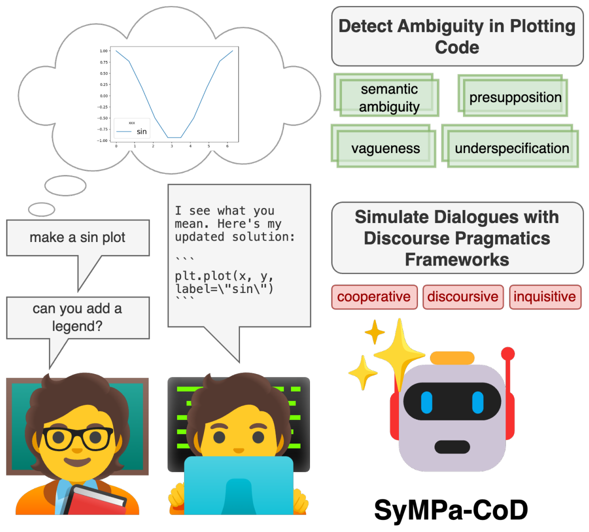
Identifying and Resolving Ambiguous Intents in Coding Instructions using Discourse Frameworks

Anonymous ACL submission

Abstract

Large Language Models (LLMs) are used by many to generate code in a pair-programming-like setting. However, intents in users' coding instructions are ambiguous, and models are limited in their ability to use dialogue to disambiguate intent to produce unambiguous code. This presents a fundamental difficulty in code generation, wherein the ambiguity in natural language can lead to seemingly correct programs that are different from the intended. We propose to use dialogue to reduce this ambiguity, specifically in the plotting domain, and contribute an analysis of the different types of ambiguity that may exist in multi-modal code generation. Based on our analysis, we propose different pragmatic models to inform dialogue strategies for ambiguity resolution, including those based on Rational Speech Acts (cooperative), Discourse Theory (discoursive), and Questions under Discussion (inquisitive). Finally, we compare these dialogue strategies in a simulated dialogue setting — operationalizing the pragmatic models via prompting. Our findings suggest that discoursive and cooperative reasoning styles show the best results regarding executability and disambiguation, while inquisitive reasoning performs the best in disambiguation for vagueness. These suggest that simulated dialogues with pragmatic frameworks can resolve ambiguities in code generation.



make a sin plot

can you add a legend?

```
I see what you
mean. Here's my
updated solution:
```
plt.plot(x, y,
label=\"sin\")
```
```

Figure 1: This figure summarizes the contributions of this paper. We analyze ambiguity in categories that exist in the plotting domain of coding instructions. Then, we address these ambiguities using simulated dialogues with discourse pragmatics. We make this simulated dataset of coding dialogues publicly available.

1 Introduction

While we have made great strides in code generation, current generative models still fail to provide a human-like pair-programming experience for their users (Sarkar et al., 2022). A user's description of their intent (in natural language) is often misconstrued by the model, forcing the user to switch back and forth between the typical pair-programming roles in human-human interactions, e.g., the coder and the director (Williams, 2001). This, we hypothesize, is the result of natural language coding instructions being inherently ambiguous – mapping to multiple possible code implementations. Indeed, we are not the first to make this observation (Vaithilingam et al., 2022). Code generation models need a way to resolve this ambiguity while also respecting the pragmatic rules expected by their (human) users. We define pragmatics similarly to Fried et al. (2023), where pragmatic communication is one that uses environmental context and inferences about other agents' unspoken goals.

To achieve this goal, we frame the natural language to code problem as a two-player cooperative dialogue. A **director** (typically the user) specifies their intent in natural language and a **coder** (typically an automated coding assistant) generates code with the functionality the director had in mind. A goal for a pair-programming setting is to have a coder agent that can interact with the director agent to resolve ambiguity and generate code.

To study this problem, we simulate this dialogue between two machine agents, drawing inspiration from theories of pragmatics to guide the dialogue acts these agents take. We show how (pragmatic) dialogues can help coders resolve ambiguity. We focus on problems where code is used to generate a visual output – `matplotlib` code generation – so we can simulate the dialogues where the intent can be specified in multiple ways (as the intended code solution, or the intended plot image).

We propose to identify ambiguous natural language intents by sampling outputs from a language-to-code language model. Using our method, we identify ambiguity in 57% of Matplotlib problems in the DS-1000 dataset (Lai et al., 2022). We propose resolving ambiguity through discourse, where the code generation model uses multiple turns of dialogue to clarify intent and resolve ambiguity. We instantiate this approach with iterative prompting and show that ambiguity is resolved for 50% of those cases. Contributions of this paper are as follows:

1. in-depth analysis of different ambiguity types in coding instructions (especially in plotting)
2. characterization and structuring of pragmatics theories–such as Segmented Discourse Representation Theory (SDRT), Rational Speech Acts (RSA), and Questions Under Discussion (QUD)– around LLMs for dialogue generation to be used in pair programming instructions
3. a technique based on sampling from code LLMs to quantify functional uniqueness and detect ambiguity in natural language coding instructions
4. a new dataset of simulated dialogues called `SyMPa-CoD` between a coder and a director that resolves ambiguities in coding instructions, and we define the new task of Coding Dialogues based on this dataset.

From our analyses, we find that certain pragmatic frameworks perform the best at disambiguating specific categories of ambiguity and that pragmatic frameworks generally increase the executability of the generated code. We make our code, and annotations publicly available for the camera-ready version of this paper.

2 Related Work

Code Generation Large language models of code have shown strong performance on natural language to code generation (Chen et al., 2021; Rozière et al., 2024; Lozhkov et al., 2024, *inter alia*). Increasingly, attention has turned to more realistic uses of code models, beyond single-turn code generation tasks. For instance, Jimenez et al. (2024) introduce SWE-bench, which evaluates models' ability to solve real-world GitHub issues. However, a line of work has investigated how users interact with current code generation models, finding that impressive benchmark performance does not always translate to improved task outcomes for users (Sarkar et al., 2022; Vaithilingam et al., 2022; Ma et al., 2023; Mozannar et al., 2024; Nguyen et al., 2024). Some of this gap can be attributed to the ambiguity inherent to human interactions with code models: Sarkar et al. (2022) observe that user utterances are often underspecified and ambiguous, forcing users to repeatedly refine their prompts and adapt their thought processes to match the LLM; likewise, Mozannar et al. (2024) observe that users often provide fuzzy instructions, motivating a clarification feature.

Ambiguity in Code Generation Recent work has studied ambiguity resolution for code LLMs via clarification questions. Mu et al. (2023) introduce ClarifyGPT, a pipeline for code generation with selective clarification. Similar to our work, their pipeline detects semantic ambiguity via execution of the code model's initial generated code. However, we investigate the use of a broader set of discourse strategies for this task. Li et al. (2023) studies clarification for open-domain code generation in a scaffolded setting.

