LLM-Assisted Control of the Clearpath Jackal UGV via Natural Language

Kenneth Berry III[1] Kapi Ketan Mehta[1]

kberry13@vt.edu kamehta@vt.edu

[1]Virginia Tech

Abstract

Natural language interfaces significantly enhance usability and intuitiveness in robotic systems. This paper introduces a version of PlannerLLM, an innovative framework that utilizes large language models (LLMs) integrated with reinforcement learning (RL) for precise robotic control using natural language commands. Our approach consists of two distinct stages: initially, a zero-shot classification LLM trained on the Multi-Genre Natural Language Inference (MNLI) dataset interprets textual instructions and maps them to specific robot actions, extracting parameters such as distances, angles, or absolute positions via regular expressions. Each intent directly corresponds to dedicated trajectory-building functions, producing clear and actionable paths. Subsequently, these generated trajectories serve as inputs to a robust RL-based actor network trained with the Twin Delayed Deep Deterministic Policy Gradient (TD3) algorithm. This network translates trajectories into precise wheel velocity commands tailored for the non-holonomic Jackal robot in the MuJoCo simulation environment.

Our primary contributions include demonstrating an effective integration of zero-shot LLM classification with reinforcement learning for direct robotic control without relying on manual demonstrations, clearly quantifying trajectory accuracy through positional and directional metrics, and providing an open-source implementation complete with training scripts and performance visualization tools. We validate our method through rigorous simulation experiments and provide extensive evaluations, including 3D trajectory visualizations, confirming the reliability and adaptability of our system to diverse natural language instructions.

1. Introduction

Human-robot interaction remains fundamentally tied to natural language communication, yet most robotic systems still rely on rigid numeric command interfaces. Unmanned ground vehicles (UGVs), in particular, are increasingly incorporated into diverse applications such as delivery, surveillance, and land surveys. These tasks require operation within complex, uncontrolled, and human-centric environments, navigating dynamic interactions with static and moving objects. Although these robotic agents are typically constrained to controlled movements within the 2D plane, they can navigate environments with smooth transitions in surface altitude. However, piloting these vehicles under such conditions through conventional numeric interfaces can be laborious, unintuitive, and impractical for non-technical users.

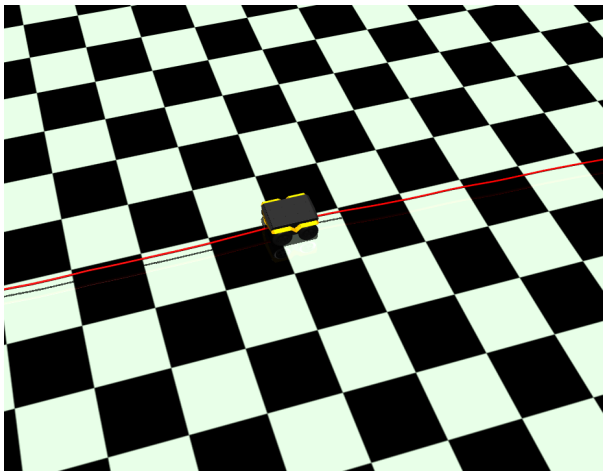


Figure 1: Jackal in Mujoco Environment with planned trajectory

Recent advancements in large language models (LLMs) have unlocked significant potential for translating free-form text into structured action plans, enabling intuitive task specification without specialized knowledge [2]. There have been multiple efforts to connect language commands directly to robotic system operations using LLMs and natural language transcription [4]. Modern LLMs' enhanced capability to process multi-modal inputs further expands

