Discovering the Syntax and Strategies of Natural Language Programming with Generative Language Models

Ellen Jiang
Google Research
Cambridge, Massachusetts, USA
ellenj@google.com

Edwin Toh
Google Research
Mountain View, California, USA
edwintoh@google.com

Alejandra Molina
Google Research
New York, New York, USA
alemolinata@google.com

Kristen Olson
Google Research
Seattle, Washington, USA
kristenolson@google.com

Claire Kayacik
Google Research
Mountain View, California, USA
cbalgemann@google.com

Aaron Donsbach
Google Research
Seattle, Washington, USA
donsbach@google.com

Carrie J. Cai
Google Research
Mountain View, California, USA
cjcai@google.com

Michael Terry
Google Research
Cambridge, Massachusetts, USA
michaelterry@google.com

ABSTRACT

In this paper, we present a natural language code synthesis tool, GenLine, backed by 1) a large generative language model and 2) a set of task-specific prompts that create or change code. To understand the user experience of natural language code synthesis with these new types of models, we conducted a user study in which participants applied GenLine to two programming tasks. Our results indicate that while natural language code synthesis can sometimes provide a magical experience, participants still faced challenges. In particular, participants felt that they needed to learn the model's "syntax," despite their input being natural language. Participants also struggled to form an accurate mental model of the types of requests the model can reliably translate and developed a set of strategies to debug model input. From these findings, we discuss design implications for future natural language code synthesis tools built using large generative language models.

CCS CONCEPTS

• **Human-centered computing** → **Human computer interaction (HCI)**; *User studies*; • **Computing methodologies** → *Artificial intelligence*; • **Software and its engineering** → *Software notations and tools.*

KEYWORDS

generative language models, prompt programming, code synthesis

ACM Reference Format:
Ellen Jiang, Edwin Toh, Alejandra Molina, Kristen Olson, Claire Kayacik, Aaron Donsbach, Carrie J. Cai, and Michael Terry. 2022. Discovering the Syntax and Strategies of Natural Language Programming with Generative Language Models. In *CHI Conference on Human Factors in Computing Systems (CHI '22), April 29-May 5, 2022, New Orleans, LA, USA.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3491102.3501870

1 INTRODUCTION

Generative models (e.g., [13, 15, 28]) are designed to create plausible continuations of their input. For example, given the text, "After work, I went to the", a language model could then generate sentences or even paragraphs of text that resemble the start of a story or conversation.

Recent large language models (LLMs), such as GPT-3 [15], have demonstrated that they can perform a wide range of text-based tasks by carefully crafting the input to the model. For example, to produce HTML from a natural language description, one can prime the model using text containing examples of goals (expressed in natural language) and their corresponding HTML (see online example [7]). For instance, one can provide the text shown below:

```
description: a red button that says stop
html: <button style = 'color: white; background-color: red'>Stop</button>
description: A textfield that says Hello in its placeholder
html: <input type='text' placeholder='Hello'></input>
description: a blue button with white text that says Submit
html:
```

Given this text as input, the model is likely to produce the HTML corresponding to "a blue button with white text that says Submit"[1]. What is notable about the above example is that it is merely the text-based input to the model: the model is able to follow the pattern established by the input, and produce HTML code corresponding to the natural language description at the end of the input. In effect, the input enables one to "customize" the LLM to perform specific tasks, such as code synthesis.

[1] When providing this input to the model used in this research, it produced the following code: <button style = 'color: white; background-color: blue'>Submit</button>

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.
CHI '22, April 29-May 5, 2022, New Orleans, LA, USA
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9157-3/22/04…$15.00
https://doi.org/10.1145/3491102.3501870

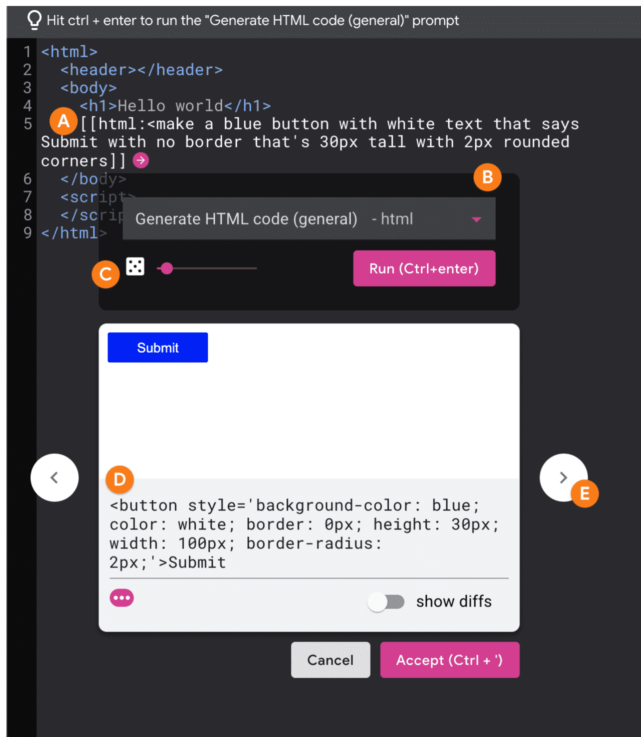

Figure 1: GenLine provides a front-end interface to task-specific prompts for the generative language model. In this example, the GenLine tool is using a prompt that converts a natural language description into HTML. The user input (A) is demarcated using a double-bracket notation. The user can choose which prompt to apply in a drop-down menu (B). The generated content is visible in (D), and is editable before insertion into the document. The user can request that the model produce more content by clicking the ellipses on the bottom left of the dialog box. By default, the model will generate alternatives, which can be navigated using the buttons indicated by (E). The degree of variety in the output is variable with a temperature slider (C).

This type of guiding input is often called a *prompt* [2, 15]. At the most basic level, a prompt is nothing more than the text inputted to an LLM. However, a number of compelling GPT-3 demos [15] demonstrate that prompts can be written to customize a single model to perform a wide range of tasks, such as transforming natural language instructions into fiction, SVG graphics, or source code (among many other types of output) [2–4, 6]. Note that the model itself does not change—only the input provided to the model changes to achieve these results. The ability for these models to perform a wide variety of tasks, without needing to retrain them, makes them of inherent interest to the HCI community, as they provide new ways to customize and integrate AI into everyday tasks.

In this paper, we examine the user experience of using an LLM to produce code from natural language descriptions. To investigate this experience, we created GenLine, a tool that provides inline support for transforming natural language requests into code (Figure 1). Using GenLine, users can input requests such as "make an OK button", or "make this button blue <button>OK</button>". GenLine transforms these requests into code by incorporating the user's input into a task-specific prompt (e.g., a prompt to create HTML, or a prompt to produce JavaScript), then feeding the prompt as input to an LLM. Because users explicitly express a request to the LLM (i.e., "make an OK button"), and choose a specific prompt for interpreting that input (e.g., a prompt to produce HTML, or a prompt to produce JavaScript), using GenLine is more akin to invoking a command, as opposed to a code completion mechanism.

To understand how this new class of large language models may affect users' software development practices, we conducted a user study in which 14 participants were asked to use GenLine to create two small web-based applications: a static search page and an interactive flashcard app. The results of our study provide key insights into the user experience of producing code with this new class of generative language model. In particular, we find that while the natural language code synthesis capabilities can be useful for certain tasks (e.g., creating boilerplate code, or doing the equivalent of an API lookup), users can still encounter challenges in several areas: 1) learning the model's natural language "syntax" (i.e., *how* to effectively phrase a natural language request), 2) knowing *what* they can reliably ask of the model, including *how much* they can request, and 3) *debugging* the model when it doesn't produce the desired results.

Collectively, these results suggest a number of implications for design. In particular, future systems may benefit from techniques that *automatically reformulate user input* (similar to what is done in information retrieval systems [27]) and *automatically vary model temperature* to increase the likelihood of producing useful code (where temperature can roughly be thought of as the "randomness" of the model's output[2]). For large and/or ambiguous requests, there may be an opportunity to leverage these models' capabilities to engage in a more structured *conversation* with the user to help scope and refine requests. Future systems may also benefit from *providing suggestions* for what could be requested, given a particular context. Finally, both *debugging tools* (such as interpretability tools) and *AI onboarding* [17] (training users on the model's relative strengths, weaknesses, and quirks) could be useful to help users debug, better understand, and predict model behavior.

In sum, the paper's contributions are as follows:

(1) We present an end-user tool, GenLine, that translates natural language requests into code, using a 137-billion parameter LLM and LLM prompts designed to synthesize code.
(2) We report results from a study investigating use of GenLine for two tasks. Our results highlight:
   (a) A sense of needing to learn the system's "*syntax*," despite model input consisting of natural language.
   (b) Challenges in forming an accurate mental model of the types of requests the model can reliably translate to code (*what* can be requested and *how much* can be requested).

[2] We make use of temperature sampling to sample tokens from the model. Setting the temperature parameter to 1 is equivalent to sampling from the natural distribution of the model. Lowering the temperature results in sampling outputs with higher probabilities, making the model output more predictable (and thus, less variable).

    This uncertainty led to requests ranging in scale and specificity from "create a flashcard app" (a large, under-specified request) to "create a blue button with label flip in white text" (a very specific request).
    (c) Participants' *model debugging strategies* to debug model input: Removing information, adding information, rewording, varying model temperature, and introducing keywords like "hello world" or "test".
(3) From the study findings, we derive a number of implications for design, including:
    (a) Providing *automated input reformulation* and *automated variation of model temperature*.
    (b) Leveraging the generative language model to engage in a more structured *conversation* with the user to gather a more precise and more tightly scoped request.
    (c) Providing *suggestions* of natural language requests a user could make in a given context.
    (d) Offering *debugging tools*, such as interpretability tools, and *AI onboarding* to help users better understand, debug, and predict model behavior.

