LaMPilot: An Open Benchmark Dataset for Autonomous Driving with Language Model Programs

Yunsheng Ma[1*], Can Cui[1*], Xu Cao[2*], Wenqian Ye[3], Peiran Liu[1], Juanwu Lu[1],
Amr Abdelraouf[4], Rohit Gupta[4], Kyungtae Han[4], Aniket Bera[1], James M. Rehg[2], Ziran Wang[1]
[1]Purdue University [2]University of Illinois Urbana-Champaign
[3]University of Virginia [4]InfoTech Labs, Toyota Motor North America
{yunsheng, cancui, liu3820, juanwu, aniketbera, ziran}@purdue.edu,
{xucao2, jrehg}@illinois.edu, wenqian@virginia.edu,
{amr.abdelraouf, rohit.gupta, kt.han}@toyota.com

Abstract

*We present **LaMPilot**, a novel framework for planning in the field of autonomous driving, rethinking the task as a code-generation process that leverages established behavioral primitives. This approach aims to address the challenge of interpreting and executing spontaneous user instructions such as "overtake the car ahead," which have typically posed difficulties for existing frameworks. We introduce the LaMPilot benchmark specifically designed to quantitatively evaluate the efficacy of Large Language Models (LLMs) in translating human directives into actionable driving policies. We then evaluate a wide range of state-of-the-art code generation language models on tasks from the LaMPilot Benchmark. The results of the experiments showed that GPT-4, with human feedback, achieved an impressive task completion rate of 92.7% and a minimal collision rate of 0.9%. To encourage further investigation in this area, our code and dataset will be made available.*

1. Introduction

Making wise and informed decisions in complex traffic scenarios is crucial for autonomous vehicles. It necessitates a common-sense understanding of traffic and knowledge about the surrounding environment. As a result, Large Language Models (LLMs) have gained popularity among researchers in this field due to their comprehensive knowledge of the world [19] and strong reasoning abilities [50]. The use of LLMs in autonomous driving, particularly for decision-making and motion planning, is gaining momentum [5, 10, 25, 32, 51, 54].

To generate a specific task plan for an autonomous vehicle, existing planners rely on clear objectives and constraints to guide their decisions. However, when faced with arbitrary commands like ”overtake the car in front of me,” existing planners struggle to handle them effectively. This highlights the potential of LLMs to address such challenges.

Nevertheless, there are several limitations associated with the use of LLMs in autonomous driving:

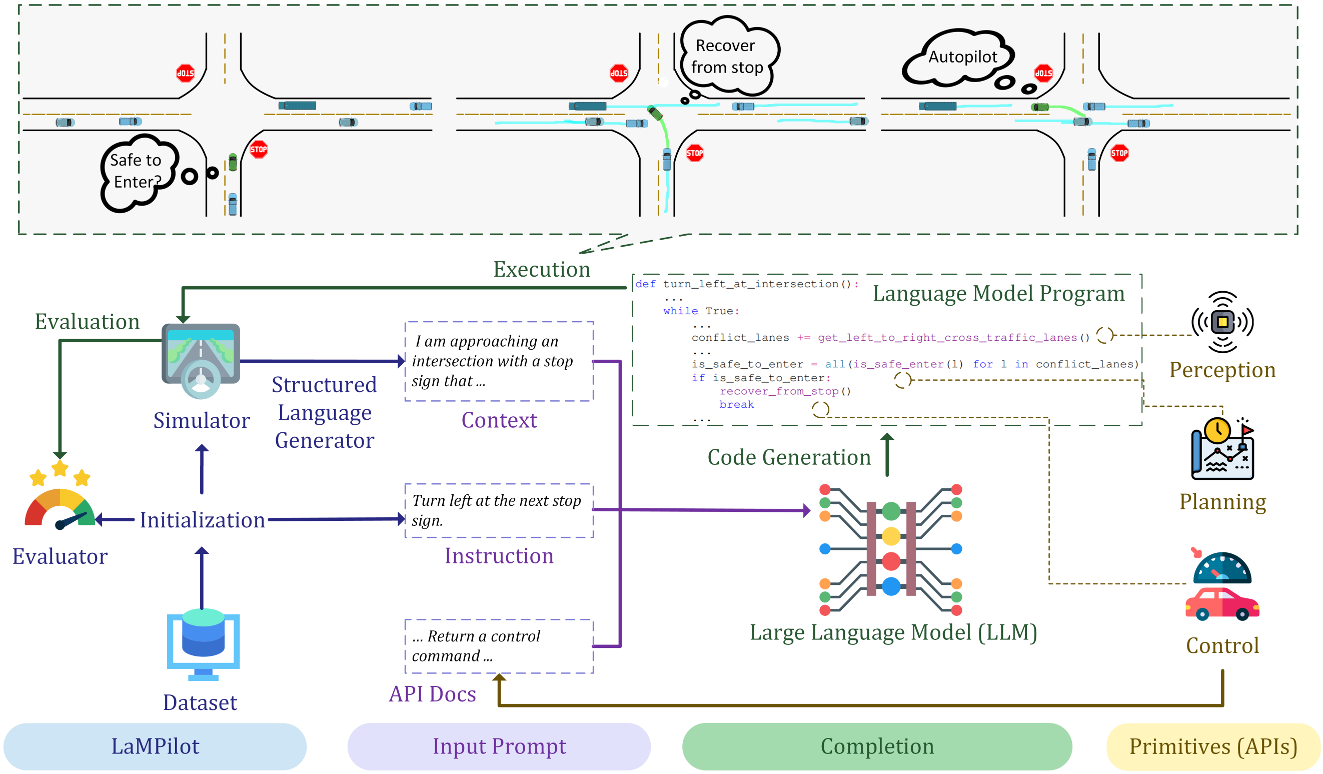
- There is a lack of datasets specifically for evaluating and comparing the capacity of LLM-based models in the context of autonomous driving.
- Controlling autonomous vehicles requires careful consideration due to its safety-critical nature. Existing frameworks often prioritize successful execution of actions predicted by LLMs without adequately addressing safety concerns.

