Simplifying Robotic Home Assistance: Translating Human Intent into Robotic Action-User Translation

RBE594-Fall 2023

Keith Chester

December 15, 2023

Abstract

Millions of people requiring physical assistance reside outside institutional settings with full time assistance available. The idea of in-home robotic assistance has been elusive for decades due to the complexities of unstructured home environments and difficulty of human-robot communications. This project demonstrates an end-to-end concept for in-home robotic item retrieval for non-technical users. By employing Large Language Models (LLM)s to determine the user's desired object, we achieved a simplified interface for users to task the robot and enable communication with a high-level planner to dynamically create and control the robot action. The high-level planner is also utilizing the gains of contextual awareness that these LLMs demonstrate to allow better understanding through implied heuristics to shape its planning. We demonstrate autonomous mapping of the simulated environment through a Simultaneous Localization and Mapping (SLAM) algorithm and utilize an off-the-shelf object recognition platform (YOLOV8) to identify objects in the simulated environment. From there, objects can be categorized and queried from a state management module. The task planner synthesizes the desired object, the obstacle map and the states of known objects to design tasks to seek the desired object, retrieve, and return to the user. All of this is demonstrated within a ROS2 powered simulation of a robot and home in Gazebo.

Keywords

robotics, LLM, task planning, embodied agents, egocentric planning

Introduction

Robotics is an integration problem of enormous complexity, marrying mechanical engineering, electrical engineering, and computer science just to have a platform. Even if a platform could solve a task, navigating the world logically with task planning - or the set of decisions and actions performed to achieve a given objective - requires specialists to incorporate the state of the world with how to approach a given objective in the form of programming. This requires highly specialized individuals to prepare a robot to work in a given environment, and is a significant reason why robots are still limited to highly controlled environments with low adoption in small organizations, let alone by individuals in the home setting.

Recent advancements in large language models (LLMs) promise to provide agents that can reason about their world and create plans as a human programmer would. This would allow task planners to generalize about and handle unpredictable human-oriented environments and develop a set of plans using robot motion primitives to solve the given task, without need of specialist programmers.

This project sets out to prove that this it is possible for generalized robotic AI via new LLMs. To this end, we focused on creating a simple "fetch" robot to act as a disabled persons or elderly individual assistance robot. The goal was to take a robot platform that could realistically interact and move through a human occupied environment and fetch items for its user. To increase ease of use amongst elderly users, we also sought to demonstrate the use of human language as an interface instead of relying on an external app or integrated control screen. Since this author was primarily focused on the planning work, my goal of the project was to create realistic and successful plans to search for and recover items upon request of its user in natural language.

We now take a look at active research in this area which influenced technical decisions and overall direction of this project.

LLM as Reasoning Agents

There has been significant reesearch in exploring techniques to elicit reasoning from LLMs. Expanding upon

basic one- and n-shot (few-shot, henceforth) techniques is *Chain of Thoughts*, which proposes an alternative to simply asking the model to generate an answer, wherein we ask the model to work through problems step by step, as if it was thinking through the problem.[1] A follow up paper expands on this by using parallel threads to generate multiple lines of reasoning, banking on the correct answer or approach being more probable (as LLMs are probability engines), called *Self-Consistency*. [2]. This parallel approach heavily influenced our own. *ReAct* introduced multiple step reasoning paired with tools in the form of knowledgebases. [3] *Tree of Thoughts* expands on this by branching out parallel threads in this multi step reasoning process. [4]

Given the importance of prompt engineering, there has additionally been research in letting LLM agents develop their own prompts to achieve a set of tasks. [5]. Additional efforts have been used to generate either fine tuning datasets or additional prompt engineering as well. To learn how to use external tools via function or tokenized calling (the latter being a specifc token repurposed in fine tuning to act as a function name), Toolformer, MRKL, and TALM all explored applying reinforcement learning techniques to improve model and prompt performance. [7] [6] [8]. Discoveries from these papers influenced our own approach to prompts even without the time and resources to appropriately apply an iterative generation of prompts.