Ambiguity in NLP Tasks Ambiguity has been studied across a wide array of NLP tasks, including coreference resolution, question answering and machine translation (Poesio and Artstein, 2005; Min et al., 2020; Iyer et al., 2023; Niwa and Iso, 2024) . Current language models generally struggle when applied directly to tasks with ambiguity (Liu et al., 2023); by default, they do not recognize ambiguity in instructions, nor do they seek clarification or engage in proactive dialogue to resolve ambiguity (Deng et al., 2023). However, recent sampling-based methods have shown promise in detecting ambiguity (Kuhn et al., 2023b; Cole et al., 2023; Lin et al., 2024), while prompting and self-improvement methods have proven effective for clarifying ambiguity with LLMs (Krasheninnikov et al., 2022; Kuhn et al., 2023a; Andukuri et al., 2024).

Pragmatics One approach to resolving ambiguity is to assume the speaker is a rational agent playing a cooperative game (Grice, 1975) where they are choosing an utterance that gives the code generation model the best chance of recovering the program they have in mind. This form of inference has been formalized in the Rational Speech Acts framework (Frank and Goodman, 2012) This framework has been productively applied to programming tasks where a user specifies their intent using examples (Pu et al., 2020, 2023; Vaduguru et al., 2024). Similar approaches to disambiguation also been applied to code generation from natural language using large language models (Zhang et al., 2023). Other pragmatic theories of discourse work include RSA for referential communication in a game of color (McMahan and Stone, 2020), question under discussion (Ko et al., 2023), and discourse theories as applied to dialogue settings (Asher et al., 2016; Chi and Rudnicky, 2022; Atwell et al., 2021, 2024, 2022). The frameworks we use to implement our dialogue agents are inspired by these in this work.

3 Ambiguity in Plotting Code

We identify multiple categories of ambiguity – ways in which a natural language instruction may map to multiple code implementations.

- **Semantic ambiguity:** certain wordings and their meaning are not clear in the coding question and can have multiple interpretations.
  e.g. "full line", "solid red", "regular matplotlib style plot"
- **Presupposition:** even though not mentioned explicitly in the prompt (question), the reference code writer has assumed certain things.
  e.g. knowing the default parameter values of the functions
- **Vagueness:** Wording is not precise or a number is not provided.
  e.g. "enough" space between axes.
- **Parameter underspecification:** color, shape, and other multimodal parameters are left to the coder's choice or not mentioned explicitly in the prompt. This has effects on the visual end-result or the code.
  e.g., title is set to be "xxx" even though not mentioned in the coding question
- **Function underspecification:** non-explicit instructions on which function to use. This has effects on the visual end-result or the code.
  e.g. a coding question asks to show a heatmap, which could be implemented by one of multiple functions: `imshow` or `pcolor`.

3.1 Detecting Ambiguity

We do a preliminary analysis of the distribution of these categories of ambiguity in the DS1000 dataset, specifically with matplotlib library questions (Lai et al., 2022). This dataset consists of natural language prompts based on StackOverflow questions related to the Matplotlib library of Python (Hunter, 2007). Our annotators are experts in ambiguity in dialogue, and we ask them to annotate 150 coding instructions from the DS1000. Even though the DS1000 dataset consists of specifically hand-picked questions that are unambiguous, we find that 57% of the plotting questions fall under one or more of the categories we have defined above. Table 1 shows the distribution of different categories.

Ambiguity Category	Distribution
semantic ambiguity	23.8%
presupposition	11.9%
vagueness	11.9%
parameter underspec.	45.2%
function underspec.	16.7%

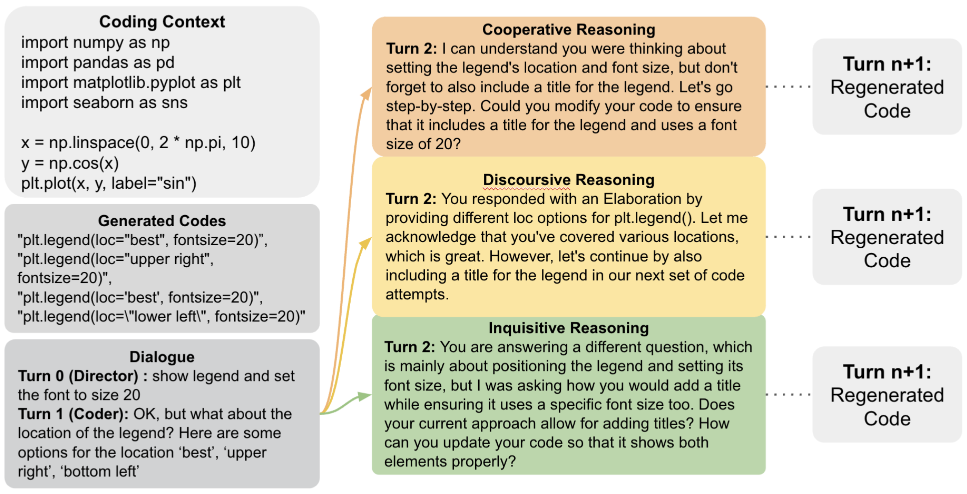
Table 1: This table shows the ambiguity category distribution of the DS1000 dataset based on our annotations.

3.2 Sampling for Ambiguity

One way to measure ambiguity in a given prompt is to count different programs that may obey the constraints specified in the prompt. If two programs are both appropriate responses to the same prompt, then they differ in some way that is not specified – and hence left ambiguous – in the prompt. Since it is infeasible to determine exactly the set of all programs consistent with the prompt, we measure ambiguity by proxy by considering the variation in a sample of programs drawn from a large code language model.

Given a `prompt`, we sample a `response` $\sim P_{\text{LLM}}(\cdot|\texttt{prompt})$ k times. We measure the ratio between the number of distinct `responses`[1] and the total number of `responses` as a proxy for the number of different ways a code model interprets the prompt. The idea of using multiple samples from a code model to measure ambiguity has been

[1]We measure distinctness of responses by comparing their parse trees; details of this Abstract Syntax Tree-based algorithm are given in Appendix B.



