Craft an Iron Sword: Dynamically Generating Interactive Game Characters by Prompting Large Language Models Tuned on Code

Ryan Volum Sudha Rao Michael Xu Gabriel DesGarennes Chris Brockett
Benjamin Van Durme Olivia Deng Akanksha Malhotra Bill Dolan
Microsoft Corporation

sudhra@microsoft.com

Abstract

Non-Player Characters (NPCs) significantly enhance the player experience in many games. Historically, players' interactions with NPCs have tended to be highly scripted, to be limited to natural language responses to be selected by the player, and to not involve dynamic change in game state. In this work, we demonstrate that use of a few example conversational prompts can power a conversational agent to generate both natural language and novel code. This approach can permit development of NPCs with which players can have grounded conversations that are free-form and less repetitive. We demonstrate our approach using OpenAI Codex (GPT-3 finetuned on GitHub), with Minecraft game development as our test bed. We show that with a few example prompts[1], a Codex-based agent can generate novel code, hold multi-turn conversations and answer questions about structured data. We evaluate this application using experienced gamers in a Minecraft realm and provide analysis of failure cases and suggest possible directions for solutions.

Figure 1: A Minecraft player interacting with a Codex-powered NPC in in two scenarios: question answering (top) and task completion (bottom).

1 Introduction

The recent advent of large pre-trained language models such as GPT-2 (Radford et al., 2019) and GPT-3 (Brown et al., 2020) has fostered spectacular advances in text-generation. In this work, we focus on the potential application of these large language models in video games. In games, Non-Player Characters (NPCs) enhance the player experience by providing interaction, often involving conversation. Currently players' conversations with NPCs are highly scripted: in a typical scenario players must select from a set of preset responses that they can give to the NPC. Moreover, this interaction is limited to natural language responses, and does not directly involve dynamic game state change as part of the interaction. Below, we explore some first steps towards creating functionally agentive NPCs with which players can hold free-form conversations that are grounded in the game and which players can instruct to perform actions that change the game state *by having the NPC adaptively generate code* that calls functions exposed by the game API. This is done by a single language model that generates both natural language and code. To this end, we use OpenAI Codex (Chen et al., 2021) (a GPT-3 model finetuned on GitHub data). We demonstrate that by simply including examples of both natural language conversations and code in the prompt, Codex can generalize to interesting new settings, opening up intriguing possibilities for enhanced player experiences and game development.

We employ Minecraft as our test bed. First, this

[1]We release the prompt and the wrapper code in this repository https://github.com/microsoft/interactive-minecraft-npcs

```
`// This file contains Minecraft bot commands and the code needed to accomplish them using the Mineflayer
// JavaScript library. If asked something conversational, the bot should use bot.chat() to answer.

// come to me
goToPlayer(bot, 3, username)

// Now follow me!
goToPlayerInterval = setInterval(() => goToPlayer(bot, 3, username), 3000)

// stop
bot.clearControlStates();
clearInterval(goToPlayerInterval)

// good work!
bot.chat("Thanks!")

// How are you?
bot.chat("I'm great! How about you?");

// Open the chest
locateBlock(bot, 'chest', 1)
.then(chestBlock => openChest(bot, chestBlock))
.then(chestBlock => chestBlock ? bot.chat('Chest is opened!') : _throw('Chest is not opened!'))

// What items are in it?
locateBlock(bot, 'chest', 1)
.then(chestBlock => listItemsInChest(bot, chestBlock))
.then(response => bot.chat('Looks like ' + response))

// Craft a furnace
getIngredients(bot, 'furnace')
.then(ingredients => createQueryPrompt(bot, ingredients, 'Craft a furnace'))
.then(queryPrompt => model.getCompletion(queryPrompt))
.then(completion => evaluateCode(completion, true))
.then(_ => craftItem(bot, 'furnace', 1))
.then(_ => equipItem(bot, 'furnace'))
.then(success => success ? bot.chat("I made a furnace!") : _throw("I couldn't make the furnace"));
```

Figure 2: Sample prompt given to the Codex model to power an NPC in Minecraft.

is an open-world game where players creatively build artifacts in the environment. This makes Minecraft a good use case for providing NPCs that can converse and perform tasks for the player, something that Minecraft currently does not do. Second, Minecraft has rich game APIs in scripting languages,[2] that permit models to write function calls that allow the NPC perform in-game actions.

We investigate these Codex-powered NPCs through an exploratory user study. We ask experienced gamers to interact with the NPC to accomplish tasks in a Minecraft realm: obtain crafting recipes, mine resources, craft items and, lastly, break out of two escape rooms. Figure 1 shows two sample interactions. We analyze these interactions, and discuss fail cases and what modifications might be needed to handle them. We also present discussion of some interesting avenues of future research in the gaming space that might be achieved by fine-tuning on game APIs.

[2]We use the open-source Mineflayer API. Microsoft recently released a first-party API with similar functionality in its GameTest Framework SimulatedPlayer class. This is still under development and was not available for us at the time we conducted our experiments.

2 Related Work

The Minecraft gaming environment is increasingly widely used as a platform for researching agents and machine-human collaboration. MALMO (Johnson et al., 2016) is a test-bed for machine learning architectures trained on reinforcement learning. (Rose, 2014) showcases dialog in which players provide NPCs with information and the NPCs retain episodic memory and identify player's sentiments. (Szlam et al., 2019) lays out the motivation for building assistants in Minecraft. (Gray et al., 2019) describes a framework for dialog-enabled interactive agents using high-level, hand-written composable actions. (Jayannavar et al., 2020) study collaborative conversation between a builder and an architect about structure building. IGLU: Interactive Grounded Language Understanding in a Collaborative Environment has emerged a competition to explore interactions in a Minecraft environment. (Kiseleva et al., 2021).

The model we explore here is distinct from the previous Minecraft-related work in that it generates novel code that allows the NPC 1) to perform contextually viable actions (moving around, mining, crafting, etc), 2) to answer questions about structured Minecraft data (such as crafting recipes) and

3) to engage in multi-turn conversations.

This richness does not emerge in a vacuum: it draws on several convergent lines of research. Large pre-trained language models (PLMs) such as GPT-2 (Radford et al., 2019), GPT-3 (Brown et al., 2020) and GPT-J have become the predominant paradigm for text generation. Research in neural modelling of dialog has focused on powerful new models derived from these, such as DialoGPT(Zhang et al., 2020), Meena (Adiwardana et al., 2020), PLATO-XL(Bao et al., 2021), and LaMDA (Thoppilan et al., 2022) that offer rich potential for open-ended conversational applications.

The application of new prompting functions to large PLMs enables them to perform few-shot or zero-shot learning to adapt to new scenarios with little or no data (Liu et al., 2021). This approach, too, is rapidly being mainstreamed in dialog generation. (Madotto et al., 2021) employ prompting to select different dialogue skills, access multiple knowledge sources, generate human-like responses, and track user preferences. (Zheng and Huang, 2021) use prompt-based few-shot learning for grounded dialog generation, in an approach similar to ours.

PLMs that have been tuned on code repositories, typically GitHub, are have begun to be used to automate coding processes and generate code according to programmer's textual specifications, e.g., (Chen et al., 2021) and PaLm-Coder (Chowdhery et al., 2022). (Shin and Durme, 2021) suggest that models pre-trained on code may also benefit semantic parsing for natural language understanding. (Nijkamp et al., 2022) explore conversational program synthesis within this framework, and is close in spirit to the current work by virtue of its focus on emergent conversational properties.

