Large Language Models and Simple, Stupid Bugs

Kevin Jesse
UC Davis
Davis, USA
krjesse@ucdavis.edu

Toufique Ahmed
UC Davis
Davis, USA
tfahmed@ucdavis.edu

Premkumar T. Devanbu
UC Davis
Davis, USA
ptdevanbu@ucdavis.edu

Emily Morgan
UC Davis
Davis, USA
eimorgan@ucdavis.edu

2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR) | 979-8-3503-1184-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/MSR59073.2023.00082

***Abstract*—With the advent of powerful neural language models, AI-based systems to assist developers in coding tasks are becoming widely available; Copilot is one such system. Copilot uses Codex, a large language model (LLM), to complete code conditioned on a preceding "prompt". Codex, however, is trained on public GitHub repositories, *viz.*, on code that may include bugs and vulnerabilities. Previous studies [1], [2] show Codex reproduces vulnerabilities seen in training. In this study, we examine how prone Codex is to generate an interesting bug category, single statement bugs, commonly referred to as simple, stupid bugs or SStuBs in the MSR community. We find that Codex and similar LLMs do help avoid some SStuBs, but do produce *known, verbatim* SStuBs as much as 2x as likely than *known, verbatim* correct code. We explore the consequences of the Codex generated SStuBs and propose avoidance strategies that suggest the possibility of reducing the production of known, verbatim SStubs, and increase the possibility of producing known, verbatim fixes.**

***Index Terms*—language models, prompting, deep learning, software engineering**

I. Introduction

The rise of language-model based AI coding tools promises to change programming practice. Developers can now use AI coding tools, which inherit their power from models trained on enormous corpora of open-source code. Copilot, a language-model based coding assistant [3], is available in many integrated development environments (IDEs). Copilot uses a model named Codex [4] to generate code completions. The full power of Codex is still being learned: it can already perform a diverse set of tasks including: code completion [4], automatic program repair (APR) [5], comment generation [4], [6], program synthesis [7], [8], and incident management [9].

Copilot is free to use, and is widely adopted. It is an attractive tool for developers at different skill levels; it helps provide starting points for developers [10] and can start functions from just input, output examples [7]. The capabilities of Codex and similar models [11]–[14] have raised many concerns, and have given rise to different research thrusts. Active avenues of Copilot research include how developers work with it: researchers report concerns on an over-reliance and unwarranted trust in Copilot-generated code [10], [15], the quality of the completions [16], [17], security implications [2], [18]–[20], and copyright infringement [21]. These research topics are motivated by the free access and popularity of Copilot, particularly, the use of code it generates may give rise to broader ethical and functional concerns.

Codex has been found to work for program repair [5], [22], problem solving [21], [23], math [24], [25], and translation from natural language to various target languages [4], [23], [26], [27] to name a few applications. With many use cases, Codex is a double-edged sword of utility and risk and ultimately we should find ways to minimize the risk and maximize the utility of Codex.

To that objective, this work examines Codex on the 2021 MSR mining challenge dataset ManySStubs4J [28]. The dataset consists of single statement 'simple, stupid bugs' (SStuBs) mined from Maven projects. Karampatis and Sutton found that SStuBs have a frequency of 1 in every 1,600 LOC and that static analyzers cannot detect them [28], [29]. Mosolygó *et al.* [29] determined that SStuBs appear more in larger chunks of code authored by the same developer, perhaps due to loss of attention or misunderstanding of code functionality. We see this exact phenomena in Codex studies [10], [15] where developers use Codex to generate large blocks of code and, *if* a bug is found, dive into a time-consuming rabbit hole to fix the code [10]. Worse, this study reports that developers often blindly trust generated code, or optimistically hope to fix problems later.

Surprisingly, so-called "simple, stupid" bugs can survive a long while; in the SStuBs dataset, fixes take around 240 days [29], [30]. More worryingly, when Codex generates code, an 'agent' other than the active IDE user is actually 'coding', and thus the number of commits to fix the SStuB might be even longer (see Mosolygó *et al.* [29]). This disappointing possibility is supported by surveys [10], [15], suggesting that developers don't always understand generated code, and struggle to fix any bugs therein.

The performance of Codex has been extensively benchmarked [4], [22], checked for security vulnerabilities [18], [18]–[21], and empirically evaluated [2], [10], [15], [17]. However, Codex has not been evaluated against SStuBs, which are a special kind of bug [31]. To understand SStuBs related to AI-supported programming, we evaluate whether Codex and other code completion models produce SStuBs, or their fixes; also we look at the consequences of such bugs in code bases. Finally, we present a Codex experiment aimed at *automatically* communicating developer *intent*, using *generated* comments,

This material is based upon work supported by the National Science Foundation under Grant NSF CCF (SHF-MEDIUM) No. 2107592. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

2574-3864/23/$31.00 ©2023 IEEE
DOI 10.1109/MSR59073.2023.00082

to help avoid introducing simple, stupid bugs, and also producing commented code.

Our research questions are as follows, all primarily evaluated using the ManySStuBs4J dataset:

- **RQ1:** How often do Codex and similar language models (CodeGen [14] and PolyCoder [13]) produce simple, stupid bugs?
- **RQ2:** When Codex generates the same simple, stupid bug that a human does, how much time does the SStuB originally take to fix?
- **RQ3:** Does Codex produce buggy or correct code more confidently (*viz.* at higher probability)?
- **RQ4:** Does adding automatically generated comments to the prompt help Codex and akin language models avoid SStuBs? Do other types of prompt improvements help reduce SStuBs?
- **RQ5:** How do bug-derived code comments, when inserted in the prompt, affect code generated by Codex and other LLMs?

Our key findings are: (1) LLMs do help avoid some SStuBS in our dataset! (2) Codex and other LLMs do produce *known* SStuBs, and at a rather high rate (perhaps twice as often as they produce *known* correct, bug-fixing code); (3) When Codex generates a known SStuB, it's associated (historically) with longer fix-times; (4) Codex-generated completions appear equally 'natural' [32], regardless of whether they match buggy code or the related fix; if they match neither, they are less natural. (5) Automatically generated comments, when added to prompts, appear to reduce the known SStuB production rate for most models and improve the bug/patch ratio; (6) Even buggy comments help to reduce the bug/patch ratio in Codex, suggesting just attempting to comment code helps. The improvement in avoiding SStuBs with comments from neural comment generation model CodeTrans [33], suggest that using these models *with* Codex would be beneficial to avoid SStuBs.

Data from this study is available here[1].

II. Related Work

We discuss related work on language models and code quality.

A. Simple, Stupid Bugs

Simple, stupid bugs (SStuBs) are bugs that have single-statement fixes that match a small set of bug templates. They are called "simple" because they are usually fixed by small changes and "stupid" because, *once located*, a developer can usually fix them quickly with minor changes. However, locating SStuBs can be time-consuming [28]. Karampatsis and Sutton [28] published a collection of SStuBs mined from a set of template bug types, *e.g.*, `CHANGE_IDENTIFIER` or `DIFFERENT_METHOD_SAME_ARGUMENTS`. Through their study of the dataset, Karampatsis found that SStuBs are prevalent in code bases, accounting for 33% of single statement bugs detected in 1000 Maven projects. They found that these bugs occur every 1,600 lines of code, and were not detected by static analysis. MSR once used ManySStuBs4J dataset as a mining challenge, to study these bugs, and manage their impact.

Mosolygó *et al.* [29] studied the history of SStuBs. They find that SStuBs are more frequent in code modified by the same developer, often when s/he writes large chunks of code. This is perhaps because such large coding tasks strain focus and attention. They found that only 40% of SStuBs were fixed by the same author in a median time of 4 days; when the SStuB is fixed by a different author, the SStuB took 136 days to find and fix! We hypothesize that if comments were present in a Codex prompt, we'd get better code completions, *and* the entire chunk would be easier to read & fix.

Zhu and Godfrey [34] studied how developers fix SStuBs. Similar to Mosolygó *et al.* [29] they found that developers fix their own bugs quicker, whereas bugs from other developers take significantly more time to fix. This suggests that Codex generated SStuBs *may* take longer to fix since they come from an artificial 'developer'. Codex-generated code-snippets that include SStuBs may require extensively debugging to be patched in a similar fashion. A Copilot study [10], found that users had trouble debugging generated code from Codex spending considerable time and effort to fix, for instance, a generated regular expression.