To address these gaps and effectively implement LLMs in autonomous driving, we propose a novel framework called LaMPilot. LaMPilot is the first interactive environment and dataset designed for evaluating LLM-based agents in a driving context. The dataset consists of 4.9K human-annotated instruction-scene pairs with diverse language styles, including full sentences and concise commands.

Taking inspiration from Code as Policy [20], which utilizes code-writing LLMs to write robot policy code, LaMPilot uses Language Model Programs (LMPs) as the action space instead of low-level control commands like acceleration and steering. This decision is motivated by the inherent capability of programs to represent temporally extended and compositional actions, which are crucial for addressing complex, long-horizon driving tasks such as *overtaking*.

Specifically, we equip LLM-based agents with APIs that cover various functional driving primitives, including perception-related functions like *object detection* and vehicle control functions such as *lane keeping*. The key idea of the LaMPilot framework is to enable LLM agents to connect natural language instructions to detailed strategic actions through code generation. The code snippets serve three main purposes:

[*] Equal Contribution



I am approaching an intersection with a stop sign that ...

Turn left at the next stop sign.

... Return a control command ...

```
def turn_left_at_intersection():
    ...
    while True:
        ...
        conflict_lanes += get_left_to_right_cross_traffic_lanes()
        ...
        is_safe_to_enter = all(is_safe_enter(l) for l in conflict_lanes)
        if is_safe_to_enter:
            recover_from_stop()
            break
        ...
```

Figure 1. This diagram shows the workflows of the LaMPilot framework, illustrating the process from initialization to evaluation. Once initialized, a structured language generator interprets the simulator environment to establish the language context. The central LLM handles input prompts, utilizing predefined APIs and generating LMP. The LMP is then executed in the simulator. The executed policy is assessed by the evaluator, which provides a final score for the policy.

- Interact with perception APIs to gather relevant environment information.
- Plan future movements to fulfill human commands (tasks).
- Parameterize control APIs to actuate the plan.

The overarching objective for agents operating within the LaMPilot framework is to accomplish assigned tasks in a safe and efficient manner. Additionally, LaMPilot incorporates an interactive simulator for evaluation purposes, featuring programmatic scoring mechanisms to assess policy performance. Furthermore, the simulator offers the flexibility of generating scenarios that can be controlled and tuned as needed. To the best of our knowledge, LaMPilot is the first of its kind in executing and evaluating language model programs in autonomous driving.