3 Methodology

Our model is based on few-shot prompting of a large language model, in which a small number of sample instances in the prompt generalize to new unseen input (Brown et al., 2020). We use Codex (the code-davinci-002 model) and the Mineflayer API, together with MineCraft (Java Edition v.1.17.1). Our goal is to have the NPC respond to the player's input appropriately according to whether the input requires a purely natural language response or a call to a function to perform some action. Figure 2 shows a section of the prompt we provide to the model. The prompt begins with the following statement: "*This file contains Minecraft bot commands and the code needed to accomplish them using the Mineflayer JavaScript library. If asked something conversational, the bot should use bot.chat() to answer.*" This tells the model that the prompt includes natural language commands and the code needed to accomplish them. We include in the prompt the natural language commands and the code that need to be generated to enable basic NPC functionalities[3].

A new command from the player is appended to this seed prompt and sent to the Codex model. In the abstracted code, we evaluate the generated completion. When the completion includes a function call to the game API, the corresponding action is performed by the NPC inside the game. When it includes a call to the bot.chat() function, (discussed below) the response string is displayed on the chat interface. For each subsequent input, the prompt includes the seed prompt plus the previous player commands and model completions. When the prompt exceeds the allowed token limit (2048 tokens), we revert to the seed prompt and report to the player that the context has been reset.

We further refine this prompting approach using the following strategies:

Using a stop sequence: Since we want only to generate NPC responses (and not an entire conversation), we use a stop sequence (comment operator). Player input always starts with the stop sequence.

Syntactic sugaring: The Mineflayer API contains lower-level functions that might be hard to map to a natural language command. We therefore wrap it in more abstract code[4] to be handled by the Codex model, e.g., the functions locateBlock, openChest and listItemsInChest in Figure 2.

Using the bot.chat() function: We use the chat interface within the Minecraft game for interaction between player and NPC. The model calls the bot.chat() function whenever the NPC needs to respond using natural language.

Function chaining: Player instructions may require the NPC to perform multiple actions, in particular, map to multiple function calls where subsequent calls depend on the success or failure of previous calls. In Figure 2, the instruction "open the chest" triggers a chain of functions where the bot first locates the chest, opens it, then finally responds with the result.

[3] A full list of these functions is provided in Appendix A

[4] We release the wrapper code in this GitHub repository https://github.com/microsoft/interactive-minecraft-npcs

```
`// This script will answer questions given an input recipe and inventory. The script must make sure that all items
// required for a recipe are present in the inventory, otherwise it will say it is missing items.
// The script must make sure to ignore extraneous items in the inventory when answering questions.

// recipe = {}
// inventory = [{"name":"water_bucket","count":5}]
// What is in your inventory
result = "I have 5 water buckets in my inventory"
// Justification: There are 5 water buckets in the inventory.

// recipe = {}
// inventory = [{"name":"dirt","count":5}]
// Do you have any cobblestones in your inventory
result = "No, I dont have any cobblestones in my inventory"
// Justification: There are no cobblestone items in the inventory.

// recipe = {"output":"iron_helmet",amount:1,ingredients:[{"name":"iron_ingots","count":7}]}
// inventory = [{"name":"iron_sword","count":1}]
// Can you craft an iron helmet
result = "No, I don't have any iron ingots in my inventory"
// Justification: The script should ignore extraneous items in the inventory, there are no iron ingots in the inventory,
// but the recipe requires 7 iron ingots, so the script should say it is missing items.
```

Figure 3: Sample of the second prompt that gets called during question answering.

Autoregressive prompting: Also known as Retrieval-Augmented Generation (RAG) (Lewis et al., 2020). For prompts that require knowledge of game state, e.g., inventory/crafting queries, we create a call through Codex that first gathers the requisite information and then self-generates a call to itself with the needed information.The last command shown in Figure 2 responds to crafting questions by first obtaining an ingredient list, then calls createQueryPrompt to generate a secondary completion on a sub-prompt (Figure 3) using the data.

Figure 4: Participant interacting with the NPC to get the crafting recipes of different resources

4 User Study

We conduct a user study to evaluate our NPC. We invite eight participants who have previously played Minecraft. Each participant had an hour to complete the study and to answer the post study survey questions. The study consists of five parts:

4.1 Get Crafting Recipe

Participants interacted with the NPC to determine the ingredients and their count necessary to craft 5 items: wooden pickaxe, furnace, clock, pumpkin pie, and any resource of their choice.

Table 4 shows sample player inputs and generated code. We find that participants used different phrases (see table 1) to frame their questions and the NPC was usually able to correctly map these to the right function call. It was able handle minor variation in the resource name ('wood pickaxe' instead of 'wooden pickaxe'), misspellings ('fornace'), contextual phrasing ('what goes into a pumpkin pie' since its a food item), and non-question phrasing ('recipe for clock'). Some participants held extensive conversations with the NPC where they asked additional questions about the resource or its ingredients[5].

Overall, across all participants, excluding cases where the resource did not have a crafting recipe, the success rate for the 4 specified resources was 85%, whereas that for the 'any resource' category was 75%. The fail cases split into two categories: those where resources had a crafting recipe and those where resources did not. When a resource had a crafting recipe, all fail cases belonged to the 'natural language response instead of function call' category. This happened primarily when the query was preceded by a lengthy language-only conversation without calls to code. Table 2 includes fail cases for resources without a crafting recipe. In the first two cases, the NPC's response ("could not find recipe") seems appropriate but uninformative. In the third case the NPC hallucinates a recipe: *nether portal* is not an object that can be crafted.

[5]Analysis included in the Appendix.

Resource	User query	NPC Response
Wooden pickaxe	what do I need to create a wood pickaxe?	You need 3 oak planks and 2 sticks to make a wooden pickaxe.
Furnace	how can I build a fornace?	You need 8 cobblestones to make a furnace.
Pumpkin pie	what goes into a pumpkin pie?	You need 1 pumpkin, 1 egg and 1 sugar to make a pumpkin pie.
Clock	what items are needed to build a clock, yo?	You need 4 gold ingots and 1 redstone to make a clock.
Clock	recipe for clock	You need 4 gold ingots and 1 redstone to make a clock.
Beacon	what do I need for a beacon?	You need 5 glass, 1 nether star, and 3 obsidian to make a beacon.

Table 1: Different participant queries and NPC responses to get ingredients and count required to create resource

User utterance	NPC Response	Reason for failure
How do I make a sword?	Could not find recipe for sword	There are multiple types of sword.
how to craft saddle	Could not find recipe for saddle	You can't craft a saddle.
how to build nether portal	You need 3 dark oak planks to build a nether portal	Nether portal is not something you can craft.

Table 2: Participant queries that failed to generate the expected response from the NPC.

4.2 Mine Resource

Participants asked the NPC to mine 1 dirt and 4 oak logs. Table 3 shows sample interactions. The last column shows the code generated with each NPC response. In the first case, in response to the player command "can you get me 1 dirt?", the NPC checked whether it had dirt in its inventory. Since it didn't, it responded "I don't have any dirt in my inventory" and then mined the dirt for the player. However, after mining the dirt, it failed to give it to the player by dropping it near them. In the second case, when the user said "go cut down an oak log", the model called the function cutTree that did not exist. When the user followed up by being more Minecraft-specific ("mine the tree"), the model was able to generalize tree to "oak log". Additional example interactions and their analysis may be found in the appendix.