Madeiral and Durieux [35] discussed SStuBs in the context of code clones [36] and the changes that introduce them, *viz.* "change clones". They found that 29% of change clones introduced SStuBs by matching the 16 SStuB patterns. Since Codex is a language model that tends to repeat code it's seen, it could conceivably generate SStuBs in multiple locations, increasing the repair effort.

Peruma and Newman [37] examined SStuBs in unit test files. They found that SStuBs tend to occur in non-test files and that developers fix the bugs separately despite test and non-test files being functionally related. Peruma and Newman also discovered that developers prioritize non-test files and the fixes in tests are associated with asserts.

Latendresse *et al.* [30] and Hua *et al.* [31] addressed the detection of SStuBs. Latendresse *et al.* [30] found that continuous integration (CI) tools cannot catch any SStuBs. Hua *et al.* [31] found that deep learning vulnerability detectors were suboptimal compared to traditional vulnerability detectors on SStuBs. Our results confirm that models as large as Codex, find SStuBs to be equally regular to the patches it generates.

Mashhadi *et al.* [38] applied CodeBERT [39] (fine-tuned for patching) to SStuBs and could fix 19% of de-duplicated Many4SStuBs4J dataset. Mashhadi *et al.* mentions an advantage of using CodeBERT: no special tokens are required like in SequenceR [40]; similarly, many APR techniques with Codex [5], [22] rely on a prior of knowing a bug exists or even, more specifically, the bug location [40]. Adding comments to the prompt does not require making any of these assumptions.

Finally, PySStuBs [41] is a Python simple, stupid bug dataset. The more recent TSSB-3M [42] is also a Python SStuBs dataset mined at scale. Our study focuses on the established Java SStuBs patterns from the everpopular

[1]https://doi.org/10.5281/zenodo.7676325

ManySStuBs4J dataset. We plan to expand our findings to other languages and SStuB patterns, resources permitting. Currently OpenAI restricts usage of Codex to 20 requests per minute. Inference on large models for ManySStuBs4J takes just over a day.

B. Examining Codex Completions

Vaithilingam *et al.* [10] studied the developer experience with Codex. The key findings were that most participants preferred Codex to Intellisense in Visual Studio IDE. Participants preferred to use Codex as a starting point in lieu of searching online. Unfortunately participants over-relied on Codex and then struggled when generated code was buggy. The authors reported three major issues: (1) participants often didn't understand and assess the correctness of generated code, (2) participants underestimated the repair-effort required when generated code was buggy, (3) the prompts used by participants were quite varied, sometimes resulting in undesired code completions.

Sarkar *et al.* [15] wrote an extensive review of programming with an AI assistant Codex. Sarkar *et al.* surveys previous work citing Codex's reliability, safety, and security implications. The review covers studies in Codex usability, design, and user reports. Sandoval [18] and Perry [19] examine Codex security implications.

Yetistiren *et al.* [16] and Nguyen *et al.* [17] empirically studied Copilot's code suggestions. Yetistiren *et al.* [16] found Copilot mostly generated valid code and Copilot improved it's correctness with further input from the developer; sample examples, unit tests, docstrings, and prompts increased correctness further. Nguyen *et al.* [17] found Copilot correctness varies by programming language and does not differ in complexity (cognitive and cyclomatic) among programming languages.

Prenner *et al.* [5], Pearce *et al.* [1], [22], Karmakar *et al.* [21], and Ahmed *et al.* [6], [9] apply Codex in various settings: automatic program repair (APR) [5], security vulnerability prediction [22], HackerRank challenges [21], code summarization [6], and incident management [9]. In APR, Prenner *et al.* tried engineering prompts to find a way to push Codex to generate a non-buggy version of the code. Codex performed competitively to recent work but was sensitive in the prompt. Pearce *et al.* found Codex could repair 58% of real world security vulnerabilities. Karmaker *et al.* applied Codex on HackerRank problems with great success; some of the success was attributed to Codex already knowing the solution despite an incomplete prompt, in other words, memorizing the solution. Ahmed *et al.* trains Codex on few-shot project-specific code to achieve state-of-the-art code summarization. Ahmed *et al.* found success in using Codex to help engineers diagnose and mitigate production incidents. All of these works use prompting to illicit a desired response from Codex.

Prompting instructions with natural language or code is often implemented as a comment to the prompt passed to Codex. The prompts are implemented as comments to improve generated code, *e.g.*, natural language instructions in docstrings

JBTabsImpl.java

```
1524        if (paintBorder.top > 0) {
1525            if (isHideTabs()) {
1526                if (isToDrawBorderIfTabsHidden()) {
1527                    g2d.setColor(borderColor);
1528                    g2d.fill(shaper.reset().doRect(boundsX, boundsY, boundsWidth,
1529                            1).getShape());
1530                }
1531            }
1532            else if (isStealthModeEffective()) {
1533                //g2d.setColor(borderColor);
1534                //g2d.fill(shaper.reset().doRect(boundsX, boundsY - 1,
1535                // boundsWidth, 1).getShape());
1536            }
1537            else  {
1538                Color tabFillColor = getActiveTabFillIn();
1539                if (tabFillColor == null) {
1540                    tabFillColor = shape.path.transformY1(shape.to, shape.from);
1541                }
1542
1543                g2d.setColor(tabFillColor);
1544                g2d.fill(shaper.reset().doRect(boundsX, topY + shape.path.deltaY(1),
1545                        boundsWidth, paintBorder.top).getShape());
1546
1547                //If the top of the border is a non-paint border, then the border is painted.
1548                if (
1549 Bug ->             paintBorder.top >= 1
1550                    paintBorder.top > 1   <- Fix
1551                ) {
1552                    g2d.setColor(borderColor);
1553                    final int topLine = topY + shape.path.deltaY(paintBorder.top - 1);
1554                    g2d.draw(shaper.reset().doRect(boundsX, topLine, boundsWidth - 1,
1555                            1).getShape());
1556                }
1557            }
1558        }
1559
1560        g2d.setColor(borderColor);
1561
1562        //bottom
1563        g2d.fill(shaper.reset().doRect(boundsX, Math.abs(shape.path.getMaxY() -
1564                shape.insets.bottom - paintBorder.bottom), boundsWidth,
1565                paintBorder.bottom).getShape());
1566
1567        //left
1568        g2d.fill(shaper.reset().doRect(boundsX, boundsY, paintBorder.left,
1569                boundsHeight).getShape());
1570
1571        //right
1572        g2d.fill(shaper.reset().doRect(shape.path.getMaxX() - shape.insets.right -
1573                paintBorder.right, boundsY, paintBorder.right,
1574                boundsHeight).getShape());
1575
1576    }
```

Fig. 1: The orange highlighted code is the candidate single line completion that Codex can match to the automatic evaluation either the known bug or fix. Blue highlighted code is the prompt *a.k.a.* the text proceeding SStuB statement that Codex uses for completion. The purple highlighted code is the code after the SStuB.

or input-output examples. Prompting relies on knowledge *a priori* that Codex should adopt into its generations. A prompt could be: an example completion, a problem description, input format, code that is vulnerable, a docstring documenting a bug location, input-output pairs, and snippets of bugs and corresponding patches (few-shot). In this work, we explore similar prompting techniques from Prenner [5], *and* focus on *prior-free* prompting through traditional commenting practices. We find that good commenting practices can guide Codex to more SStuB-free completions.

III. Methodology

In this section, we describe the methodology for evaluating Codex [4], PolyCoder [13], and CodeGen [14] on SStuBs.