For many of our design implications, we show how the model itself may be able to assist with input rewriting through a set of "fallback prompts" that transform requests into forms more likely to yield useful results.

The rest of the paper contextualizes this research in the larger body of literature, describes the GenLine tool and implementation, and presents our study and study findings. We conclude with a discussion of implications for design.

2 RELATED WORK

Our work builds on recent advances in generative language models, specifically their ability to generate or modify code from natural language instructions through prompt programming. In this section, we review prior work in generative language models, software engineering, and end-user and natural language interfaces for AI, showing how this prior work informs and inspires our research.

2.1 Generative Language Models

Generative language models such as GPT-3 [15] demonstrate the ability to produce useful or interesting content from high-level natural language inputs. For example, OpenAI and GPT-3 users have shown GPT-3's ability to convert natural language descriptions to SVG [6], SQL [8], shell scripts [4], and Python [3].

One of the compelling aspects of these demonstrations is that the results are achieved by providing a text-based input, or *prompt*, which influences the model output, with no model retraining necessary. For example, to generate SVG graphics code, it may be sufficient to provide a prompt that employs a pattern like the following: "Q: «SVG description» A: «SVG code» Q: «Desired SVG to generate» A:" (where text and code replaces content indicated by the «» symbols, as in this demo [6]). This ability to produce custom functionality through high-level descriptions has given rise to *prompt programming* (for example, see Gwern.net's essays on prompt programming [2] or OpenAI's Prompt Library [5]). This research builds on this prior work, creates a number of task-specific prompts to assist with software development, and examines their use in a user study.

2.2 User Tools for Generative Models

Research in generative models has produced a number of models that produce content good enough to be used in some real-world contexts (e.g., generative music models [28]). Given these capabilities, there is additional interest in how to integrate this functionality into end-user applications to support co-creation [16]. For example, Cococo [33] integrates a generative music model into an end-user application. In the realm of software development, GitHub Copilot [1] and Codex [18] demonstrate how code synthesis capabilities could be exposed to users in an editor or a chat-like interface, respectively. In studying co-creation with AI, a number of design recommendations have been developed, such as a suggestion to avoid overwhelming the user with AI-generated content [16, 33].

GenLine's design builds upon and utilizes this prior research in a number of ways. Specifically, GenLine produces content in discrete chunks (to reduce the likelihood of overwhelming users with content), provides facilities to navigate alternatives, and allows users to edit content before requesting more generated content from the model. In contrast to Copilot (which provides an autocomplete-like interaction) and Codex (which provides a conversational, chat-like interface), GenLine's interaction is more akin to invoking a command within a text editor.

2.3 Software Engineering Tools

In the area of software engineering, there is active research examining how generative models (and deep learning models in general) may be used to assist with software development. Recent papers survey the state-of-the-art in this field [10, 21], which includes targeted work in code synthesis (e.g., from natural language to code, and code to natural language), detecting code defects, end-user programming, and code translation (from one programming language to another). Interpreting natural language specifications (or intent), in particular, has been a long-standing goal in the research community, with prior work demonstrating systems that can transform natural language descriptions to spreadsheet macros [26], shell scripts [31, 32], SQL queries [36], and data visualizations [34] (among many other targets). We take inspiration from this prior work, and examine the natural language code synthesis capabilities of an LLM from an end-user's perspective.

Transforming natural language specifications to code is considered a form of inductive specification [42]. Programming-by-example is also a form of inductive specification in which users define a function by providing input-output examples (e.g., [19, 39, 42]). One of the known issues with inductive specification is the potential for ambiguity in interpreting the user's input [10, 25, 42]. For example, the user's input to the system may under-specify their actual intent, leading to multiple, valid interpretations of their input. Thus, these systems often include mechanisms to help refine or discover the user's true intent, such as generating additional examples for the user to evaluate in programming-by-example systems [42]; showing alternative, valid interpretations to the end-user for them to choose from [25]; or providing the specification with a combination of natural language and examples [37]. Outside the

realm of software engineering tools, other work examines how to recover from, and disambiguate within, conversational breakdowns (e.g., [30]).

Recognizing the potential for ambiguity in interpreting the user's input, the GenLine tool produces multiple outputs for the user to choose from. Our study results also reinforce the observation that it can be useful to provide mechanisms to discover the user's true intent, particularly when they make ambiguous requests, or when they provide a request that the model cannot correctly interpret.

Research has also examined the *user experience* of using code synthesis capabilities. For example, Weisz et al. [40] ran a design scenario study with developers to examine their willingness to use generative AI in different use cases, and found that developers expressed openness to working with AI for tasks like code migration and translation. Recognizing that errors are inevitable, the researchers note that actual adoption likely hinges on "how many errors are present and the nature of those errors" [40]. Participants in our study similarly highlighted concerns over model accuracy, and suggested the importance of being able to continually improve a model to reduce its errors over time.

To understand the potential for pair programming with an intelligent agent, Kuttal et al. [29] conducted a Wizard-of-Oz study. Their results indicate that agents may serve as useful pair programming partners and alleviate barriers to expertise, albeit at the potential cost of code creativity. Our study explores this question further by examining use of an operational, modern generative language model capable of synthesizing code.

Xu et al. [41] studied developers' use of their natural language code synthesis prototype (NL2Code) for completing a range of programming tasks. One finding from the study was that participants were open to a more constrained syntax, if it would yield more reliable results. Our study results similarly suggest the potential utility in providing clear guidance (or restrictions) on what can be requested from a model, and how to formulate those requests.

Prior research has shown the importance of web search in software development, and illustrated the value of more tightly integrating web search with software development environments. For example, Brandt et al. [14] showed how web search can be integrated with a development environment to ease the process of finding and applying web-based examples to code. When using a generative model trained on a corpus that includes code, it is possible to achieve similar results for some information-seeking needs. More specifically, the user can describe their intent using natural language, with the generative language model producing the relevant code (e.g., as illustrated by GPT-3 demos that convert natural language to code, or that continue writing code for the user [3, 4]). In this context, the natural language input to the model is similar to a search query. However instead of returning related web pages, the model generates code. Our study results suggest that users find this "API lookup" use case a compelling scenario for code synthesis performed by modern generative models.

3 GENLINE

GenLine provides in-editor, inline support for accessing and using generative language model prompts that produce code. Figure 1 provides an overview of GenLine's user interface components.

3.1 Interface

User input to GenLine is a single string of text that can be 1) a natural language text, 2) content (code) to modify, or 3) a mixture of natural language text and content to modify. Given this input, GenLine executes the chosen prompt and produces multiple alternatives, which are de-duplicated and presented to the user (users can cycle through each unique output).

Users specify model input by surrounding it with double brackets[3]. For example, given the following code in an HTML code editor:

```
<button>Submit</button>
```

The user can wrap the code with the following instruction:

```
[[add a border to this <button>Submit</button>]]
```

In our implementation, this bracket notation is automatically recognized by the editor: When the user enters the ending brackets or clicks in a double-bracketed region, the GenLine tool automatically appears.

Importantly, GenLine's design provides an interaction style more similar to invoking a command, and less like autocomplete: the user must explicitly specify the input to send to the model (surrounding code context is not passed to the model), and they must choose which prompt to apply to that input. This interaction design differs from the interaction styles found in GitHub Copilot [1], which provides autocomplete-like functionality, or Codex [18], which enables a more conversational style of code creation. While each design has its trade-offs, we focused on the command-like interaction because it enables creation of prompts that support very specific, targeted tasks (such as changing the styling of an existing HTML element). This interaction style also allows users to both create code from scratch and modify existing code in-place, using the same tool. However, one limitation of our current implementation is that users must explicitly provide code context if they wish the model to make use of that context in interpreting their request.

To streamline the process of invoking a specific prompt, GenLine allows the prompt definition to include a "tag" for invoking the prompt. This tag can then be inserted in the double-bracketed content, pre-pending the actual input to pass to the prompt (e.g., "[[html: make an OK button]]"). When a tag is detected, GenLine automatically loads the specified prompt, saving the user the need to choose it from a menu.

Model input can be interactively edited, and model output can also be edited before inserting it into the text editor. This ability to interactively and iteratively construct output enables interactions similar to live programming [22] or exploratory programming [12]. For example, the user may first produce an "OK" button by typing "make an OK button". After producing an output, they can then edit the original input to include additional requirements, such as styling: "make a light blue OK button that is 30px tall". In our study, we observed participants making use of this ability to incrementally build a request to produce the final, desired output.

Beyond editing the model input, users can adjust the "temperature" using a slider, where a lower temperature indicates the model's higher certainty in its top choices (this can be thought of as varying

[3]Depending on the environment, this double bracket notation may conflict with a language's syntax (e.g., lists in programming languages are often defined using square brackets). For a given environment, a delimiter should be chosen so that it does not (or rarely) conflicts with actual content.

the randomness of the output, with higher temperature values more likely to lead to greater variety of output).

For prompts that produce HTML, GenLine renders the model output in an HTML iframe, providing a way to validate the output at a glance.

3.2 Model and Prompts

GenLine is backed by a version [20] of the model described in [9], which is a 137-billion parameter generative language model. The model's training data includes code, but the model was not specifically trained to support software development.