4.3 Craft Item

Participants asked the NPC to craft an oak plank, a crafting table and a chest. The third example in table 3 shows an interaction involving crafting an oak plank. When the player asked if the NPC can craft an oak plank, the NPC generated a chain of functions where it first obtained the ingredients of an oak plank and then created a query for the question answering prompt where it checked whether it had those ingredients in its inventory. Since it did not have them, it threw an error message indicating that it couldn't make the plank. The player followed up by asking what is needed to make the plank and the NPC responded with the recipe for making oak plank. The player then asked the NPC to get the oak log and make the plank. The NPC

Figure 5: Player instructing the NPC to craft items

then mined the oak log and crafted the oak plank. See the appendix for further examples.

4.4 Gather Minecraft knowledge

To get an idea what kinds of questions a typical Minecraft player might want to ask an NPC, we had participants ask freeform general knowledge questions of the NPC. Figure 6 shows a sample interaction. Participants tested the NPC's knowledge of of the game. They were interested in whether the NPC can get numerical facts correct, how much it knows about where to mine or find resources, whether can it answer comparative questions, does it know the purpose of different things in Minecraft, and does it have a personality of its own. More example queries are found in the appendix.

4.5 Escape from Two Rooms

Participants collaborated with the NPC to successfully escape from two rooms. In the first, both player and NPC had to stand on two separate pressure plates simultaneously to release a diamond pickaxe that the player could use to break out. Dif-

Player Utterance	NPC Response	Generated Code

can you get me 1 dirt?	No, I don't have any dirt in my inventory; I got your dirt	if(isInInventory(bot, 'dirt')) {
		goToPlayer(bot, username)
		.then(() => dropItem(bot, 'dirt'))
		.then(() => bot.chat("I got your dirt"))
		} else {
		bot.chat("No, I don't have any dirt in my inventory;")
		.then(() => mineBlock(bot, 'dirt'))
		.then(() => goToPlayer(bot, username))
		.then(() => bot.chat("I got your dirt"))
		}

go cut down an oak log		cutTree is not defined
mine the tree	Here you go!	mineBlock(bot, 'oak_log') .then(() => goToPlayer(bot, username)) .then(() => bot.chat("Here you go!"))
give me the oak log	Here you go!	dropItem(bot, 'oak_log') .then(() => goToPlayer(bot, username)) .then(() => bot.chat("Here you go!"))
can you craft an oak plank?	I'll craft you an oak plank! I couldn't make the oak plank.	getIngredients(bot, 'oak_planks') .then(ingredients => createQueryPrompt(bot, ingredients, 'Can you craft an oak plank')) .then(queryPrompt => model.getCompletion(queryPrompt)) .then(completion => evaluateCode(completion, true)) .then(() => craftItem(bot, 'oak_planks', 1)) .then(() => equipItem(bot, 'oak_planks')) .then(success => success ? bot.chat("I made an oak plank!") : _throw("I couldn't make the oak plank"));
can you get an oak log and then make the oak plank?	Here you go! I made an oak plank	getIngredients(bot, 'oak_planks') .then(ingredients => createQueryPrompt(bot, ingredients, 'Can you get an oak log and then make the oak plank')) .then(queryPrompt => model.getCompletion(queryPrompt)) .then(completion => evaluateCode(completion, true)) .then(() => mineBlock(bot, 'oak_log', 1)) .then(() => craftItem(bot, 'oak_planks', 1)) .then(() => equipItem(bot, 'oak_planks')) .then(success => success ? bot.chat("I made an oak plank!") : _throw("I couldn't make the oak plank"));

Table 3: Player and NPC interaction for mining 1 dirt, mining 4 oak logs and crafting an oak plank.

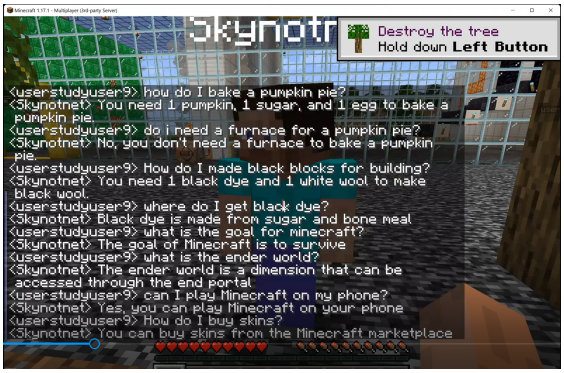

Figure 6: Player asking NPC questions about Minecraft

ferent participants used different strategies to get the NPC to stand on the plate. In most cases, participants used the phrase "stand on the pressure plate". In a few cases, they used phrases such as "stand on this square" and "stand on that purple tile" which failed because the NPC lacks the ability to map square or purple tile to the pressure plate. Similarly, the phrase "stand where I am looking" also failed because NPC lacks visual capabilities. All participants were able to escape the room, with the average time being 2 mins 30 secs.

In the second room, the player was trapped in a cobweb. A chest placed at a distance held the ingredients necessary to make an iron sword. To escape, participants needed to get the NPC to make the sword and give it to them. Table 4 shows a sample interaction between participant and NPC in this room. We found that the NPC is able to generate complex code (with novel function chains) during these interactions. Of the 8 participants, 7 successfully escaped with the help of the NPC,

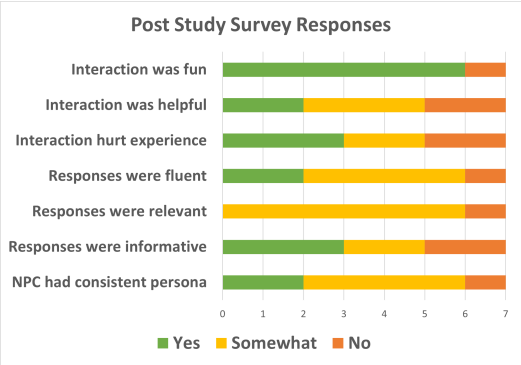

Figure 7: Participant responses for different questions in post study survey.

taking an average 4 mins 40 secs.

Upon finishing, if time permitted, participants were allowed to interact freely with the NPC.

4.6 Post Study Survey

Participants filled out a survey where they rated their overall experience interacting with the NPC. They were asked to judged the fluency, relevancy and informativeness of the NPC responses, compare this NPC experience with other in-game NPC experiences, and finally provide feedback on how this experience could be improved.[6]

Figure 7 shows the results of the survey. 6 out of the 7 participants[7] found the interaction with the NPC fun and said they would interact with it if it existed in a game, thus showing value in pursuing this direction further. Some thought the NPC was helpful and its responses were fluent and informative. Most thought the responses were only somewhat relevant and also often hurt the game experience. This suggests the need for further improving the proposed model and working on the shortcomings of prompt-based approach.

5 NPC Capabilities

On analyzing interactions in the user study, we find that the NPC exhibits the following capabilities:

Parse unseen commands: The NPC can understand player commands that are not in the prompt but correspond to an existing functionality. It can map the command 'make me a chest' (not in the prompt) to the right function calls and create the chest or explain why it can't.

Generalize to new functionality: For some low-level functionality, the NPC can generalize to unseen functions. For example, since the command 'move forward' is included in the prompt, the NPC knows how to call the right functions to move in other directions (backward, right and left).