LLM Agents and Robotics

There is additional research looking at the application of LLMs in robotics. *SayCan* explores the use of taking pretrained large language models with additional and then attaching image inputs to create a model for finding affordance functions to judge whether a given action would succeed given what it sees, grounding the actions of the robot as prompted by another LLM in success likelihood. [9] *Inner Monologue* proposes an approach of embodied reasoning planning, specifically using a frozen LLM model with incorporated environmental feedback to allow better control in robotic environments, hoping the approach augments LLMs to handle control of robotic agents in complex tasks without fine tuning or significant changes requiring retraining. [10] *Code as Policies* noted that LLMs have particular skill at code generation, and demonstrated some temporal and geospatial reasoning, which, through the presented framework, could then generate policy code for robotic control.[11] Similarly, *ProgPrompt* discussed a programmatic prompt approach to facilitate plan generation across different environments and robotic applications. [14] *Statler* utilizes LLMs for embodied reasoning for state management. Succinctly, Statler proposes a method to create a representation of the world in text that acts as an injectable memory and maintained over time entirely through LLM interaction, allowing long horizon task planning for robotic agents. [13]. *RT2* explored using transformers pretrained on internet data to create a Vision Language Action (VLA) model using existing large language models; RT2 will, given natural language commands, robotic state, and vision, have a singular combined multi-headed model output direct robotic control. [12]

Methodology

This project was approached with the goal of creating a realistic simulation of a robot in a home environment using industry standard tools, then focus on proving that LLM agents can excel at creating high level task plans that benefit from both state knowledge of its environment but derive additional contextual insight as an emergent property. Similarly, we approached the use of the robot in two phases - the first, our initialization phase - assumed the robot was a recent addition to the household, and is given the freedom to explore and learn its environment. The second - our utilization phase - assumes that the robot has mapped the environment, seen some set of interactable items throughout the house, and thus can reasonably create a plan to perform actions as requested by its human user. Finally, the project work itself can be divided into three categories as well - the initial setup, the simulation and robot, and finally the AI agent integration. This author had significant contributions in each of these steps and can claim it as their work.

Environment Setup

We begin with the creation of tools to manage our development and tooling environments. ROS2 and Gazebo are notoriously dependent on operating system integrations, requring specific versions of not only C/C++ dependencies, but is also specific of the Ubuntu operating system and resulting installed libraries to operate. This can cause significant friction to a distributed team, resulting in hours lost trying to find version differences to solve infamous "works on my machine" issues. By creating a repeatable containerized and/or virtualized environment, each team member can have rapid deployment repeatable environments that match across each of our machines irregardless of host machines.

I developed a series of tools automating the downloading, configuration, and installation of virtual machines, containers, and dependencies as per the team's needs. Teammates could run a single command to have their environment recreated or updated to match everyone else's. This proved crucial to avoiding complications in initial setup, allowing us to instead focus on core components.

Once the environment is set up, we look at the interactive development environment, or the IDE. To this end, we wished to ensure that each team member had the appropriate tools set up and integrated in the easiest possible way, without requiring significant setup or differences between users. To accomplish this, I developed a visual studio code tunnel setup that would integrate into the virtual environments and allow teammates to work in their native OS, but have equivalent IDE setups with full integration into built ROS2 Python modules and APIs. This is significant as ROS2 does not work cleanly with Python virtual environments, breaking compatibility with normal Python development patterns. Using GitHub remote tunnels through VS Code and having it properly synced across the multiple virtual environments mitigated this problem.

Explorer Module

We needed to generate maps of our initial environment for navigation purposes, as well as build an initial "lay of the land" view of where items were to generate our room labels for the planner. While we initially did this by manually driving our robot through the map manually via keyboard controls, we wanted to try and automate the process to emulate a more realistic "out of the box" feel for a robotic product. To this end I built the explorer service. Once started, the explorer service would subscribe to */odom* and */map* to receive updates of the robot's current location and the state of the occupancy map generated by the ROS2 *slam_toolbox* module. We ingested the occupancy map from the broadcast, convert it to more usable 2d numpy array, and then used A* search to find the closest desirable free spot. The heuristic cost for the A* planner was distance of the total path traveled (preventing snaking paths from being preferred over straight shots). When considering nodes, additional criteria were considered, processing each cell based on a 2d filter applied across the map. This filter would designate how close each unknown spot was to an obstacle; a spot designated "unknown" surrounded by too many obstacle pixels was not considered. This was implemented when an earlier version became obsessed with trying to navigate to the interior unknown pixels of a concrete pillar, and again when unknown pixels behind an object that could not be navigated around presented a small amount of unknown pixels on the map.