With this model, we created prompts to: produce HTML and JavaScript; fix code; style code; and add unique IDs to HTML elements. See the Appendix for example prompts. The majority of these prompts were designed using a few-shot prompting pattern (see Figure 2). For example, a prompt may set up a pattern of making a natural language request for HTML code (e.g., "description: An OK button"), with the response being the corresponding HTML (e.g., "html: <button>OK</button>").

The few shot prompts we developed that translate natural language descriptions to code also allow users to mix natural language and code in their input. For example, the user might enter a request such as, "Make this button 30 px tall <button>OK</button>". We call this form of input *mixed inputs*, to capture the notion that users can mix conceptually different types of input in their request to the model (e.g., mixing code and natural language). Notably, we found that the prompts do not need to include examples that mix natural language and code in the natural language "description" fields of the examples: Even without examples that mix natural language and code in the "description" field, the model can often successfully interpret these types of mixed inputs and produce only code as output.

While model performance itself is not the focus of this paper, we did observe through our own testing that the model could produce reasonable HTML and JavaScript code through natural language prompts for simple tasks (see also [11], which examines the model's ability to synthesize Python code). The results of the user study provide further insight into how frequently model output was of use to study participants.

4 USER STUDY

To understand how recent LLMs could affect the software development process, we conducted a study in which participants used GenLine to complete two tasks.

4.1 Study Design and Methodology

The study consisted of a remote pre-interview, two tasks to be completed over the course of one week, and a remote post-interview. Interviews focused on 1) the different strategies used to synthesize code and 2) opportunities for writing coding with natural language.

During the 30 minute pre-interview, participants were shown a 4 minute video overview of the GenLine tool that described the prompts (how they were written), the GenLine tool, and a demonstration of it used to build a todo app.

After watching the video, participants had an opportunity to ask the researcher questions, and participants were asked about their initial impressions. Next, participants were given access to a slide deck that contained slides detailing the capabilities of GenLine. Participants then performed a tutorial task to generate stylized text (specific instructions can be viewed in Appendix C.1). Participants were again asked about their impressions of the tool, and whether they had any additional questions. Finally, researchers described the tasks they would perform over the course of a week. Researchers told participants that the GenLine tool was a prototype and that they may encounter quirks during usage that they should come prepared to discuss in the final interview.

Participants had one week to attempt two tasks. The first task (T1) was to create a static search page with a textbox and a logo. The second task (T2) was to create a flashcard app that changed a card from front to back with the click of a button. See Appendix (section C.2) for task figures. These tasks were chosen to represent a fairly basic programming problem (T1, creating a static web page), and a slightly more complex problem that requires participants to include multiple interactive elements in the app (T2).

Participants were asked to record their use of GenLine (i.e., video recordings of their screen) whenever they worked on these tasks throughout the week. Participants uploaded their recordings to an individual online folder shared with the study researchers. During the post-study interview, participants evaluated their overall experience of coding with natural language and their impressions of using GenLine. All interviews were conducted remotely and were recorded.

To analyze the data, two researchers transcribed the natural language requests from the participant-recorded videos of GenLine usage. We used methods from grounded theory [24] to code the data and characterize participants' request strategy, repair strategy, the content of the request, and the request complexity. The two researchers reviewed each other's notes in a shared spreadsheet, met several times to discuss the codes, and iteratively refined and grouped the codes into higher-level themes.

4.1.1 Participants. To provide a cross-section of user experiences, we recruited participants with differing levels of front-end coding experience. Participants were recruited through an internal message board and completed a screener asking them to identify their front-end coding proficiency. Each participant received a $50 gift card (or an option to donate the gift card equivalent to a charity) for participating.

Our participants comprised the following demographics:

- **Role:** UX Researcher (2), UX Designer (1), Interaction Designer (5), Software Engineer (2), UX Engineer (4)
- **Location:** US (13), India (1)
- **Gender:** Female (7), Non-binary (1), Male (6)
- **HTML/CSS experience:** Somewhat experienced (4), Experienced (4), Very experienced (3), Extremely experienced (3)
- **JavaScript experience:** Not at all experienced (4), Somewhat experienced (2), Experienced (3), Very experienced (2), Extremely experienced (3)

In reporting results, we indicate the participant number and the letter N, I, or E to indicate whether they self-reported novice,

```
description: make a red button that says stop
html: <button style = 'color: white; background-color: red'>Stop</button>

description: give code for an html button with a margin
html: <button style='margin:10px;'></button>

description: <<user input>>
html:
```

Figure 2: A few-shot prompt for transforming natural language descriptions to HTML.

intermediate, or expert knowledge of front-end software development. There were a total of 5 novice[4], 4 intermediate, and 5 expert participants[5].

4.2 Results

A total of 7 hours and 42 minutes of video was recorded by participants. There were 2 hours, 23 minutes of video for T1, and 5 hours, 19 minutes of video for T2. Participants spent a median of 10 minutes on T1 and 25 minutes on T2. A total of 227 model requests were issued for T1, and 301 for T2. For T1, participants individually issued a median number of 12 model requests, and a median of 22 model requests for T2. Twelve out of 14 participants uploaded videos of both tasks, and all completed the interviews.

Seven out of twelve (7/12) participants were able to fully complete T1, and 6/12 fully completed T2, where "fully completed" means that they were able to produce the required interface along with the desired styling. Of the remaining, 4/7 and 1/7 were able to partially complete T1 and T2, respectively (partial completeness was judged as producing the required interface, but without the desired styling). All self-reported experts successfully completed both tasks. See Table 1 for more details.

Across both tasks, fewer than half of the model outputs were eventually accepted by participants (with our without edits), with most being edited or rejected (Figure 4). To help understand the types of inputs participants provided, and the code generated by the model, Appendices C.3 and C.4 show example requests of novices and experts, respectively, and the output produced by the model.

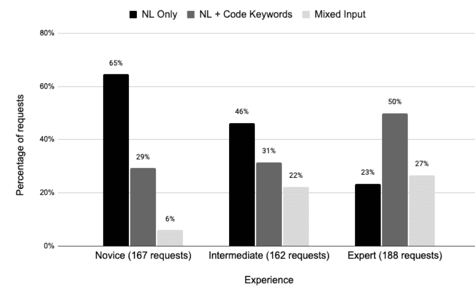

Figure 3: Request strategy according to participants' self-reported level of coding experience

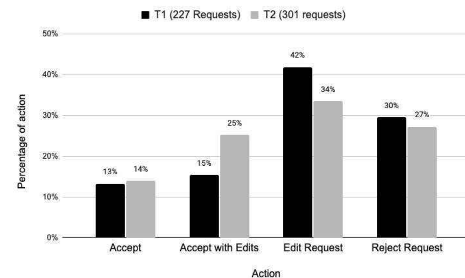

Figure 4: Distribution of actions taken in T1 and T2 (N=12)

4.2.1 Characterizing Request Content. The introductory video we presented participants at the start of the study demonstrated that GenLine could accept both natural language and a mixture of natural language and code. However, participants used natural language by itself as their primary strategy for synthesizing code, with some variability observed for different experience levels (see Figure 3). More specifically, we observed that novices tended to rely primarily on natural-language-only requests, whereas experts were more likely to mix natural language and code in their requests. Neither natural language alone nor a mixture of language and code[6] yielded higher acceptance rates (where acceptance rate refers to the likelihood of a participant clicking the tool's "Accept" button after invoking the model).

Among all natural language requests, there was a nearly even split between natural language requests and natural language requests that also included domain-specific keywords, such as "input", "div", and "padding". For some participants, the use of keywords was an explicit attempt to add more specificity to the request: "I noticed in the beginning I was very natural, like, 'make a square that's bigger than it is taller' and it [GenLine] was getting it wrong for me a lot. I pivoted to using CSS and element language, like, 'make a div' and I would get immediate results" (P10-E).

While participants' inputs sometimes included very precise language, there was also a desire for the model to be able to interpret more vague requests: "If I can give a very vague, not very specific instruction... 'Can you make it a little more warmer? Can you make

[4]Self-reported novice participants may have more prior (non-front-end) programming experience.
[5]To derive the novice, intermediate, and expert labels, we map participants' self-reported expertise as follows for HTML/CSS and JavaScript: "Not at all experienced" and "Somewhat experienced" map to the "novice" label, "Experienced" to the "intermediate" label, and "Very experienced" and "Extremely experienced" to the "expert" label.
[6]The total number of requests was 528 across tasks, 11 of which contained code only, and are not featured in the final experience strategy graph.

Table 1: Task Completion. Bolded entries indicate participants who fully completed both tasks.

Status	T1 - Participants	T2 - Participants
Fully completed	P13-N, P14-I, **P4-E**, **P6-E**, **P7-E**, **P10-E**, **P12-E**	P8-I, **P4-E**, **P6-E**, **P7-E**, **P10-E**, **P12-E**
Partially completed	P1-N, P3-N, P8-I, P9-I	P9-I
Did not complete	P5-N	P1-N, P5-N, P3-N, P13-N, P14-I

it a little more dense?' That comes more naturally, more easily in natural language..." (P11-N).