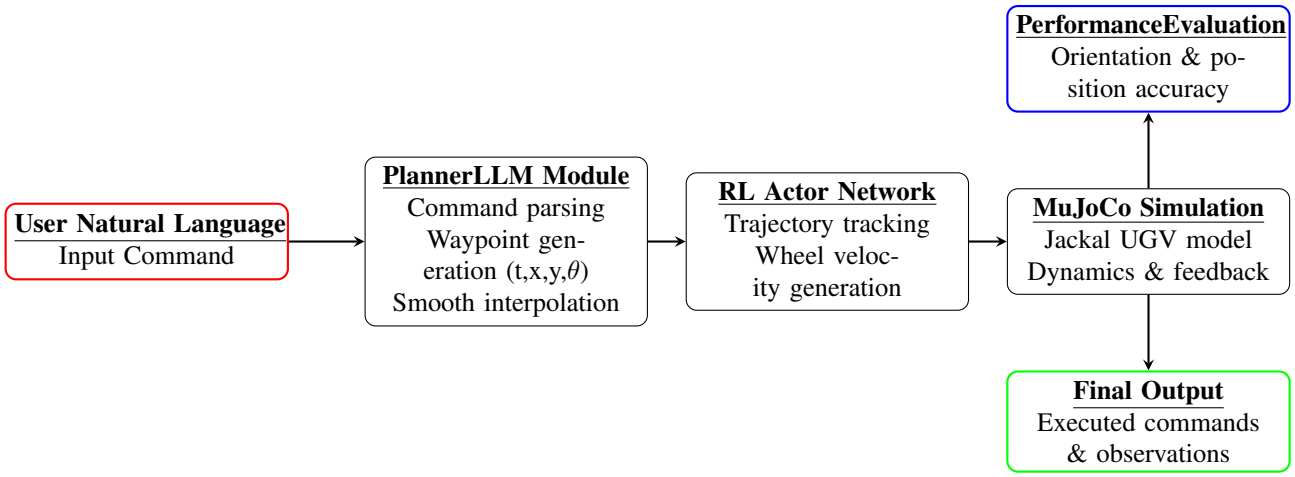


Figure 2: PlannerLLM end-to-end control pipeline: natural language inputs are parsed into waypoints by PlannerLLM, translated into wheel commands by an RL actor, executed in simulation, and then evaluated for performance before producing final outputs.

these possibilities, allowing real-time environment sensing to inform navigation commands directly. Despite these advances, effectively translating high-level, informal instructions into precise robot actions remains challenging, especially regarding precision, adaptability, and real-time performance required in robotic movements.

To address these challenges, we introduce an implementation of **PlannerLLM**, a practical and effective solution structured around a clear two-stage process. Initially, a zero-shot natural language inference (NLI) model classifies the intent of the command using MNLI-based fine-tuning, and then extracts relevant numerical parameters, such as distance, angle, or target position, using regular expressions. These are used to directly invoke one of several intent-specific trajectory generation functions, which produce smooth, discretized sequences of COM states representing the desired robot motion.

Subsequently, generated trajectories are processed by the Actor Network, a reinforcement learning (RL)-trained neural network based on the TD3 algorithm, which translates these trajectories into precise wheel velocity commands. This design targets the non-holonomic Clearpath Jackal robot operating within the MuJoCo simulation environment [7]. In contrast to traditional control techniques such as model predictive control, our approach reduces complexity by implicitly modeling robot dynamics within the learned policy network, improving both execution speed and robustness.

Our primary contributions include the first open-source demonstration of an end-to-end integration of LLM-commanded navigation with a MuJoCo-simulated Jackal UGV. We provide thorough quantitative evaluations of trajectory translation and rotation accuracy across diverse linguistic commands, demonstrating the robustness and effectiveness of our method. Additionally, the modular nature of our framework clearly separates high-level trajectory planning from detailed low-level motor control, offering considerable flexibility for adaptation and integration with various LLM or RL models.

Our methodology advances existing research in language-guided navigation [7] by refining trajectory smoothing and incorporating real-time physical feedback directly into the language-robotics interface. Through detailed visual validations using 3D plotting techniques, we further demonstrate the practical effectiveness and reliability of our system. Ultimately, PlannerLLM significantly simplifies robot control for users without technical expertise, making human-robot interactions more natural, intuitive, and broadly accessible.

2. Methods

2.1. Model Selection

In the formulation, a model defines the state transition which is dependent on the state and action. For autonomous driving problem for clearpath's Jackal UGV, it is modeled using the differential drive model which consists of two wheel velocity inputs and the transition model incorporates converting them to linear and angular velocities which are integrated using euler method.

2.2. Observation Formulation

We build the observation structure such that trajectory tracking can be done with the neural network based controller. The observation $\mathbf{O} = [o_t, o_s, o_m]$ consists of three parts. The first part of the observation o_t consists of the trajectory points to a horizon h. The second part of the observation o_s is the state information which is pose independent. And lastly, the third part of the observation o_m

consists of parameters of the actual robot such as wheel base and wheel radius.

