Alchemist: LLM-Aided End-User Development of Robot Applications

Ulas Berk Karli  
ulasberk.karli@yale.edu  
Yale University  
New Haven, CT, USA

Juo-Tung Chen  
jchen396@jhu.edu  
Johns Hopkins University  
Baltimore, MD, USA

Victor Nikhil Antony  
vantony1@jhu.edu  
Johns Hopkins University  
Baltimore, MD, USA

Chien-Ming Huang  
cmhuang@cs.jhu.edu  
Johns Hopkins University  
Baltimore, MD, USA

ABSTRACT

Large Language Models (LLMs) have the potential to catalyze a paradigm shift in end-user robot programming—moving from the conventional process of user specifying programming logic to an iterative, collaborative process in which the user specifies desired program outcomes while LLM produces detailed specifications. We introduce a novel integrated development system, Alchemist, that leverages LLMs to empower end-users in creating, testing, and running robot programs using natural language inputs, aiming to reduce the required knowledge for developing robot applications.

We present a detailed examination of our system design and provide an exploratory study involving true end-users to assess capabilities, usability, and limitations of our system. Through the design, development, and evaluation of our system, we derive a set of lessons learned from the use of LLMs in robot programming. We discuss how LLMs may be the next frontier for democratizing end-user development of robot applications.

CCS CONCEPTS

• Human-centered computing. • Computer systems organization → Robotics;

KEYWORDS

robot programming, end-user development, code generation
ACM Reference Format:

1 INTRODUCTION
Robots have the potential to significantly boost productivity across diverse fields from factory floors to research laboratories. Yet, deploying robots can be complex and costly, often requiring teams of experts for system initialization and task-specific programming. Consider a scenario where a team of researchers is striving to enhance the efficiency of photocatalysts for hydrogen production from water, a pursuit often described as the “holy grail” of chemistry [48]. This endeavor may require extensive experimental exploration of a diverse set of candidates and configurations; for instance, evaluating a search space with ten variables could present approximately 98 million potential candidates. Even with optimization strategies applied to narrow down this search space, as many as 688 experiments might still be required to identify superior photocatalyst mixtures. Robotic chemist [10] performed these experiments and exemplified how robots can automate the repetitive, low-level testing procedures, liberating scientists to focus on high-level tasks.

Nevertheless, robots deployed in such specialized roles are often laboriously developed for specific tasks and environments, posing a barrier to the widespread integration of robots in complex and dynamic workflows such as life science laboratories. Unlike computers, which can be readily acquired and customized with various software applications, the accessibility and customizability of robotic platforms and programs by end-users, such as scientists, remain elusive. To democratize the use of robots, several end-user programming systems and frameworks have been proposed [2]. Approaches for lowering barriers to programming robots include utilizing behaviour trees [38], illustration-based systems [40, 41], flow-based systems [5], programming by demonstration [3, 24], block-based systems [43] and a mix of methods that utilize high-level coding abstractions to program robots [23]. However, these abstractions still require an understanding and explicit specification of the programming logic which presents a significant barrier.

Large Language Models (LLMs), with their ability to generate code from natural language inputs [17], could help catalyze a paradigm shift in end-user robot programming from the conventional process of user specifying programming logic to an iterative, collaborative process in which the user specifies desired program outcomes while LLM produces detailed specifications. This paradigm shift conceptually regards end-user programming systems as collaborators rather than tools. By enabling a more intuitive and collaborative robot programming experience for end-users, LLMs may help lower barriers to the deployment and effective use of robots in diverse settings.

To explore the potential of using LLMs for end-user robot programming, we developed Alchemist, an end-to-end system that aims to streamline the complex process of robot programming by empowering users to create, debug, test, and execute robot programs using natural language dialog through a one-stop interface. Alchemist integrates RViz for robot visualization, a chat-box for interacting with the LLM, and a terminal to run generated code (Fig. 1). It is designed to be robot-platform and LLM agnostic to support various settings and technical advancements. It further supports authoring robot applications that involve automated processes (e.g., robotic chemist) and human-robot interaction scenarios.