Participants used a limited number of verbs to synthesize code, including "add", "create", "generate", and "make". In the final interviews, participants expressed that adding constraints or having some design affordance to confirm recognition of a word would be more helpful compared to having unlimited syntax, consistent with other natural language invocation research [41]: "Even just having like a reference for the language in another tab that it can accept for different outcomes. Having a list it uses to do these basic things...so a glossary, really." (P5-N)

Some participants' requests could be considered "contextless," where the model would not need to refer to surrounding code or recent history to correctly interpret the participants' input (e.g., "generate a text input with the label 'Front:' and the value 'Hello'"). Other requests implicitly assumed that the model could access and utilize surrounding context (e.g., "Add a blue border", with no indication of *what* to add a blue border to). Notably, more than a third of the requests fell into this latter category. Participants' mental model, and desire, was that this surrounding context (including recent actions) would be used by the model, consistent with other past work [41]. When the model didn't take this context into account, participants were frustrated: "I found that I could ask GenLine for specific things, which is of course awesome, but I couldn't interact with things that I had asked for previously... So for example...let's say I ask GenLine to provide the Google logo. It wouldn't be a problem. But then I couldn't then say, 'Please center it', 'Please make it this percentage size and then place a box underneath it" (P3-N). In part, participants' frustrations are due to our interface design not effectively communicating what information is (or is not) sent to the model. However, the more important observation is that participants desired a style of interaction similar to having a conversation with a colleague, where previous context is taken into account with each new request. This style of interactively constructing code by referencing prior code could be useful in many contexts, such as specifying the layout of an interface (e.g., "place the Cancel button to the left of the OK button").

One concern raised was how inclusive tools like GenLine would be for multilingual users: "My first language is not English. I actually prefer to use the HTML language instead of typing the English because I'm worried about, 'What if I make a grammar mistake—will it still generate the code for me?'" (P4-E). Communicating the ability of the model to deal with grammar mistakes, as well as ensuring code synthesis tools work with diverse language input, will be useful as these tools continue to evolve.

4.2.2 *How Much Code Was Requested.*

We observed a wide range of strategies with respect to the level of abstraction of the code-generation request, as well as the amount of code effectively being requested. Some participants attempted to get as close as possible to their overall goal, all within their initial request, as in this example: "create a rectangle with the word "hello" in the middle and a blue button underneath the rectangle with the text "flip" on it. when I press the "flip" button, show the text "hola" in the middle instead of "hello"" (P1-N). Another participant similarly requested a lot of code to be generated in their first request, though with less detail: "Create flashcard webapp" (P3-N). These strategies could be considered "top-down" strategies, where participants attempt to create large units of code all at once.

However, the majority of participants had more reserved initial requests, with many requests being roughly equivalent to a line of code, as in these examples: "create an input field with label front" (P13-N) and "make a div with width 100% and height 100%" (P10-E). These requests can be considered more akin to "bottom-up" strategies, where the strategy is to create small units that are then assembled together. P12, an expert, described their strategy as follows: "I approached it like I was ... actually coding line by line... So I would just say ... okay, make this line for me, and then from there, I'll move on. But I think it would be nice to ... not have to take that step-by-step approach—to really ... take full advantage of the generated code" (P12-E).

While the granularity and specificity of requests varied, participants were excited when high-level requests were successfully interpreted by the model: "Another thing was just getting an image from the internet. All I had to say was 'insert the Google logo' here and it did the rest of finding the image and putting the URL into HTML" (P7-E).

4.2.3 *Strategies Observed in Each Task.*

In Task 1 (create the search page), participants only used natural language or natural language and keywords (from the programming language) for their requests. If participants were unable to generate the logo within their first few requests, they tried to create the word in the logo letter-by-letter. Once they were able to create the logo, either through an image or letter-by-letter, requests focused on creating a search bar, before moving onto formatting.

Task 2 presented more challenges given the need to specify interactivity using JavaScript. No novice was able to complete Task 2. Both intermediate and expert participants were able to fully complete Task 2, but expressed that it felt tedious to try to formulate requests in natural language: "Sometimes it took me a little bit longer to find the words to describe what I wanted it to do as opposed to me just doing it myself" (P12-E). For participants who

were able to complete Task 2, the majority of their time was spent editing output from GenLine or leveraging their prior knowledge to write code.

As with Task 1, the majority of requests for Task 2 were natural language only, but intermediate and expert participants began to include code in their requests. This strategy was often leveraged to add more specificity to styling requests as in this request to add a top margin to an existing element: "Add 50px top margin <div style='width: 300px; height 250px; background-color: white; border: 1px solid black;'></div>". Participants also used a mixture of natural language and code to specify JavaScript functionality: "bind the text in the <input type='text'> to <div id='front-word'></div>". However, we did not observe participants accepting model output at a higher rate using this strategy compared to requests consisting only of natural language.

Request complexity increased slightly for Task 2 with more experienced participants attempting to execute multiple discrete actions with their requests, as in this example: "give the value of front-input as a text content of card on page load" or this example: "when <button id='flipButton' style='background-color:blue; color:white;height:30px;margin-top :15px>Flip</button> is clicked, hide <div id='frontWord'></div> if it's visible".

4.2.4 *Coping with Model Failures / Repair Strategies.*

In our study, participants accepted generated results outright or with modifications less than 50% of the time. When participants did not receive the desired results, they employed the set of strategies listed in Table 2 to produce the desired results.

The most frequently employed strategy was rewording a request, which consisted of changing, adding, or dropping a word used in the initial request (e.g., going from "create a div and center all elements" to "add a div and center all elements"), or reordering the words from the request.

Participants also attempted to *expand* the scope of the request in an attempt to improve the output (e.g., going from "Add a button" to "Add a button and textbox"). This strategy of adding more information to the request was often observed as one of the final strategies employed to obtain the desired model output. (Participants also were observed adding information to an initial request that was successful. However this strategy is not intended to fix a request, but rather, to iteratively construct a larger request, bit by bit.)

Participants sometimes reduced the scope of requests, but requests were generally of a relatively small scope to begin with (75% of requests were coded to be of low complexity). However, novice users often asked for a significant amount of code in their first request, before scoping down the request.

Participants also changed parts of the input to target an outcome they felt would be easier for the model, such as common tutorial content (e.g., using "hello world" as a placeholder for a more specific phrase). This strategy can be thought of as an attempt to "reverse engineer" what a model was trained on, and craft a request to better match what they consider to be in the training data.

Interestingly, participants would also simply try re-running the same input, making no other changes. This is similar to "rolling the dice," since the same language model can sometimes produce different output on different model runs.

Finally, participants would sometimes re-run the original request using a different model temperature, in an attempt to get more variety in the responses (higher temperature), or less variety in the responses (lower temperature).

Overall, we did not observe one strategy clearly leading to a higher likelihood of participants accepting the generated code.

4.2.5 *Developing Mental Models of the AI and Its Syntax.*

As participants interacted with the model, they seemed to grapple with forming a mental model of what the model can "understand": "Mostly it just seemed like it didn't actually pick up on what I was trying to communicate to it or what I wanted it to do. So, there was a lot of trial and error, or I'll accept something that's close and then just tweak the result afterwards" (P9-I).

Participants also cited challenges in learning the "syntax" of the AI assistant, despite its input consisting of natural language: "It didn't feel like natural language, it was more of like—what's the right magic phrasing to get the model to do what I want it to do" (P6-E). P8 echoed this sentiment: "I feel it's kind of like learning a new language, except maybe it's an easier one to learn than JavaScript, but it still I think requires learning" (P8-I).

P10 provides a specific example of the challenges of precisely expressing intent through natural language alone: "Equals is a great example: I want a value to 'equal' another value, I want to replace the value of variable 1 with variable 2. That can also be interpreted by this model as 'does value 1 equal value 2 as a Boolean [...] operator. Trying to do that and realizing after the fact that 'Oh, it interpreted this as equal and not this as take on this value' makes you kind of feel like a) we're not speaking the same language here, b) if I don't know how to interpret something in a way you're [the model] going to understand it, I can't reliably count on any sort of improvement to my workflow" (P10-E).

In the final interviews for the study, participants cited the lack of feedback from the system as a primary reason for feeling that coding with natural language was unintuitive and unreliable (Figure 5): "In the natural language case, I'm always worried if the machine is going to understand my language or not" (P8-I). This led to a mixture of feelings toward the AI's capabilities, or as P5 put it, a "combination of fun and frustrating" (P5-N).

4.2.6 *Envisioned Use Cases.*

Participants saw potential utility in the tool for 1) API lookups, 2) minimizing the tedium of boilerplate code, 3) as a means for two or more teams to collaborate, and 4) as an educational tool. We expand on these envisioned use cases below.

Supporting the equivalent of API look-ups resonated with some participants: "It made things pretty easy, especially for adding event listeners and adding listeners that did what I wanted it to do. That worked really well and it was easy to implement. I didn't have to go and Google 'how to do the specific action' that I wanted when that event happens" (P7-E).

Others saw the capabilities useful for reducing otherwise tedious work: "Saying 'make sure tests from this package are run, whenever this other file is touched,' that's a thing saying out loud is a very clear sentence, but to do that in code is very tedious" (P2-I).