The contributions are summarized as follows:

- **LaMPilot Benchmark**: We introduce the first benchmark for evaluating LLM-based agents in autonomous driving contexts.
- **Interactive Simulation Environment**: The framework includes an open simulator for executing policies, equipped with scoring mechanisms.
- **Comprehensive API Suite for Driving Primitives**: LaMPilot provides LLM agents with a set of APIs that cover essential driving functions, where safety criteria are loaded into the APIs.
- **Connecting Natural Language to Strategic Actions**: LaMPilot enables the translation of natural language instructions into actionable strategies through code generation.

2. Related Work

2.1. Large Language Models for Robot Planning

The use of language in planning tasks has a longstanding history in robotics, dating back to the use of lexical parsing in natural language for early demonstrations of human-robot interaction [52], and it has been widely studied being used in the robotics area. There exists comprehensive review works on this topic [22, 40]. The ability of robotic systems to generalize to new tasks through language-based planning and control has been demonstrated in various works [1, 13]. Achieving specific planning or control tasks or policies, including model-based [2, 27, 34], imitation learning [23, 35], and reinforcement learning [11, 14, 26], has been extensively explored.

Recently, Using LLMs for code generation in robotics planning has attracted increasing attention. Voyager [43] introduces lifelong learning by incorporating three key components: an automatic curriculum that maximizes exploration, a skill library for storing and retrieving complex behaviors, and a new iterative prompting mechanism that generates executable code for embodied control. Socratic Models [56] employs visual language models to replace perceptual information within the language prompts used for robot action code generation. [20] introduces an approach that uses LLMs to directly generate policy code for robots to do control tasks, specify feedback loops, and write low-level control primitives. However, the utilization of LLMs for code generation in the autonomous driving domain remains relatively underexplored. Our primary objective is to pioneer this field and bridge the existing gap.

2.2. Language in Driving

Recent studies have highlighted the significant role of language interaction in enhancing self-driving technology. Companies like Wayve are using natural language to improve the learning and explainability of driving models, with their LINGO-1 system integrating vision, language, and action [48]. Other approaches [8, 9, 15, 54] use LLMs in various aspects of autonomous driving, from vehicle control and trajectory planning to integrating detailed imagery and language for better decision-making. Safety and explainability are also key in these applications. For instance, Talk2Car [7] demonstrated the use of LLMs to identify specific objects based on natural language commands. LanguageMPC [32] incorporated LLMs to explain planned actions or adjust low-level control parameters. However, most existing models either provide high-level, non-executable instructions like turning and lane changing or involve very detailed planning and control tasks, such as adjusting steering angles. Our work is unique in using LLMs to generate codes that control autonomous driving operations, fully leveraging their reasoning capabilities for a more efficient driving experience.

2.3. Large Language Models for Code Generation

With the advent of powerful large language models, programming languages are able to be leveraged to perform various tasks across many domains such as code generation [55], information extraction [44, 45], robotics [12, 20, 36], and vision [24, 39, 53]. Furthermore, code provides a foundation for logical planning [21, 46, 47], which when leveraged by LLMs, has produced ground breaking results in many longitudinal tasks. This combination of LLMs and code has revealed a new paradigm that can be applied to numerous engineering and scientific domains.

3. LaMPilot Benchmark

3.1. Problem Statement

We define the benchmark $\mathcal{B}$ as the tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, I \rangle$, comprising:

- $\mathcal{S}$: The comprehensive set of all possible states.
- $\mathcal{A}$: The set of actions that can be executed by the ego vehicle.
- $\mathcal{T}$: The transition model, formulated as $\mathcal{T} : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$, encapsulating the dynamics of the environment.
- I: High-level instructions that guide the planning for the ego vehicle.

For each task in benchmark $\mathcal{B}$, we begin from an initial state $\mathbf{b}$, aiming to reach a goal state in $\mathbf{g} \in \mathcal{G}$. The benchmark outlines a policy rollout for each task. This rollout is steered by the conditional probability $p(a_{t+1}|s_t, I)$ and concludes upon arrival at a goal state $g \in \mathcal{G}$. Note that the agent does not have direct access to the goal state $\mathbf{g}$, but only a high-level instruction I.