Coding Context

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

x = np.linspace(0, 2 * np.pi, 10)
y = np.cos(x)
plt.plot(x, y, label="sin")
```

Generated Codes

```
"plt.legend(loc="best", fontsize=20)",
"plt.legend(loc="upper right",
fontsize=20)",
"plt.legend(loc='best', fontsize=20)",
"plt.legend(loc=\"lower left\", fontsize=20)"
```

Dialogue

Turn 0 (Director) : show legend and set the font to size 20

Turn 1 (Coder): OK, but what about the location of the legend? Here are some options for the location 'best', 'upper right', 'bottom left'

Cooperative Reasoning

Turn 2: I can understand you were thinking about setting the legend's location and font size, but don't forget to also include a title for the legend. Let's go step-by-step. Could you modify your code to ensure that it includes a title for the legend and uses a font size of 20?

Discoursive Reasoning

Turn 2: You responded with an Elaboration by providing different loc options for plt.legend(). Let me acknowledge that you've covered various locations, which is great. However, let's continue by also including a title for the legend in our next set of code attempts.

Inquisitive Reasoning

Turn 2: You are answering a different question, which is mainly about positioning the legend and setting its font size, but I was asking how you would add a title while ensuring it uses a specific font size too. Does your current approach allow for adding titles? How can you update your code so that it shows both elements properly?

Turn n+1: Regenerated Code
Turn n+1: Regenerated Code
Turn n+1: Regenerated Code

Figure 2: This figure shows the dialogue flow for a pragmatic director, where the initial intent of the dialogue is given on the left, and the different responses generated using separate reasoning styles are given in the middle.

considered in Shi et al. (2022). In that work, they start with the hypothesis that models split probability mass across different semantically equivalent (but syntactically distinct) responses. While their method uses variation and agreement across samples to break ties and choose a single response, we instead use it to measure ambiguity.

One observation is that this notion of ambiguity is model-dependent and actually reflects how ambiguous the instruction is from the model's perspective. For instance, a model may incorrectly represent the instruction as ambiguous because of improper training: it does not return the optimal (unambiguous) solution and instead returns diverse incorrect ones. For this reason, we refer to this as Model Ambiguity. This notion of ambiguity is still useful, especially when we assume we are working with a (fixed) large language model as a code generator. It is an easily measured form of ambiguity, which we can attempt to resolve. We report a corrected form of model ambiguity, which restricts computation to only correct model solutions (passes test cases).

4 Pragmatic Modeling of Disambiguation in Coding Dialogues with LLMs

We propose resolving ambiguity in natural language specifications of intent with multi-turn dialogue. Each coding task is defined by the natural language intent I (see Figure 2). We have a sequence of alternating dialogue turns $\mathbf{u} = [u_1^D, u_2^C, u_3^D, u_4^C, \ldots, u_{n-1}^D, u_n^C]$ between the **director** D and the **coder** C.

The dialogue proceeds by each agent taking their turn. The director generates a response $u_i^D = f_D(I, \mathbf{u}^*_{:i-1})$ based on the intent and the dialogue history up to that turn. The coder takes the next turn and generates a natural language response $u_{i+1}^C = f_C(\mathbf{u}^*_{:i-1})$ based on only the dialogue history. At the end of the interaction, the coder model conditions on the entire dialogue history and generates the code solution $t = g_C(\mathbf{u})$ to the task. We enforce a turn-based dialogue structure that takes the following pattern, as also given in Figure 2:

- Director presents initial instruction (a prompt from DS1000).
- Coder takes a dialogue act with access to a set of generated answers to the instruction
- Director responds, using access to the target
- (repeat previous two steps as desired)
- Coder generates code

Dialogue acts can be clarification questions, e.g., "C: what location should I put the legend," which evokes a specific response, "D: The top left corner" or can be more general declarations that start a sub-topic of conversation "C: I'll plan for the default legend arguments. D: Hmm. Keep it on the top left. What else can you change?". We integrate pragmatic frameworks into the turns of the dialogue at each turn of the director and the coder. We give details of this setup in Section §4.3. We propose different ways of instantiating f_D and f_C by prompting large language models inspired by ideas from theories of pragmatics.

In a pair programming task, the director would be a human user, who has an intent specification in their mind (this may not be explicitly available) and they interact with a coder agent to generate code. In this work, we explore ambiguity in code generation using a simulated director and coder. Both agents are simulated by instruction-tuned large language models prompted in different ways (and hence with access to different information about the problem).

While most existing datasets for NL-to-code translation are single-turn, we argue that dialogue can be a principled way to reduce ambiguity. This section details the dialogue structure we assume, as well as the mechanisms through which we execute the discussed pragmatic frameworks. We simulate our dialogues for plotting code using LLMs and based on the algorithm given in Algorithm 1. In the algorithm, f_D and f_C are defined based on different pragmatics strategies as given in detail in Section §4.3.

Algorithm 1 Dialogue Simulation with LLMs