A. Experimental Setup

ManySStuBs4J

This dataset consists of a small and large dataset with 10,231 and 63,923 single-statement bug-fix changes (*a.k.a* SStuBs) mined from 100 and 1000 popular repositories respectively. These SStuBs must match one of 16 bug templates. The goal is to collect bugs that are difficult to locate, but easy to fix. It's natural to wonder if automated coding tools based on language models could introduce such single-statement bugs.

```
    if (WRITE_TYPE.isCache()) {
      try {
        int tLen = len;
        int tOff = off;
        while (tLen > 0) {
          if (mCurrentBlockLeftByte == 0) {
            getNextBlock();
          }
          if ("
```

Bug:

```
"mCurrentBlockLeftByte > tLen"
```

Fix:

```
"mCurrentBlockLeftByte >= tLen"
```

Model Davinci:

```
"tLen >= mCurrentBlockLeftByte ){
    mCurrentBlockOutputStream.write(b, tOff, mCurrentBlockLeftByte);
    tLen -= mCurrentBlockLeftByte;
    tOff += mCurrentBlockLeftByte;
    mCurrentBlockLeftByte = 0;"
```

bug | fix | incorrect | unsure

(a) This completion is incorrect.

```
  public static void checkMinimalPollingPeriod(TimeUnit pollUnit, int pollPeriod)
      throws IllegalArgumentException {
    int period = (int) MINIMAL_POLL_UNIT.convert(pollPeriod, pollUnit);
    Preconditions.checkArgument("
```

Bug:

```
"period < MINIMAL_POLL_PERIOD"
```

Fix:

```
"period >= MINIMAL_POLL_PERIOD"
```

Model Davinci:

```
"period >= 1, "Polling period %s %s is smaller than 1%s", pollPeriod, pollUnit, MINIMAL_POLL
}

/**
  * Checks if the poll period is smaller that the minimal poll period which is 1 second"
```

bug | fix | incorrect | unsure

```
",
        "Polling period %d %d is below than minimal polling period", pollPeriod, pollUnit);
  }
```

(b) This completion matches the fix. Earlier in the code, the constant `MINIMAL_POLL_PERIOD` is set to 1.

Fig. 2: This annotation tool helps mark Codex completions that do not match any SStuB directly. This guarantees our evaluation is not missing reasonable alternatives to the SStuB that could be deemed a bug or fix.

For our study, we use the ManySStuBs4J *large* dataset of 63,923 samples and use `git checkout` to obtain versions of the bug prior and after being fixed. We use `git blame` to determine when the bug was introduced and capture key statistics such as the number of commits to fix the SStuB. The bug locations within the files are indexed with fields `bugNodeStartChar` and `bugNodeLength`. Using these fields we can find the code before the bug, the bug, the fix, and the code after the bug/fix. Our experiment focuses on giving Codex a piece of code prior to the bug and seeing if Codex generates the correct code (fix) or incorrect code (bug). A large concern in this experiment is to make sure Codex has an equal opportunity to generate the known bug or patch. Therefore, we remove SStuBs that have other changes besides fixing the SStuB, which could otherwise *condition* or bias Codex to make a decision; for example, a new variable only exists in the buggy

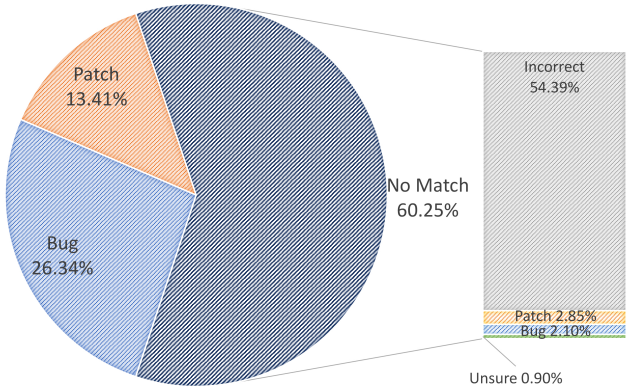


Fig. 3: Match rate of Codex Davinci (left). Completions that do not match a patch or SStuB are validated by hand (right).

version so Codex completes the bug. This leaves 34,595 bugs prior to deduplication. Then we drop duplicates for bugs that share the exact same prefix, bug, and fix. It is important to not inflate results by duplicate code examples [43]. The remaining 16,899 SStuBs are used for evaluating all models.

Codex, PolyCoder, CodeGen

Large language models like Codex, PolyCoder, and CodeGen are demonstrably useful in code completion tools like Copilot, and are available for experimentation. Other models exist, like AI21 Jurassic; however they are not free, and would be costly at our scales, so were excluded from our study. We also didn't have the computational resources to run very large models locally. For these reasons, we follow the methodology from Xu *et al.* [13] and use CodeGen, PolyCoder, and Codex.

Codex derives from GPT-3; its training data consists of natural language and source code from available sources like public GitHub repositories. Codex has two sizes one called *cushman-codex* and *davinci-codex* [44]. To query *cushman-codex* and *davinci-codex* models, we must make API requests using the OpenAI API (free, but rate-limited to 20 requests a minute). While the exact number of parameters is unknown, for both *cushman-codex* and *davinci-codex* models, prior work suggests sizes of 12B and 175B parameters [4], [5], [26] for *cushman-codex* and *davinci-codex* respectively. Codex is primarily trained on Python [4].

To determine if our results generalize to other large language models (LLMs) for code, we evaluate two additional families of models on SStuBs. These models are trained with different procedures and are readily available. CodeGen [14] is an auto-regressive transformer model, trained with the next-token prediction objective on a corpus of code and natural language from GitHub. CodeGen is trained on multiple languages, but Python is the primary language. PolyCoder [13] is based on GPT-2 architecture and is trained on 250GB of code across 12 programming languages with C, C++, and Java being the primary language. PolyCoder outperforms all other code LLMs in C including Codex. Codex, PolyCoder, and CodeGen represent a diverse set of models all with several

model size versions. Testing our the SStuBs hypothesis on Codex, PolyCoder, and CodeGen highlights potential risks of inducing SStuBs while using LLMs. While we cannot say for certain, other LLMs trained on similar data could show similar behavior.

We use the aforementioned models to generate completions by prompting with the code before the SStuB. When models complete the prompt, we can analyze the completion, by matching the known bug, or fix, from the SStuBs dataset. To compare completions to the bug and patch ground truths, we use substring matching (ignoring whitespace and formatting). To verify the accuracy of the results, a survey was conducted where the authors determined if sampled completions (n=401) match the bug, fix, or no match in a manner the automatic evaluation could not capture. For each model family, the best performing model completions were subjected to finer scrutiny; it is important that semantic equivalents are properly counted, such as Codex replacing a constant for the equivalent literal value. The manual survey interface[2] is screen-shot in Figure 2. Figure 2a shows a completion that is logically incorrect. In Figure 2b, Codex actually replaces a constant with its literal value which matches the fix. 401 randomly selected SStuBs are evaluated across each of the three model families to guarantee a confidence level of > 95%. The overwhelming majority of completions are semantically incorrect to the bug or patch, see Figure 3.

```
// Fix bugs in the below function.
...
g2d.setColor(tabFillColor);
g2d.fill(shaper.reset().doRect(boundsX, topY + shape
    .path.deltaY(1),
    boundsWidth, paintBorder.top).getShape());

if (
```

Listing 1: Prompting Codex with hint.

```
// Fix bugs in the below function

// Buggy Java
paintBorder.top >= 1

//Fixed Java
paintBorder.top > 1
...
g2d.setColor(tabFillColor);
g2d.fill(shaper.reset().doRect(boundsX, topY + shape
    .path.deltaY(1),
    boundsWidth, paintBorder.top).getShape());

if (
```

Listing 2: Prompting Codex the bug and fix.

Prompting LLMs with Comments

Large language models were found to be surprisingly effective with good prompting [45]. "Prompt engineering" is the process of constructing a text prompt, either just a textual prefix and/or set of explicit instructions to induce generation of desired text. Prompt engineering has shown an effect on fixing programs, and generating solutions to coding questions [21], [22], [46]; see Listing 1 for an example. Effective prompts may include sample input-output pairs [5] or SQL queries [26], [27]; Listing 2 is an example of bug-fixing comments according to the OpenAI API instructions [44]. This form of traditional hard-prompting [46], named hard because it uses hard-coded language, requires a *prior viz.* some known information about the input, *e.g.*, it is buggy. In Codex experience surveys [10], it appears that prompt engineering is not useful *in situ*, for actual coding. Still, prior work suggests prompt engineering can sometimes be useful [5], [46], [47].