Consider the instruction I =*"Make a right lane change"*. If the initial state $\mathbf{s}$ includes the ego vehicle in the current 5lane $l \in \mathbf{s}$, the policy `set_target_lane (get_right_lane(ego))` will enact a series of state-action pairs. This process transitions the vehicle from its current state $l \in \mathbf{s}$ to a new state $l' \in \mathbf{s}'$, where l' is the lane immediately to the right of l, assuming such a lane exists.

In this context, a goal state $\mathbf{g} \in \mathcal{G}$ might be defined by specific criteria that confirm the ego vehicle's proper placement in the right lane l'. This would indicate that the sequence of actions has successfully transitioned the vehicle from $l_0 \in \mathbf{s}$ to $l'_0 \in \mathbf{s}'$, consistent with instruction I.

3.2. LaMPilot

Based on the problem statement, we present LaMPilot, a new benchmark designed to evaluate instruction following in autonomous vehicles. LaMPilot consists of three key components: a simulator, a dataset, and an evaluator. Within LaMPilot, driving policies are articulated through Python Language Model Programs (LMPs). The primary approach involves creating prompts based on human-annotated instructions within LaMPilot. These prompts are processed by LLMs, which generate corresponding code. This code is then executed and its performance is evaluated within the LaMPilot framework, as shown in Fig. 1. In this subsection, we will provide an overview of the simulator and the dataset, while the evaluator will be discussed in Sec. 3.6.

Simulator The simulator is an essential component of the LaMPilot environment and is built on HighwayEnv [18], which is widely used in research related to autonomous driving and tactical decision-making. It offers various driving models and simulates realistic interactions between multiple vehicles. Originally designed for training reinforcement learning agents, HighwayEnv is expanded in LaMPilot in this study to include interfaces suitable for LLM-based agents. In addition, we implement custom intersections with different configurations to increase the diversity of decision-making scenarios. This expansion complements the existing "highway" environment and enhances the overall functionality of the simulator, enabling a wider range of driving scenarios to be evaluated. The simulation

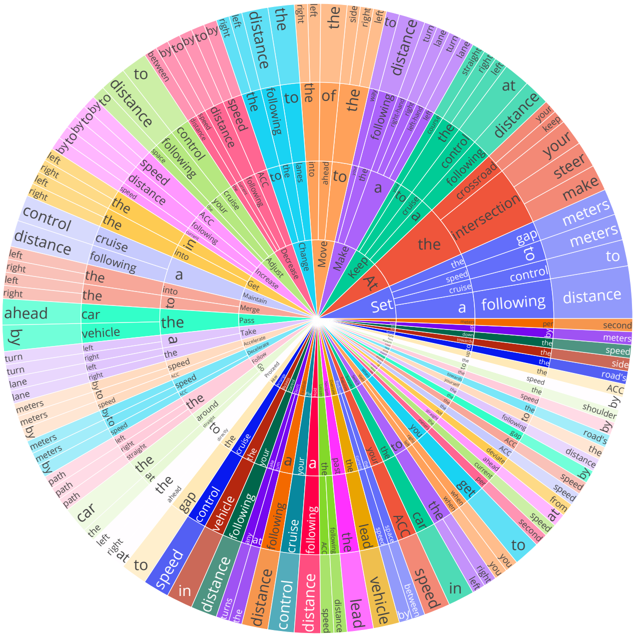


Figure 2. Instruction distribution in the LaMPilot dataset.

includes randomly generated traffic patterns with various density settings, covering both congested peak traffic hours and less crowded road conditions.

Dataset The LaMPilot dataset consists of 4,900 human-annotated traffic scenes, where each data sample includes three elements:

- An instruction I, which is the high-level task description.
- An initial state $\mathbf{b}$, utilized to initialize the simulator.
- A set of criteria for determining goal states $\mathcal{G}$, which aligns with I.

The dataset is divided into three sets: training (3,900 samples), validation (500 samples), and testing (500 samples). The dataset contains a variety of instructions that reflect the diversity and unpredictability of real-world requirements. These instructions are categorized by maneuver types, such as turning, lane changing, and overtaking, as well as scenario types, including both highway and intersection settings. Fig. 2 shows a distribution of the first four words in the instructions, highlighting the diversity of instructions in our dataset. The detailed statistics are shown in Tab. 1.