This work makes three key contributions: (1) An open-source, end-to-end system that utilizes LLMs to enable a collaborative and intuitive robot programming experience for end-users. (2) An exploratory study to test and understand system capabilities and usability. (3) A set of lessons learned to inform the design and development of future LLM-powered robot programming systems.

2 RELATED WORK
Recent strides in LLMs and their code generation capabilities provide an opportunity for developing a new end-user robot programming paradigm. In this section, we summarize prior work and identify gaps in end-user robot programming systems, LLM-enabled code generation and LLMs for robot programming.

2.1 End-User Robot Programming Systems
Robot programming traditionally demands domain-specific expertise [22, 45], posing a barrier for the general population. Several different modes of programming have been explored to enable end-user robot programming [2], aiming to democratize the utilization of robots. A common mode is visual programming including block based programming [4, 15, 23, 53, 59], flow diagrams [7, 19], rule based programming [33], and behaviour trees [8, 39]. Mixed reality-based end-user programming systems have also been developed and evaluated [21, 27, 35]. Apart from these, a hybrid approach utilizing natural language chat and block based programming has been shown to be possible [20]. These existing approaches, while valuable, often still require end-users to possess some knowledge of programming logic, limiting their accessibility to the broader population.

The rise of large language models (LLMs) presents an unique opportunity to address the limitation posed by programming logic complexity; by leveraging the capabilities of LLMs, this paper explores how natural language can serve as an intuitive means for individuals to interact with and program robots.

2.2 LLM-enabled Code Generation
As LLMs code generation capabilities have improved significantly, there has been a shift in utilizing them as programming assistants such as GitHub Copilot [17]. Unlike these coding assistants which assist in the process of writing code, our system, Alchemist, generates code autonomously as it is designed for users with minimal coding skills. Additionally, Alchemist is tailored specifically for robotics applications. LLMs such as GPT-3 exhibit substantial code generation capabilities [13, 14]; however, there are notable limitations [44, 46]. Usability studies reveal that while LLM-based tools can provide helpful starting points, understanding, editing, and debugging the generated code can be challenging for programmers [50]. Furthermore, evaluations underscore concerns about functional correctness, suggesting that not all code generated by LLMs is error-free [30, 37]. Therefore, there is a need to identify common mistakes made by LLMs [12] and develop a general approach to

rectify these errors effectively. To address errors in generated code, Alchemist includes quality assurance methods (Section 3.3.3).

### 2.3 LLMs for Robot Programming

The incorporation of LLMs into robot programming has recently gained considerable attention, opening up possibilities for enhancing human-robot interactions. Generating code from natural language descriptions facilitates the control of robots through human-readable instructions. These capabilities have the potential to streamline robot programming. For instance, ChatGPT has been used to generate code for controlling robots using natural language instructions in a zero-shot fashion [51]; however, one limitation of this work is the absence of an integrated code editor and visualization panel, which makes it challenging for users to directly interact with the system and debug errors when they occur. Other studies have also investigated the application of pre-trained LLMs in robotic task planning [16, 25, 49, 55] and reasoning [1, 18, 28, 56]. While these studies demonstrate the potential of LLMs in robotics, they do not provide a straightforward means for end-users to interact with robots or troubleshoot issues in case of errors or unintended actions. In our system, we address these limitations by providing an end-to-end, one-stop system that incorporates a visualization panel and an embedded code editor to facilitate iterative and collaborative interaction between the user, the LLM, and the robot.

Perhaps the work by Inagaki et al. [26] is closest to ours; it explores the integration of LLMs in robot programming, particularly focusing on its application in automating biological laboratories. Although it showcases the potential of using LLMs to enable individuals with limited robotics experience to program robots, it does not fully address the challenge of preventing undesired actions or errors in the code generated by LLMs. Our system addresses this challenge by employing grounded prompting and a code verification process to enhance the code quality.

### 3 ALCHEMIST: LLM-POWERED END-USER ROBOT PROGRAMMING SYSTEM

**Alchemist** is an open-source, end-to-end system that empowers users to create, edit, and test programs for robots through natural language-based dialog by using large language models as its code generation backbone. We outline our core design objectives (DO) for Alchemist in Table 1 and present an overview of the system followed by details on system implementation in this section.