In the context of working with others, one participant indicated that it could be useful as a "universal translator" when two teams are collaborating: "One team is using JavaScript and one team is

Table 2: Repair Strategies

Repair Strategy	Example	Count	Participants
Reword (add, drop, change, or reorder words)	make width 250px <input type='text'> → change to width 250px <input type='text'>	131	all
Expand scope of request	write Google in bold blue → write Google in bold blue with a search box under	50	P1-N, P5-N, P3-N, P13-N, P8-I, P9-I, P14-I, P6-E, P10-E P12-E
Retries (reruns request)	change the size of the google logo to 80% smaller → change the size of the google logo to 80% smaller → change the size of the google logo to 200px (moves on to different strategy)	31	P5-N, P3-N, P13-N, P8-I, P9-I, P14-E, P6-E, P7-E, P10-E
Reduce scope of request	add search box to page with search button and a button that says 'i'm feeling lucky → add search box to page with search button	29	P5-N, P3-N, P13-N, P8-I, P9-I, P4-E, P6-E, P7-E
Adjust temperature	add a text input and a submit button → does not get desired result, increases temperature to 0.5 and reruns request	15	P5-N, P13-N, P7-E
Recalibrate specific targets with "easier" targets	blue Arial 25px text G → blue Arial 25px text hello world	13	P5-N, P8-I, P9-I, P14-I

using React. How could you make sure the teams can collaborate together, because both of them are definitely using natural language" (P4-E).

Finally, some suggested that these capabilities could be a useful tool to educate people on how to program: "I can see it being of great use to people who are not that proficient in coding to be ... a good sort of introduction, or to really ... lower the barrier to entry for coding" (P12-E).

In addition to these use cases, participants also considered *who* would benefit the most from the ability to code with natural language: "I think the best use case, it's me, which is like, I know code, but I haven't used code in a really long time. I know the mechanics. I know how it works, but I don't remember a lot of the syntax and ... then that's great...For ... a complete beginner, it would be a nightmare. They wouldn't be able to do it. And I think if you do that with someone that codes every day, they might be like, yeah, I'm going to be faster just doing it myself" (P13-I). Thus, an ideal use case for natural language coding might be for people who are somewhat familiar with coding, but are unfamiliar with the depths of a specific language, rusty on syntax, or rarely use a specific library or API.

In spite of the current challenges of learning to code through natural language, participants were optimistic about the possibilities of using natural language to reduce barriers to interacting with technology (see Figure 5). "Let's not waste time with these specific kinds of tools that are asking you to think in a certain way because they think it's the right way (and it might be), but I think just speaking about what you want is much better" (P3-N).

4.3 Limitations

Participants were instructed they could use GenLine as much or as little as they needed to complete the tasks, but the novelty of the tool and participation in the study may have led participants to use GenLine more than they might in regular practice. Five participants encountered a bug in the prototype where they were unable to view the preview window/code output for a portion of their usage, and some experienced a few instances of a bug where code was cut off when it was inserted into the editor window. However, we do not believe these bugs affect our overall results. Finally, given the early stage of this tool, we were not able to integrate GenLine in participants' daily work, and thus needed to test in an experimental context.

5 DISCUSSION

In theory, LLMs' ability to translate natural language to code would seem to provide a welcome capability to assist with software development, by enabling users to express goals more intuitively through natural language. However, the unexpected model responses led to study participants feeling like they needed to learn the "syntax" of

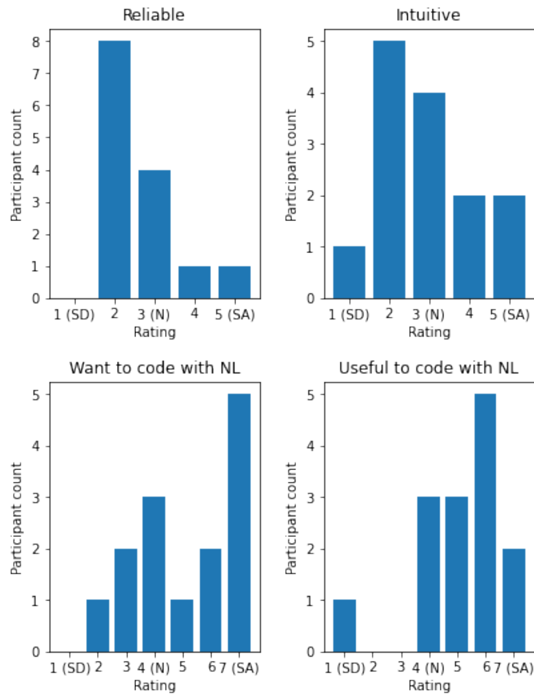

Figure 5: Graph of participant ratings on 1) whether writing code with natural language is reliable, 2) whether writing code with natural language is intuitive, 3) whether participants wanted to write code using natural language, and whether participants believe writing code using natural language is useful (where 1 is Strongly Disagree, 3 is Neutral, and 5 (or 7) is Strongly Agree).

the model—the specific words to say and the specific phrasings to produce the desired output. Similarly, given the expansive scope of natural language, participants did not know how much code they could reasonably ask for, with requests ranging from the production of an entire app ("make a flashcard app") to a single line of code. Thus, while these models accept any text as input (conceptually offering an "unbounded" syntax), in reality, there is a *latent "syntax" and latent problem space* in which the model can reliably perform, and users needed to discover both.

While participants encountered challenges in achieving their desired results with the model, the inherent versatility and flexibility of LLMs suggests that the model itself may be able to help end-users cope with some of the challenges they experienced. For example, users sometimes modified a request by simplifying it (e.g., removing specific objectives), a fallback strategy that an LLM may be able to perform itself. More generally, the ability to rapidly customize this new generation of models using prompt programming opens up the possibility for end-users to create highly targeted prompts to support specific tasks, including prompts that help the user recover when the model does not produce the desired output. Our work thus brings to light the dual challenges and opportunities of large language models: Although the unbounded syntax of natural language can be difficult for users to grapple with, there is also the possibility of leveraging the inherent flexibility of LLMs to address some of these challenges.

In this section, we consider the key user challenges arising from the expansive nature of natural language programming, and suggest implications for future work. For several of these design implications, we also describe potential LLM-based remedies to aid future researchers and designers (example prompts can be viewed in the Appendix). We conclude this section by considering how our results may generalize beyond the specific tool we built and tested.

5.1 Providing Suggestions for an Unbounded Syntax

As we observed, participants often had difficulty determining what they could ask of the model. To address this issue, it could be useful for systems to provide suggestions of the types of natural language that could be used in a particular context. For example, the prompts we seeded the system with (e.g., to generate HTML, or to generate JavaScript) provide a sense of what types of tasks the model is likely to be able to handle. However, future systems could go beyond this one strategy. For example, as part of its documentation or onboarding materials [17], a system could provide a variety of example requests that are likely to work well, for each prompt presented (e.g., for the "Generate HTML" prompt, the system could surface examples such as "make a blue button" or "make a text field that alerts 'hello' when clicked"). These examples could help give users a sense of the *level of abstraction and granularity* they should target in their requests. A system could also surface suggestions of common successful requests relevant to the existing code on the page. For example, given the code for an HTML element (e.g., "<button>OK</button>"), the system could present: "To change the color of the button, try: 'Make this blue <button>OK</button>'". These types of suggestions may additionally aid novices who are new to the problem domain, and thus may not know how to solve a problem or how to express their goal in a way likely to produce a useful result (i.e., addressing the vocabulary problem [23]). Conversely, in onboarding materials, it may be helpful to show different categories of requests that users may *expect* to work well, but that surprisingly *don't*, to help calibrate expectations.

To test this idea of offering suggestions, we created a "Suggestion" prompt that suggests sub-tasks for the user's request (see Appendix B.1). As can be seen from the examples provided in the Appendix, its suggestions could prove to be helpful for people who are unfamiliar with the problem space.

5.2 Automated Input Variation and "Fallback Prompts"

When the model did not produce expected results, participants often rephrased the request, added information, reduced information, and/or changed the temperature. Notably, many of these strategies can be automated.

For example, the system could invoke a set of *fallback prompts* that transform requests into simpler requests. For example, a prompt may transform the user's input into smaller sub-tasks, as in the Suggestion prompt above (see Appendix B.1).

A prompt could also transform the user's input into a *simpler request* with less information, mimicking the strategy of participants

who would sometimes *take out* information from their request. A prompt that demonstrates this concept, along with example output from the model, is shown in Appendix B.2. The sample results produced suggest that this technique could be useful in producing some basic code that can then be built upon.

Finally, another potential fallback strategy would be to *rephrase the user's input* or to produce multiple interpretations of the input. The prompt in Appendix B.3 demonstrates these concepts and shows sample output from the model.

While the fallback prompts developed here show promise, a larger research need is to determine which strategies, including user strategies and the fallback prompts described here, reliably lead to improved results. In our study, we did not observe one strategy appearing to be more effective than another (and we did not offer fallback prompts to participants during the study). Future research would benefit from empirical data establishing which strategies and fallback prompts can effectively improve model results for end-users. These data can then be used to determine which strategies to automate on behalf of the user. For example, if a fallback prompt that simplifies the request is shown to reliably produce useful output, it could automatically be run, with its results included in the set of results returned to the user.

In addition to this empirical data, we also expect that users will naturally invoke all of the strategies we observed in our study. Effectively communicating which strategies are likely to lead to better results, and which aren't, can help users optimize their time and avoid pursuing approaches that are not likely to work.

5.3 Improving Request Quality Through Conversation

As we found in the study, user requests can range from extremely large, under-specified requests (e.g., "create a flashcard app") to highly targeted, tightly scoped requests (e.g., "make a div with width 100% and height 100%"). For large, ambiguous requests, it can be challenging to debug the model when it doesn't produce useful output.

In these circumstances, a more structured, conversational interaction with the model may be useful. For example, given the request "create a flashcard app," the system could initially respond, "Describe what is in the flashcard app." After describing the interface, the model could ask, "Describe the behavior of each part of the interface." Since participants naturally tried to interact with the tool in a conversational style, this type of interaction in which the model helps the user derive specifications may dovetail nicely with existing user expectations. Deriving specifications in this manner may also help novices break down a problem into more reasonably-sized chunks.