The result of this module was a robot that would explore an area efficiently, exploring various nooks of a map until it was sufficiently covered for navigation and other tasks.

Room Segmentation

Once the occupancy map was generated, it was fed manually into a classical computer vision pipeline. While a deep learning solution would be more robust and produce better results, there was not sufficient time or resources to generate a dataset and train a model.

The occupancy map is loaded, resized, and blurred to emphasize lines representing walls. We perform a binary thresholding to limit our map to merely black/white pixels, and generate a mask around the outline of the mapped area so we do not generate rooms in the "unknown" outside area. We finally perform a distance transform to extenuate rooms to their centroids, find the contours of the resulting shapes, and feed it into OpenCV's Watershed algorithm. The resulting markers and index labels are then checked against total area represented; small area rooms might be small obstacles (such as a couch, in our simulation). We took the approximate area of a typical couch/desk, and thresholded our areas; any area smaller than this limit was instead merged into their surrounding index by finding the largest surrounding room label. We resize the result back to the size of the original occupancy map for a 1:1 overlap of pixel coordinate to room segmentation index. We finally save the map, providing it to the state management's room service, which tracks knowledge about rooms and can be queried about the room a given coordinate belongs to. While the map is not perfect, it is accurate enough for our semantic labeling purposes.

We then load all items in memory, converting their real world meter coordinates to associated pixel coordinates. We look up the associated label on our segmentation map and group all items by that index. We finally ask the LLM model in a few-shot example prompt what room label we should apply to a room with the given objects in it. This resulted in accurate results, though some rooms, like "living room", occasionally had synonyms that might change performance; "lounge" and "den" appeared, for instance. All items are then tagged with the resulting label, so that when we generate the state prompt we can specify what items are in what rooms.

Litterbug

Litterbug is a service designed to handle multiple issues that arose throughout, its featureset expanding as issues arose. Initially, Litterbug's purpose was to handle tracking and managing items throughout the map. Upon startup, it would populate the map with a set of items in specified in a CSV file, allowing rapidly changing the environment setup to match what is needed for a test run. Litterbug also would handle interactions, limited for this project to picking up object and giving objects to the human user. Picking up objects were as-

sumed successful if the robot was within 1 meter of the object, and giving objects to the user was successful if the robot had previously picked up the object and was within 1 meter of the human user. When an object was picked up, it was appropriately removed from the simulation.

With issues in the vision integration occurring, it became probable we needed an additional way to simulate vision. Since Litterbug was already tracking and managing items, it made sense to have Litterbug adopt this responsbility as well. To simulate vision, a copy of the occupancy map developed during the initialization phase via the exploration service is fed in. Litterbug has an internal tracking of this map, conversion of pixel coordinates to real world meter coordinates (including initial base frame offset), and item and human coordinates within it. Litterbug would then, at a set rate similar to our camera ($24Hz$) take the current location of the robot and simulate vision via a projected cone. To accomplish this, first we check the distance of each item against the current location of the robot, finding all within the "view range" of the robot (for our purposes, 8 meters). All items that could fall within that range then have their angle relative of the robot's own orientation checked. The camera simulation is given a 45 degree field of view (FoV), similar to our chosen camera. We calculate the angle of a line between the centroid of the item and the robot, noting its orientation in the world frame. If the robot's orientation is $\pm\frac{FoV}{2}$ (with special handling of rollover at 2π) of the line generated, then the item also falls within the FoV of the robot. We then generate a line with Bresenham's algorithm along the occupancy map from the centroid of the robot to the item, checking each pixel cell the existence of a known obstacle. This would mean that the view of the object is obstructed. After these checks, we know that it is possible that the item was seen. We then run a probability check, giving the chance that a false negative would occur and thus not broadcasting it. All objects have the following odds of being broadcasted based on its distance from the robot:

% Vision Range	% chance seen
$0-40\%$	95%
$40-60\%$	75%
$60-80\%$	50%
$80-100\%$	25%

Table 1: Probability to be seen

If the item is going marked as seen after this, we then randomly generate a ± 0.5 meter error to each coordinate dimension of the object's location, allowing us to test the robustness of our system dealing with noise. It is processed in this order due to likelihood of eliminating objects early for additional processing and the cost of each calculation.