#### 3.1 System Overview

Using Alchemist, users initiate the robot programming process by placing AR markers to identify objects of interest in robot’s workspace. These markers enable the system to track, update, and visualize the robot’s world model. Subsequently, users prompt the LLM to generate code via the Chat Panel. They can then edit and debug the generated programs, either by prompting the LLM further or by utilizing the Editor. Once satisfied, users can preview, execute and test the programs via the Terminal Panel.

Alchemist has two distinct task-level capabilities:

1. **Automation**: Alchemist allows users to automate entire processes (see Fig. 2-a). This can be done by either breaking tasks into smaller sub-tasks and then creating and integrating individual code segments for each sub-task or by devising a single comprehensive program for the entire task; this choice depends on user preference and task complexity.

2. **Collaboration**: Alchemist allows users to program the robot to be collaborative in a way that they can verbally instruct the system to execute specific actions, aiding them in task completion or they can specify how or when they would need the system to do certain actions in response to user actions or changes in environment (see Fig. 2-b).

#### 3.2 Front-End Components

We designed Alchemist’s interface (Fig. 1) to be plain and functional to simplify the programming process and lower barriers to robot programming for end-users. It consists of three primary panels: 3D visualization panel, Terminal panel, and Chat panel. Moreover, the system incorporates two supplementary panels (Text Editor and File Tree), accessible to users via toggle buttons (top menu bar).

1. **3D Visualization Panel**: This panel has an embedded RVIZ interface to visualize the information from the motion planning system regarding the robot’s physical environment (see Fig. 1-a). Here, users can identify and address discrepancies between the physical

### Table 1: Alchemist’s Design Objectives (DO) and Rationales

<table>
<thead>
<tr>
<th>DO</th>
<th>Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DO1: Facilitate Programming with Natural Language</strong></td>
<td>We seek to use LLMs to facilitate an intuitive robot programming experience through natural language communication, aiming to reduce the need for programmatic thinking of end-users.</td>
</tr>
<tr>
<td><strong>DO2: Enable End-to-end Robot Development Workflow</strong></td>
<td>Robot programming is a complex multi-step process. We seek to simplify this process by providing a one-stop shop for development, testing, debugging, and execution of robot programs.</td>
</tr>
<tr>
<td><strong>DO3: Support Varied Programming Proficiencies</strong></td>
<td>End-users of robotic systems have varying programming preferences. We aim to offer a dynamic framework that adjusts the level of code generation abstraction, allowing users to choose between more flexible, general-purpose code that requires increased oversight and debugging, and more rigid, task-specific code that demands minimal oversight.</td>
</tr>
<tr>
<td><strong>DO4: Visualize Robot World and Actions</strong></td>
<td>Robot programs have real-world physical outcomes which necessitate previewing of program execution for ensuring safety and supporting debugging. We visualize the real-time robot world model and enable preview of actions to prioritize user understanding, control and safety over programs.</td>
</tr>
<tr>
<td><strong>DO5: Ensure System Modularity</strong></td>
<td>We seek to accommodate various LLMs and robot platforms with ease. Our system-level design ensures adaptability to evolving needs or technological updates without altering the interaction paradigm.</td>
</tr>
</tbody>
</table>

---

1. [https://tinyurl.com/alchemist-github](https://tinyurl.com/alchemist-github)
Alchemist’s back-end has several components to achieve the desired functionalities: Function Library, LLM Initialization Prompting, and Code Safety Mechanisms. We built upon the design principles laid out in Vemprala et al. [51] for the function library and LLM prompts but with enhanced functionalities tailored for end-user robot programming. Fig. 3 provides a high-level overview of the back-end operations; the implementation details are provided below.

### 3.3 Back-End Components

#### 3.3.1 Function Library

The function library is a platform-specific code library designed to be a broad set of tools for the LLM to use. It is a layer of abstraction over underlying ROS functions and services specific to achieve general actions (e.g., moving the robot, operating the gripper). The LLM is provided descriptions of these functions in its initial prompt and is instructed to use only these functions. We assign descriptive names to the functions in this library and their input to further leverage the natural language understanding capabilities of LLMs [51].