[2]The annotation tool is a fork from localturk, a tool designed to emulate Amazon's Mechanical Turk. https://github.com/danvk/localturk

Original Code

```
else  {
    Color tabFillColor = getActiveTabFillIn();
    if (tabFillColor == null) {
        tabFillColor = shape.path.transformY1(shape.to, shape.from);
    }

    g2d.setColor(tabFillColor);
    g2d.fill(shaper.reset().doRect(boundsX, topY + shape.path.deltaY(1),
            boundsWidth, paintBorder.top).getShape());

    if (
```

One Comment

```
else  {
    Color tabFillColor = getActiveTabFillIn();
    if (tabFillColor == null) {
        tabFillColor = shape.path.transformY1(shape.to, shape.from);
    }

    g2d.setColor(tabFillColor);
    g2d.fill(shaper.reset().doRect(boundsX, topY + shape.path.deltaY(1),
            boundsWidth, paintBorder.top).getShape());

 (1) //If the top of the border is a non-paint border, then the border is painted.
    if (
```

Two Comments

```
else  {
    Color tabFillColor = getActiveTabFillIn();
    if (tabFillColor == null) {
        tabFillColor = shape.path.transformY1(shape.to, shape.from);
    }

 (2) //This assumes that the tabs are filled with a background color.
    g2d.setColor(tabFillColor);
    g2d.fill(shaper.reset().doRect(boundsX, topY + shape.path.deltaY(1),
            boundsWidth, paintBorder.top).getShape());

 (1) //If the top of the border is a non-paint border, then the border is painted.
    if (
```

Fig. 4: Adding neural-generated comments, step by step, in the prompt preceeding the SStuB. The first added comment induces greatest improvement in generated code.

We hypothesize including comments within a code prompt *might* pre-condition the model better, to produce more relevant and better overall code, in a couple ways: (1) generated code, if relevant, will be easier to understand; previous works show that comments help code comprehension [10], [48], [49]. (2) comments will help generated code be more maintainable [50], [51]. This led us to use comments to augment the readability and maintainability of the code prior to the SStuB for Codex. Figure 4 illustrates the incremental addition of comments starting around the SStuB, and then to surrounding code. We *automatically* generate comments using CodeTrans comment generation model [33], trained on the DeepCom dataset [52]. The comment generation model [33] can use any number of statements to condition its outputs on; we chose to use two statements for each comment, plus the buggy or fixed line, to keep the comment related to the SStuB.

Comments can be generated from either the buggy or fixed version of code. Comments generated from the fixed

version of the code should represent the correct logical steps through the single statement bug. On the other hand, comments generated from the buggy version should represent mostly correct steps with a minor single statement mistake. We test all models using *both versions* of generated comments; the fixed version representing a "non-buggy" comment and the buggy version a "buggy" comment; this facilitates an evaluation of how non-buggy vs. buggy comments influence Codex's ability to avoid/make a single-statement mistake. We suspect that commenting in general, regardless of minor mistakes in natural language descriptions, will still condition Codex to use more reliable, well commented data for generation. Figure 4 shows the incremental addition of comments with automatic comment generation tools. We find that the comments *are* beneficial for Codex models, irrespective of the developer's minor misunderstanding as conveyed in comments.

Prompt input, length, and SStuB completion

The code prior to the SStuB, if available, is the conditional input to the model or prompt. The prompt, or code prior to the SStuB, is identical for all 16,899 SStuBs which guarantees the model is not biased towards the bug or the fix. When commenting the SStuB, an automatically generated comment from CodeTrans is placed above the lines used to generate it, properly tabulated, such that the comment appears natural as a developer would place it; see Figure 1. The prompt includes the code prior to the bug and up to the maximum allowed amount for each model. The length of any comments reduces the available input we can pass to the LLMs due to the fixed-length token window. The token window for Davinci is 8000 and the other models, Cushman, PolyCoder and CodeGen, are 2048. The token window is ultimately reduced further by the code completion length as the model performs generation in an auto-regressive fashion.

LLM code completions can span several lines. Codex *can* use a stop token, such as the newline character, to terminate completion early. We found that LLMs, like Codex, often add arbitrary newlines and whitespace to completions; thus terminating completion on a newline might otherwise leave unmatched completions. Instead, we ask the LLMs to complete a length of 64 tokens, which is sufficient for almost all SStuBs; SStuBs have a mean length 29 tokens and a median length 25 tokens. After generating a sequence of length 64, the completion is compared to the SStuB ignoring whitespace. The generated sequence *must* match the SStuB completely to count as a bug or fix.

In the next section, we present the results from our findings: how often Codex and LLMs produce SStuBs, the number of commits to fix the generated SStuBs, and how annotating code with comments can improve performance on SStuBs.

IV. Results

A. SStuB production in LLMs (RQ1)

Table I is the number of bugs, patches, and non-matching completions from studied LLMs. We use the bug/patch ratio, as a metric to universally compare models as the overall number of successful completions, either a SStuB or a patch, varies between models. Codex, PolyCoder, and CodeGen 350M all produce nearly 2x as many bugs as patches. *Davinci-codex* and *cushman-codex* perform surprisingly poorly given their size, and extensive training. It is plausible that the Codex models recapitulate bugs seen in training data [21], however, many of these SStuBs will have been addressed per their inclusion in the ManySStuBs4J corpus two years ago; *viz.* there is a fix. Although Codex produces a high rate of SStuBs, Codex is capable of avoiding 13.41% of SStuBs in the dataset.

TABLE I: SStuB production rate on off-the-shelf LLMs

(a) LLM completed SStuBs vs. correct code.

Model	Bugs	Patches	No Match	Bug/Patch	Match Rate (%)
PolyCoder 160M	3429	1635	11835	2.10	29.97
PolyCoder 0.4B	3672	1852	11375	1.98	32.69
PolyCoder 2.6B	3924	2096	10879	1.87	35.62
CodeGen 350	3709	1911	11279	1.94	33.26
CodeGen 2B	4102	2756	10041	1.49	40.58
CodeGen 6B	4168	2944	9787	1.42	42.09
CodeGen 16B	4299	3296	9304	1.30	44.94
Cushman 12B	3775	1833	11291	2.06	33.19
Davinci 175B	4452	2267	10180	1.96	39.76

(b) Manually examined model predictions when neither bug or patch *a.k.a.* "No Match" is detected.

Model	**PolyCoder 2B**		**CodeGen 16B**		**Davinci**	
	counts	%	counts	%	counts	%
Incorrect	361	90.02	357	89.03	362	90.27
Patch	19	4.74	28	6.98	19	4.74
Bug	15	3.74	10	2.49	14	3.49
Unsure	6	1.50	6	1.50	6	1.50

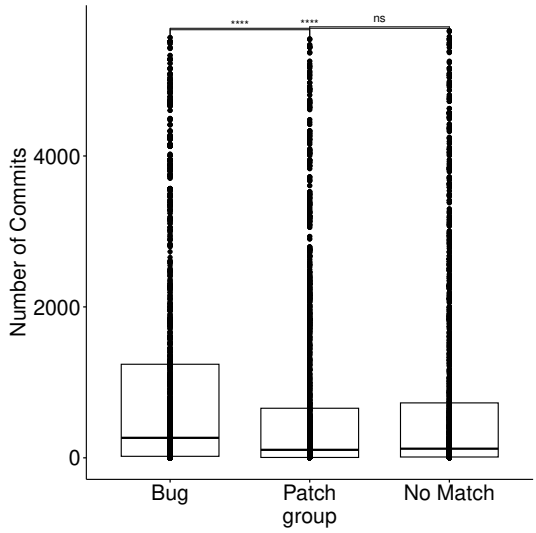


Fig. 5: Developers take more time, measured in commits, to resolve SStuBs that Codex generates. All differences are pairwise statistically significant to $p \leq 0.0001$.

RQ1: Codex and LLMs produce twice as many SStuBs as correct code. Codex manages to avoid 13.41% of SStuBs.

TABLE II: Bug Rate and Patch Rate after adding a comment around the SSTUB.