Hold multi-turn conversation: The NPC can retain the context (both code and language) and maintain a multi-turn conversation in which the NPC both responds using natural language and takes actions within the game.

Generate language about code: The NPC can remember multi-turn context and answering questions about the code (and the language) generated in previous contexts. It can answer questions such as "What did you just do?" and "What directions have you moved?" .

Switch between code & language generation: Depending on the player command and the previous context, the NPC is able to automatically decide when to respond using natural language and when to generate a function call.

Question answering: The NPC is able to answer questions about its inventory (e.g. "what do u have in your inventory?", "do you have X?", "how many of X do you have" etc), about crafting recipes (e.g. "how can I make a chest?", "how many cobblestones do I need to make a furnace") and answer questions that require both inventory and crafting recipe information (e.g. "how many more cobblestones do I need to make a furnace?"). It can also answer questions generally about Minecraft. (The training data for GPT-3 includes Minecraft information available on the web.).

Generate novel function chains: Depending on the player command, the NPC is able to generate novel function chains by combining functions in an order unseen in the prompt. Table 3 includes multiple examples of such novel function chains.

6 Issues with Prompting

We also observed issues in our prompt-based approach. Many of are known issues in large language models, and more specifically in prompt engineering (Reynolds and McDonell, 2021; Liu et al., 2022) and longer conversations with agents in general (Xu et al., 2021). Principled solutions pose interesting avenues for future investigation.

Calling non-existent functions: In response to command, the NPC may attempt to call a function that does not exist in the API. For example, when the user asks the NPC to put a block down, the NPC calls the placeBlock function which is not in

[6] Appendix B contains the full list of questions.

[7] Only 7 of the 8 participants filled the survey.

Player Utterance	NPC Response	Generated Code
get what is on the chest	Looks like 64 iron ingot and 64 stick	locateBlock(bot, 'chest', 1) .then(chestBlock => listItemsInChest(bot, chestBlock)) .then(response => bot.chat('Looks like ' + response))
take everything		takeAllItemsFromChest is not defined
take the 64 iron ingots and the 64 sticks	I got your iron ingots and sticks!	locateBlock(bot, 'chest', 1) .then(chestBlock => getItemFromChest(bot, chestBlock, 'iron_ingot', 64)) .then(chestBlock => getItemFromChest(bot, chestBlock, 'stick', 64)) .then(success => success ? bot.chat("I got your iron ingots and sticks!") : _throw("I couldn't get your iron ingots and sticks!"))
craft an iron sword	OK I'll craft you an iron sword! I made an iron sword!.	getIngredients(bot, 'iron_sword') .then(ingredients => createQueryPrompt(bot, ingredients, 'craft an iron sword')) .then(queryPrompt => model.getCompletion(queryPrompt)) .then(completion => evaluateCode(completion, true)) .then(() => craftItem(bot, 'iron_sword', 1)) .then(() => equipItem(bot, 'iron_sword')) .then(success => success ? bot.chat("I made an iron sword!") : _throw("I couldn't make the iron sword"));
give me the iron sword	Here you go!	goToPlayer(bot, username) .then(() => dropItem(bot, 'iron_sword', 4)) .then(() => bot.chat("Here you go!"))

Table 4: Player utterance, NPC response and the code generated during the second escape room interaction.

the codebase. This might solved by providing the model with a list of existing functions (perhaps in the prompt), but a more principled solution may lie in fine-tuning the model on the game API itself.

Context exceeding prompt token limit: When the conversation exceeds the prompt's token limit, the prompt needs to be reset. This makes the NPC lose the context of the conversation. Instead of erasing the conversational history, it may be possible to prune irrelevant parts of the context to keep within the token limit. Some form of multi-stage prompting (Liu et al., 2022) may provide a solution.

Conversational response instead of function call: The NPC sometimes responds conversationally when the correct behavior would be to call a function. We observe that this happens when the player's command is preceded by a long language-only conversation thus priming the model for a language only response. On the other hand, if the preceding context includes function calls, then the same user command triggers the right function call. It may be difficult to fix this issue purely by prompt engineering. A better solution may be to fine-tune the Codex model on curated player NPC conversations that include by function calls.

Factual Inaccuracies: When the player asks general questions about Minecraft, the NPC gets the answer wrong. Table 11 includes instances of factual inaccuracies. A potential fix could be to incorporate a mechanism whereby the NPC can refer to an external knowledge source, e.g., as in retrieval-augmented methods (Lewis et al., 2020).

Inconsistencies: The NPC does not always have a consistent persona. In a few cases, it responds with a different answer for the same user query depending on the context, even when the question pertains to something that shouldn't change with the context. This could be addressed by enforcing a strategy wherein the NPC maintains its persona throughout the conversation; Again, multi-stage prompting (Liu et al., 2022) may help.

Repetition: The NPC starts repeating itself. This is especially likely when player and NPC engage in a long conversation that doesn't involve calls to code. In table 11, the player queries "what have you built?" and "have you built a house?", receive the same response: "I have built a lot of things". This may be addressable by metaprompt programming (Reynolds and McDonell, 2021) or multi-stage prompting (Liu et al., 2022).

Recency bias: The NPC can be biased by the most recent context and answers questions incorrectly. For example, if player has been conversing about things found in an ocean, and then asks "where is the best place to look for diamonds?", the NPC responds incorrectly "The best place to look for diamonds is in the ocean". Retrieval-augmented methods, e.g., (Lewis et al., 2020; Xu et al., 2021), may provide the needed factual grounding.

7 Conclusion

Codex-powered NPCs can integrate both conversational and task-oriented language interactions almost seamlessly with code generation in asset-rich contexts, and suggest huge potential for new kinds of gaming experience, including the generation of side quests (Appendix F). Gaming, moreover, is a rich sandbox-like environment for exploring complex agent interactions with code and addressing issues faced by large language models. The behavior of NPCs shed light many of the challenges encountered by large pretrained models of language and code in sustaining persona, goals, and intents over the course of interactions. It remains to be seen whether solutions can be found within existing training and tuning strategies or whether they must be sought outside these models. These are important, ongoing research questions, as are the huge challenges remaining in mapping these interactions to image recognition and to game state.

Ethical Considerations

The use of very large language models runs the risk of exposing users to offensive or sensitive language that might be contained in training data. Potential harms include, but are not limited to, offensive references to classes of people and beliefs, encouragement of violence outside the game, and socially inappropriate sexual references. Any implementation outside a sandboxed research environment will need to build guardrails appropriate to the audience and game environment, and especially to provide protections for minors. In addition, implementations must be able to handle adversarial probes designed to elicit offensive language.

A further concern is that this technology may make it easier for users to manipulate NPCs to perform in socially inappropriate ways or to construct socially inappropriate objects. Longer-term, the ability to enable users themselves to generate code that can affect game state may pose security threats.

Acknowledgments

We would like to thank the eight participants from the XBOX team at Microsoft who volunteered to the user study. Their insights and feedback helped us shape our analysis. We would also like to thank Ali Kazim-Zaidi, Vlad Naoumoff, Paige North, Richard Shin, Georgia Clarke, Karan Gulati and Haiyan Zhang who were the participants of a Hackathon at Microsoft that generated many of the ideas in this project. We also thank Nebojsa Jojic, Jorge Leandro, Brannon Zhand, Steve Latta and Mike Ammerlaan for all the useful discussions. We would also like to thank the two anonymous reviewers for their useful comments and suggestions.