```
Require: Problem instance $(u_1^D, I)$
Require: Director model $f_D$
Require: Coder model for NL response $f_C$
Require: Coder model to generate code $g_C$
Require: Number of samples $k$
Require: Number of rounds of dialogue $n$
 1: $S \leftarrow \{s_i \sim g_C(u_1^D) \mid 1 \leq i \leq k\}$
 2: $\mathbf{u} \leftarrow [\,]$
 3: for $n$ times do
 4:     $u^C \leftarrow f_C(\mathbf{u})$
 5:     $\mathbf{u} \leftarrow \mathbf{u} + [u^C]$
 6:     $u^D \leftarrow f_D(I, \mathbf{u})$
 7:     $\mathbf{u} \leftarrow \mathbf{u} + [u^D]$
 8: end for
 9: $c \sim g_C(\mathbf{u})$
10: return $\mathbf{u}$, $c$
```

4.1 Generating director responses

We prompt the director model f_D to generate instructions and clarifications that guide a coder model toward the correct solution. Since we work with an artificial director agent, we source intents from the DS-1000 dataset. We present the intent to the director in one of two ways – as the code for a reference solution or the plot generated by the code presented as an image. Since a natural language instruction accompanies the DS-1000 problem instances, we seed the interaction using that interaction as the first director turn (u_1^D). We prompt the model to use different strategies to generate responses.

4.2 Generating coder responses

We prompt a coder model to generate responses that clarify intent and resolve ambiguity. To provide the coder model with an explicit representation of its ambiguity, we first sample candidate solutions by instructing the model to generate code to solve the problem based on the initial turn of the interaction, i.e. $g_C(u_1^D)$. We then list these solutions and instruct the coder model to engage in dialogue to resolve ambiguity and arrive at a single solution.

4.3 Pragmatics Frameworks

After defining ambiguity in plotting code and our ways of detecting it using sampling language models, this section proposes possible strategies for modeling disambiguation with follow-up coding dialogues between a coding agent and a director. We ground these strategies in several Pragmatics frameworks and analyze how they can mitigate and disambiguate various categories of ambiguity differently.

We operationalize our pragmatics frameworks by turning our dialogue setting into role prompting, persona prompting, and in-context learning as described in Wang et al. (2023); Schulhoff et al. (2024), and Zheng et al. (2023) to implement our proposed pragmatics reasoning styles using role-playing capabilities of agent-level LLMs. Also, we employ prompt-boosted Theory-of-Mind capabilities as described in (Moghaddam and Honey, 2023). In the following sections, we describe our formalizations of this transformation and role prompting for three different reasoning styles: cooperative, discoursive, and inquisitive.

Cooperative Reasoning The first framework we use is based on Grice's maxims of cooperative dialogue partners (Grice, 1975; Horn, 1984; Levinson, 2000; McMahan and Stone, 2020). Here, the interlocutors are pragmatic agents, where they recursively engage in interaction and model each other's state of mind while responding to an utterance. The Rational Speech Act (RSA) framework is the most well-known Bayesian implementation of cooperative reasoning. Inspired by this, we define the disambiguation mapping in coding instructions as the following optimization function,

$$f_D(I, \mathbf{u}) = \text{argmin}_n(U_D(I, \mathbf{u}) \wedge U_C(\mathbf{u})) \quad (1)$$

where U represents a cooperative utterance that considers the other interlocutor's beliefs and reasoning styles. Overall, the function tries to minimize the number of turns in the dialogue, which consists of cooperative utterances.

Discoursive Reasoning The second pragmatics framework is based on Discourse theories. Here, the interlocutors are not necessarily responding strategically. Still, each of their utterances is related to the history of the conversation and the coding context with a set of coherence relations. Hence, when a pragmatic coding director produces an utterance, the utterance relates to the reference solution. When a coder produces an utterance, it relates to the set of solutions they have sampled and what the director has said in the previous turn. This definition of discourse is mostly similar to SDRT-like dialogue-based relation categories (Ko et al., 2023; Asher et al., 2016; Fu, 2022; Atwell et al., 2024; Alikhani et al., 2023). Inspired by this, we propose the following discourse-based disambiguation function,

$$f_D(I, \mathbf{u}) = f_D(I, \mathbf{u}) \bowtie_C f_C(\mathbf{u}) \tag{2}$$

Here, $\bowtie$ represents the coherence relations that exist between the coder and the director agents' utterances. These relations should be from the set of $\bowtie_C \subset$ {Comment, Clarification Question, Elaboration, Acknowledgment, Continuation, Explanation, Conditional, Alternation, Result, Background, Narration, Correction, Parallel, Contrast} as defined in Chi and Rudnicky (2022).

Inquisitive Reasoning The third Pragmatics framework is relative to discourse theories but focuses more on question-type relations. In this case, each interlocutor's utterance explicitly answers an implicit question the other interlocutor poses. This discourse framing is described in (Clifton and Frazier, 2012) under the umbrella term of Question Under Discussion (QUD). When a director gives a coding instruction, the pragmatic coder with QUD understanding recognizes an implied question under the instruction and gives an answer satisfying that question. This happens over a dialogue that optimizes the semantic distance between the implied question understood by the coder and the actual instruction given by the director. The following disambiguation function can represent this,

$$f_D(I, \mathbf{u}) = \text{argmin}_u |Q_D(I, \mathbf{u} + [u]) - Q_C(\mathbf{u} + [u])| \tag{3}$$

where Q_D represents the actual instruction given by the director, and Q_C is the understood implicit question by the coder, and $| \cdot |$ represents the semantic distance between the questions. The disambiguation function aims to minimize the difference between these two questions over dialogue turns.

4.4 Generating dialogues

Pragmatic Director In this scenario, we hypothesize that a pragmatic coding director chooses their utterances according to the frameworks we presented in Section §4. To simulate a dialogue between a pragmatic director and a coder, we carry out in-context learning and prefix-tuning using two separate instruction-tuned LLM agents (using GPT-4o in this case) and generate utterances for both of the interlocutors.

We use a system prompt for the pragmatic director, which instructs the agent's purpose and outlines the requirements and the structure of the dialogue that is taking place between two interlocutors. The details of the prompt are given in Appendix A.

Pragmatic Coder We present a second scenario, which investigates different reasoning strategies applied to the coder. The dialogue setup for the pragmatic coder and regular director is similar to the previous scenario but with the key difference of having multiple reasoning types for the coder instead of the director. We first extract the code context and the coding instructions from the DS1000 dataset and then convert it into a dialogue format as described in section § 4. Then, using CodeLLaMA-13B, we generate codes that respond to the original instruction (sampled k times)[2].

To the pragmatic coder, we present a set of possible unique answers it can choose from the generated codes and the dialogue history that is happening and ask for a follow-up utterance for the coder to converge to the solution that the director is describing. We then instruct it to give three solutions based on the reasoning types. For the regular director, we provide the reference code (or the reference plot in the case of a multimodal model) and the dialogue history and ask to generate a follow-up utterance to converge to a solution without giving away the answer. All the details of the prompts are given in Appendix A.

Dialogue Policy We employ a rule-based dialogue policy to choose one of the three utterances we generated for each strategy in the simulation.

[2]We mix code-specialized LLMs (CodeLLaMA) with dialogue-specialized LLMs (GPT-4o) in our experiments

For the first turn of the dialogue, we do not use any LLM generations but directly use the coding instruction from the DS1000 dataset. For the following turns, we generate three different utterances, one for each of the pragmatic director's reasoning ways, and then generate a single utterance without any pragmatic reasoning prompting for the coder for each of the three responses of the director. We use the number of turns as a hyperparameter to generate the dialogue and perform ablation experiments on it. We do not mix reasoning styles across the dialogue's turns, but we choose a single reasoning style for the overall dialogue. We also experiment with providing the reference image or the reference code to the director to see how clarity of instructions affects execution.

4.5 Dataset Creation

We create the first synthetic dialogue dataset for plotting codes using the aforementioned strategies. We call this dataset `SyMPa-CoD` (**Sy**nthetic **M**ultimodal **Pra**gmatic **Co**ding **D**ialogues), where we provide three different dialogues with multiple turns (t=3,4,5) for each Matplotlib question in the DS1000 dataset. In total, we provide 450 pragmatic coder with regular director, and 450 pragmatic director with regular coder dialogues based on the 150 Matplotlib coding questions.

5 Experiments & Findings