AI Agent Planning

The planner service handles the AI agent feature. It consists of three pieces - the service itself - which provides the input and output channels, the AI, which coordinates the LLM models, prompt generation, and the process of generating and rating prompts, and the robot engine - which takes the pythonic plan code and executes it when instructed. We will move through each piece.

Service The ROS2 service exposes a ROS2 endpoint */objective*, wherein the human's instruction to the robot is provided. Once an objective is received, it is processed by the AI module. If another objective comes in, the AI planner is cancelled. If a plan is actively running via the robot engine, it will be cancelled to trigger the new objective. The service also broadcasts several helpful channels. */objective/status* shares the current process of the planning/execution project for a given objective. */objective/plan* announces the resulting plan from the AI agent module. */objective/ai/out* broadcasts all print statements generated by the AI (it is encouraged to explain to the user its actions and thoughts in print statements within prompts). Finally */objective/action* broadcasts whenever a control function is called by the AI's plan.

AI Module The AI module's job is to generate a plan to achieve a given objective. Upon being asked to generate a plan, it creates a set of prompts. The prompts for how its control functions work is prepared ahead of time with few-shot examples. A state prompt is generated by querying for items correlated to the objective ranked by generated embedding vector distances. The state prompt contains the names of each available room and their distance from the robot, and then items are presented in a manner collated by what room they are in and their distance from the robot.

Once prompts are generated, a parallel process is triggered wherein we call the LLM provider N times (5 for our purposes) to generate multiple competing plans at once. We then scan through each plan, parsing out non Python code where possible; models commonly would write markdown around the plans, explaining its task, and then wrapping Python code in a markdown *"'python* tag; when this happens we merely cut out the code within those tags. Once returned, we then perform an AST syntax check, eliminating plans that failed to generate recognizable Python from contention. We append the surviving plans together in a few-shot example prompt rating the plans. The prompt also encourages the agent to explain why it rated the plan as it did. Plans are ultimately rated with a score of 1 to 5, with 5 being the best. The plan ratings are returned via a JSON response which is parsed for its scores and reasoning. This is done in parallel as well (again, $N = 5$), and resulting scores are averaged to select the best plan. Due to the lack of a JSON to-

kenizer in PaLM 2 and the resulting difficulty getting repeatable parseable JSON from it, this step skipped and only a single plan is generated and immediately used.

A tree of thoughts approach was tried instead of this approach, but results were poor, likely due to poor feedback and rating leading to essentially random branching and chaotic plans. This could perhaps be revisited if significant time was dedicated towards generating a more robust "rating mechanism" LLM for individual branching of steps.

Robot Engine Finally, we have the robot engine, which parses our python code for the execution by controlling the robot. I first prepared what we considered a domain-specific language (DSL) and parser, despite being based in Python code. The goal was to create a language that was easy to convey but also allowed complicated logic to be created to control the robot for its tasks. Ultimately I discovered that I was spending too much time implementing basic features that are available in other languages. Similarly, if I make the DSL too close to Python, LLM models would simply assume it to be Python and thus assume language attributes that were not implemented. To convince the models otherwise would take either significant context and effort in prompting or significant time fine-tuning.

With this in mind we swapped to generating Python code, thus benefitting from the impossibly large trove of training data for Python code generation. We developed a set of simply named Python functions with documentation to convey to the model its purpose.

- **move_to_room** - move to the centroid of a labeled room
- **look_around_for** - check recent vision history; if items within are not recently spotted, spin in place $\frac{\pi}{4}$
- **move_to_object** - move to the closest object specified if it is known
- **pickup_object** - pick up the object if the robot is within 1 meter of it
- **give_object** - give the object to the human if you are within 1 meter of them and have it in the robot's posession
- **return_to_human** - move to the last known location of the human

The Python code plans are executed in a sub process, with these functions injected. The functions are not passed straight, but rather wrapped in lambdas and classes that handle the asynchronous system for handling ROS2 components, while also allowing remote cancellation to quickly interrupt the actions if another objective came in. *print* statements were broadcasted out by redirecting standard out buffers. Exceptions thrown within the control code would be captured and announced in a controlled manner for safety.