References

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. *CoRR*, abs/2001.09977.

Siqi Bao, Huang He, Fan Wang, Hua Wu, Haifeng Wang, Wenquan Wu, Zhihua Wu, Zhen Guo, Hua Lu, Xinxian Huang, Xin Tian, Xinchao Xu, Yingzhan Lin, and Zhengyu Niu. 2021. PLATO-XL: exploring the large-scale pre-training of dialogue generation. *CoRR*, abs/2109.09519.

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*, volume 33, pages 1877–1901. Curran Associates, Inc.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison 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, Joshua 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. *CoRR*, abs/2107.03374.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton,

Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*.

Jonathan Gray, Kavya Srinet, Yacine Jernite, Haonan Yu, Zhuoyuan Chen, Demi Guo, Siddharth Goyal, C Lawrence Zitnick, and Arthur Szlam. 2019. Craftassist: A framework for dialogue-enabled interactive agents. *arXiv preprint arXiv:1907.08584*.

Prashant Jayannavar, Anjali Narayan-Chen, and Julia Hockenmaier. 2020. Learning to execute instructions in a Minecraft dialogue. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2589–2602, Online. Association for Computational Linguistics.

Matthew Johnson, Katja Hofmann, Tim Hutton, and David Bignell. 2016. The malmo platform for artificial intelligence experimentation. In *IJCAI*.

Julia Kiseleva, Ziming Li, Mohammad Aliannejadi, Shrestha Mohanty, Maartje ter Hoeve, Mikhail Burtsev, Alexey Skrynnik, Artem Zholus, Aleksandr Panov, Kavya Srinet, Arthur Szlam, Yuxuan Sun, Katja Hofmann, Michel Galley, and Ahmed Awadallah. 2021. Neurips 2021 competition iglu: Interactive grounded language understanding in a collaborative environment.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474. Curran Associates, Inc.

Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2021. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *CoRR*, abs/2107.13586.

Zihan Liu, Mostofa Patwary, Ryan Prenger, Shrimai Prabhumoye, Wei Ping, Mohammad Shoeybi, and Bryan Catanzaro. 2022. Multi-stage prompting for knowledgeable dialogue generation.

Andrea Madotto, Zhaojiang Lin, Genta Indra Winata, and Pascale Fung. 2021. Few-shot bot: Prompt-based learning for dialogue systems. *arXiv preprint arXiv:2110.08118*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A conversational paradigm for program synthesis. *arXiv preprint*.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners.

Laria Reynolds and Kyle McDonell. 2021. Prompt programming for large language models: Beyond the few-shot paradigm. In *CHI '21: CHI Conference on Human Factors in Computing Systems, Virtual Event / Yokohama Japan, May 8-13, 2021, Extended Abstracts*, pages 314:1–314:7. ACM.

Caroline M. Rose. 2014. Realistic dialogue engine for video games. *Electronic Thesis and Dissertation Repository*, 2652.

Richard Shin and Benjamin Van Durme. 2021. Few-shot semantic parsing with language models trained on code. *CoRR*, abs/2112.08696.

Arthur Szlam, Jonathan Gray, Kavya Srinet, Yacine Jernite, Armand Joulin, Gabriel Synnaeve, Douwe Kiela, Haonan Yu, Zhuoyuan Chen, Siddharth Goyal, et al. 2019. Why build an assistant in minecraft? *arXiv preprint arXiv:1907.09273*.

Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, YaGuang Li, Hongrae Lee, Huaixiu Steven Zheng, Amin Ghafouri, Marcelo Menegali, Yanping Huang, Maxim Krikun, Dmitry Lepikhin, James Qin, Dehao Chen, Yuanzhong Xu, Zhifeng Chen, Adam Roberts, Maarten Bosma, Yanqi Zhou, Chung-Ching Chang, Igor Krivokon, Will Rusch, Marc Pickett, Kathleen S. Meier-Hellstern, Meredith Ringel Morris, Tulsee Doshi, Renelito Delos Santos, Toju Duke, Johnny Soraker, Ben Zevenbergen, Vinodkumar Prabhakaran, Mark Diaz, Ben Hutchinson, Kristen Olson, Alejandra Molina, Erin Hoffman-John, Josh Lee, Lora Aroyo, Ravi Rajakumar, Alena Butryna, Matthew Lamm, Viktoriya Kuzmina, Joe Fenton, Aaron Cohen, Rachel Bernstein, Ray Kurzweil, Blaise Aguera-Arcas, Claire Cui, Marian Croak, Ed H. Chi, and Quoc Le. 2022. Lamda: Language models for dialog applications. *CoRR*, abs/2201.08239.

Jing Xu, Arthur Szlam, and Jason Weston. 2021. Beyond goldfish memory: Long-term open-domain conversation.

Yizhe Zhang, Siqi Sun, Michel Galley, Yen-Chun Chen, Chris Brockett, Xiang Gao, Jianfeng Gao, Jingjing Liu, and Bill Dolan. 2020. DIALOGPT : Large-scale generative pre-training for conversational response generation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 270–278, Online. Association for Computational Linguistics.

Chujie Zheng and Minlie Huang. 2021. Exploring prompt-based few-shot learning for grounded dialog generation. *arXiv preprint arXiv:2109.06513*.

A NPC functionalities

We include in the prompt the natural language commands and the code that should be generated to enable the following basic NPC functionalities:

- Move forward
- Jump
- Look at the player
- Come to the player
- Follow the player
- Locate a block
- Mine a block
- Get the crafting recipe of an item
- Craft an item
- Open a chest
- Take items from chest
- Put items into the chest
- Close the chest
- List all items in inventory
- Check if item is in the inventory
- Get count of item in inventory
- Give item to player

B Post user study survey questions

Following the study, the participants filled out a survey that had the following questions:

(a) How would you rate your skill in Minecraft?

(b) How would you rate your skill as a gamer in general?

(c) Did you have fun while interacting with the NPC in this study?

(d) How did you find this NPC interaction in comparison to the interactions you might have had with dialog-capable NPCs in other games (e.g, Skyrim)?

(e) How fluent were the NPC's responses?

(f) How relevant were the NPC's responses?

(g) How informative were the NPC's responses?

(h) Did you feel like you were interacting with an NPC with a consistent persona?

(i) When the NCP's response was incorrect, how much did that hurt your game experience?

(j) How magical did the experience of interacting with an NPC feel? If it didn't, can you explain why not?

(k) What could be done to improve the experience of interacting with the NPC?

(l) How useful were your interactions with this NPC in helping you better understand how to play Minecraft?

(m) More generally, would you prefer interacting with such an NPC to learn a new game, as opposed to a tutorial or an FAQ?

(n) Would you ever want to interact with an NPC like this generally in any game?

(o) What are some useful applications of this kind of NPC to enrich game experiences (in general, not only in Minecraft)?

(p) If this technology could reach the level of a companion (like an AI-gamer friend) that you could take to any game, would you want to use it?