Model Name	Model Size (Billions)	Bug Change	% Change	Patch Change	% Change	Bug/Patch Ratio Change	% Change	Match Rate Change
PolyCoder 160M	0.16	-427	-12.45	150	9.17	-0.42	-20.42	-1.64
PolyCoder 400M	0.4	-376	-10.24	250	13.50	-0.41	-21.94	-0.75
PolyCoder 2.6B	2.6	-407	-10.37	323	15.41	-0.42	-23.23	-0.50
CodeGen 350M	0.35	-427	-11.51	256	13.40	-0.43	-21.70	-1.01
CodeGen 2.7	2.7	-655	-15.97	129	4.68	-0.29	-18.73	-3.11
CodeGen 6B	6.1	-742	-17.80	-174	-5.91	-0.18	-11.59	-5.42
CodeGen 16B	16.1	-759	-17.66	-81	-2.46	-0.20	-10.24	-4.97
Codex Cushman	12.0	800	21.19	2329	127.06	-0.96	-44.90	18.52
Codex Davinci	175.0	877	19.70	2882	127.13	-0.93	-46.08	22.24

TABLE III: LLM completed SStuBs vs correct code with a comment prior.

Model	Bugs	Patches	No Match	Bug\Patch	Match Rate (%)
PolyCoder 160M	3002	1785	12112	1.68	28.33
PolyCoder 0.4B	3296	2102	11501	1.57	31.94
PolyCoder 2.6B	3517	2419	10963	1.45	35.13
CodeGen 350M	3282	2167	11450	1.51	32.24
CodeGen 2B	3447	2885	10567	1.19	37.47
CodeGen 6B	3426	2770	10703	1.24	36.66
CodeGen 16B	3540	3215	10144	1.10	39.97
Cushman 12B	4575	4162	8162	1.10	51.70
Davinci 175B	5329	5149	6421	1.03	62.00

B. Number of commits to fix LLM produced SStuBs (RQ2)

Figure 5 shows the number of commits to fix of SStuBs where Codex generates the original human-created 'Bug', or a 'Patch', or something else ('No Match'). For each of these categories, we examine the version-control history to examine how long (count of commits from introduction to fix, using `git blame`) developers took to fix them. Unfortunately, the number of commits to fix when Codex (re)produces SStuBs (bugs) *is significantly longer* than in other cases. The median number of commits to fix for the bugs, patches, and no match is 265, 106, and 121 commits respectively. Significance of pairwise t-tests was sustained even after the conservative Bonferroni correction. This finding suggests that *when Codex generates SStuBs, these might inherently take human developers longer to fix*! If used widely in open-source code, Codex might spout SStuBs that live longer (in version history) and further pollute future Codex training data. We believe future, detailed investigation in the 4452 matching SStuBs might help improve Codex.

RQ2: The ManySStuBs4J data suggest that in cases where Codex wrongly generates simple, stupid bugs, these may take developers significantly longer to fix than in cases where Codex doesn't.

C. SStuB regularity (RQ3)

The significant number of commits to fix SStuBs vs. patches (RQ2) motivates a comparison of the "naturalness" of bugs, patches, and no match group of SStuBs. Figure 6 shows that there is little difference between the negative log-likelihood of bugs and patches. As expected, the 'no-matches' have a higher negative log-likelihood, since these completions were presumably not seen in the training set. The similar negative log-likelihood of SSTuBs and patches suggests that

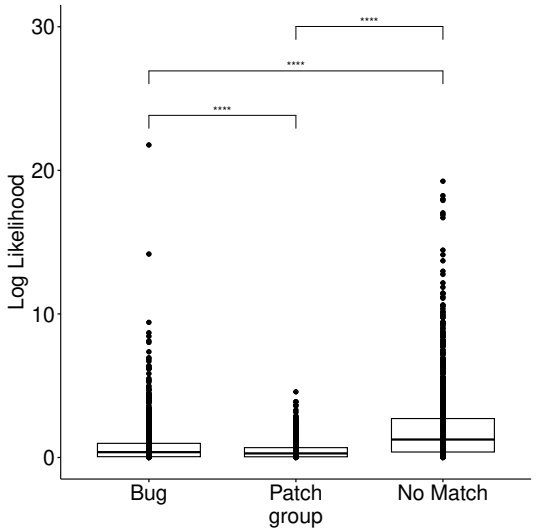


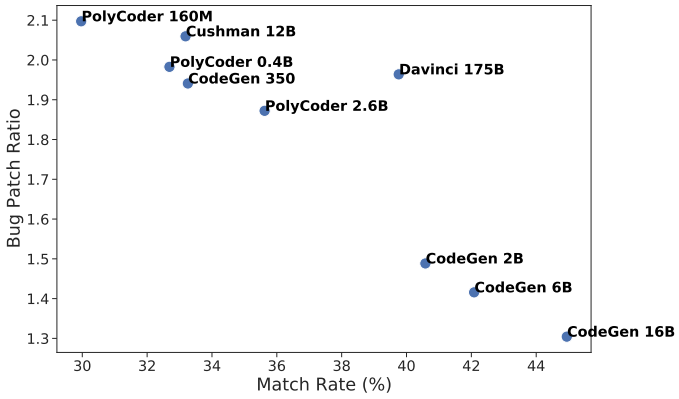
Fig. 6: SStuB bugs and patches from Codex are equally more natural than other code it generates. All differences are pairwise statistically significant to $p \leq 0.0001$.

it may be challenging to fine-tune Codex to detect or avoid SStuBs, since Codex rates them both equally 'natural'; we leave this for future work. However we do study if proper prompt engineering (*e.g.,* with comments), might help matters.

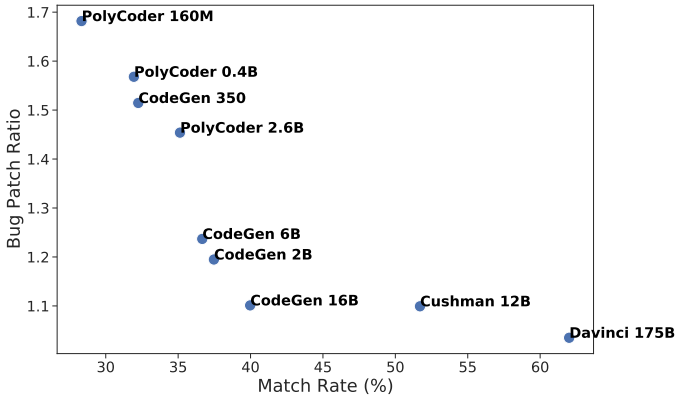
RQ3: Codex log-probabilities indicate that SStuBs (bugs and patches) are regular and natural, thus making detection difficult.

D. Avoiding SStuBs (RQ4)

We now turn to the question of whether adding natural language comments to the prompt suppresses SStuB generation by Codex. Table III shows the results of inserting a single comment into the prompt. Table II captures the rate change in bugs, patches, and bug/patch ratio after adding a comment prior to the SStuB. First, Codex and other LLMs PolyCoder and CodeGen behave differently, namely in the match rate; Codex match rate increases by 19-22% where as other LLMs do not change much. In PolyCoder and CodeGen, the number of bugs decreases from 10-18% and the patch rate increases. The bug/patch ratio in all models improves! Codex Cushman and Davinci generate 20% more bugs, but then produce *127%* more patches. The bug/patch ratio is cut by almost half. This



(a) Codex models (Cushman & Davinci), without comments, perform not well on bug/patch ratio and the match rate.



(b) Commenting code helps Codex achive best performance across SStuBs while improving the match rate;

Fig. 7: Prompting with comments should be used to both avoid SStuBs

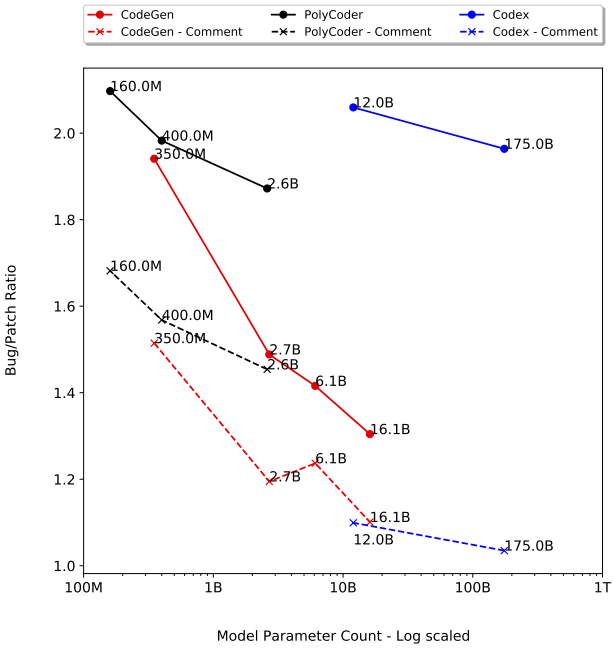


Fig. 8: Bug/Patch Ratio vs. Parameter Count. Three model families at various sizes. The largest difference is the addition of 1 comment prior to the SStuB.

suggests that developers get better results by commenting code while using Codex: this amounts to about 3000 more patches (5149 vs. 2267).