We experiment with different LLMs, code generation models, pragmatic scenarios, and the number of turns during and after the dataset's construction. In this section, we provide details of these experiments. We pose multiple research questions and report our findings in combination with our experiments. We first describe the automatic metrics based on our sampling approach for ambiguity as described in Section §3, and then follow-up with experimentation based on the dialogue approach to coding we described in Section §4.

5.1 Automatic Metrics

We measure the system's success using two automatic metrics: mean pass@1 and sampling diversity. Pass@1 score is used directly from the DS1000 unit tests, with 30 samples from the coding LLM. This score measures how many of the samples execute and pass the unit tests that were specifically hand-written for the questions. Sampling diversity is calculated by using the code generations from the coding LLMs with the following formula:

$$d_s = \sum \frac{\#\text{Unique code completions}}{\#\text{Total samples per question}} \quad (4)$$

We use these two metrics as proxies for evaluating successful, executable disambiguation and present our quantitative results using them.

			Pass@1↑	d_s↓
Baseline		No Dialogue	0.422	0.744
Pragmatic Director	with code	Cooperative	0.484	**0.569**
		Discoursive	**0.500**	0.609
		Inquisitive	0.407	0.796
	with image	Cooperative	0.447	0.600
		Discoursive	0.353	0.611
		Inquisitive	0.362	0.722
Pragmatic Coder	with code	Cooperative	0.427	0.640
		Discoursive	0.467	0.613
		Inquisitive	0.396	0.716
	with image	Cooperative	0.447	**0.584**
		Discoursive	**0.493**	0.624
		Inquisitive	0.393	0.711

Table 2: This figure shows the main results of our experimentation between pragmatic director and pragmatic coder. Here, we give the metrics for both executability and sampling diversity. Having a dialogue generally performs better than the baseline code completion without any dialogue. For each pragmatic setting, we experiment with all the reasoning styles and have an image or code as the reference solution for the director.

What is the best model temperature to depict uniqueness for code completion in coding dialogues? We noted previously that we work with functional uniqueness as a proxy for diversity in the generated answers and the ambiguity. However, this is a proxy governed by multiple parameters, one of which is the temperature used to sample from the code generation model. When the coding instruction is very specific, even if temperature is increased, the solutions tend to be very similar, but by default, an increase in temperature results in an increase in uniqueness. From our analyses, as we present them in Appendix Figure 3, we observe that higher temperatures have higher variability in their uniqueness but they produce more unique codes compared to lower temperatures. This also changes the executability and the representation of ambiguity in the answers by the model.

What is the difference in ambiguity resolution and executability between a pragmatic coder and a pragmatic director? In order to answer this question, we run multiple experiments with

Coding Question	Ambiguity	Baseline	Pragmatic Director			Pragmatic Coder		
draw a line (with random y) for each different line style	underspecification	0.000	0.267	0.000	0.000	0.000	0.200	0.000
draw a full line from (0,0) to (1,2)	semantic ambiguity	0.000	0.000	0.000	0.000	0.067	0.000	0.000
make seaborn relation plot and color by the gender field of the dataframe df	parameter underspecification	0.067	0.733	0.700	0.667	0.533	0.000	0.000
highlight in red the x range 2 to 4	vagueness	0.667	0.833	0.700	0.667	0.967	1.000	0.167

Table 3: This table shows a breakdown of the final executability scores for different questions in the DS1000 dataset, with their annotated ambiguity categories. The examples are picked to show when most models have low scores, or to show the performance according to different categories of ambiguity. From left to right, both pragmatic director and pragmatic coder have three columns corresponding to Cooperative, Discoursive and Inquisitive reasoning styles.

pragmatic director and coder scenarios separated, which is presented in Table 2. The details of their prompting are given in Section §4. We can observe two main trends in the overall results.

First, cooperative and discoursive reasoning styles get higher executability scores and lower sampling diversity, meaning that they disambiguate better than other strategies. The pragmatic director has the highest executability, especially with reference code, in comparison to the pragmatic coder, and this is because when one side of the conversation has full access to the reference solution and different reasoning styles, it performs the best. This scenario is not very realistic, as it is modeling a user of a code generation system and has full access to the final solution code.

Secondly, if we focus on the pragmatic coder setting, we can see that reference image-based generations are better than reference code. We postulate that in this scenario, the director does not give away the solution easily but can process the ambiguities in the final image to give meaningful instructions to satisfy the final execution tests. For the pragmatic coder with reference images, the highest executability is with the discoursive reasoning, and the lowest sampling diversity is with the cooperative reasoning – the same trend was observed with the pragmatic director with reference code. This may be due to cooperative reasoning trying to minimize the number of turns while discoursive can correctly identify the ambiguous intents (coherence relations) to improve the executability.

The inquisitive reasoning style does not yield the best results in both cases. In certain cases, its performance falls below the baseline, which has no dialogue. To understand the reasons for the low inquisitive performance, we do a detailed error analysis with a breakdown of each question in Section §5.2.

What is the effect of the number of turns on the disambiguation efficiency of the generated dialogue? We experimented with several numbers of turns, from 2 to 4 turns. The mean pass@1 scores for 2 turns were the lowest (0.400 for most strategies), while 3 turns yielded the results we have previously presented. Hence, the number of turns affects the overall performance, but it is inconclusive if more turns are better for executability.

5.2 Error Analysis

We provide a detailed breakdown of the performance of different dialogue strategies in Table 3. The key takeaway is that some questions are still hard even after dialogue, but specific ambiguity categories have a best-performing pragmatic strategy. Nearly none of the frameworks got the first underspecification question correct (mean pass@1: 0.000). Only cooperative reasoning got it sometimes (mean pass@1: 0.267). Interestingly, sometimes, additional dialogue made the performance worse for all the frameworks. Inquisitive reasoning performed the best with vagueness categories, while discoursive and cooperative performed the best for parameter underspecification.

6 Conclusion

Overall, in this paper, we have proposed a dialogue-oriented perspective to code generation. We characterized various pragmatics frameworks in relation to pair-programming-like dialogues that happen between a director and a coder. We then analyzed the effects of having dialogues with different reasoning strategies on the executability and disambiguation of the final generated code. As having a dialogue based on code is becoming the norm with LLMs, focusing on the pragmatics of dialogue opens up new venues for developing dialogue systems, datasets, and evaluation mechanisms for code generation.

7 Limitations

We proposed using pragmatic dialogue for code generation, but the major limitation is from the side of human data collection and evaluation. We resorted to automatic metrics already being used or developed for this study to evaluate our setup without relying on human annotators. However, this entails that the evaluations may not be human-like and may not show the most accurate representations even though they show improvements in generally accepted code executability standards. Further, we did not deploy a dialogue system to study our approach. Instead, we resorted to simulations using LLMs, which may or may not accurately represent how a human interlocutor would act in a real-world setting. We wanted to minimize this by using large parameter models for dialogue generation and StackOverflow-based code instructions from the DS1000 dataset.

8 Ethics Statement

In our simulation process we have used GPT-4o, and this is a closed-source LLM, and we are aware that this model can propagate its own training biases. The scientific community does not have access to any information regarding how this model is trained or what the dataset consists of. This may result in a deficient evaluation of the final performance and human-likeness of the generated dialogue. This is a simulated analysis study to identify and characterize pragmatics frameworks with possible LLM behavior in a pair programming setting. Hence, we do not involve humans in our current setup. The biases propagated by GPT-4o are the responsibility of OpenAI and should be held accountable by their and the scientific community's ethical standards.

References

Malihe Alikhani, Baber Khalid, and Matthew Stone. 2023. Image–text coherence and its implications for multimodal AI. *Front. Artif. Intell.*, 6:1048874.

Chinmaya Andukuri, Jan-Philipp Fränken, Tobias Gerstenberg, and Noah D. Goodman. 2024. Star-gate: Teaching language models to ask clarifying questions. *ArXiv*, abs/2403.19154.