Our system is designed to cater to a wide spectrum of users, from those proficient in coding to individuals with little to no coding experience. We achieve this variability in LLM code output based on the chosen abstraction level through our initial prompt, rather than by altering the function library itself. In our current implementation, the function library contains functions with two levels of abstraction: high-level and low-level. This approach makes our implementation of the function library simple, easy to interpret, and interchangeable, resulting in a modular system.

High level abstraction support novice users or those with limited coding experience. These high-level functions are closely interconnected, serving specific, task-oriented purposes, and possess a limited number of input parameters. An example high-level function is `pour(target_name)` which takes the target container name to pour as an input and pours the container that is gripped by the robot into the target container given in the input, as can be seen this is a highly engineered, abstract function. Conversely, low-level functions are more versatile, encompassing a broader range of potential uses and interactions. They boast a greater degree of generality and offer more extensive functionality, while tending to be more error-prone. An example low-level function is `move(x,y,z,roll,pitch,yaw)` which takes the end-effector pose and moves the end-effector to that pose, as can be observed from this example this function is a small wrapper over the underlying robot API and not as abstract;

#### 3.3.2 Chat Panel

This panel allows users to interact with the LLM to generate the code required for their tasks (Fig. 1-2). It provides an input box for users to communicate with the LLM and contains the complete chat history. Once the LLM generates code and provides an explanation, only the explanation is presented in the chat history. Additionally, the chat panel allows users to communicate via voice input in addition to manual typing; this is enabled by speech-to-text functionality implemented using OpenAI’s Whisper [42].

#### 3.3.3 Terminal Panel

This panel is a Python terminal enriched with built-in functions to let users inspect, run, and save code and reset the system as needed (see Fig.1-3). The built-in functions and their usage descriptions are presented in Appendix A.

#### 3.3.4 Text Editor and File Tree Panels

The text editor constitutes a comprehensive text editing tool enriched with syntax highlighting features tailored for Python code (Fig. 1-4). Meanwhile, the file tree panel exclusively displays files generated by LLM and subsequently saved by the user. Its principal function is to facilitate users in monitoring the quantity of files they have saved, enabling them to rerun these files through the terminal when needed. By default, these panels are hidden, and users can unhide them by selecting the “Editor” or “File Tree” buttons located within the upper horizontal menu bar. This design approach, wherein the panels remain hidden by default, serves a specific purpose: to prevent users from feeling overwhelmed by an excessive array of panels and code.

---

**Figure 2: Example development and execution flow of different system capabilities.**

and the virtual worlds used in motion planning. Additionally, the panel is designed to be versatile: when paired with a Gazebo ROS node, it can act as a fully simulated environment.

#### 3.2 Chat Panel

This panel allows users to interact with the LLM to generate the code required for their tasks (Fig. 1-2). It provides an input box for users to communicate with the LLM and contains the complete chat history. Once the LLM generates code and provides an explanation, only the explanation is presented in the chat history. Additionally, the chat panel allows users to communicate via voice input in addition to manual typing; this is enabled by speech-to-text functionality implemented using OpenAI’s Whisper [42].

#### 3.2.3 Terminal Panel

This panel displays users’ saved code files. Initially, these panels are hidden, and users can unveil them by selecting the “File Tree” button located within the upper horizontal menu bar. This design approach, wherein the panels remain hidden by default, serves a specific purpose: to prevent users from feeling overwhelmed by an excessive array of panels and code.

#### 3.2.4 Text Editor and File Tree Panels

The text editor constitutes a comprehensive text editing tool enriched with syntax highlighting features tailored for Python code. Meanwhile, the file tree panel exclusively displays files generated by LLM and subsequently saved by the user. Its principal function is to facilitate users in monitoring the quantity of files they have saved, enabling them to rerun these files through the terminal when needed. By default, these panels are hidden, and users can unhide them by selecting the “Editor” or “File Tree” buttons located within the upper horizontal menu bar. This design approach, wherein the panels remain hidden by default, serves a specific purpose: to prevent users from feeling overwhelmed by an excessive array of panels and code.
notice that the move function can enable similar outcome as the \texttt{pour(target\_name)} function but with more granular control over the specifics of the movement. More examples of different abstraction level libraries can be found in supplementary materials.