Figure 7 are scatter plots showing the relationship between match rate and bug/patch ratio. Ideally, models will have a high match rate (less unknown cases) and a low bug/patch ratio. Per Figure 7a, Codex models Cushman and Davinci were not competitive to other off the shelf models. After adding comments to the SStuB prone code, Figure 7b, all models perform better and Codex performs *much* better than the next best model CodeGen 16B.

Figure 8 shows the effect of adding comments to the bug/-patch ratio with model parameter counts. Adding a comment in the prompt helps more than increasing parameter counts! A 160M parameter PolyCoder with a comment outperforms (by 14% improvement in bug/patch ratio) both the Codex Cushman 12B and Davinci 175B, without comments.

Finally, Figure 9 shows bug/patch ratio (Figure 9a) and number of patches (Figure 9b) by the bug type (given in the ManySStuBs4J dataset) ranked left to right by the frequency in the dataset. The largest improvements in bug/patch ratio in a sufficient set of samples are `LESS_SPECIFIC_IF`, `MORE_SPECIFIC_IF`, `OVERLOAD_METHOD_MORE_ARGS`, and `DIFFERENT_METHOD_SAME_ARGS`. The bug types that gained the most patches are `CHANGE_IDENTIFIER` and `DIFFERENT_METHOD_SAME_ARGS`. `SWAP_ARGUMENTS` got worse, but only consists of 4 total examples, and it is hard to make any conclusions from this.

E. Commenting vs Traditional Prompting (RQ4 cont.)

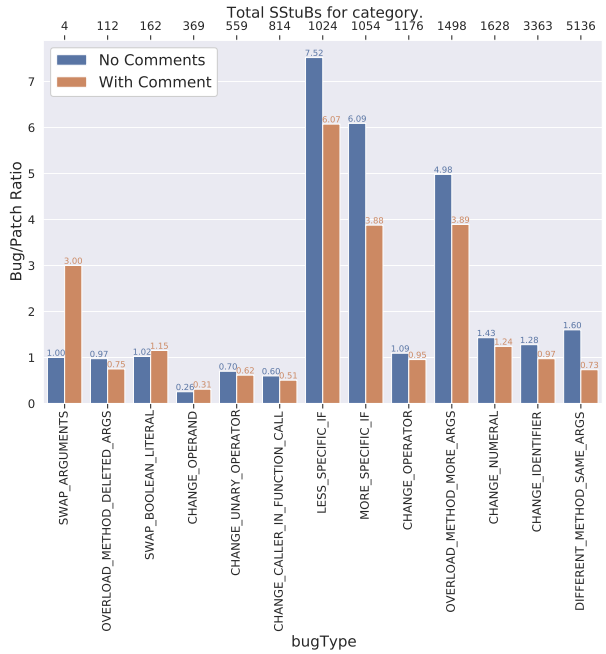
Prompting LLMs both in NLP and SE applications often come in the form of instructions prior to the input. Previous empirical studies of Codex [1], [5], [7], [16]–[20], [22], [46] prompt Codex with instructions, input-output pairs, or examples prior to the input that the model is conditioning the output on. We compare in a similar fashion to Prenner *et al.* [5] by providing Codex with hints (Listing 1), and SStuBs (Listing 2). We remind the reader that both prompting techniques found commonly in previous works, require knowing something about the snippet of code that will be generated; *e.g.*, a location in the code that has a SStuB (hint) and what the SStuB is and how to fix it (bug and fix).

While both approaches improve the bug/patch ratio and match rate, Table V, neither does as well as adding natural language comments. Furthermore the number of bugs greatly increases in the traditional prompting techniques.

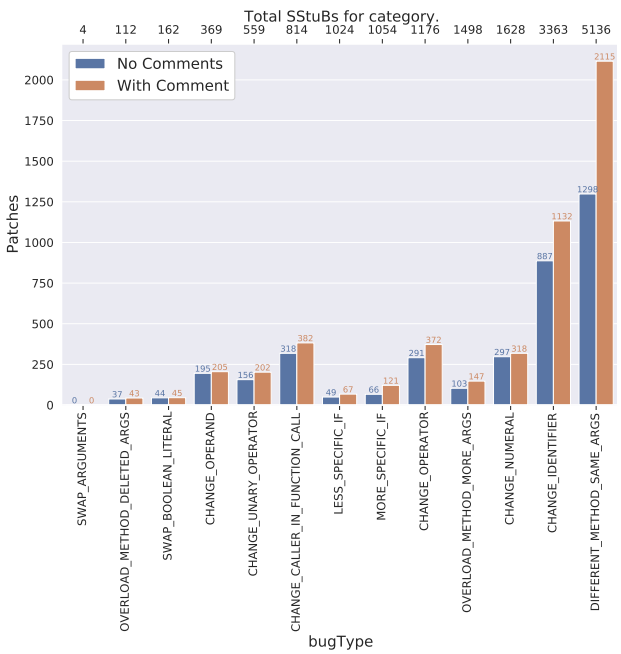
RQ4: Commenting code can lead to less generated SStuBs and more generated patches. Codex models improve the most from code comments.

TABLE IV: Bug Rate and Patch Rate after adding erroneous comments around SSTUB.

Model Name	Model Size (Billions)	Bug Change	% Change	Patch Change	% Change	Bug/Patch Ratio Change	% Change	Match Rate Change
PolyCoder 160M	0.16	59	1.72	-237	-14.50	0.40	18.97	-1.05
PolyCoder 400M	0.4	171	4.66	-132	-7.13	0.25	12.69	0.23
PolyCoder 2.6B	2.6	234	5.96	-145	-6.92	0.26	13.84	0.53
CodeGen 350M	0.35	110	2.97	-178	-9.31	0.26	13.54	-0.40
CodeGen 2.7	2.7	-78	-1.90	-420	-15.24	0.23	15.74	-2.95
CodeGen 6B	6.1	-264	-6.33	-603	-20.48	0.25	17.79	-5.13
CodeGen 16B	16.1	-95	-2.21	-558	-16.93	0.23	17.72	-3.86
Codex Cushman	12.0	1511	40.03	1135	61.92	-0.28	-13.52	15.66
Codex Davinci	175.0	1620	36.39	1613	71.15	-0.40	-20.31	19.13



(a) Comment effect on bug/patch ratio. Lower is better. Top axis is total SStuB count, little significance placed on bug categories with less than 100 samples.



(b) Comment effect on patches. Higher is better. Top axis is total SStuB count, little significance placed on bug categories with less than 100 samples.

Fig. 9: Effect of adding comments to Codex Davinci prompt.

TABLE V: Codex (Davinci) performance across various prompting techniques.

Prompt	Bugs	Patches	No Match	Bug/Patch	Match Rate (%)
Hint	6228	4814	5857	1.29	65.34
Bug & Fix	6565	6055	4279	1.08	74.68
Comment	5329	5149	6421	1.03	62

F. What if we insert a 'buggy' comment? (RQ5)

Finally, we try inserting a *buggy* comment in the prompt, to mimic the developers description of the SStuB. The natural language comment for buggy code is created by conditioning CodeTrans on the SStuB (bug) rather than correct code (patch). Table IV shows the difference in bugs, patches, bug/patch ratio, and match rate with a buggy comment before the SStuB. Surprisingly, Codex is robust and still improves bug/patch ratio over the "no comments" case. This suggests that the mere presence of relevant comments in the prompt sufficiently pushes the model to produce better code. It's also interesting to note that the lower capacity models (and also the 16B parameter CodeGen) tend to be misled by the 'buggy' comments, whereas the larger capacity, well-trained Codex models are not.

RQ5: Misleading comments still condition Codex to produce less SStuBs. Commenting appears beneficial irrespective of the developer's understanding of the SStuB.