The Suggestion and Simplify prompts (Appendix B.1, B.2) explore these ideas. The example outputs from these prompts suggest that the model could be used to help the user re-scope requests into smaller sub-tasks.

5.4 Debugging Tools and AI Onboarding

Given the model's variability in translating natural language requests to code, it may reside in somewhat of an "uncanny valley" for users since it can sometimes correctly translate natural language requests into code (much like a human counterpart could), and other times not. Mechanisms that help to illuminate the model's understanding of its input may enable users to climb out of this uncanny valley and form a more accurate mental model of how to optimally interact with the model.

To help build a more robust understanding of the model and its capabilities, model attribution or interpretability techniques like those found in the Language Interpretability Tool [38] could be useful. For example, showing how each input influenced each output could provide insight into the model's behavior. Providing some transparency into model behavior may also help users better reason about the model and why it produces the output that it does.

AI onboarding [17], which describes a model's capabilities viz-a-viz a typical person, could also be useful. In particular, describing scenarios where it is known to work well, and situations in which it may produce unexpected behavior, are likely to be welcome to end-users.

5.5 Looking Beyond GenLine

Research has shown that large generative language models can translate natural language requests into code (e.g., [4, 7, 11, 18]), continue code (e.g., Copilot [1]), produce code through a conversation [11, 18], and (as this research shows) modify code through a mixture of natural language and existing code (e.g., "Make this button blue: <button>OK</button>"). These examples highlight the flexibility of this new generation of models, and their ability to support a *wide range of qualitatively different modes of interaction*—autocomplete-like functionality, conversational styles of code construction, command-like tools (as with GenLine)—all by using the same underlying model, customized through prompt programming.

While it is tempting to ask which of these interaction styles is the "best," each design has its own unique set of affordances. For example, the user interface afforded by GenLine allows users to invoke very targeted operations (such as changing the styling of an existing HTML element) more easily, by selecting the markup of the HTML element to apply the GenLine prompt to. Additionally, the ability to author new GenLine prompts enables users to extend the tool to support specific use cases and needs. However, despite differences across tool designs, what these tools all share in common is that they all offer the potential to further bridge the Gulf of Execution [35] (e.g., by translating natural language requests to code) and to streamline existing practices (e.g., by performing the equivalent of looking up an API call and inserting it into code).

This research has explicitly examined the user experience of working with a large generative language model to assist in producing front-end web code (HTML, JS, CSS). While some of our study results clearly are 1) a function of GenLine's design (e.g., participants expected that the tool would take into account surrounding code or recent requests) or 2) a function of the capabilities of current models (e.g., the variability in code correctness, as also seen in empirical studies [11, 18]), other findings and their implications generalize across tool designs, and are thus more broadly applicable to the larger HCI research community. For example, the challenges in "debugging" interactions with the model, or the uncertainty participants faced in how to best phrase requests, will almost certainly

apply to any tool that makes use of these new models. Furthermore, these issues will likely remain relevant, even as these models continue to improve in their ability to synthesize code (e.g., there will always likely be model failures that users wish to debug). Developing tools that help people leverage these new models' unique capabilities (e.g., the ability to rapidly customize them through prompt programming), while offering reliable means to cope with common challenges, thus represents a rich problem space requiring active HCI research. Our study results, design implications, and examples showing how LLMs may be able to help address these issues, collectively provide a foundation for this future research.

6 CONCLUSION

This paper introduces GenLine, a tool for accessing and applying large generative language model prompts within a code editor. Our user study examines how people interact with a natural language code synthesis tool backed by a 137 billion parameter LLM, and highlights challenges and opportunities this kind of tool introduces. In particular, participants felt they needed to learn the "syntax" of the model (despite its input being natural language), as well as which specific tasks could be reliably performed (and how).

When model output was not what was desired, participants employed a range of strategies to coax the model to produce the desired output. These strategies included reducing the scope of the request, increasing the scope of the request, rewording or rephrasing the request, and even introducing keywords such as "test" or "hello world".

The challenges participants encountered suggest a number of implications for design, as well as future research. In particular, an important, open research need is to develop (and demonstrate) reliable "error recovery" techniques for natural language code synthesis. We propose a number of possibilities that make use of the LLM itself, such as the use of "fallback prompts." The black box nature of the model also suggests that interpretability tools may be useful in helping users understand and debug the model.

ACKNOWLEDGMENTS

We thank all our participants for their thoughtful feedback and Dr. Julie Anne Séguin for data analysis support.

REFERENCES

[1] [n.d.]. GitHub Copilot. https://copilot.github.com/. Accessed: 2021-09-02.
[2] [n.d.]. GPT-3 Creative Fiction. https://www.gwern.net/GPT-3. Accessed: 2021-03-30.
[3] [n.d.]. OpenAI API: Code Completion. https://beta.openai.com/?app=productivity&example=4_4_0. Accessed: 2021-03-30.
[4] [n.d.]. OpenAI API: Natural Language Shell. https://beta.openai.com/?app=productivity&example=4_2_0. Accessed: 2021-03-30.
[5] [n.d.]. OpenAI Prompt Library. https://openai.com/blog/gpt-3-apps/. Accessed: 2021-03-30.
[6] [n.d.]. Tweet: 'First work with #GPT3 , I asked it to draw an image. I gave it seed SVG code and asked it to generate an SVG code by itself. Turns out it drew something resembling a Floppy Disk.'. https://twitter.com/fabinrasheed/status/1284052438392004608. Accessed: 2021-03-30.
[7] [n.d.]. Tweet: 'I only had to write 2 samples to give GPT-3 context for what I wanted it to do. It then properly formatted all of the other samples. There were a few exceptions, like the JSX code for tables being larger than the 512 token limit.'. https://twitter.com/sharifshameem/status/1282692481608331265. Accessed: 2021-04-07.
[8] [n.d.]. Tweet: 'Meet Marz. Like @ProjectJupyter, but closer to Earth. No-code data notebook to go from 'natural language' question to SQL to insight, powered by @OpenAI's GPT3. Built with @barrnanas @idavidgoldberg @imfanjin as part of @beondeck's Build Weekend!'. https://twitter.com/albertgozzi/status/1320526310729539584. Accessed: 2021-03-30.
[9] Daniel Adiwardana, Minh-Thang Luong, David R. So, Jamie Hall, Noah Fiedel, Romal Thoppilan, Zi Yang, Apoorv Kulshreshtha, Gaurav Nemade, Yifeng Lu, and Quoc V. Le. 2020. Towards a Human-like Open-Domain Chatbot. arXiv:2001.09977 [cs.CL] Accessed: 2021-08-12.
[10] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *ACM Comput. Surv.* 51, 4, Article 81 (July 2018), 37 pages. https://doi.org/10.1145/3212695
[11] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. arXiv:2108.07732 [cs.PL]
[12] M. Beth Kery and B. A. Myers. 2017. Exploring exploratory programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 25–29. https://doi.org/10.1109/VLHCC.2017.8103446
[13] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, Shyamal Buch, Dallas Card, Rodrigo Castellon, Niladri Chatterji, Annie S. Chen, Kathleen Creel, Jared Quincy Davis, Dorottya Demszky, Chris Donahue, Moussa Doumbouya, Esin Durmus, Stefano Ermon, John Etchemendy, Kawin Ethayarajh, Li Fei-Fei, Chelsea Finn, Trevor Gale, Lauren Gillespie, Karan Goel, Noah D. Goodman, Shelby Grossman, Neel Guha, Tatsunori Hashimoto, Peter Henderson, John Hewitt, Daniel E. Ho, Jenny Hong, Kyle Hsu, Jing Huang, Thomas Icard, Saahil Jain, Dan Jurafsky, Pratyusha Kalluri, Siddharth Karamcheti, Geoff Keeling, Fereshte Khani, Omar Khattab, Pang Wei Koh, Mark S. Krass, Ranjay Krishna, Rohith Kuditipudi, and et al. 2021. On the Opportunities and Risks of Foundation Models. *CoRR* abs/2108.07258 (2021). arXiv:2108.07258 https://arxiv.org/abs/2108.07258
[14] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. 2010. *Example-Centric Programming: Integrating Web Search into the Development Environment.* Association for Computing Machinery, New York, NY, USA, 513–522. https://doi.org/10.1145/1753326.1753402
[15] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901. https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf
[16] Daniel Buschek, Lukas Mecke, Florian Lehmann, and Hai Dang. 2021. Nine Potential Pitfalls when Designing Human-AI Co-Creative Systems. *arXiv preprint arXiv:2104.00358* (2021).
[17] Carrie J. Cai, Samantha Winter, David Steiner, Lauren Wilcox, and Michael Terry. 2019. "Hello AI": Uncovering the Onboarding Needs of Medical Practitioners for Human-AI Collaborative Decision-Making. *Proc. ACM Hum.-Comput. Interact.* 3, CSCW, Article 104 (Nov. 2019), 24 pages. https://doi.org/10.1145/3359206
[18] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]
[19] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2020. Multi-Modal Synthesis of Regular Expressions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 487–502. https://doi.org/10.1145/3385412.3385988
[20] Eli Collins and Zoubin Ghahramani. 2021. LaMDA: our breakthrough conversation technology. https://blog.google/technology/ai/lamda/ Accessed: 2021-07-14.
[21] Prem Devanbu, Matthew Dwyer, Sebastian Elbaum, Michael Lowry, Kevin Moran, Denys Poshyvanyk, Baishakhi Ray, Rishabh Singh, and Xiangyu Zhang. 2020. Deep Learning & Software Engineering: State of Research and Future Directions. arXiv:2009.08525 [cs.SE]
[22] Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2020. *Small-Step Live Programming by Example.* Association for Computing Machinery, New York, NY, USA, 614–626. https://doi.org/10.1145/3379337.3415869

[23] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. 1987. The Vocabulary Problem in Human-System Communication. *Commun. ACM* 30, 11 (Nov. 1987), 964–971. https://doi.org/10.1145/32206.32212

[24] Barney G. Glaser and Anselm L. Strauss. 1967. *The Discovery of Grounded Theory: Strategies for Qualitative Research.* Aldine de Gruyter, New York, NY.