3.3.2 Initial Prompting of the LLM. For our initial release and exploratory study, we employed OpenAI’s GPT-4 [36]; we chose GPT-4 due to its state-of-the-art reasoning capabilities and simple API usage. The key to effectively leveraging an LLM for programming is initializing the model with well-defined function library, a system role, and an environment prompts [51].

We instruct the LLM to act as a robot programming helper. We also specify the robot in terms of name, number of degrees of freedom, end effector, and type of code that it needs to return with some warnings, rules, and caveats (e.g., “always use floating numbers” and “you are not to use any other hypothetical functions that you think might exist”). The function library prompt contains the names, inputs, functionalities, and outputs of each function that is available to the LLM along with axis and unit conventions; this prompt acts as a descriptive code documentation.

The environment prompt is used to inform the system about the presence of physical objects within the workspace. For our exploratory study, we utilized various-sized beakers and graduated cylinders, all of which are listed in the environment prompt with their respective dimensions. If a sophisticated perception system, such as a vision language model [58], is used, the environment prompt could be omitted.

The final component of the initial prompt is the example user prompt, coupled with the corresponding example code output. This element holds significant importance in enabling the LLM to generate high-quality code. In our exploratory study, the provided example is thoughtfully crafted to incorporate a wide array of functions from the function library. The example given in our system is designed in a way that utilizes most of the functions in the library to reinforce the correct usage of each function by the LLM.

Lastly, to overcome the token limit issue arising from a lengthy conversation history inherent to iterative programming, we selectively truncate the middle segment of the conversation history and resend the API call to aid in error recovery.

3.3.3 Safety and Quality Assurance Methods. During the development process, we identified several common mistakes made by the LLM during code generation [12]. These errors can be roughly categorized into two classes: \textbf{errors in interpretation} and \textbf{errors in execution} (see Appendix B for details). Errors in interpretation, such as import errors, are usually detected by the program interpreter or compiler, and are relatively straightforward to rectify. On the other hand, errors in execution, such as adding an unnecessary action, are less obvious because they do not necessarily lead to an immediate code breakdown. Instead, these errors become apparent only when undesirable task outcomes are observed. We explored various methods, such as parsing the generated code or adding rules to the initial prompt, to address errors during execution; grounded prompting was the most effective approach.

Alchemist relies on grounded prompting (e.g., [54]) to reinforce specific rules for code output towards ensuring safety, quality, and executability of the generated code. We have two general grounding prompts that are added to all prompts issued by the users: (1) “By using the function library you are provided,” is added to the beginning of each prompt to reinforce library usage and prevent the LLM from using imaginary functions; and (2) “make sure to move back to home after the task is finished.” is added to the end of each prompt to ensure the robot’s proper positioning between tasks.

In addition to these two groundings, we use three conditional groundings more specific to our exploratory study. If there is an “add” word in the user prompt, then we append the user input with “Make sure to use marker location.”. If there is a “pour” word in the user prompt, then we add “Don’t move above the beaker before pouring, just call the pour function. Also, after pouring, make sure you place the object back to where it was on the table and then open the gripper to release it.”. If there are any of “function”, “generic”, “code” words in the user prompt, we add “If you wrote a function, remember to add an example function call at the end.”. These groundings were all added on an as-needed basis. The first one is to make sure when objects are being added to the virtual space, GPT uses markers as location indicators; the second is to address several issues in its call of pour function and assumptions it makes; and the third is to make sure when generic functions are written by GPT there is also an accompanying example function call.

Other than prompt grounding, we added code verification. The primary objective of the code verification is to examine the code produced by GPT and make necessary general corrections such as imports (e.g., not importing rospy, etc.), ROS node and function

Figure 3: Illustration of back-end components of Alchemist: the function library, initialization prompt and grounded prompts
library initialization, as well as Python version check. This verification ensures that the code created by GPT does not include errors resulting in failure of code interpretation.

3.4 System Modularity

3.4.1 Robotic Platform. Alchemist features modularized, interchangeable function library and initial prompts, allowing for the easy configuration to different robotic platforms with support for both physical and simulated robots and LLM used. In our initial system release, we provide implementations for two manipulators (Universal Robots UR5 and Franka Emika Panda) and one mobile manipulator (PAL Robotics TIAGo).