V. Discussion

A. Implications of Findings

Implications of Codex Producing SStuBs

The good news: Our study suggests that LLMs like Codex do help avoid a significant number of SStuBs in our dataset, out-of-the-box, and even more with non-buggy comments! But they do produce simple, stupid bugs. Even Codex produces up to 2x more SStuBs as patches, if used directly, nor does increasing model size (see Table I) necessarily help.

To better understand why Codex still produce SStuBs, one must further examine the training data (sadly, not available for many LLMs); we hope training data will become more available. Previous work [2], [7], [20] studying Codex-generated vulnerabilities blames Codex's language modeling roots, which push it to produce the most "likely completion (for a given prompt) based on the encountered samples during training". Also, SStuBs are capable of lasting for long periods of time [29] and are not detected by continuous integration [30] or static analysis [28], [29], [31] which explains why Codex recapitulates them (from it's training data). Simple, stupid bugs are likely regularly injected by devs; training Codex without SStuBs would be challenging, given training data is drawn from 54 million repositories [4]. The effect of Codex produced SStuBs is significant, and troubling. The number of commits to fix Codex produced SStuBs versus the avoided SStuBs is significant, taking more than twice as long to fix. Still we should bear in mind that Codex avoids 2,267 bugs on its own or 13.41% of the dataset, indicating an AI paired programmer is helpful in avoiding SStuBs too.

Avoiding SStuBs with LLM

Codex and other LLMs respond unpredictably to prompts, and developers often struggle to get LLMs to generate desired code [10]. Studies suggest that breaking coding tasks into manageable sub-problems helps [10], [27]. NLP tasks work similarly; chain-of-thought [53] and reasoning step-by-step [54] improve problem-solving rate. Commenting is ideally a form of step-by-step reasoning, explaining high level steps, or clarifying confusing code. The generated comments we used appear to be high-level descriptions and not deep technical commentary of computed values and algorithmic mechanisms. Our work suggests minimal effort techniques, like automatically generated documentation, may help avoid SStuBs when using an AI programming assistant like Copilot. Not only can comments be automatically generated as documentation, but comments can be used directly as a prompt for Codex. To the best of our understanding, Codex might condition the generated code on a smaller search-space of non-buggy solutions, thus helping the developer avoid introducing SStuBs.

Lastly, comments can sometimes be used to check the implementation consistency given the desired functionality [55]. Future work could examine if SStuBs can be detected with the same tools given a set of generated comments.

In our experiments the placement of comments is uniform, and further work should be done to determine best possible comment placement in a density that is adequate and not excessive; automatic methods exist [56]. Excessive commenting is typically symptomatic of a lack of understanding of the code and a "code smell" [57].

Maintaining AI Generated Code

Language models for code like Codex, PolyCoder, CodeGen, and others [39], [58]–[61] will become bigger, and better at code completion [62]. In a world where AI programming assistants learn from data at scale [63], it is hard to say how much of novel programming projects or code reuse [43] will guide such tools. Fundamentally, there is a need for improved readability and comprehensibility in AI-generated code [10]. Code comments can improve comprehensibility of inserted code, especially of more difficult statements. Code that is more readable and understandable is much more maintainable. Our work suggests that comments help avoid SStuBs, in addition to the traditional role of improving code readability. Prior work indicates that nearly half of SStuBs are fixed by another developer and with greater effort [34]; note that any code (buggy or not) generated by a language model may be unfamiliar to a developer.

Lastly, the preliminary successes on SStuBs warrants further research in comment generation with AI programming assistants. Comment generation models like DeepCom [52] and CodeTrans [33] are fully automated and could function in a variety of roles for AI programming assistants. For example, comment generation models could serve as automatic code commenting for Codex completions, be used to check for implementation consistency and accuracy, and improve the quality of training data for Codex to name a few. Our approach of using comments with Codex should be reexamined under a variety of applications including program repair and defect prediction. This is an interesting future direction.

VI. Threats to Validity

A. Internal Validity

ManySStuBs4J

We assume the samples in this dataset are mostly actual bugs. The authors report that changes related to refactoring, are removed, but some non-bug-fixing commits may remain.

Manual Inspection

We use automated matching to determine whether the models produce a known SStuB or patch. However, it is possible the automatic evaluation misses semantically equivalent but syntactically different bugs or patches. This could potentially hide the true number of bugs and patches. To reduce this threat to our results, we have three independent raters (the authors) inspect random samples from *davinci-codex* completions (for the cases with no-comments, and the cases with non-buggy comments) that matched neither bug nor fix (we call this unmatched subset "dark matter").

With fair agreement, Fleiss Kappa 0.40, the independent raters found the vast majority (over 80%) to be inappropriate code completions (neither bug nor fix — just wrong), and sparsely little bugs or patches for the no-comment case with Davinci Table Ia. With moderate agreement, Fleiss Kappa 0.6, the independent raters found a smaller majority (about 70%) of inappropriate code generations in the "dark matter sample" for the non-buggy comment case, Table III. The independent ratings all found that, even in the "dark matter" sample, adding comments in the prompt resulted in substantial increase in patches. While we acknowledge that Codex non-match completions pose a threat to our findings, our sampled examination of this "dark matter" in both settings (with and

without comments) suggests that adding comments does help LLMs avoid the generation of SStuBs.

Data Leakage from LLM Training

We cannot independently verify that the ManySStuBs4J dataset is excluded in the training of the models since none of the models' training data is published. "Data leakage" is traditionally a concern when evaluating the performance of language models as data seen during training might artificially inflate results. In our case, data leakage will bias the model towards an outcome either the bug or the fix. We examined the latest fix date for the studied SStuBs and found 100% of the SStuBs were fixed by February 2019 *and* the earliest data collected for training is 2020 (*cushman*) and 2021 (*davinci*). If there is data leakage from ManySStuBs4J, we postulate the models would most likely see the *fixed* version of the code, but intriguingly, the models still produce 2x more SStuBs!

Reproductions of Generations

Depending on hyper-parameters, Codex models are nondeterministic in their text generation (for the same prompt). We sampled the top-1 completion for each ManySStuBs4J sample across all models (Codex, PolyCoder, and CodeGen), with and without comments.

B. External Validity

ManySStuBs4J

Generalizability is subject to the limits of ManySStuBs4J. The dataset consists of Java single statement bugs; our results may not generalize to other languages, or less simple bugs. PySStuBs [41] and TSSB-3M [42] are larger, and cite different SStuB patterns. The ManySStuBs4J dataset is the appropriate size given our constraints on available compute, and also API access to Codex. We were limited by OpenAI's rate ceiling of 20 requests per minute; on local hardware, the largest model, CodeGen 16B takes over a day for a run with a single prompt on ManySStuBs4J.

Models at Scale

Language models are getting ever larger. Results may vary with the next generation of models.

VII. Conclusion

Most importantly, we find that Codex and other large language models *significantly help avoid human-produced simple, stupid bugs*! In our best case, around 30% (5149) SStuBs were actually patched (avoided) by Codex Davinci. Still, we find that large language models might produce many more SStuBs than patches. *First*, Codex and PolyCoder produce nearly twice as many SStuBs as fixes. *Second*, and very worryingly, Codex generated SStuBs apparently took significantly longer to resolve and that the SStuBs appear to be as natural as the correct statements. Our results show that AI pair programming can introduce SStuBs, and the manner in which developers are known to use such tools is not conducive to avoiding SStuBs. Still, though the models were somewhat SStuB prone, even out-of-the-box LLMs could have avoided as many as 2,300 SStuBs had developers used code completion instead of writing them.

Since the simple, stupid bugs are quite obvious after detection, we explore the idea of guiding AI assistants by adding comments describing high-level functionality. The proposed strategy of communicating functional intent to Codex with comments improved the bug/patch ratio substantially. Finally, we explore minor misunderstandings in the intended functionality by using buggy comments and find that Codex may not require strict correctness in comments to avoid SStuBs. Our results suggest that good commenting practices, even in an automatic setting, can help other developers and Codex, especially in an era where AI generated code is regularly committed.

Overall, our findings are somewhat promising, LLMs may help avoid *at least some* simple, stupid bugs!

References

[1] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2022, pp. 1–18.