[25] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 317–330. https://doi.org/10.1145/1926385.1926423

[26] Sumit Gulwani and Mark Marron. 2014. NLyze: Interactive Programming by Natural Language for Spreadsheet Data Analysis and Manipulation. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 803–814. https://doi.org/10.1145/2588555.2612177

[27] Marti A. Hearst. 2009. *Search User Interfaces* (1st ed.). Cambridge University Press, USA.

[28] Cheng-Zhi Anna Huang, Ashish Vaswani, Jakob Uszkoreit, Ian Simon, Curtis Hawthorne, Noam Shazeer, Andrew M. Dai, Matthew D. Hoffman, Monica Dinculescu, and Douglas Eck. 2019. Music Transformer. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rJe4ShAcF7

[29] Sandeep Kaur Kuttal, Bali Ong, Kate Kwasny, and Peter Robe. 2021. Trade-Offs for Substituting a Human with an Agent in a Pair Programming Context: The Good, the Bad, and the Ugly. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 243, 20 pages. https://doi.org/10.1145/3411764.3445659

[30] Toby Jia-Jun Li, Jingya Chen, Haijun Xia, Tom M. Mitchell, and Brad A. Myers. 2020. Multi-Modal Repairs of Conversational Breakdowns in Task-Oriented Dialogs. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '20)*. Association for Computing Machinery, New York, NY, USA, 1094–1107. https://doi.org/10.1145/3379337.3415820

[31] Xi Victoria Lin. 2017. Program Synthesis from Natural Language Using Recurrent Neural Networks. http://victorialin.net/pubs/tellina_tr_2017.pdf Accessed: 2021-04-06.

[32] Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D. Ernst. 2018. NL2Bash: A Corpus and Semantic Parser for Natural Language Interface to the Linux Operating System. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*. European Language Resources Association (ELRA), Miyazaki, Japan. https://www.aclweb.org/anthology/L18-1491

[33] Ryan Louie, Andy Coenen, Cheng Zhi Huang, Michael Terry, and Carrie J. Cai. 2020. Novice-AI Music Co-Creation via AI-Steering Tools for Deep Generative Models. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3313831.3376739

[34] A. Narechania, A. Srinivasan, and J. Stasko. 2021. NL4DV: A Toolkit for Generating Analytic Specifications for Data Visualization from Natural Language Queries. *IEEE Transactions on Visualization and Computer Graphics* 27, 2 (2021), 369–379. https://doi.org/10.1109/TVCG.2020.3030378

[35] Donald A. Norman. 2002. *The Design of Everyday Things.* Basic Books, Inc., USA.

[36] Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. 2003. Towards a Theory of Natural Language Interfaces to Databases. In *Proceedings of the 8th International Conference on Intelligent User Interfaces* (Miami, Florida, USA) *(IUI '03)*. Association for Computing Machinery, New York, NY, USA, 149–157. https://doi.org/10.1145/604045.604070

[37] Kia Rahmani, Mohammad Raza, Sumit Gulwani, Vu Le, Dan Morris, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. 2021. Multi-modal Program Inference: a Marriage of Pre-trained Language Models and Component-based Synthesis. In *OOPSLA*. https://www.microsoft.com/en-us/research/publication/multi-modal-program-inference-a-marriage-of-pre-trained-language-models-and-component-based-synthesis/

[38] Ian Tenney, James Wexler, Jasmijn Bastings, Tolga Bolukbasi, Andy Coenen, Sebastian Gehrmann, Ellen Jiang, Mahima Pushkarna, Carey Radebaugh, Emily Reif, and Ann Yuan. 2020. The Language Interpretability Tool: Extensible, Interactive Visualizations and Analysis for NLP Models. arXiv:2008.05122 [cs.CL]

[39] Gust Verbruggen, Vu Le, and Sumit Gulwani. 2021. Semantic programming by example with pre-trained models. In *OOPSLA*. ACM. https://www.microsoft.com/en-us/research/publication/semantic-programming-by-example-with-pre-trained-models/

[40] Justin D. Weisz, Michael Muller, Stephanie Houde, John Richards, Steven I. Ross, Fernando Martinez, Mayank Agarwal, and Kartik Talamadupula. 2021. Perfection Not Required? Human-AI Partnerships in Code Translation. *26th International Conference on Intelligent User Interfaces* (Apr 2021). https://doi.org/10.1145/3397481.3450656

[41] Frank F. Xu, Bogdan Vasilescu, and Graham Neubig. 2021. In-IDE Code Generation from Natural Language: Promise and Challenges. arXiv:2101.11149 [cs.SE]

[42] Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L. Glassman. 2020. *Interactive Program Synthesis by Augmented Examples.* Association for Computing Machinery, New York, NY, USA, 627–648. https://doi.org/10.1145/3379337.3415900

A PROMPT PROGRAM CODE LISTINGS

Note: The following examples are formatted to be easier to read; when input to the model, newlines are removed. Also note that prompts include a "[code]" string. While the model was not explicitly trained to perform translation from natural language to code, the training data did prepend a "[code]" string to training data from a source that included source code.

A.1 Prompt: Generate HTML code (general)

Description: Generates HTML code given a natural language description. Note: This prompt derives from a demo posted on Twitter [7].

Prompt template:

```
[code] description: make a red button that says stop
html: <button style = 'color: white; background-color: red'>Stop</button>

description: a blue box that contains 3 yellow circles with red borders
html: <div style = 'background-color: blue; width: 150px; height: 150px;'>
  <div style = 'background-color: yellow; border-radius: 50\%; border: 5px solid red;
  width: 30px; height: 30px;'></div>

description: create two buttons that are centered in a div
html: <div style='text-align:center;'> <button>1</button> <button>2</button> </div>

description: give code for an html button with a margin
html: <button style='margin:10px;'></button>

description: make a pink button that says type and a textarea
html: <button style='background-color: pink;'>Type</button> <textarea></textarea>

description: <<user input>>
html:
```

A.2 Prompt: Generate JavaScript code (general)

Description: Generates JavaScript code (general). Note the lack of space between the keys ("description" and "javascript"). This was a typo, was not an issue for the model.

Prompt template:

```
[code] descriptionadd the variable el to the div with id 'hi'
javascriptdocument.getElementById('hi').appendChild(el);
descriptionmake a button with text 'hi'
javascriptvar button = document.createElement('button'); button.innerHTML = 'hi';

description <<user input>>
javascript
```

A.3 Prompt: HTML (zero shot)

Description: Generates HTML and JavaScript from a zero-shot prompt.

Prompt template:

```
description of web code:  <<user input>> generated html and inline javascript: [code] <!DOCTYPE html>
```

A.4 Prompt: Refactor code

Description: Refactors inline CSS styles.

Prompt template:

```
description: <div style='background-color: green; width: 100px; height: 100px;'></div>
refactored:
  html: <div class='green-square'></div>
  css: .green-square { background-color: green; width: 100px; height: 100px; }

description: <<user input>>
refactored:
```

```
html:
```

A.5 Prompt: Fix Code

Description: Fixes small errors in existing code.

Prompt template:

```
[code] original: <button>hi
fixed: <button>hi</button>

original: <div style = 'background-color: green; width: 150px; height: 150;
fixed: <div style='background-color: green; width: 150px; height: 150px;'></div>

original: <ol> <li>Coffee</li> <li>Tea</li> <li>Milk</li
fixed: <ol><li>Coffee</li><li>Tea</li><li>Milk</li></ol>

original:  <<user input>>
fixed:
```

A.6 Prompt: Generate HTML and JavaScript

Description: Generates HTML and JavaScript code together. Note the typo ("with and") was not an issue for the model.

Prompt template:

```
[code] input: make a button in html with and add a clickhandler in javascript to alert hello world
output: html: <button id='hi'>hello</button>
javascript: document.getElementById('hi').onclick = (e) => {alert('hello world');}

input: <<user input>>
output:
```

Prompt template:

```
Here's a stack overflow response on how to <<task you want>>: <<copy and paste stack overflow response>>

Use this previous example to <<task you want>> in <<your existing code>>, and return the <<type of code>> code:
```

B FALLBACK PROMPTS

This section presents a set of *fallback prompts* intended to address commonly occurring issues when synthesizing code using a large generative language model. These prompts' strategies include suggesting sub-tasks to consider solving instead of the original request, breaking down the task into simpler tasks, and helping the user clarify vague input. We provide example model output for the prompts using inputs collected from the user study. The sample outputs provide an indication of the potential value of these fallback prompts, but research in this space would benefit from further study on the overall effectiveness of these strategies. (Note that these fallback prompts were not available to participants during the study—these prompts were developed after the study to explore how the model could be used to improve outcomes.)