The first part of the observation describes the trajectory information of the environment. For the network to be independent of pose of the system, the trajectory information passed to the network should be egocentric version of the trajectory. To create the egocentric version of trajectory, a slice of the trajectory $\{z_d^i\}$ of length of horizon h is extracted starting at the current timestep. The reference trajectory slice is then transformed to the body coordinates of the robot. The transformations are described as

$$z_d^i = \{z_d^t | t = i, i+1, ..., i+h-1\}$$
$$o_t = \{\mathbf{M}_{world \to body}.[z; 1] | z \in z_d^i\}$$
$$M_{world \to body} = \begin{bmatrix} R(-\theta) & -R(-\theta)^T * p \\ 0^T & 1 \end{bmatrix}$$
$$R(\theta) = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix}$$
$$p = \begin{bmatrix} x_i & y_i \end{bmatrix}$$

where (x_i, y_i, θ_i) are the robot pose in world frame.
The second part of the observation which describes the system information independent of the pose of the robot at current timestep. We use the velocity of Center of Mass (COM) of the robot as the system observation at the current feedback.

$$o_s = \{\nu_i\}$$

The third part of the feedback gives information about the robot which forms the dynamics of the robot as the differential drive model. These act as priors for the transition model which the network learns and are described as

$$o_m = \begin{bmatrix} l_{robot} & w_{robot} & r_w \end{bmatrix}$$

where l_{robot}, w_{robot}, and r_w represents length of vehicle, wheel base of the vehicle and radius of the wheel.

2.3. Reward Design

The reward acts as a feedback from the environment. The environment provides high reward in response to the actions which result in desired states and low reward when action results in undesired state.

For our problem, the reward design was done to incorporate the following goals:

1. Waypoint position and orientation tracking
2. Trajectory lateral drift compensation
3. Smooth control input
4. Smooth velocity

Keeping the goals in mind, the reward is formulated as

$$R(s) = w^t * R_t + w^o * R_o + w^v * R_v + w^c * R_c + w^l * R_l + R_N$$

where R_t, R_o, R_v, R_c, R_l, and R_N represent the position penalty, orientation penalty, velocity penalty, control penalty, lateral drift penalty and terminal reward respectively.
For the formulation of R_t, R_o, and R_v, we calculate the Euclidean distance between the reference waypoint and the respective system state. For R_c, we use the negative norm of min-max normalized action to penalize the large inputs.
For R_l, we calculate the perpendicular distance between waypoint vector and the robot at current timestep. And R_N is a exponential reward centered at the final waypoint and activates in a certain range of final waypoint.

2.4. PlannerLLM Trajectory Generator

The core trajectory generator is defined in `improved_jackal_trajectory.py` and imported via `from improved_jackal_trajectory import JackalTrajectoryGenerator`

An instance is created and its internal position is reset with `gen = JackalTrajectoryGenerator (model_path="facebook/bart-large-mnli")` `gen.reset_position()`. Calling `trajectory = gen.generate_trajectory(command_text)` returns a `List[np.ndarray]` of discrete COM waypoints $[x, y, \theta]$. Internally, `generate_trajectory` proceeds as follows:

1. **Intent classification.** The input string is lowercased and passed through a zero-shot NLI pipeline to select one of the predefined intents (e.g., "move forward", "turn left", "go to", "stop").
2. **Parameter extraction.** Distance, angle, or position values are parsed via regular expressions (e.g., `self.distance_pattern`).
3. **Trajectory assembly.** The appropriate builder function (e.g., `_generate_forward_trajectory`, `_generate_left_turn_trajectory`, `_generate_goto_trajectory`) divides the motion into time-stepped segments based on `max_linear_speed`, `max_angular_speed`, and `time_step`, producing a sequence of COM offsets.
4. **State management.** The methods `reset_position()` and `set_position(np.ndarray)` allow external control of the generator's internal $[x, y, \theta]$ state between calls.

2.5. Integration Pipeline

The high-level planner is invoked via `jackal_nlp_interface.py`, imported as `from jackal_nlp_interface import get_trajectory, reset_position`

A new command session begins with `reset_position()` `trajectory = get_trajectory("move forward 1 meter")` which returns a list of $[x, y, \theta]$ waypoints. In the integration script, these waypoints feed into the wheel-command model, with a basic interface built in for testing.

This design cleanly decouples natural-language parsing and trajectory generation from wheel-level actuation, allowing either component to be replaced without altering the interface for issuing or processing waypoints.