C User study analysis

C.1 Get Crafting Recipe

In this part of the user study, some participants had a longer conversation with the NPC around the resource. Table 5 includes such an interaction around wooden pickaxe. The NPC is able to effectively switch between code calling and natural language response. In this particular example, the participant asked "can I only use oak to make a wooden pickaxe, why not pine?" to understand if the NPC knows that wooden pickaxes can be of various different types. Interestingly, the NPC first called the recipe function to respond that you need sticks in addition to oak planks to make the pickaxe. And then it generated a natural language response where it said you can use any wood to make the pickaxe. This shows the kind of complex behavior that can be generated by the codex model.

Table 6 is an example of a fail case for question answering. Although the initial query 'how do I make a furnace' correctly mapped to a recipe function call, the later query 'How do I make a clock' failed to do so since it was preceded with a length language-only conversation. The participant, however, was able to recover by asking a more specific question 'in terms of raw materials, what do I need to make a clock?'.

C.2 Mine Resource

Table 7 shows sample interactions for mining task. In the first case, the user's query was similar to that in table 3, however this time the NPC only checked for dirt in its inventory and when it didn't find any dirt, it said "I have no dirt". But it did not

Player utterance	NPC response
I want a wooden pickaxe	I don't have the ingredients needed for a wooden pickaxe! I couldn't make the wooden pickaxe
how do I make a wooden pickaxe	You need 3 oak planks and 2 sticks to make a wooden pickaxe.
where do I get planks?	You need to craft oak planks from oak logs. You need to chop down some oak trees.
how many logs do I need for 3 planks?	You need 3 logs to make 3 planks
does it have to be oak?	Yes, you will need 1 oak log to make 4 oak planks. Yes, you will need to use oak logs to make oak planks.
can I only use oak to make a wooden pickaxe, why not pine?	No, you can't make a wooden pickaxe with only oak planks, you need to use oak planks and sticks. You can use any wood to make a wooden pickaxe.

Table 5: Interaction between a participant and NPC around creation of a wooden pickaxe

Player utterance	NPC response
how do I make a furnace	You need 8 cobblestones to make a furnace.
where do I find cobblestone?	You can find cobblestone in the ground
does depth matter?	No, depth does not matter.
do I need anything to collect cobblestone?	You need a shovel to collect cobblestone
how do I pick up cobblestone?	You need to mine cobblestone
how do you mine cobblestone?	You need a pickaxe to mine cobblestone
How do I make a clock?	You need a clock to tell the time.
is there a way to make a clock	I think you can make a clock by combining a clock and a compass
in terms of raw materials, what do I need to make a clock?	You need 4 gold ingots and 1 redstone to make a clock.

Table 6: Sample interaction where the NPC failed to map user query to recipe function for a resource that has a crafting recipe. The user query preceded with lengthy context that required purely natural language response.

follow it with mining the dirt as it did in the first case. When the user explicitly used the term 'mine' in their next utterance, the NPC mined the dirt for them. This suggests that the model is not able to always generalize "bring me" to the mine action. In the second case, the player was more direct and said "mine dirt" and in response the NPC mined the dirt. Next the player said "drop dirt" and the NPC went to the player and dropped the dirt close to the player.

The third case is similar to the first. The notable difference is that the model is able to understand the compound instruction "can you mine some dirt and give it to me" and correctly generates the code to first mine and then drop the item near the user. The fourth case depicts an instance of fail case belonging to the 'natural language response instead of function call' category. Although this player used the same utterance as in the first case, the NPC's response was purely natural language. When the player followed this with "get me one dirt" and "i want one dirt", both times, the model tried to call a function getItem() that did not exist. After doing a reset of the prompt, however, the user was able to get the NPC to mine them the dirt.

The fifth case is a successful interaction around mining of oak logs. In additon to generating the correct code, the NPC's responses ("I'm chopping the oak logs" and "I dropped the oak logs") were customized to the player's phrasing ("please chop oak logs" and "drop oak logs"). The sixth case depicts a success case for mining of 4 oak logs. The last case depicts a failure case. when the user says "bring me some oak logs please", they were expecting the NPC to mine the logs and then give them to the user. However, the model only generated the code for mining the oak logs. When the user followed it up with a verbose utterance ("you are holding the logs, please throw them at my feed"), the model was not able to map this to dropping of the oak logs. Likewise, the phrase "pass me the logs" also did not map to dropping of the oak logs. Instead, in both these cases it generated a purely natural language response ("I am holding the oak logs").

C.3 Gather Minecraft Knowledge

Table 11 contains example queries where participants tested NPC's general knowledge about Minecraft. We group them by different aspects. In the first four aspects, the models gets several questions wrong, suggesting the need for the integration of Minecraft specific knowledge base into the model. The last aspect (personality of the NPC)

Figure 8: Clues for the first escape room: ‘I wonder what these pressure plates do?’, ‘I think we need to stand on both at the same time’.

Figure 9: Clues for the second escape room: ‘Maybe that chest has what I need to get out’, ‘I think I can get out if I get my hands on an Iron Sword’

suggests that users would like the NPC to have a persona and a history of its previous game experience.

D Details of Escape Rooms

Both rooms had clues written on the wall or on boards that helped users figure out the path to escape. Figures 8 and 9 shows the clues written on the wall/boards in the two rooms.

E Need for Visual Capabilities

During the user study, we found multiple instances where participants were expecting the NPC to be able to see the things in the games, just like a player does. Figure 10 shows one such instance where player pointed at a crafting table and asked the NPC “what is this”. The NPC responded “this is a chat” since it had only textual context. Figure 11 shows another instance where participant pointed at a location and told the NPC “place the crafting table here”. Lastly, during the escape room interaction, one participant looked at the pressure plate and told the NPC “stand where I am looking”. These cases strongly suggest the need for the NPC to have visual capabilities.

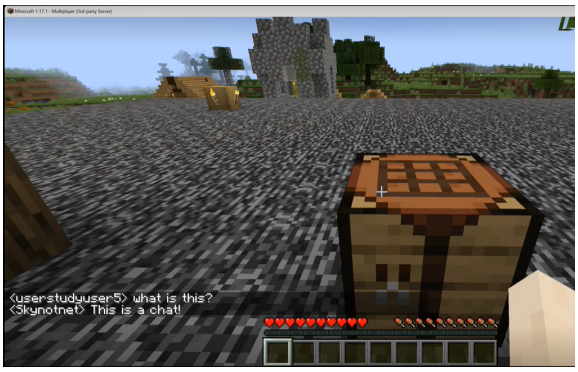

Figure 10: Player pointing at an object and asking NPC ‘what is this?’

Figure 11: Player pointing at a location and asking the NPC to place a block there.

Figure 12: Player asking the NPC to stand where they are looking.

F Side Quest Generation

We also experiment with the idea of side quest generation. Side quests represent a large amount of content that players engage with, they spend hours wandering worlds based on instructions given by NPCs. The more quests we can create automatically, the longer the player can spend exploring their favorite world. This technology is meant for open-world games like in Fallout, the OuterWorlds, or Skyrim. We design a separate prompt with few sample quests. When the player asks for a quest, the NPC generates a new quest where it requests one or more items from the player and devises a relevant backstory as to why the requested items are needed.The NPC then checks whether or not the player has completed the quest, and provides a reward if so. The necessary code to execute this logic is generated by the Codex model.

1. For example, if the player asks, "Do you have a quest for me?". User (U) says hello to Quest-Giver (Q), which is a prompt-based model that is given some summary of the state of the game (including items that exist in the game).