Nicholas Asher, Julie Hunter, Mathieu Morey, Benamara Farah, and Stergos Afantenos. 2016. Discourse structure and dialogue acts in multiparty dialogue: the STAC corpus. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, pages 2721–2727, Portorož, Slovenia. European Language Resources Association (ELRA).

Katherine Atwell, Remi Choi, Junyi Jessy Li, and Malihe Alikhani. 2022. The role of context and uncertainty in shallow discourse parsing. In *Proceedings of the 29th International Conference on Computational Linguistics*, pages 797–811, Gyeongju, Republic of Korea. International Committee on Computational Linguistics.

Katherine Atwell, Mert Inan, Anthony B. Sicilia, and Malihe Alikhani. 2024. Combining discourse coherence with large language models for more inclusive, equitable, and robust task-oriented dialogue. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 3538–3552, Torino, Italia. ELRA and ICCL.

Katherine Atwell, Junyi Jessy Li, and Malihe Alikhani. 2021. Where are we in discourse relation recognition? In *Proceedings of the 22nd Annual Meeting of the Special Interest Group on Discourse and Dialogue*, pages 314–325, Singapore and Online. Association for Computational Linguistics.

Shobhit Chaurasia and Raymond J. Mooney. 2017. Dialog for language to code. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 175–180, Taipei, Taiwan. Asian Federation of Natural Language Processing.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. Codet: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations*.

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. *Preprint*, arXiv:2107.03374.

Ta-Chung Chi and Alexander Rudnicky. 2022. Structured dialogue discourse parsing. In *Proceedings* *of the 23rd Annual Meeting of the Special Interest Group on Discourse and Dialogue*, pages 325–335, Edinburgh, UK. Association for Computational Linguistics.

Charles Clifton, Jr. and Lyn Frazier. 2012. Discourse Integration Guided by the 'Question under Discussion'. *Cognit. Psychol.*, 65(2):352.

Jeremy Cole, Michael Zhang, Daniel Gillick, Julian Eisenschlos, Bhuwan Dhingra, and Jacob Eisenstein. 2023. Selectively answering ambiguous questions. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 530–543, Singapore. Association for Computational Linguistics.

Yang Deng, Lizi Liao, Liang Chen, Hongru Wang, Wenqiang Lei, and Tat-Seng Chua. 2023. Prompting and evaluating large language models for proactive dialogues: Clarification, target-guided, and non-collaboration. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 10602–10621, Singapore. Association for Computational Linguistics.

Ahmed Elgohary, Saghar Hosseini, and Ahmed Hassan Awadallah. 2020. Speak to your parser: Interactive text-to-SQL with natural language feedback. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2065–2077, Online. Association for Computational Linguistics.

Ahmed Elgohary, Christopher Meek, Matthew Richardson, Adam Fourney, Gonzalo Ramos, and Ahmed Hassan Awadallah. 2021. NL-EDIT: Correcting semantic parse errors through natural language interaction. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 5599–5610, Online. Association for Computational Linguistics.

Michael C. Frank and Noah D. Goodman. 2012. Predicting pragmatic reasoning in language games. *Science*, 336(6084):998–998.

Daniel Fried, Nicholas Tomlin, Jennifer Hu, Roma Patel, and Aida Nematzadeh. 2023. Pragmatics in language grounding: Phenomena, tasks, and modeling approaches. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 12619–12640, Singapore. Association for Computational Linguistics.

Yingxue Fu. 2022. Towards unification of discourse annotation frameworks. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics: Student Research Workshop*, pages 132–142, Dublin, Ireland. Association for Computational Linguistics.

H. P. Grice. 1975. Logic and Conversation. In *Speech Acts*, pages 41–58. Brill, Leiden, The Netherlands.

Laurence Horn. 1984. Toward a new taxonomy for pragmatic inference: Q-based and r-based implicature.

J. D. Hunter. 2007. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95.

Vivek Iyer, Pinzhen Chen, and Alexandra Birch. 2023. Towards effective disambiguation for machine translation with large language models. In *Proceedings of the Eighth Conference on Machine Translation*, pages 482–495, Singapore. Association for Computational Linguistics.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? *Preprint*, arXiv:2310.06770.

Wei-Jen Ko, Yating Wu, Cutter Dalton, Dananjay Srinivas, Greg Durrett, and Junyi Jessy Li. 2023. Discourse analysis via questions and answers: Parsing dependency structures of questions under discussion. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 11181–11195, Toronto, Canada. Association for Computational Linguistics.

Dmitrii Krasheninnikov, Egor Krasheninnikov, and David Krueger. 2022. Assistance with large language models. In *NeurIPS ML Safety Workshop*.

Lorenz Kuhn, Yarin Gal, and Sebastian Farquhar. 2023a. Clam: Selective clarification for ambiguous questions with generative language models. *Preprint*, arXiv:2212.07769.

Lorenz Kuhn, Yarin Gal, and Sebastian Farquhar. 2023b. Semantic uncertainty: Linguistic invariances for uncertainty estimation in natural language generation. In *The Eleventh International Conference on Learning Representations*.

Igor Labutov, Bishan Yang, and Tom Mitchell. 2018. Learning to learn semantic parsers from natural language supervision. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1676–1690, Brussels, Belgium. Association for Computational Linguistics.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2022. Ds-1000: A natural and reliable benchmark for data science code generation. *Preprint*, arXiv:2211.11501.

Stephen C. Levinson. 2000. *Presumptive Meanings: The Theory of Generalized Conversational Implicature*. The MIT Press, Cambridge, MA, USA.

Haau-Sing (Xiaocheng) Li, Mohsen Mesgar, André Martins, and Iryna Gurevych. 2023. Python code generation by asking clarification questions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14287–14306, Toronto, Canada. Association for Computational Linguistics.

Yuntao Li, Bei Chen, Qian Liu, Yan Gao, Jian-Guang Lou, Yan Zhang, and Dongmei Zhang. 2020. "what do you mean by that?" a parser-independent interactive approach for enhancing text-to-SQL. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6913–6922, Online. Association for Computational Linguistics.

Zhen Lin, Shubhendu Trivedi, and Jimeng Sun. 2024. Generating with confidence: Uncertainty quantification for black-box large language models. *Transactions on Machine Learning Research*.

Alisa Liu, Zhaofeng Wu, Julian Michael, Alane Suhr, Peter West, Alexander Koller, Swabha Swayamdipta, Noah Smith, and Yejin Choi. 2023. We're afraid language models aren't modeling ambiguity. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 790–807, Singapore. Association for Computational Linguistics.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024. Starcoder 2 and the stack v2: The next generation. *Preprint*, arXiv:2402.19173.

Qianou Ma, Tongshuang Wu, and Kenneth Koedinger. 2023. Is ai the better programming partner? human-human pair programming vs. human-ai pair programming. *Preprint*, arXiv:2306.05153.

Brian McMahan and Matthew Stone. 2020. Analyzing speaker strategy in referential communication. In *Proceedings of the 21th Annual Meeting of the Special Interest Group on Discourse and Dialogue*, pages 175–185, 1st virtual meeting. Association for Computational Linguistics.

Sewon Min, Julian Michael, Hannaneh Hajishirzi, and Luke Zettlemoyer. 2020. AmbigQA: Answering ambiguous open-domain questions. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 5783–5797, Online. Association for Computational Linguistics.

Shima Rahimi Moghaddam and Christopher J. Honey. 2023. Boosting theory-of-mind performance in large language models via prompting. *Preprint*, arXiv:2304.11490.

Hussein Mozannar, Valerie Chen, Mohammed Alsobay, Subhro Das, Sebastian Zhao, Dennis Wei, Manish Nagireddy, Prasanna Sattigeri, Ameet Talwalkar, and David Sontag. 2024. The realhumaneval: Evaluating large language models' abilities to support programmers. *Preprint*, arXiv:2404.02806.

Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, Chenxue Wang, Shichao Liu, and Qing Wang. 2023. Clarifygpt: Empowering llm-based code generation with intention clarification. *Preprint*, arXiv:2310.10996.

Sydney Nguyen, Hannah McLean Babe, Yangtian Zi, Arjun Guha, Carolyn Jane Anderson, and Molly Q Feldman. 2024. How beginning programmers and code llms (mis)read each other. CHI '24, New York, NY, USA. Association for Computing Machinery.

Ayana Niwa and Hayate Iso. 2024. Ambignlg: Addressing task ambiguity in instruction for nlg. *Preprint*, arXiv:2402.17717.

Massimo Poesio and Ron Artstein. 2005. The reliability of anaphoric annotation, reconsidered: Taking ambiguity into account. In *Proceedings of the Workshop on Frontiers in Corpus Annotations II: Pie in the Sky*, pages 76–83, Ann Arbor, Michigan. Association for Computational Linguistics.

Yewen Pu, Kevin Ellis, Marta Kryven, Josh Tenenbaum, and Armando Solar-Lezama. 2020. Program synthesis with pragmatic communication. In *Advances in Neural Information Processing Systems*, volume 33, pages 13249–13259. Curran Associates, Inc.

Yewen Pu, Saujas Vaduguru, Priyan Vaithilingam, Elena Glassman, and Daniel Fried. 2023. Amortizing pragmatic program synthesis with rankings. *Preprint*, arXiv:2309.03225.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code llama: Open foundation models for code. *Preprint*, arXiv:2308.12950.

Advait Sarkar, Andrew D. Gordon, Carina Negreanu, Christian Poelitz, Sruti Srinivasa Ragavan, and Ben Zorn. 2022. What is it like to program with artificial intelligence? *Preprint*, arXiv:2208.06213.