2.6. NLP Intent Trainer

The intent classification component is realized by fine-tuning a pretrained sequence-classification model on a custom labeled dataset of user commands. We employ the Hugging Face Transformers `Trainer` API to optimize the model parameters via supervised learning.

Input examples consist of pairs (x_i, y_i), where x_i is a user-issued command string and $y_i \in \{1, \ldots, K\}$ its intent label. The dataset is split into training and validation sets (typically 80/20). Texts are tokenized using an `AutoTokenizer`, with truncation to a maximum sequence length of 128 tokens and dynamic padding per batch via `DataCollatorWithPadding`.

We load `facebook/bart-large-mnli` and replace its classification head to output K logits. The resulting model θ produces class probabilities

$$p_i = \mathrm{softmax}\big(W_{\mathrm{cls}}\, h_{[\mathrm{CLS}]} + b_{\mathrm{cls}}\big)$$

where $h_{[\mathrm{CLS}]}$ is the pooled hidden state.

Training minimizes the cross-entropy loss

$$\mathcal{L}(\theta) \;=\; -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K} \mathbf{1}\{y_i = k\}\, \log p_{i,k}\,.$$

We use the AdamW optimizer with weight decay 0.01, an initial learning rate of 1×10^{-5}, and a linear scheduler with warmup over 10% of total steps. Mini-batches of size 64 are processed for 32 epochs. Gradients are clipped to a maximum global norm of 1.0 (`max_grad_norm=1.0`) to stabilize training.

Every 500 training steps, the model is evaluated on the validation set. Metrics logged include classification accuracy and macro-averaged F1-score. The checkpoint with the lowest validation loss is retained as the best model; early stopping is triggered if no improvement occurs over 5 consecutive evaluations.

At inference time, the fine-tuned model tokenizes each incoming command, computes logits, and selects

$$\hat{y} = \arg\max_{k}\; p_k\,,$$

providing a robust intent classification layer for downstream trajectory generation.

2.7. Environment and MuJoCo Simulation

The learning environment was build using the OpenAI Gymnasium by implementing the elements described above. We choose an horizon h of 10 steps.
Each waypoint consists of (t, x, y, θ) which results into a 40-D vector to represent the reference trajectory in the observation. And observation encodes the 2-D state velocities in the form of linear and angular velocities and a 3-D dynamics parameters. Hence, each observation is of size 45.
Details of the environment is described below.

- **JackalEnv:** A custom Gymnasium MuJoCo environment implementing the Jackal UGV kinematics and dynamics (see `jackal_env/jackal_env.py`).
- Standard States: $(t, x, y, \theta, \dot{x}, \dot{y}, \dot{\theta})$.
- States: $(z_i^{body}, \nu_i, \theta_{dynamics})$.
- Actions: differential wheel velocities $(v_L, v_R) \in [-2, 2]$.

2.8. Policy

We choose TD3 algorithm and the implementation done in stable-baselines3 library. We used the multi-layered perceptron (MLP) to build the policy network(actor) and the value network(critic).
The policy network has the shape of $(256, 256, 128, 128, 64, 64)$ and the critic network has the shape of $(1024, 512, 256, 128)$. The batch size was set to 4096 with training frequency set to 4 episodes, soft update factor to 0.0001 and the discount factor to 0.99. The training was done for $3 * 10^6$ total timesteps. The network was trained by sample trajectories of 6 secs with time steps of 0.01.

3. Experimental Results

This section contains a brief overview of the experiments done to train the models.

3.1. NLP Training

The NLP component was trained to classify natural language commands into predefined motion intents using the `facebook/bart-large-mnli` model. After evaluating smaller transformer models, such as `distilbert-base-uncased`, which failed to differentiate between intent classes during training, we transitioned to a more robust architecture. The chosen model was fine-tuned on a curated set of labeled commands corresponding to basic robot maneuvers (e.g., user intents such as "move forward" and "turn left"). Training accuracy and validation confirmed the model's suitability for zero-shot classification. As you can see from Figure 3, the model is generally well trained by 50 steps, and after 200 the loss is less than 0.0001. For time-independent tracking, left and right turns achieve 100% success but exhibit instability after reaching the target, while forward movement succeeds only 37% of the time. With time-dependent tracking, left and right turns fail completely (0%) and forward targets are reached only 8% of the time.