2. Q specifies in response: (a) backstory; (b) reward; (c) quests; and (d) code logic that checks for quest completion. Quests are items to retrieve. An example of a quest backstory would be: "The recent festival was wonderful, but I fear my children were just a bit too rambunctious. I hear there is a new potion for sale that would slow down anyone who drinks it. I would like to try it out on my kids. Please get me some", in reference to a slowness potion.

3. U gets an item X

4. U says to Q some paraphrase of "I got the X that you asked for", this triggers code that results in U /give X to Q, followed by Q inspecting inventory to verify it was given X and that it previously asked for X

5. If all quests completed, Q says "here is your reward", with an associated /give to U

Player Utterance	NPC Response	Generated Code

Please bring me 1 dirt	I have no dirt	if(isInInventory(bot, ‘dirt’)) {
		goToPlayer(bot, username)
		.then(() => dropItem(bot, ‘dirt’))
		.then(() => bot.chat(“Here you go!"))
		} else {
		bot.chat(“I have no dirt");
		}

please mine 1 dirt for me	Here you go!	mineBlock(bot, ‘dirt’) .then(() => goToPlayer(bot, username)) .then(() => bot.chat(“Here you go!"))
mine dirt	I mined the dirt!	mineBlock(bot, ‘dirt’) .then(() => goToPlayer(bot, username)) .then(() => bot.chat(“I mined the dirt"))
drop dirt	I dropped the dirt!	goToPlayer(bot, username) .then(() => dropItem(bot, ‘dirt’)) .then(() => bot.chat(“I dropped the dirt!"))

can you get me a piece of dirt	I have no dirt	if(isInInventory(bot, ‘dirt’)) {
		goToPlayer(bot, username)
		.then(() => dropItem(bot, ‘dirt’))
		.then(() => bot.chat(“Here you go!"))
		} else {
		bot.chat(“I have no dirt");
		}

can you mine some dirt and give it to me	Here you go!	mineBlock(bot, ‘dirt’) .then(() => goToPlayer(bot, username)) .then(() => dropItem(bot, ‘dirt’)) .then(() => bot.chat(“Here you go!"))
can you get me one dirt?	I can get you some dirt!	
get me one dirt		getItem is not defined
i want one dirt		getItem is not defined
reset		

bring me one dirt	I have no dirt	if(isInInventory(bot, ‘dirt’)) {
		goToPlayer(bot, username)
		.then(() => dropItem(bot, ‘dirt’))
		.then(() => bot.chat(“Here you go!"))
		} else {
		bot.chat(“I have no dirt");
		}

mine one dirt	I got your dirt	mineBlock(bot, ‘dirt’) .then(() => goToPlayer(bot, username)) .then(() => bot.chat(“I got your dirt"))
please chop oak logs	I’m chopping the oak logs	mineBlock(bot, ‘oak_log’) .then(() => bot.chat(“I’m chopping the oak logs"))
drop oak logs	I dropped the oak logs	dropItem(bot, ‘oak_log’) .then(() => goToPlayer(bot, username)) .then(() => bot.chat(“I dropped the oak logs"))
I need 4 oak logs. Can you help?	Here you go!	mineBlock(bot, ‘oak_log’, 4) .then(() => goToPlayer(bot, username)) .then(() => bot.chat(“Here you go!"))
please give me the oak logs	Here you go!	dropItem(bot, ‘oak_log’, 4) .then(() => goToPlayer(bot, username)) .then(() => bot.chat(“Here you go!"))
bring me some oak logs please	I got your oak logs	mineBlock(bot, ‘oak_log’) .then(() => goToPlayer(bot, username)) .then(() => bot.chat(“I got your oak logs"))
you are holding the logs,	I am holding the oak logs	
please throw them at my feet		pass me the logs
I am holding the oak logs		

Table 7: Player utterance, NPC response and generated code during mining interaction.

Player Utterance	NPC Response	Generated Code
craft an oak plank	OK, I’ll craft you an oak plank! I made an oak plank	getIngredients(bot, ’oak_planks’) .then(ingredients => createQueryPrompt(bot, ingredients, ’Craft an oak plank’)) .then(queryPrompt => model.getCompletion(queryPrompt)) .then(completion => evaluateCode(completion, true)) .then(() => craftItem(bot, ’oak_planks’, 1)) .then(() => equipItem(bot, ’oak_planks’)) .then(success => success ? bot.chat("I made an oak plank!") : _throw("I couldn’t make the oak plank"));
give me the oak plank	Here you go!	goToPlayer(bot, 3, username) .then(success => success ? dropItem(bot, ’oak_planks’, 1) : _throw("I couldn’t drop the oak plank")) .then(success => success && bot.chat("Here you go!"));
how do you make an oak plank?	You need 1 oak log to make 1 oak plank. I made an oak plank!	getIngredients(bot, ’oak_planks’) .then(ingredients => createQueryPrompt(bot, ingredients, ’How do you make an oak plank’)) .then(queryPrompt => model.getCompletion(queryPrompt)) .then(completion => evaluateCode(completion, true)) .then(() => craftItem(bot, ’oak_planks’, 1)) .then(() => equipItem(bot, ’oak_planks’)) .then(success => success ? bot.chat("I made oak planks!") : _throw("I couldn’t make the oak plank"));
make me an oak plank	I’ll make some oak planks! I couldn’t make the oak plank	getIngredients(bot, ’oak_planks’) .then(ingredients => createQueryPrompt(bot, ingredients, ’Make an oak plank’)) .then(queryPrompt => model.getCompletion(queryPrompt)) .then(completion => evaluateCode(completion, true)) .then(() => craftItem(bot, ’oak_planks’, 1)) .then(() => equipItem(bot, ’oak_planks’)) .then(success => success ? bot.chat("I made oak planks!") : _throw("I couldn’t make the oak plank"));
<user dropped 1 oak log>	<NPC picked it up the oak log>	
make me an oak plank	OK, I’ll make you an oak plank. I made oak planks!	getIngredients(bot, ’oak_planks’) .then(ingredients => createQueryPrompt(bot, ingredients, ’Make an oak plank’)) .then(queryPrompt => model.getCompletion(queryPrompt)) .then(completion => evaluateCode(completion, true)) .then(() => craftItem(bot, ’oak_planks’, 1)) .then(() => equipItem(bot, ’oak_planks’)) .then(success => success ? bot.chat("I made oak planks!") : _throw("I couldn’t make the oak plank"));

Table 8: Player utterance, NPC response and generated code for interaction around crafting an oak plank.