3.4.2 Vision System. The vision system in Alchemist offers an easy, fast and low cost way for quick testing and development. We modify the AR Track Alvar [34] package to have object grasping orientation information encoded in each marker. For this purpose, we created markers that have a pointy end indicating how the robot’s gripper should approach and align itself with the marker inspired by Sefidgar et al. [47] (for details see supplementary materials). Importantly, these markers play a pivotal role in establishing a connection between physical objects and their virtual counterparts. They are intentionally designed to be independent of specific objects, offering users the flexibility to utilize them as reference points for object positioning. Users can strategically position these markers anywhere within the workspace, allowing for various orientations and placements. It is worth noting that Alchemist is engineered to support seamless integration of a full-fledged perception system that has multi-modal sensing through its modular system architecture.

4 AN EXPLORATORY STUDY

We conducted an exploratory study to gauge the usability of our system and understand its limitations. Details of the experimental task and settings are provided in the supplementary materials.

4.1 Context and Task

Integrating robotic assistants in life sciences laboratories is emerging as a promising application domain [10, 11, 29]; the nature of lab experiments in life sciences requires precise and repetitive work with long operation duration and involves various health hazards. However, experts in life sciences rarely have experience programming or working with robots. Changes in an experimental protocol require robotics experts to set up or update the configuration of the robotic system to the new experimental requirements. We propose Alchemist as an alternative since its collaborative end-user programming paradigm is designed to help users with little to no robotics experience to be able to program robots to perform desired tasks. Thus, we chose a common biochemistry experiment as the basis for the programming task in our study.

We based our task on the LB Media preparation experiment, which is performed to create plates for growing live cultures in biochemistry and related field labs. We modified LB Media preparation task into a toy experiment by replacing reagents with beads and glass lab equipment such as beakers and graduated cylinders with plastic ones for safety reasons. At a high level, this task is about picking various graduated cylinders that have different types of reagents and pouring them into a beaker to create mixture.

In our main experimental task, we asked participants to use the system to generate general functions for re-usability but let them decide whether to use general functions or step by step prompting for the task execution. Before the main task, we used a training task to familiarize users with the system. Our training task was designed to give participants a complete overview of the system by requiring all system functionalities to be used.

4.2 Procedure

Participants were asked to fill out the consent form, and right after, they were given a user manual and asked to read it entirely. After reading the user manual, they were asked to watch a short tutorial video that explains the training task and the system. The experimenter then gave the participants the sheet containing the training task and went over it once to ensure the participants understood everything. Then, participants started the training task and the experimenter guided the participants along the tasks to familiarize them with the system. Upon completing the training task, the experimenter gave the participant the main task sheet, and started the video and screen recording. After that the experimenter went behind a divider to let participants work independently with the system. From this time till task end, the experimenter did not intervene unless there was a significant failure with the system usage or participants have spent more than 45 minutes on the task alone. If the experimenter stepped in, the experimenter was instructed to give just enough support to have participants go on with the task. A post-interaction questionnaire that collected System Usability Scale (SUS) [9] responses and demographic information including self-reported expertise in programming and robot programming was administered. After the questionnaire, participants had a semi-structured interview to conclude the study. The user manual, study procedure, interview questions, training task sheet, and main task sheet can be found in the supplementary materials.

4.3 Measures

We collected a range of metrics to understand the user experience of our system during the exploratory study (see Appendix C for details). Measures such as total programming time, debugging time, idle time, task completion time, number of errors, editor use, debugging method, and use of general functions are extracted using post-study labeling of the video and screen recordings of the participants interacting with the system and the robot during the study.

4.4 Participants

In this study, we recruited 5 (1 male, 4 female) graduate students or postdocs who work in biology, chemistry and biophysics fields (“novice”). Their age ranged from 25 to 29 (\(M = 26.8, SD = 1.48\)). We additionally recruited 5 (4 male, 1 female) graduate students who work in the robotics field. Their age ranged from 25 to 28 (\(M = 26.4, SD = 1.34\)). Participants who work in the robotics field are denoted as “experts” and provided us with additional insights about our system. Participants self-reported expertise levels were rated on a scale of 1 to 5 for coding and robot coding, with 1 indicating novice and 5 indicating expert. Novice users typically rated themselves...
as 2 for coding ($M = 2.42, SD = 1.51$) and 1 for robot coding ($M = 1.71, SD = 1.25$), while expert users rated themselves as 4 for coding ($M = 4.25, SD = 0.5$) and 4 for robot coding ($M = 4, SD = 0.82$).