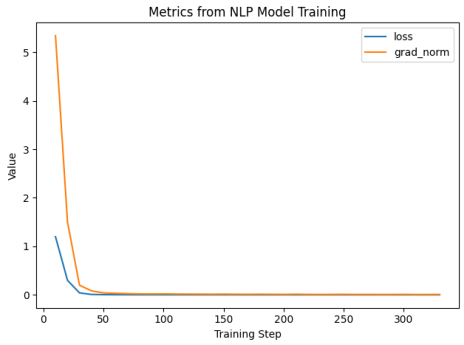
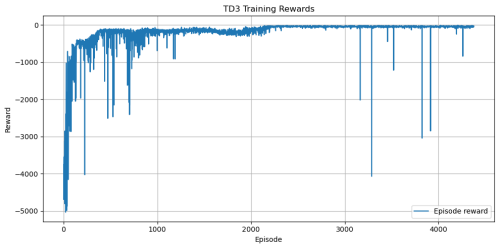
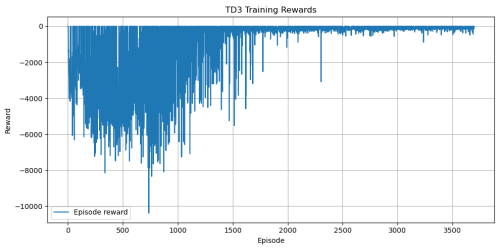


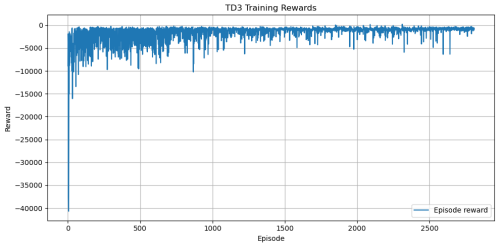
Figure 3: Loss and Gradient Norm (before clipping) over time during Training of the NLP



(a) Reward: Tracking Error Only + $1e^{-4}$ Learning Rate

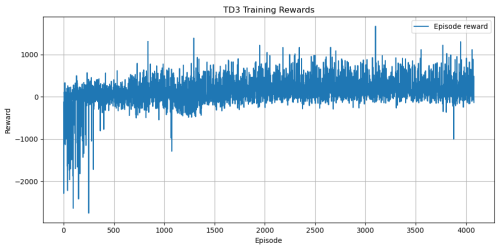


(b) Tracking Error Only + $1e^{-6}$ Learning Rate

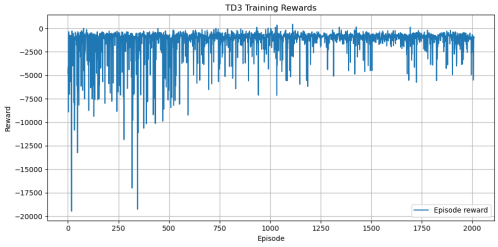


(c) Tracking Error + $1e^{-4}$ Learning Rate + Fine Tuned Velocity Control

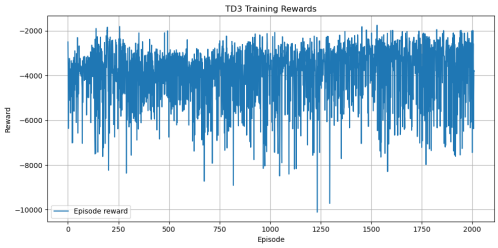
Figure 4: Evaluation Results for the main three hyperparameter settings.



(a) Experiment: 1e-6, 2.0 0.01



(b) Experiment: 1e-6, 2.0 0.05 0.75



(c) Experiment: 1e-4, 2.0 0.05 with Exponential Gain to Target

Figure 5: Few experiments in training [Each name describes learning rate, weight of control and weight of lateral drift in order.].

3.2. RL Network Training

The reinforcement learning model was implemented using the Twin Delayed Deep Deterministic Policy Gradient (TD3) algorithm, selected for its stability and proven performance in continuous control tasks. The actor network was trained to follow randomly generated stable trajectories and later fine tuned to reference trajectories generated from the NLP stage by converting them into corresponding wheel velocity commands suitable for the Jackal UGV's dynamics.