Sander Schulhoff, Michael Ilie, Nishant Balepur, Konstantine Kahadze, Amanda Liu, Chenglei Si, Yinheng Li, Aayush Gupta, HyoJung Han, Sevien Schulhoff, Pranav Sandeep Dulepet, Saurav Vidyadhara, Dayeon Ki, Sweta Agrawal, Chau Pham, Gerson Kroiz, Feileen Li, Hudson Tao, Ashay Srivastava, Hevander Da Costa, Saloni Gupta, Megan L. Rogers, Inna Goncearenco, Giuseppe Sarli, Igor Galynker, Denis Peskoff, Marine Carpuat, Jules White, Shyamal Anadkat, Alexander Hoyle, and Philip Resnik. 2024. The prompt report: A systematic survey of prompting techniques. *Preprint*, arXiv:2406.06608.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. 2022. Natural language to code translation with execution. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 3533–3546, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Yuan Tian, Zheng Zhang, Zheng Ning, Toby Li, Jonathan K. Kummerfeld, and Tianyi Zhang. 2023. Interactive text-to-SQL generation via editable step-by-step explanations. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 16149–16166, Singapore. Association for Computational Linguistics.

Saujas Vaduguru, Daniel Fried, and Yewen Pu. 2024. Generating pragmatic examples to train neural program synthesizers. In *The Twelfth International Conference on Learning Representations*.

Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *CHI EA '22: Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*, pages 1–7. Association for Computing Machinery, New York, NY, USA.

Zekun Moore Wang, Zhongyuan Peng, Haoran Que, Jiaheng Liu, Wangchunshu Zhou, Yuhan Wu, Hongcheng Guo, Ruitong Gan, Zehao Ni, Jian Yang, Man Zhang, Zhaoxiang Zhang, Wanli Ouyang, Ke Xu, Stephen W. Huang, Jie Fu, and Junran Peng. 2023. RoleLLM: Benchmarking, Eliciting, and Enhancing Role-Playing Abilities of Large Language Models. *arXiv*.

L. Williams. 2001. Integrating pair programming into a software development process. In *Proceedings 14th Conference on Software Engineering Education and Training. 'In search of a software engineering profession' (Cat. No.PR01059)*, pages 27–36.

Ziyu Yao, Yu Su, Huan Sun, and Wen-tau Yih. 2019. Model-based interactive semantic parsing: A unified framework and a text-to-SQL case study. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 5447–5458, Hong Kong, China. Association for Computational Linguistics.

Tianyi Zhang, Tao Yu, Tatsunori Hashimoto, Mike Lewis, Wen-Tau Yih, Daniel Fried, and Sida Wang. 2023. Coder reviewer reranking for code generation. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 41832–41846. PMLR.

Mingqian Zheng, Jiaxin Pei, and David Jurgens. 2023. Is "a helpful assistant" the best role for large language models? a systematic evaluation of social roles in system prompts. *Preprint*, arXiv:2311.10054.

Ruiqi Zhong, Charlie Snell, Dan Klein, and Jason Eisner. 2023. Non-programmers can label programs indirectly via active examples: A case study with text-to-SQL. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 5126–5152, Singapore. Association for Computational Linguistics.

A Prompting Details

A.1 Pragmatic Director

Here, we provide the prompts we used for the pragmatic director and regular coder.

A.1.1 System prompts:

Director: You are a coding director. The things you say depend on your persona. You have the following different personas (reasoning styles):

- Cooperative Persona (Pragmatic): You want to converge on the solution as quickly as possible and follow Grice's Maxims when choosing your words. You anticipate the coder's cooperative reasoning. You possess theory-of-mind capabilities and common sense. You MUST start your utterance with variations of: "I can understand you were thinking about [coder's cooperative reasoning]. Let's go step-by-step."

- Linguistic Reasoning Persona (Literal): You choose words according to semantic differences. You elaborate or describe the task more to target the ref code and exclude distractors. You elaborate, describe the target, and exclude the alternatives in the generated code. You MUST start your utterance with variations of: "Let's move away from [distractors in the generated answers]. Let me elaborate on my question."

- Questioning Persona: Everything you say has an implicit question underneath it. You MUST start your utterance with variations of: "You are answering a different question, which is [implicit question]."

You have a final product in mind. This is going to be named the REF CODE. You want a coder to write the codes for this final product. For the first turn of the dialogue, you give a specific instruction

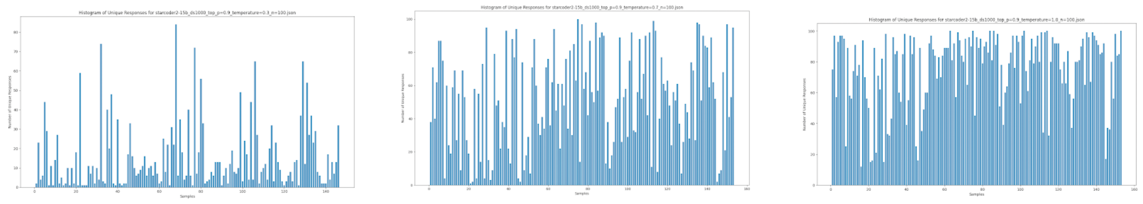


Figure 3: This figure shows the change in unique responses of code completions depending on the temperature of the model. From the left, the plots are showing histograms for 0.3, 0.7, and 1.0 temperatures. The horizontal axis is the question number from the DS1000 matplotlib dataset. It is observable that the uniqueness is high for higher temperatures, expectedly. However, very high temperatures may have minor differences that increase the overall uniqueness. Hence, a moderate temperature like 0.7 gives more reliable results for further experimentation.

or a question about the final product. Then, the coder will give you some answers, and then you will have another turn to refine the codes.

Coder: No system prompt.

Coder: You are given the following dialogue that happens between a coding director and a coder:

Director: Here is a code snippet: [code-context]
Director: coding-question
Coder: Here are the unique generated codes:
Solution 1: [CODE] Solution 2: [CODE] ... Director: generated-dialogue
Give the next turn in the dialogue for the coder with a new code solution. No unnecessary explanations, and give the code in a code block with “‘ CODE “‘. Short replies only, just give the dialogue turn.

A.2 Pragmatic Coder

A.2.1 System Prompts:

Director: You are a coding director. There is another coding agent you are going to have a dialogue with. You have a final product in mind. This is going to be named the REF CODE. You want a coder to write the codes for this final product. For the first turn of the dialogue, you give a specific instruction or a question about the final product. Then, the coder will give you some answers, and then you will have another turn to refine the codes.

Coder: You are a coding agent. There is another director agent you are going to have a dialogue with. The things you say depend on your persona. You have the following different personas (reasoning styles):

- Cooperative Persona (Pragmatic): You want to converge on the solution as quickly as possible and follow Grice's Maxims when choosing your words. You anticipate the director's cooperative reasoning. You possess theory-of-mind capabilities and common sense.

- Discourse Reasoning Persona: Everything you say is connected to the previous turn with a relation. The possible discourse relations are Comment, Clarification Question, Elaboration, Acknowledgment, Continuation, Explanation, Conditional, Alternation, Result, Background, Narration, Correction, Parallel, Contrast. You try to identify the relation between the utterance of the director in the previous with your utterance. Then you reply with an utterance that has the appropriate relation.

- Questioning Persona: Everything you say has an implicit question underneath it. You should tell what the director is actually asking for (the question under their instruction), and give your answer to that implicit question.

The director has a final product in mind. You, as the coder, write the codes for this final product or have a dialogue about the instruction. For the first turn of the dialogue, the director gives a specific instruction or a question about the final product. Then, you will give some answers, and then the director will have another turn to refine the codes.

user prompts:

Director: REF CODE: “‘+ ref-code “‘ + DIALOGUE HISTORY:" + dialogue-history + What can you say on the follow-up turn for the coder to converge to the reference code? Do not mention anything about the REF CODE, and don't give away the answer.

Coder: POSSIBLE GENERATED CODES: Solution 1: “‘CODE“‘ Solution 2: “‘CODE“‘ ....

DIALOGUE HISTORY: + dialogue-history + What can you say on the following turn as the coder to converge to the solution that the director has in mind? Give responses for all types of your personas. Personas must not give the same solution! Your solution MUST NOT contain any new code. You can talk about the provided code.

B Abstract Syntax Tree (AST) Functional Uniqueness Algorithm

In this section, we detail the AST-based function uniqueness comparison algorithm between two separate generated functions. The code for the algorithm is given in Listing 1. We find this form of comparison to be appropriate for plotting tasks as the lines of code of interest are generally the calls to library functions, particularly those provided by the `matplotlib` API.

C Temperature Adjustments

We present our experimentation results for the temperature tuning in Figure 3.

D Additional Related Work

Interactive Semantic Parsing Prior work on interactive semantic parsing has also extensively studied ambiguity resolution for tasks adjacent to code generation, including text-to-SQL. These works can be broadly separated into three modes of user interaction: (1) asking clarification questions (Chaurasia and Mooney, 2017; Yao et al., 2019; Li et al., 2020) (2) requesting natural language feedback on simplified representations of the parse (Labutov et al., 2018; Elgohary et al., 2020, 2021; Tian et al., 2023) and (3) requesting labels for specific inputs (Zhong et al., 2023; Chen et al., 2023). Our work aligns most closely with the first mode; however, we study a wider set of discourse strategies, beyond just clarification questions.