4.5 Findings

From our exploratory study we observed some interesting similarities and differences in how novice and expert users interacted with the system (Table 2). Below, we highlight these observations.

Novices and experts both had similar average task completion times which might be due to different prior knowledge of both groups, one group possessing prior knowledge of the task well whereas the other group possessing prior knowledge on coding and robots. This is supported by how the expert group on average took less time programming and debugging but more idle time than the novice group; idle time refers to time spent on the task other than using the system (e.g., thinking, reading the user manual, etc.).

Novice users tended to debug their program by prompting the LLM further rather than using the editor; this behavior can be attributed to novice users avoiding direct dealing with code due to their unfamiliarity as reflected by N2: “It would be difficult for me to troubleshooting by myself, as I lack the confidence to examine the code to determine exactly what’s going on.” Similarly, we observed that novice participants chose not to use general functions; N3: “Since I’m not comfortable with coding, I prefer providing direct step-by-step instructions to the robot rather than using a generic function.” A prompt example by novice N3 illustrates this observation: “Pour the water from the 250ml graduated cylinder to the 500ml beaker. Put the 250ml graduated cylinder back to its original place gently.” On the other hand, a prompt example by expert E1 for the same task shows that experts tended to create generic functions and reused them: “Can you write a generic function called 'pick_and_pour', which allows me to insert the input, and the robot will based on the input to grab the target and pour it into the 500ml beaker?”. We provide more examples of user prompts in supplementary materials of this paper.

Overall, all novice participants had positive comments about the potential of the collaborative programming paradigm in democratizing robot programming especially in specialized domains such as life sciences research laboratories. N3 commented that “Working in a biochemistry lab, we need to prepare media almost every day, often in large quantities. It’s a routine protocol that takes up a significant amount of time. However, if a system like this could be used to automate the process, it could save a substantial amount of time.” N5 also commented that “I think the idea behind this project is really novel. It can be used extensively, especially in biochemistry labs, where tasks like these are a frequent occurrence. I think this system could be extremely helpful.”. This feedback validates our rationale behind developing Alchemist: empowering true end-users to intuitively create, edit, and test robot programs for custom uses.

The main limitations of the system arose due to problems with motion planning and visual perception resulting in failures in robot action which occurred independent of the generated code itself; our findings highlight the value of this collaborative programming paradigm while underscoring the need to improve reliability in task execution. Participant N4 experienced a significant number of vision and planning errors, which explains the outlier SUS score of the participant; N4: “I don’t feel confident using this in a lab because it keeps knocking over things several times.”. Yet, N4 still acknowledged the value of a system like Alchemist: “I would love to have a liquid handling system in our lab where I could simply press a button and say ‘go’ without worrying about it failing, but I don’t think it’s at that stage yet.”.

We note that although the occurrence of errors described in Appendix B has decreased due to our code verification mechanisms, errors have not been completely eliminated.

5 LESSONS LEARNED

Our system design and development process, along with our observations in the exploratory study, elucidated a range of effective practices for and limitations of our end-user programming approach. We are sharing a set of lessons learned to guide future work on LLM-powered end-user robot programming.

5.1 LLMs Can Output Unreliable Code

Robustness of LLM-generated code is critical for a successful end-user programming experience [30, 50]. We adopted two key strategies to enhance the reliability of the generated programs: (1) code verification through simple parsing and error handling mechanisms helped eliminate most interpretation errors; and (2) grounded...
prompting helped reduce execution errors and refine LLM responses to fit domain-specific requirements. These approaches helped minimize errors ensuring that even novice users were able to successfully complete our study task; yet, some errors still persisted introducing overhead in the task completion time by necessitating debugging (see Table 2). Noticing that novice users largely relied on prompting the LLM for debugging further underscored the need for reliable code generation and iterative refinement to enable end-users to effectively author robot programs. Incorporating advanced formal software verification methods [32, 52] into LLM-enabled end-user programming systems can further bolster the reliability of LLM-generated programs and enhance the efficiency of our proposed approach.