3.3. Functional Primitives

Driving policies involve API calls to functional primitives, which are specialized APIs designed to support LLMs in generating actionable plans based on instructions. The main objective is to utilize LLMs for strategic planning while intentionally avoiding their direct involvement in low-level control tasks. Due to the autoregressive nature of LLMs, generating long completions can introduce significant latency. This characteristic makes them less suitable for time-critical tasks like object avoidance, which require quick responses. To address this, our API design strategically offloads safety-critical tasks, allowing LLMs to focus on decision-making based on common sense, which aligns better with their capabilities [49].

Statistic	Value
Total scenarios	4,900 (100%)
Distance-related	1,200 (24.5%)
Speed-related	1,200 (24.5%)
Pulling over	200 (4.1%)
Routing	1,500 (30.2%)
Lane changing	400 (8.2%)
Overtaking	400 (8.2%)
Highway	3,400 (69.4%)
Intersection	1,500 (30.6%)
Average Instruction Length	7.6
Maximum Instruction Length	14
Minimum Instruction Length	2

Table 1. Main statistics in the LaMPilot dataset.

Our suite of functional primitives is developed with insights from Responsibility-Sensitive Safety (RSS) [33]. These APIs facilitate the construction of driving policies that minimize the risk of accidents resulting from improper policies. This approach sets our work apart from many existing initiatives that may prioritize API functionality [56] without a strong emphasis on safety feasibility.

The API suite is categorized into four main types: (1) Ego APIs: These APIs provide information about the status of the ego vehicle, such as its speed and position; (2) Perception APIs: These APIs include functions like object and lane detection, which can be called to acquire information about the surrounding environment; (3) Planning APIs: These APIs offer capabilities to generate routes when provided with a destination; (4) Control APIs: These APIs translate LLM-generated code into low-level commands for the vehicle, implementing proper responses [33] to avoid collisions. In the event of an exception during the execution of a generated policy, the ego vehicle automatically switches to a predefined `autopilot` mode to prevent undefined behaviors. The full list of APIs is provided in the Appendix.

3.4. Input Prompt

In addition to the main instructions, we also provide environment contexts and API Docs as part of the input prompt for LLMs.

Environment Contexts The environment contexts include relevant information about the driving environment. These details are represented as numerical vectors in the simulator. We develop an interface that allows LLMs to use this information by converting the numerical vectors into natural language descriptions. These vectors contain semantic attributes from the simulation, such as information about other vehicles on the road, the ego vehicle, and the map. This approach sets our work apart from other recent advancements in the field [10], where numerical vectors are directly fed into LLMs without any contextual translation. Specifically, we employ a structured language generator, as illustrated below:

$$g(\mathbf{s}) \rightarrow \tag{1}$$

$$\left\{\begin{array}{l}\text{My current speed is 31.3 m/s.}\\ \text{I am driving on a highway with 5 lanes in my direction.}\\ \text{I am in the 4th lane from the right.}\\ \text{The right-most lane is an emergency lane.}\\ \text{There is a car in front of me in my lane,}\\ \text{at a distance of 25.7 m, with a speed of 22.1 m/s.}\\ \vdots\end{array}\right.$$

This function converts the numerical data of the state vector into a narrative format. This narrative does not require any additional fine-tuning for interpretation by the LLM. It provides a comprehensive information of the agent's environment, including other road users, ego-vehicle state, and relevant map information.

API Docs Consider the example instruction *make a right lane change*, where an LLM might suggest the action `change_lane_right`. However, the controller may not directly support this as a primitive action. To bridge this gap, we include API documentation in our prompts. This documentation equips LLMs with essential information about the available APIs and guides their correct usage. These documents include not only the input and output specifications but also provide descriptions of their usage and the logic underlying these functions. By integrating API documentation into the prompt, we ensure that the policy code generated by the LLM is aligned with the actual capabilities of the ego vehicle, thereby facilitating executable action plans.

3.5. Completion and Execution

As shown in Figure 1, the LLM receives prompts and is responsible for generating a completion. The completion produced by the LLMs is anticipated to be valid functions, written using the provided APIs. These functions could range from straightforward, one-off functions to more complex generator functions. A simple example is demonstrated below, where the target speed of the ego vehicle is altered:

```python
def decrease_speed_by_5():
    set_target_speed(get_target_speed() - 5)
```

Additionally, LaMPilot supports the use of control structures such as *if-else* and *loop* statements, enabling the LLM to create dynamic feedback policies. An example of this is a while-loop for making a left lane change:

```python
def make_left_lane_change():
    ...
    while True:
        if is_safe_enter(left_lane):
            set_target_lane(left_lane)
            break
        yield autopilot()
```

In this example, the policy continuously evaluates the feasibility of changing to the left lane. It executes the lane change when conditions are safe; otherwise, it defaults to the `autopilot` function.