[2] O. Asare, M. Nagappan, and N. Asokan, "Is github's copilot as bad as humans at introducing vulnerabilities in code?" *arXiv preprint arXiv:2204.04741*, 2022.

[3] A. Ziegler, "https://github.blog/2021-06-30-github-copilot-research-recitation/." [Online]. Available: https://github.blog/2021-06-30-github-copilot-research-recitation/

[4] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[5] J. A. Prenner and R. Robbes, "Automatic program repair with openai's codex: Evaluating quixbugs," *arXiv preprint arXiv:2111.03922*, 2021.

[6] T. Ahmed and P. Devanbu, "Few-shot training llms for project-specific code-summarization," in *37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3551349.3559555

[7] D. Sobania, M. Briesch, and F. Rothlauf, "Choose your programming copilot: a comparison of the program synthesis performance of github copilot and genetic programming," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2022, pp. 1019–1027.

[8] N. Jain, S. Vaidyanath, A. Iyer, N. Natarajan, S. Parthasarathy, S. Rajamani, and R. Sharma, "Jigsaw: Large language models meet program synthesis," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1219–1231.

[9] T. Ahmed, S. Ghosh, C. Bansal, T. Zimmermann, X. Zhang, and S. Rajmohan, "Recommending root-cause and mitigation steps for cloud incidents using large language models," *arXiv preprint arXiv:2301.03797*, 2023.

[10] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *CHI Conference on Human Factors in Computing Systems Extended Abstracts*, 2022, pp. 1–7.

[11] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," *arXiv preprint arXiv:2204.05999*, 2022.

[12] L. Tunstall, L. von Werra, and T. Wolf, *Natural language processing with transformers*. " O'Reilly Media, Inc.", 2022.

[13] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022, pp. 1–10.

[14] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "A conversational paradigm for program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

[15] A. Sarkar, A. D. Gordon, C. Negreanu, C. Poelitz, S. S. Ragavan, and B. Zorn, "What is it like to program with artificial intelligence?" *arXiv preprint arXiv:2208.06213*, 2022.

[16] B. Yetistiren, I. Ozsoy, and E. Tuzun, "Assessing the quality of github copilot's code generation," in *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2022, pp. 62–71.

[17] N. Nguyen and S. Nadi, "An empirical evaluation of github copilot's code suggestions," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 1–5.

[18] G. Sandoval, H. Pearce, T. Nys, R. Karri, B. Dolan-Gavitt, and S. Garg, "Security implications of large language model code assistants: A user study," *arXiv preprint arXiv:2208.09727*, 2022.

[19] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do users write more insecure code with ai assistants?" *arXiv preprint arXiv:2211.03622*, 2022.

[20] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 754–768.

[21] A. Karmakar, J. A. Prenner, M. D'Ambros, and R. Robbes, "Codex hacks hackerrank: Memorization issues and a framework for code synthesis evaluation," *arXiv preprint arXiv:2212.02684*, 2022.

[22] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Can openai codex and other large language models help us fix security bugs?" *arXiv preprint arXiv:2112.02125*, 2021.

[23] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[24] L. Tang, E. Ke, N. Singh, N. Verma, and I. Drori, "Solving probability and statistics problems by program synthesis," *arXiv preprint arXiv:2111.08267*, 2021.

[25] I. Drori, S. Zhang, R. Shuttleworth, L. Tang, A. Lu, E. Ke, K. Liu, L. Chen, S. Tran, N. Cheng *et al.*, "A neural network solves, explains, and generates university math problems by program synthesis and few-shot learning at human level," *Proceedings of the National Academy of Sciences*, vol. 119, no. 32, p. e2123433119, 2022.

[26] N. Rajkumar, R. Li, and D. Bahdanau, "Evaluating the text-to-sql capabilities of large language models," *arXiv preprint arXiv:2204.00498*, 2022.

[27] I. Trummer, "Codexdb: Generating code for processing sql queries using gpt-3 codex," *arXiv preprint arXiv:2204.08941*, 2022.

[28] R.-M. Karampatsis and C. Sutton, "How often do single-statement bugs occur? the manysstubs4j dataset," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 573–577.

[29] B. Mosolygó, N. Vándor, G. Antal, and P. Hegedűs, "On the rise and fall of simple stupid bugs: a life-cycle analysis of sstubs," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 495–499.

[30] J. Latendresse, R. Abdalkareem, D. E. Costa, and E. Shihab, "How effective is continuous integration in indicating single-statement bugs?" in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 500–504.

[31] J. Hua and H. Wang, "On the effectiveness of deep vulnerability detectors to simple stupid bug detection," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 530–534.

[32] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Communications of the ACM*, vol. 59, no. 5, pp. 122–131, 2016.

[33] A. Elnaggar, W. Ding, L. Jones, T. Gibbs, T. Feher, C. Angerer, S. Severini, F. Matthes, and B. Rost, "Codetrans: Towards cracking the language of silicon's code through self-supervised deep learning and high performance computing," *arXiv preprint arXiv:2104.02443*, 2021.

[34] W. Zhu and M. W. Godfrey, "Mea culpa: How developers fix their own simple bugs differently from other developers," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 515–519.

[35] F. Madeiral and T. Durieux, "A large-scale study on human-cloned changes for automated program repair," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 510–514.

[36] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queen's School of computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[37] A. Peruma and C. D. Newman, "On the distribution of" simple stupid bugs" in unit test files: An exploratory study," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 525–529.

[38] E. Mashhadi and H. Hemmati, "Applying codebert for automated program repair of java simple bugs," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 505–509.

[39] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[40] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, 2019.

[41] A. V. Kamienski, L. Palechor, C.-P. Bezemer, and A. Hindle, "Pysstubs: Characterizing single-statement bugs in popular open-source python projects," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 520–524.

[42] C. Richter and H. Wehrheim, "Tssb-3m: Mining single statement bugs at massive scale," *arXiv preprint arXiv:2201.12046*, 2022.

[43] M. Allamanis, "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*, 2019, pp. 143–153.

[44] [Online]. Available: https://beta.openai.com/examples?category=code

[45] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *arXiv preprint arXiv:2107.13586*, 2021.

[46] C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang, and M. R. Lyu, "No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 382–394.

[47] F. Petroni, P. Lewis, A. Piktus, T. Rocktäschel, Y. Wu, A. H. Miller, and S. Riedel, "How context affects language models' factual predictions," *arXiv preprint arXiv:2005.04611*, 2020.

[48] T. Tenny, "Program readability: Procedures versus comments," *IEEE Transactions on Software Engineering*, vol. 14, no. 9, p. 1271, 1988.

[49] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen, "The effect of modularization and comments on program comprehension," in *Proceedings of the 5th international conference on Software engineering*, 1981, pp. 215–223.

[50] C. S. Hartzman and C. F. Austin, "Maintenance productivity: Observations based on an experience in a large system environment," in *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering-Volume 1*, 1993, pp. 138–170.

[51] B. P. Lientz, "Issues in software maintenance," *ACM Computing Surveys (CSUR)*, vol. 15, no. 3, pp. 271–278, 1983.

[52] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 2018, pp. 200–20 010.

[53] J. Wei, X. Wang, D. Schuurmans, M. Bosma, E. Chi, Q. Le, and D. Zhou, "Chain of thought prompting elicits reasoning in large language models," *arXiv preprint arXiv:2201.11903*, 2022.

[54] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," *arXiv preprint arXiv:2205.11916*, 2022.

[55] F. Wen, C. Nagy, G. Bavota, and M. Lanza, "A large-scale empirical study on code-comment inconsistencies," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 53–64.

[56] Y. Huang, X. Hu, N. Jia, X. Chen, Y. Xiong, and Z. Zheng, "Learning code context information to predict comment locations," *IEEE Transactions on Reliability*, vol. 69, no. 1, pp. 88–105, 2019.

[57] P. Becker, M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[58] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[59] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.
[60] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," *arXiv preprint arXiv:2103.06333*, 2021.
[61] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," in *International Conference on Machine Learning*. PMLR, 2020, pp. 5110–5121.
[62] V. J. Hellendoorn and A. A. Sawant, "The growing cost of deep learning for source code," *Communications of the ACM*, vol. 65, no. 1, pp. 31–33, 2021.
[63] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," *arXiv preprint arXiv:2006.03511*, 2020.