Integration

At its core robotics is a complicated marriage of many fields of engineering and expertise; no robotics project is complete without significant effort placed into the integration of all parts. To this end, the author found himself working through multiple integration problems throughout all services listed here as well as the environment, vision, and state management services. Each module took significant time to test and ensure they ran together smoothly.

Results

We can break results into the performance of various models generating pythonic plans, observed emergent behaviors from the models, and performance across a set of tasks to test understanding of tasks and the environment.

Model Performance

We utilized Google's PaLM 2, OpenAI's GPT-3.5-turbo, and GPT-4-turbo. The PaLM 2 model version was *text-bison-002*, GPT-3.5-turbo's version was *gpt-3.5-turbo-1106*, and GPT-4-turbo's version was *gpt-4-turbo-1106-preview*. We observed the following performance.

We define code failure as a failure to produce Python code. We detected this through two methods. The first was an AST syntax check of the string (though we do scan for and remove common pre- and post-code inclusions, wherein models often wrapped code with markdown explanations of what it set out to do). The second was to scan for inclusion of at least some of our control code functions, as their absence would imply the code would do nothing.

While code line length is not necessarily correlated with complexity or reliability of code, we saw such a correlation here. PaLM2 and GPT-3.5 performed similarly in both code length and generally complexity of approach towards presented problems, whereas GPT-4-Turbo routinely produced longer and more complex behavioral plans for the robot.

Finally inference times were similar for PaLM2 and GPT-3.5-Turbo, but significantly longer for GPT-4-Turbo. This is clearly a limiting factor for adoption of these techniques in real time applications.

	PaLM2	GPT-3.5	GPT-4
Code Failure	0%	3%	0%
Average Lines	41.75	40.7	72.12
Longest Lines	64	82	118
Shortest Lines	20	45	13
Inference Time Avg	5.00	5.28	32.82
Longest Time	11.18	14.14	66.13

Table 2: Model Metrics

Emergent Behaviors

We saw a series of "emergent behaviors", wherein we saw the planning agents develop code that did things that either we only lightly suggested in prompting or was entirely novel.

Often, agents produced tiers of back up plans, wherein the agent produced code that could deal with its first chosen action not resulting in an immediate success. For instance - if the plan said to look for the object, and immediately move to it, but had no prepared plan for what to do if the item was not immediately obvious to it, then this would be considered having no backup plan. Similarly, having the plan realize that this will not work, and then moving to another room to try there, is a backup plan. Having multiple tiers of backup plans would consist of trying multiple rooms, or looking for different kinds of objects. GPT-4-Turbo always produced a backup plan to initial failure, PaLM 2 almost always had one, and GPT-3.5-turbo rarely provided one. GPT-4-Turbo almost always had multiple levels of backup plans, which PaLM 2 rarely had, and GPT-3.5-turbo never demonstrated.

Room-by-room searching is the agent producing a set of instructions that would have the robot visit multiple locations in hunt for the desired object. GPT-4-Turbo always had some form of this, PaLM 2 usually did, and GPT-3.5-Turbo occasionally did this.

Task Performance

We created a series of tasks to test performance of each model within the same agent. The only difference in generation and prompting was PaLM 2' aforementioned lack of a JSON tokenizer preventing reliable rating agent output. This resulted in us simply taking the first plan generated with PaLM 2 instead of comparing multiple and selecting the best.

Tasks grew increasingly complex. The tasks were:

- **(1)** objects with known locations nearby to the robot
- **(2)** objects with known locations in another room
- **(3)** objects with unknown locations in the same room
- **(4)** objects with unknown locations in expected rooms
- **(5)** objects with an unknown location in unexpected rooms

For tasks **(4)**, we mean objects that the robot knows exist, but are unsure where it is, in a room one would expect it. For instance - food would typically be found in the kitchen. Along those same lines, **(5)** are objects that the robot knows about in an unknown location that makes little sense, such as a plate of salad being located in the bathroom.

The performance of each model across these tasks are shown in the following table, outlining successes across each. Failures are only considered for the sample size if the failure was an issue of planning, not from simulation or robot failures.