To execute an LMP in LaMPilot, the Python `exec` function is used. This function takes the LMP code as an input string, along with two dictionaries that define the execution scope: (i) `apis`, which includes all APIs the code may invoke, and (ii) `policies`, an initially empty dictionary that will eventually contain the synthesized functions and a `policy` variable. If the LMP is designed to return a generator, this generator is extracted from the locals dictionary after the execution of the `exec` function.

3.6. Evaluator

Our framework evaluates driving performance from the initial state $\mathbf{b}$ to the goal state $\mathbf{g} \in \mathcal{G}$ using three critical metrics: safety, comfort, and efficiency.

Safety Metric We employ the time-to-collision (TTC) as the primary safety metric. It measures the vehicle's compliance with traffic regulations and similarity to human driving behaviors by assessing its capability to maintain safe distances and react appropriately to avoid collisions. In a scenario with $n+1$ vehicles, including the ego one. We utilize the label i to specifically refer to each vehicle, with 0 being the ego vehicle. For a single time step with state $\mathbf{s}$, the time to collision (τ_i) with vehicle i is calculated using its velocity $\mathbf{v}_0 \in \mathbf{s}$ and position $\mathbf{p}_0 \in \mathbf{s}$, compared against another vehicle with velocity $\mathbf{v}_i \in \mathbf{s}$ and position $\mathbf{p}_i \in \mathbf{s}$, where $1 \leq i \leq n$. The formula is:

$$\tau_i = -\frac{(\mathbf{p}_0 - \mathbf{p}_i) \cdot (\mathbf{v}_0 - \mathbf{v}_i)}{\|\mathbf{v}_0 - \mathbf{v}_i\|^2} \tag{2}$$

To compute the TTC score, the shortest positive time to collision value τ_{min} among all n other vehicles and all the time steps is selected to represent the nearest potential collision. We use t to index each time step and T to denote the actual task completion time. Therefore, for all τ_i^t where i ranges from 1 to n and t ranges from 1 to T, the τ_{min} can be calculated as:

$$\tau_{\text{min}} = \min_{1 \leq i \leq n,\, 1 \leq t \leq T} \tau_i^t \tag{3}$$

TTC scores are given empirically based on a 2-second safety margin [6], with scores above 2 seconds considered safe and rated as 100. Specifically, the TTC score is calculated as follows:

$$\text{TTC} = \begin{cases} 100 & \text{if } \tau_{\min} > 2 \\ 100 - \frac{1}{\tau_{\min}} & \text{if } 0 < \tau_{\min} \leq 2 \end{cases} \tag{4}$$

Comfort Metric Speed variance (SV) is used to assess the smoothness of speed transitions, reflecting the comfort level of the driving policy.

We first need to calculate the speed standard deviation σ_0 for the ego vehicle. For each time step t, where $1 \leq t \leq T$, we calculate the σ_0 as follows:

$$\sigma_0 = \sqrt{\frac{\sum_{t=1}^{T} (\|\mathbf{v}_0^t\| - \mu_0)^2}{T}}, \tag{5}$$

where μ_0 is the mean speed for the ego vehicle:

$$\mu_0 = \frac{\sum_{t=1}^{T} \|\mathbf{v}_0^t\|}{T} \tag{6}$$

Then, the SV score for is defined as:

$$\text{SV} = 100 \cdot \frac{\sigma_0}{\sigma_{\text{comfort}}}, \tag{7}$$

where σ_{comfort} is the maximum comfort speed deviation.

Efficiency Metric Time efficiency (TE) score evaluates the time efficiency of the policy code, calculated as follows:

$$\text{TE} = 100 \cdot T / T_{\text{limit}} \tag{8}$$

Here, T denotes the actual task completion time, and T_{limit} is the predefined time limit.