Lesson Learned: Enhancing LLM-generated code reliability through code verification and effective prompting is critical for end-user robot programming.

5.2 Effective LLM Prompting is Difficult
A significant challenge stems from users possessing a skewed or incomplete understanding of the LLM and its capabilities [6, 31, 57]. This often leads to vague or implicit prompts, which in turn can produce undesirable code outcomes, ranging from coding errors to unintended robot behaviors. To address this issue, we implemented two strategies to improve how users prompt the LLM.

Firstly, we used guided training assets (i.e., user manual, tutorial video, a training task) in our exploratory study to better familiarize the users with methods for LLM prompting. However, effective prompting of large language models remains a challenge for users and further work is needed to develop training methods that empower end-users to create high-quality prompts.

Secondly, we also utilized grounded prompting to dynamically add contextual details to user prompts, e.g., when users prompted the LLM about a pouring task we added grounding to the prompt to state that the container once poured should be returned to its original location. However, this mechanism is domain-dependent and requires prompt modification in case of a vastly different application domain reflecting a limitation of this approach. One potential solution could be the structural integration of the LLM within the robot programming ecosystem to ensure the LLM has necessary contextual information. Moreover, enhancing dialogue between the LLM and the user about the task’s overarching purpose and characteristics could strengthen the collaborative bond and enrich the LLM’s contextual comprehension. These approaches could facilitate the automatic grounding of tasks, based on current requirements and conditions.

Lesson Learned: Effective LLM prompting requires end-user training and dynamic context-dependent prompt enhancement.

5.3 End-User Aversion to Direct Coding
End-users exhibit a wide range of knowledge regarding programming and specifically robot programming. To cater to this diversity, Alchemist features a two-level abstraction in its function library which affects how the LLM generates code. The high-level abstraction functions aimed to guide the LLM to generate less error prone and easier to debug programs while the low-level abstraction provided the end-user with more control over program specification. This distinction was important for novice users especially as we observed they tend to avoid directly viewing and editing code.

LLMs are capable of generating general purpose code that can be reusable in later stages of a task. Alchemist provides users with a terminal panel function (see Appendix A for details) that allows them to call general functions with custom inputs without editing any code; however, we still observed that novice users hesitated to use general functions. Further integrating LLMs into the programming ecosystem as a conversational assistant could allow novice users to call such functions without needing to use the command line interface, thus further enhancing a sense of collaboration. Further work is needed to design methods to empower end-users to effectively use more advanced programming notions (e.g., general functions) to maximally exploit the generative capabilities of LLMs.

Lesson Learned: Introducing abstractions to minimize code complexities while retaining programmatic expressiveness can enhance user confidence in programming.

6 LIMITATIONS AND FUTURE WORK
Though our exploratory study provided an insight into the effectiveness of Alchemist as well as how expert and novice users interacted with it, the sample size is small, limiting our ability to draw conclusions regarding how people with different levels of programming knowledge may use an LLM-based robot programming system; moreover, the stochastic nature of LLM outputs may result in various programming/user behavior and user experience [37]. Future work should investigate methods to validate LLM-based robot programming and conduct well-powered experiments. Furthermore, while the inclusion of expert users in our exploratory study offered valuable insights into how experts and novices differ in using the system, it limited our ability to focus the study on true end-users.

While true end-users (i.e., participants from life sciences) evaluated our system, their interactions occurred in a robotics laboratory, rather than a real-world deployment environment which represents another limitation. Additionally, this paper centers on a singular use case for our system. However, its potential applications span diverse settings, from varying experimental protocols to entirely distinct domains like manufacturing or customer-facing services in hospitality and food sectors. Future research should explore deploying this system or analogous ones in real-world contexts across various domains to discern both its capabilities and constraints. Lastly, future research should systematically compare our end-user programming system with existing state-of-the-art systems to better evaluate and understand their differences.

In conclusion, we see a shift toward a more collaborative paradigm for end-user robot programming as powered by large language models and vast opportunities to use LLMs in advancing end-user development of robot applications in diverse domains.

ACKNOWLEDGMENTS
This work was supported by National Science Foundation award #2143704.
REFERENCES