User Utterance	NPC Response	Generated Code

build a crafting table	I'll build you a crafting table.
	I built a crafting table
		getIngredients(bot, 'crafting_table')
		.then(ingredients =>
		  createQueryPrompt(bot, ingredients,
		  'Build a crafting table'))
		.then(queryPrompt =>
		  model.getCompletion(queryPrompt))
		.then(completion =>
		  evaluateCode(completion, true))
		.then(() => craftItem(bot, 'crafting_table', 1))
		.then(() => equipItem(bot, 'crafting_table'))
		.then(success => success ?
		  bot.chat("I built a crafting table!") :
		  _throw("I couldn't build a crafting table"))

please make a crafting tabel	OK, I'll make you a crafting table.
	I couldn't make the crafting table.
		getIngredients(bot, 'crafting_table')
		.then(ingredients =>
		  createQueryPrompt(bot, ingredients,
		  'Please make a crafting table'))
		.then(queryPrompt =>
		  model.getCompletion(queryPrompt))
		.then(completion =>
		  evaluateCode(completion, true))
		.then(() => craftItem(bot, 'crafting_table', 1))
		.then(() => equipItem(bot, 'crafting_table'))
		.then(success => success ?
		  bot.chat("I built a crafting table!") :
		  _throw("I couldn't build a crafting table"))

what do you need to make
a crafting table?
	You need 4 oak planks to make
	a crafting table.
		getIngredients(bot, 'crafting_table')
		.then(ingredients =>
		  createQueryPrompt(bot, ingredients,
		  'What do you need to make a crafting table'))
		.then(queryPrompt =>
		  model.getCompletion(queryPrompt))
		.then(completion =>
		  evaluateCode(completion, true))

make me 4 oak planks please	OK, I'll make you 4 oak planks!
	I made 4 oak planks.
		getIngredients(bot, 'oak_planks')
		.then(ingredients =>
		  createQueryPrompt(bot, ingredients,
		  'Make me 4 oak planks'))
		.then(queryPrompt =>
		  model.getCompletion(queryPrompt))
		.then(completion =>
		  evaluateCode(completion, true))
		.then(() => craftItem(bot, 'oak_planks', 4))
		.then(() => equipItem(bot, 'oak_planks'))
		.then(success => success ?
		  bot.chat("I made 4 oak planks!") :
		  _throw("I couldn't make the 4 oak planks"))

what resources do you have now?	I have 16 oak planks in
	my inventory
		createQueryPrompt(bot, ,
		  'What resources do you have now')
		.then(queryPrompt =>
		  model.getCompletion(queryPrompt))
		.then(completion =>
		  evaluateCode(completion, true))

please craft a crafting table	OK, I'll craft you a crafting table!
	I made a crafting table.
		getIngredients(bot, 'crafting_table')
		.then(ingredients =>
		  createQueryPrompt(bot, ingredients,
		  'Craft a crafting table'))
		.then(queryPrompt =>
		  model.getCompletion(queryPrompt))
		.then(completion =>
		  evaluateCode(completion, true))
		.then(() => craftItem(bot, 'crafting_table', 1))
		.then(() => equipItem(bot, 'crafting_table'))
		.then(success => success ?
		  bot.chat("I made a crafting table!") :
		  _throw("I couldn't make the crafting table"));

Table 9: Player utterance, NPC response and generated code for interaction around crafting a crafting table.

User Utterance	NPC Response	Generated Code

how do you make a chest?	You need 8 oak planks to make a chest.
	I couldn't make the chest.
		getIngredients(bot, 'chest')
		.then(ingredients =>
		  createQueryPrompt(bot, ingredients,
		  'How do you make a chest'))
		.then(queryPrompt =>
		  model.getCompletion(queryPrompt))
		.then(completion =>
		  evaluateCode(completion, true))

make oak planks	OK, I'll make you some oak planks!
	I couldn't make the oak planks
		getIngredients(bot, 'oak_planks')
		.then(ingredients =>
		  createQueryPrompt(bot, ingredients,
		  'Make oak planks'))
		.then(queryPrompt =>
		  model.getCompletion(queryPrompt))
		.then(completion =>
		  evaluateCode(completion, true))
		.then(() => craftItem(bot, 'oak_planks', 8))
		.then(() => equipItem(bot, 'oak_planks'))
		.then(success => success ?
		  bot.chat("I made oak planks!") :
		  _throw("I couldn't make the oak planks"))

chop oak	I chopped 4 oak logs!	locateBlock(bot, 'oak_log', 0)
		.then(success => success ?
		  chopTree(bot, 'oak_log', 4) :
		  _throw("I couldn't find any oak logs!"))
		.then(success => success
		  && bot.chat("I chopped 4 oak logs!"));

make oak planks	I made oak planks!	getIngredients(bot, 'oak_planks')
		.then(ingredients =>
		  createQueryPrompt(bot, ingredients,
		  'Make oak planks'))
		.then(queryPrompt =>
		  model.getCompletion(queryPrompt))
		.then(completion =>
		  evaluateCode(completion, true))
		.then(() => craftItem(bot, 'oak_planks', 8))
		.then(() => equipItem(bot, 'oak_planks'))
		.then(success => success ?
		  bot.chat("I made oak planks!") :
		  _throw("I couldn't make the oak planks"))

how many oak planks
 	
do you have?
	I have 12 oak planks in my inventory	createQueryPrompt(bot, ,
		  'How many oak planks do you have')
		.then(queryPrompt =>
		  model.getCompletion(queryPrompt))
		.then(completion =>
		  evaluateCode(completion, true))

make a chest	I made a chest!	getIngredients(bot, 'chest')
		.then(ingredients =>
		  createQueryPrompt(bot, ingredients,
		  'Make a chest'))
		.then(queryPrompt =>
		  model.getCompletion(queryPrompt))
		.then(completion =>
		  evaluateCode(completion, true))
		.then(() => craftItem(bot, 'chest', 1))
		.then(() => equipItem(bot, 'chest'))
		.then(success => success ?
		  bot.chat("I made a chest!") :
		  _throw("I couldn't make the chest"))

place chest	placeBlock is not a function	locateBlock(bot, 'chest', 1)
		.then(chestBlock =>
		  placeBlock(bot, chestBlock))
		.then(success => success ?
		  bot.chat("I placed the chest!") :
		  _throw("I couldn't place the chest")) height

Table 10: Player utterance, NPC response and generated code for interaction around crafting a chest.

User utterance	**NPC response**
Numerical facts	
how much hunger does steak restore	Steak restores 4 hunger
how much damage does a creeper do?	A creeper does 5 damage
how many mobs spawn in a raid?	There are about 20 mobs that spawn in a raid!
what is the build limit?	The build limit is 256 blocks!
how long is a Minecraft day?	A day in Minecraft is 20 minutes.
how many different potions are there?	There are currently 16 different potions.
Location of things	
where can I find sponges?	You can find sponges in the ocean!
where is the best place to look for diamonds?	The best place to look for diamonds is in the ocean
where is the best place to look for coal?	The best place to look for diamonds is in the ocean
where do I find a nether star?	A nether star is dropped by the Wither.
Where do I find zombies?	Zombies are in the nether!
Where can I find diamond	Diamond can be found in the ground
Where do I find pigs?	Pigs are in the forest!
Comparisons	
what has longer pulse, a stone or a wooden button?	A stone has a longer pulse!
which is worse, a creeper or a the dragon?	The dragon is worse than the creeper
who would win between a skeleton and a zombie?	The zombie would win!
Purpose of things	
What is redstone used for?	Redstone is used to power things like furnaces and redstone torches
What does an oven do?	An oven is uded to cook things like bread and cake
what does a tree drop	A tree drops oak logs!
what does the ground drop?	The ground drops dirt!
what is a nether star used for?	A nether star is used to cure a zombie villager.
Personality of the NPC	
are you an NPC?	I'm not an NPC, I'm a bot!
what do you like to do?	I like to build things!
what have you built?	I have built a lot of things
have you built a house?	I have built a lot of things
where did you get your name?	I was named after the bot in the movie 'Ace Ventura: Pet Detective'
What do you like to do in Minecraft?	I like to make things in Minecraft!
What kinds of things do you like to make?	I like to make things like furnaces, pickaxes, and crafting tables!

Table 11: Users testing NPC's general knowledge about Minecraft.