```python
def compare_parse_trees(response1, response2):
    """Compare the parse trees of two responses."""
    unique_function_calls = []
    unique_params = {}
    unique_keywords = {}
    try:
        tree1 = ast.parse(response1)
        functions1 = get_params(tree1)
        tree2 = ast.parse(response2)
        functions2 = get_params(tree2)
        for function in functions1.keys():
            if function not in functions2.keys():
                unique_function_calls.append(function)
            else:
                for i, arg in enumerate(functions1[function]):
                    if arg not in functions2[function]:
                        if function not in unique_params.keys():
                            unique_params[function] = []
                        unique_params[function].append(arg)
                    if isinstance(arg, dict):
                        for key in arg.keys():
                            for j in range(len(functions2[function])):
                                if isinstance(functions2[function][j], dict):
                                    if key not in functions2[function][j].keys():
                                        if function not in unique_keywords.keys():
                                            unique_keywords[function] = []
                                        unique_keywords[function].append(key)
                                    else:
                                        if arg[key] != functions2[function][j][key]:
                                            if function not in unique_keywords.keys
    ():
                                                unique_keywords[function] = []
                                            unique_keywords[function].append(key)
    except SyntaxError:
        print("Syntax Error")
    return unique_function_calls, unique_params, unique_keywords
```
Listing 1: This code snippet shows how the functions of two separate generated codes are compared using their ASTs.