, and we created two models, codex-davinci and CodeT5 that perform that task. When using those models for inference, we need associated edits. What if we don’t have any? What if we have edits that include associated edits but may have other edits? If we use edits that are not related to the target edit, would those harm and cause exact match to fall below if we had not used any associated edits? If the models are very sensitive to such noise in the associated edits, then that would increase the burden on users to find exactly the correct set of associated edits.

**RQ5. How tolerant is our approach to noise in the set of associated edits provided during inference?**

**Table 13: Noise tolerance of associated code update: codex-davinci on filtered c3po test set**

<table>
<thead>
<tr>
<th>Source of associated edits</th>
<th>Exact Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spatial</td>
<td>67.92</td>
</tr>
<tr>
<td>Spatial + one unfiltered edit from any repo</td>
<td>68.51</td>
</tr>
</tbody>
</table>

**Results:** We first consider the case when we have one additional unrelated edit in the set of associated edits. We added one unrelated unfiltered edit to the two spatially close associated edits in the c3po filtered test set. With everything else unchanged, as shown in Table 13, codex-davinci gives a 68.51% Exact Match on this modified c3po filtered test set. So, adding an unrelated edit, in fact, slightly improved the Exact Match metric from 67.92%. This small gain is possibly due to the few-shot learning feature of codex-davinci.

**Result 1:** Inclusion of an unrelated edit does not significantly affect the LLM’s Exact Match metric on the associated code update task.

We next consider the following question: How does a model trained with associated edits compare against one trained without them on the code update task? Does training with associated edits hurt the model’s performance on the code update task?

**Table 14: Effect of associated code update finetuning on code update task:**

**Test set: overwatch without spatial edits**

<table>
<thead>
<tr>
<th>Technique</th>
<th>Finetuning Prompt</th>
<th>Exact Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeT5-base-c3po-finetuned</td>
<td>without spatial edits</td>
<td>22.25</td>
</tr>
<tr>
<td></td>
<td>with spatial edits</td>
<td>24.74</td>
</tr>
<tr>
<td>CodeT5-base-overwatch-finetuned</td>
<td>without spatial edits</td>
<td>40.78</td>
</tr>
<tr>
<td></td>
<td>with spatial edits</td>
<td>38.97</td>
</tr>
</tbody>
</table>

**Results:** In order to answer these questions, we consider evaluating two CodeT5 models with different finetuning objectives on the overwatch dataset. The first model is trained with prompts containing associated edits while the second one is trained without. As shown in Table 13, the two models report comparable results on the overwatch dataset when no associated edits are provided. The model trained with associated edits reports a 2.5% higher Exact Match when finetuned on the c3po dataset and only loses 2% on Exact Match when further finetuned on the overwatch dataset. This suggests that the associated code update task can be used to train models that are robust enough to then be used out-of-the-box on the code update task.

D.3 Associated Edits Reduce Entropy

Consider the probability density function (pdf) $p_1(\omega_n)$ on document version $\omega_n$ given by $p_1(\omega_n) = P(\omega_n | L, \omega_n-1)$ and a second pdf $p_2(\omega_n) = P(\omega_n | L, \omega_n-1, \omega_n-2, \ldots, \omega_n-2, n-1)$. If the associated edits $\omega_n-1, \ldots, \omega_n-2, n-1$ contribute to the prediction of $\omega_n$, then the entropy of $p_1$ should be higher than that of $p_2$. The models without and with associated edits learn the functions $p_1$ and $p_2$ respectively. The difference between Top1 Exact Match and TopK Exact Match for a given model is a good proxy for the entropy of the probability density function it has learnt.

Table 15 shows irrespective of whether we used fine-tuned CodeT5, or codex-davinci, and independent of which training sets we used to fine-tune CodeT5, when we used associated edits in the prompt, the gap between Top1 and Top5 Exact Match was always lower than what it was when we did not use associated edits. So, this indicates that associated edits are indeed predictive of code updates.

**Table 15: Top1-Top5 Spread**

<table>
<thead>
<tr>
<th>Model</th>
<th>Test</th>
<th>Without associated edits</th>
<th>With associated edits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Top1</td>
<td>Top5</td>
</tr>
<tr>
<td>CodeT5 1</td>
<td>c3po</td>
<td>64.5</td>
<td>72.2</td>
</tr>
<tr>
<td>CodeT5 2</td>
<td>c3po</td>
<td>72.3</td>
<td>83.1</td>
</tr>
<tr>
<td>CodeT5 1</td>
<td>overwatch</td>
<td>22.25</td>
<td>48.81</td>
</tr>
<tr>
<td>CodeT5 12</td>
<td>overwatch</td>
<td>40.78</td>
<td>59.79</td>
</tr>
<tr>
<td>codex-davinci</td>
<td>c3po</td>
<td>37.1</td>
<td>40.9</td>
</tr>
<tr>
<td>codex-davinci</td>
<td>overwatch</td>
<td>31.8</td>
<td>37.2</td>
</tr>
</tbody>
</table>

In Table 15 CodeT5 1 is CodeT5 fine-tuned only on c3po unfiltered training set, CodeT5 2 is CodeT5 fine-tuned only on c3po filtered training set, and CodeT5 12 is CodeT5 1 further fine-tuned on overwatch training set. Note that the Top1-Top5 Exact Match difference is generally smaller for codex-davinci because we used a lower temperature setting (0.1) for our experiments. Additional fine tuning also helps reduce entropy of the learned probability density function. For example, the Top1-Top5 spread is extremely large for the case when we use CodeT5 fine-tuned only on c3po unfiltered training set (CodeT5 1) and evaluate it on overwatch dataset, but the spread reduces somewhat when we further fine-tune the model on overwatch train set (CodeT5 12). The above results provide additional evidence in support of our Result 1 (from the main text).
D.4 Learnings from Manual Inspection

We manually inspected a few benchmarks where some model failed to make the correct code prediction – either with associated edits, or without them, or in both cases. We next share some high level observations from these investigation.

In the set of benchmarks that succeeded with associated edits, but not without, a common case was when there was an edit in the associated edits that could be generalized into a program transformation rule and then applied to the edit location to yield the ground truth. However, often the associated edit was not a “perfect” demonstration of the required rule, but only a “noisy” demonstration. Furthermore, there were also cases where the rule had to be adapted (say, by generating some new identifier based on the pattern) in subtle ways.

In the set of benchmarks that succeeded without associated edits, but not with, the prediction with associated edits was often very close to the ground truth. There were a few failures (in presence of associated edits) that were caused due to previous versions misleading the model with some irrelevant information (such as, something about SelectExpression objects) that was close to something in the ground truth (such as, the Expression type).

In the set of benchmarks that failed both with and without associated edits, the most common case was when the edit involved generating entirely new code or a new identifier name. In other cases, generating the ground truth would have required knowing the signature of certain methods that did not appear in the associated edits. It was clear that while associated edits is very useful, it is not always enough and needs to be enhanced with “associated code context” (such as, live variables and type signatures). We leave that investigation to future work.

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