B.1 Prompt: Suggest sub-tasks

Description: Attempts to suggest sub-tasks for a main task.

Prompt template:

```
Task: making todo app
Suggestion: 1) an input for todo items
2) todo items using an unordered list
3) checkboxes for the items
Task: build a mortgage calculator
Suggestion: 1) an input for the interest rate, years and amout
2) a div to display the monthly payment
3) write mortgage calculator function with tests
Task: <<user input>>
Suggestion:
```

Example 1 For the input, 'a flashcard app', this prompt generated the following suggestions:

```
1) an input for the question and answer
2) a div to display the question
3) a div to display the answer
4) a button to flip the card
5) a button to shuffle the cards
6) a button to show the answers
```

Example 2 For the input, 'a search page', this prompt generated the following suggestions:

```
1) an input for the search term
2) a div to display the search results
3) write a search function with tests
```

These model-generated suggestions could serve as a jumping off point for new users.

B.2 Prompt: Simplify request

Description: Attempts to transform a user's input into a simpler task.

Prompt template:

```
Instruction: create a div with a search bar centered
Broken down instructions: create a div with a search bar
create a div
Instruction: create a search box and a google logo above it
Broken down instructions: create a search box
create a google logo
Instruction: create a search box and make it blue
Broken down instructions: create a search box
make the search box blue
Instruction: large field with id 'dunno' and a black border, with vertically centered text
    'what the card says'
Broken down instructions: create a large field with id 'dunno'
add a black border to field with id 'dunno'
create vertically centered text that says 'what the card says'
Instruction: <<user instruction>>
Broken down instructions:
```

Example 1: The user input "create a flashcard app" yields the following outputs from this prompt:

```
create a div with text 'flashcard'
create a div with a button that says 'create flashcard'
create a div with a button that says 'delete flashcard'
create a div with a button that says 'display flashcard'
```

If these individual lines are passed onto an HTML prompt, they yield the following code:

- <div>flashcard</div>
- <div><button>Create Flashcard</button></div>
- <div><button>Delete Flashcard</button></div>
- <div><button>Display Flashcard</button></div>

Example 2: The user input "generate a text input with a blue border and center align it" yields:

```
create a text input
add a blue border to text input
center align text input
```

Passing these outputs into an HTML prompt yields:

- <input type='text'>
- <input style='border: 2px solid blue;'>
- <div style='text-align:center;'> <input type='text'></div>

Example 3: The user input "create a rectangular text input centered under the google logo with a blue border 1px wide" yields:

```
create a rectangular text input
center the text input under the google logo
add a blue border to the text input
```

These outputs passed to an HTML prompt yield:

- <input type="text" style='border: 3px solid blue; border-radius: 10px;'>
- <div style='text-align:center;'><input type='text'> </div>
- <input type='text' style='border: 3px solid blue;'>

B.3 Prompt: Clarify Input

Description: Attempts to clarify ambiguous user input by inserting a placeholder or providing multiple options that clarify points of ambiguity.

Prompt template:

```
Instruction: create a div with a search bar centered
Broken down instructions: create a div with a search bar centered within the div with id [outer div's id]
Instruction: create a search box and a google logo above it
Broken down instructions: create a search box and a google logo in rainbow text above it
create a search box and a google logo with an image tag above it
Instruction: <<user input>>
Broken down instructions:
```

Example 1
For the input “Show containers one at a time”, this prompt produced the following output:

```
When clicking on the [container's id] div, hide all other divs except for this div
```

Example 2 For the input “text box centered blue outline”, this prompt modified the request to be:

```
create a text box with a blue outline and centered within the div with id [outer div's id]
```

Example 3 For the input, “make this centered at the page <input type='text' id='input' style='height: 200px; width 200px;'>”, the prompt returned:

```
create a search box with a height of 200px and a width of 200px and center it at the page
```

C USER STUDY DETAILS

C.1 Onboarding for Study

To help study participants learn how to use the tool, we showed them a narrated video demonstrating GenLine in action, then asked them to use GenLine to render ‘Lorem ipsum dolor sit amet’ in purple, italic font. We also provided the following tutorial text for using the tool:

To get started with GenLine:

- *Use [[double brackets]] to type your goal*
- *Select your language*
- *(Optional) Increase the temperature slider to get different results*
- *Evaluate the results*
- *View additional results as needed*
- *Ctrl + h will ‘auto format selection’*
- *Use GenLine to generate code via natural language and evaluate the outcome.*
- *We're looking for people to challenge GenLine with different use cases to better understand where it can be most useful.*
- *GenLine is not a fully functional code editor (e.g. Sublime) - it is currently in a prototype stage to learn more about the utility of natural language programming.*

C.2 Tasks in User Study

Figure 6: Task 1 in User Study

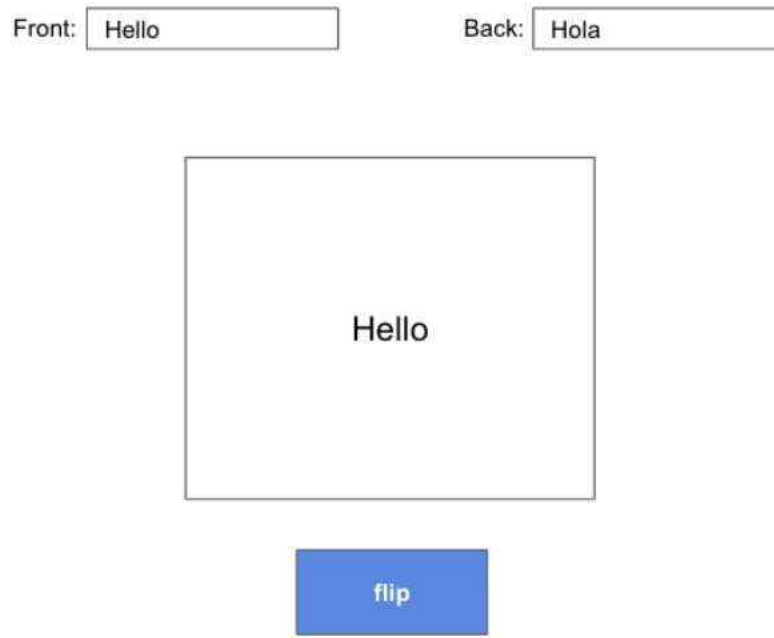

Figure 7: Task 2 in User Study

C.3 User Study Examples (Novice)

User Input	Generated Output (using HTML prompt)
create a flashcard app	<div style=’text-align:center;’> <button></button> <button></button>
create an input text box named "front:" and another input text box named "back:"	<input type="text" name="front"> <input type="text" name="back">
create a text input box with "hello" inside and create another text input box that will show "hola" when I press enter for "hello"	<input type=’text’ value=’hello’> <input type=’text’ onkeydown=’if(event.keyCode == 13)this.value="hola";’>
create a rectangle with the word "hello" in the middle and a blue button underneath the rectangle with the text "flip" on it. when I press the "flip" button, show the text "hola" in the middle instead of "hello"	<div style=’width: 100px; height: 100px; background-color: red;’> <div style=’width: 50px; height: 50px
create a white rectangle with the word "hello" in the middle	no result
create a white rectangle with the word "hello" in the middle	no result
create a white rectangle with the text "hello" in the middle	<div style=’width: 100px; height: 100px; background-color: white;’>. <div style=’text-alight:center;’>
create a rectangle with the text "hello" in the middle	<div style=’width: 100px; height: 100px; background-color: black;color:white;font-size:30px;’>Hello</div> description
when this button is clicked, make the rectangle say the text "hola" instead of "hello"	<button onclick="document.getElementByID(’rect’) .innerHTML=’hola’">Click Me</button><button onclick="document.getElementById
onclick=change the text "hello" to "hola"	‘<button onclick=’document.getElementById("hello") .innerHTML="hola"’></button>"

C.4 User Study Examples (Expert)

User Input	Generated Output (using HTML prompt)
html: generate a text input with the label ‘Front:’ and the value ‘Hello’	<input type=’text name=‘front’ value=‘Hello’> <label for=’front’>Front:</label> description
html: generate a text input with the label with the text ‘Front:’ and the value ‘Hello’	<input type=’text’ name=‘front’ value=‘Hello’> <label style=‘color: white; background-color: pink’>Front:</label>
html: generate at text input with the label that reads ‘Front:’	<input type=‘text’ name=‘front’>
html: make a box	[no result]
html: draw a box that is 200px	<div style=‘width: 200px; height: 200px; background-color: blue;’></div>
html: center align content and add 60px padding to the top	html, body {width:100%; height: 100%; margin-top: 60px;}
html: make a white box that is 200px and has a solid black border	<div style=‘width: 200px; height: 200px; border: 5px solid black; background-color: white;’></div>
html: make a white box that is 200px and has a solid black border of 1px	<div style=‘width: 200px; height: 200px; border: 1px solid black; background-color: white;’></div>
js: add a click handler to the button	button.onclick = function() {alert(‘hi’); }
js: create a varable with my button	var myButton = document.createElement(‘button’);
js: on a click event add the value from inputFront to the box	var el = document.getElementById(‘hi’); el.appendChild(document.createTextNode(document.getElementById(‘inputFront’).value));
js: set isEnglish to it’s opposite value	var isEnglish = !isEnglish;
js: create a function that toggles the text of the box	function toggleText(divId, newText){var div = document.getElementById(divId); div.innerHTML = newText;}