	PaLM2	GPT3.5-turbo	GPT4-turbo
1	10	10	10
2	10	9	10
3	9	9	10
4	5	4	9
5	0	0	3

Table 3: Model Task Performance

All models were able to handle short horizon tasks, and often short horizon searching. We did see that GPT-4-turbo could handle most short or long horizon tasks that required searching, as long as the request fell within the bounds of predictability - if the item did exist where one would expect or where the robot last saw it, it would likely succeed.

The only 3 successes in the final task was, by our observation, only possible because the robot happened to see the item as it was driving past on its way to another room, placing it into memory and triggering *do_i_see* functions to redirect the robot there. The lack of success in this category is viewed by this author as a positive - it provides demonstrable proof that the planners were indeed using their contextual understanding of their world to shape their planning.

Plans generated by the agents were able to solve many of the typical "fetch" requests we were able to give it. Often we were surprised by interesting logical jumps the agents made. Failures were equally interesting, sometimes extrapolating additional functions we did not tell it about in a similar naming style,

Discussion

Future Work

We would be remiss if we did not consider the various avenues that this work opens up for future explorations of the topic. The nascent field of LLM embodied agents

is massive, but we suggest some particular areas of interest we'd like to explore next.

Iterative Planning - Our plans, despite parallel generation and rating, was still performed in a single inference run with the state prompting generated at query time. Instead, an agent that performs iterative question and answer cycles with the state management system to refine its own understanding of the environment prior to plan generation could result in a better understanding of the world state and more complicated tasks.

Functional Development - We provided a set of core functions that allowed the robot to control robotic hardware with an easy interface. It would be interesting to give the agent smaller "building blocks" of core robot functionality, and then have it build increasingly more complex functionality iteratively based on what it wishes to do. For instance - we provided a *look_around_for* function, which spins the robot in place at intervals, querying the vision system if it recently saw an item of a set label. Instead, we can give the agent just the vision query and movement code, and expect it to eventually develop equivalent functionality as it builds up.

Rating Agent Our rating agent was "good enough" - in that it produced plausible explanations for its resulting scoring of each plan. We did not have time, however, to fully explore the effectiveness of this agent beyond initial testing. Not only would it be wise to create a human-rated dataset of plans. From this we could work to derive a targeted rating agent, with the goal of creating one that would select more desirable plans.

Automated Prompt Generation We engineered our prompts through trial and error, though recent works suggest that prompts can be generated with genetic algorithm or reinforcement learning approaches. [15] It would be a worthwhile endeavor to try to improve performance through use of these techniques to develop the best prompts for a given application and discover best practices for future projects.

In Progress Self Rating One of the original design goals was to have an LLM agent self-assess its performance or react to new information, such as spotting new objects in the environment, or realizing that a passageway has been blocked. Unfortunately, due to inference time of our language model providers, we limited replanning to failure of a plan as designated by the agent ahead of time. Agents that could assess the progress of its plan and decide to replan or incorporate new information could create far more reliable agents and be applied to more complicated tasks.

Fine-tuning Fine tuning is the process of performing additional training on a pre-trained model, orienting the agent to produce the output desired without the need of extensive prompt engineering. Generating an extensive dataset of objectives to plans could produce significantly more complex and reliable plans by the agent, resulting in a better performing robot. Similarly, recent works suggest that you can match existing performance of larger general application models with far smaller models, allowing weaker consumer oriented hardware to run models for specific applications. It was not pursued during this project due to time and resource constraints.

Tokenizers Recent applications have demonstrated the improvement of agents in applications requiring specific output formats or actions by using specialized tokenizers that only output compatible tokens. For instance- OpenAI's JSON tokenizer will ignore suggested tokens by the model that would produce invalid JSON. Similarly, other tokenizers have been produced for various languages such as Python. These increase the probability that the agent will produce correctly formatted output. Additional research into a robotic action oriented tokenizer can be done to produce more reliable results from smaller models, but would require custom models as existing model providers within this work do not allow custom tokenizers.