Multiple reward formulations were tested to refine control precision, particularly in turning behavior, which consistently proved to be a limiting factor. Reward components included: progress toward the goal, smoothness of motion (penalizing erratic acceleration or control spikes), forward velocity incentives, and lateral drift penalties. Four main reward weight variations were explored alongside different

learning rate configurations. Despite these efforts, turning behavior remained inconsistent, indicating the need for further tuning or structural adjustments to the training procedure.

4. Discussion

The objective of the project was to implement a neural network-based trajectory tracking controller for the Jackal UGV. For the project, TD3 algorithm was chosen for training the actor and the critic network. The focus of the training was not only positional tracking but also achieving smooth control, acceptable velocities, minimal lateral drift and a stable stop at the final waypoint.

As shown in the experiments section, the agent was able to learn a trajectory-following policy when only positional error was penalized. However, introduction of additional penalties such as control penalties, velocity penalties and lateral-drift penalties, prevented the model from learning early and convergence was seen at a worse policy or the same initial reward associated policy or there was no convergence with unstable training behavior throughout the episodes.

To add other penalties and rewards to the policy, the policy network was fine-tuned with each additional penalty until convergence or maximum steps of the environment. Following this paradigm of training the policy network still proved difficult to train the network to do stable and repeatable behaviors. Penalizing velocities and control inputs especially proved challenging to incorporate.

Based on the experiments and trainings, the report showcases combination of penalties and rewards is difficult to navigate and is very sensitive to the gains of control and velocity penalties especially. Interestingly even small changes in the gains, make the rewards conflicting preventing the policy to converge to a stable version. Also, the addition of a linear or an exponential reward to reach the final waypoint and stopping could not help improve the performance of the policy.

Aside from the reward design, the TD3 algorithm hyperparameters are affecting the training although default values have shown to be quite stable as well in other tasks. Nevertheless, hyperparameters of the TD3 algorithm were also experimented with slightly - the learning rate and the soft update factor. The changes to the hyperparameters of the algorithm lead to minimal effects to the training of the policy network.

Also, an interesting observation, the learned policy does achieve the final goal and tracks the trajectory although time independent but does not have a stable behavior at the end of the trajectory as stated before, maybe suggesting that the policy is stuck in a local minimum of the reward function.

Incorporating all the challenges in tuning the reward function hyperparameters, it begs to say that there is a structural challenge in the reward design paradigm than just mere tuning of gains. There may be better penalties which would propel the training to reach the waypoints faster and in a stable manner.

The few occasions that the training was able to converge was when only the tracking error was used in the reward function. This highlights the benefit of a clear, consistent reward signal directly linked to the observable outcomes. Once additional penalties and rewards were introduced, the reward landscape became more complex and potentially overwhelming and unstable for the agent's capacity to associate actions with future rewards.

The future work of this project would be to better formulate the reward, try alternate algorithms to train the networks and maybe incorporating heavier terminal costs as well as trying out physics informed training as well.

Overall these results emphasize the challenges with reward design in RL for multi-objective tasks and suggest that further exploration of algorithmic and structural improvements is necessary for robust trajectory tracking.

5. Contributions

My contribution in the project includes making the environment, reward designing, training the actor-critic network using the TD3 implementation of stable-baselines3 library. I also conducted the experiments, integrated the planner with the actor network for evaluations and conduct experiments both in training and evaluation.

Code and Data Availability

The complete source code for all environment definitions, training scripts, sample NLP Training files, and evaluation tools presented in this work is freely available on GitHub at:
https://github.com/Ketan13294/Project8-NNJackal/.

References

[1] X. Yinda. et al. DRL-Based Trajectory Tracking for Motion-Related modules in Autonomous Driving. arxiv:2308.15991, 2024

[2] A. Ahmed. MobRobGPT GitHub repository. 2024.

[3] E. Latif. 3p-llm: Probabilistic path planning using large language models. arXiv:2501.15214, 2024.

[4] S. Meng et al. LLM-A*: LLM enhanced incremental heuristic search for path planning. arXiv:2407.02511, 2024.

[5] L. Nwankwo and E. Rueckert. The conversation is the command. HRI '24, ACM, 2024.

[6] OpenAI. GPT models. 2024.

[7] B. Pan et al. LangNav: Language as a perceptual representation for navigation. 2024.

[8] J. Tang et al. Zero-shot robotic manipulation with language-guided planning. arXiv:2403.18778, 2025.