Chat Agent Integration Significant research and commercialization has recently been done to apply chat agents (or "chatbots") to existing applications as a new interaction layer between users and their applications and data. Many of these utilize techniques such as retrieval-augmented generation (RaG) to expand LLM capabilities into the chosen domain of their application. It stands to reason that if these agents can provide new human-like communication interfaces to existing applications, then robotics can also benefit. This might provide an even better interface, since robotic applications often do not have an integrated screen for traditional application control.

Conclusions

We set out to demonstrate that modern LLMs can be utilized to form embodied agents to control robotic systems and provide a high level planner of logic to solve tasks while also introducing emergent behavior derived from contextual understanding of its environment due to its training data. We believe that we demonstrated this within our project, opening up exciting new avenues of follow up research and applications.

While the technological limitations of LLMs prevent them from being readily applied to existing robotic platforms - their extensive hardware and power requirements, costs of both hardware and training, and inference time - we believe that the excited fervor around this technology will drive rapid progression on all of these fronts to become more accessible and usable. Thus, it is this author's opinion that it is only a matter of time before these techniques are further refined and adapted as a standard of robotic platforms.

References

[1] Jason Wei Xuezhi Wang Dale Schuurmans Maarten Bosma Brian Ichter Fei Xia Ed H. Chi Quoc V. Le Denny Zhou "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models" https://arxiv.org/abs/2201.11903

[2] Xuezhi Wang Jason Wei† Dale Schuurmans† Quoc Le† Ed H. Chi† Sharan Narang† Aakanksha Chowdhery Denny Zhou Self-Consistency Improves Chain of Thought Reasoning in Language Models https://arxiv.org/abs/2203.11171

[3] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, Yuan Cao "ReAct: Synergizing Reasoning and Acting In Language Models" https://arxiv.org/abs/2210.03629

[4] Shunyu Yao Dian Yu Jeffrey Zhao Izhak Shafran Thomas L. Griffiths Yuan Cao Karthik Narasimhan "Tree of Thoughts: Deliberate Problem Solving with Large Language Models" https://arxiv.org/abs/2305.10601

[5] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, Jimmy Ba "Large Language Models are Human-Level Prompt Enigneers" arvix https://arxiv.org/abs/2211.01910

[6] Timo Schick Jane Dwivedi-Yu Roberto Dessì† Roberta Raileanu Maria Lomeli Luke Zettlemoyer Nicola Cancedda Thomas Scialom "Toolformer: Language Models Can Teach Themselves to Use Tools" https://arxiv.org/abs/2302.04761

[7] Ehud Karpas, Omri Abend, Yonatan Belinkov, Barak Lenz, Opher Lieber, Nir Ratner, Yoav Shoham, Hofit Bata, Yoav Levine, Kevin Leyton-Brown, Dor Muhlgay, Noam Rozen, Erez Schwartz, Gal Shachaf, Shai Shalev-Shwartz, Amnon Shashua, Moshe Tenenholtz "MRKL Systems" https://arxiv.org/abs/2205.00445

[8] Aaron Parisi Yao Zhao Noah Fiedel "TALM: Tool Augmented Language Models" https://arxiv.org/abs/2205.12255

[9] Michael Ahn, Anthony Brohan "Do As I Can, Not As I Say: Grounding Language in Robotic Affordances" https://say-can.github.io/assets/palm_saycan.pdf

[10] Wenlong Huang, Fei Xia "Inner Monologue: Embodied Reasoning through Planning with Language Models https://arxiv.org/pdf/2207.05608.pdf

[11] Jacky Liang, Wenlong Huang "Code as Policies: Language Model Programs for Embodied Control" https://arxiv.org/pdf/2209.07753.pdf

[12] Anthony Brohan, Noah Brown "RT-2: Vision-Language-Action Models Transfer Web Knowledge to Robotic Control" https://robotics-transformr2.github.io/assets/rt2.pdf

[13] Takuma Yoneda, Jiading Fang "Statler: State-Maintaining Language Models for Embodied Reasoning" https://arxiv.org/pdf/2306.17840.pdf

[14] Ishika Singh, Valts Blukis "PROGPROMPT: Generating Situated Robot Task Plans using Large Language Models" https://arxiv.org/pdf/2209.11302.pdf

[15] Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, Tim Rocktaschel "PromptBreeder: Self-Referential Self-Improvement Via Prompt Evolution" https://arxiv.org/abs/2309.16797