Task Completion Criteria A task is considered complete when the ego vehicle successfully translates from the initial state $\mathbf{b}$ to a goal state $\mathbf{g} \in \mathcal{G}$. Specifically, it needs to fulfill specific conditions. For example, a lane change task is completed when the vehicle is in the target lane, and its heading aligns with the lane's direction within a defined threshold. The full list of criteria is outlined in the Appendix.

Overall Scoring The final score is an aggregate of all individual metrics:

$$\text{Score} = W_{\text{TTC}} \cdot \text{TTC} + W_{\text{SV}} \cdot \text{SV} + W_{\text{TE}} \cdot \text{TE} \tag{9}$$

The contribution of each metric to the overall score is determined by its respective weight W.

4. Baselines

4.1. Heuristic Baselines

In autonomous driving, rule-based models have been favored for their deterministic and interpretable nature. In this context, we employ two rule-based baseline policies: the Intelligent Driver Model (IDM) [42] and the Minimizing Overall Braking Induced by Lane Changes (MOBIL) principle [17]. IDM describes a rule for updating the acceleration of a vehicle to avoid collisions based on the proximity and relative velocity of the vehicle to the object directly in front. According to MOBIL, a lane change is executed only if the prospective new lane offers a more favorable driving scenario and the maneuver can be conducted safely. These baselines can be considered as operating on *random chance*, as the policy is independent of user instructions but follows predefined rules.

Additionally, we include a *human performance* baseline, where a licensed human driver controls the vehicle in the simulation using arrow keys on the keyboard, following the commands and visuals displayed. This baseline provides a reference for human-level performance on the LaMPilot benchmark.

4.2. Zero-Shot and Few-Shot Baselines

We conduct benchmarking using a wide range of state-of-the-art large language models, which include both open-source and proprietary solutions. The evaluated models are: Llama 2 [41], PaLM 2 [3], ChatGPT [28], GPT-4 [30], and GPT-4 Turbo [29]. The zero-shot setup is configured the mapping from API descriptions (A), user instructions (I), and the driving context (C) to executable program code (P), where the input prompt consists of concatenated tokens from (A), (I), and (C) generated by the structured language generator in Eq. (1). The output aims to provide executable program code (P) that follows the user instructions in the given driving context. In the few-shot setting, we follow the standard paradigm [4] by including k human-written exemplars $\{I_i, C_i, P_i\}_{i=1}^{k}$ before the test instance. These in-context examples help the model adapt to the tasks in the LaMPilot benchmark.

4.3. Human Feedback Baselines

LLMs have demonstrated proficiency in generating coherent solutions across various tasks without requiring additional fine-tuning. However, when it comes to code generation, especially for complex scenarios, they can produce suboptimal results. The autoregressive nature of these models poses a significant challenge, as tokens generated early in a sequence cannot be modified in the same iteration. This constraint limits the models' ability to refine initial responses, potentially impacting the effectiveness of the generated code [37, 43].

To address these challenges and enhance the performance of LLMs on tasks in the LaMPilot benchmark, we introduce a human-in-the-loop approach. This method differs from most data-driven autonomous driving systems that rely solely on experiential learning. Instead, it combines practical experience with theoretical knowledge, similar to how human drivers develop their driving skills. In this approach, LLMs serve not only as planners but also as platforms for incorporating human feedback [16, 31, 38]. As

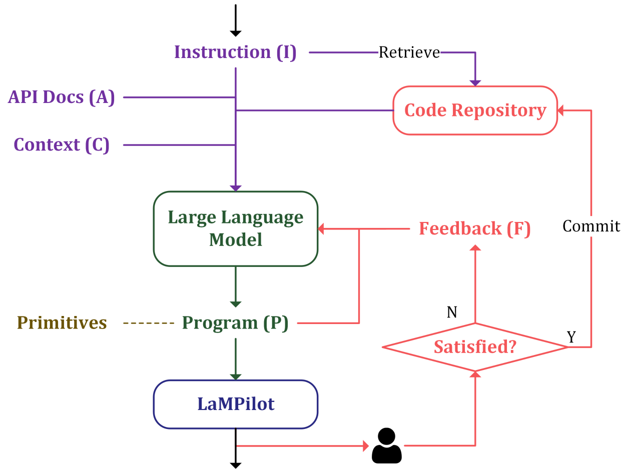


Figure 3. LaMPilot human feedback: this workflow diagram illustrates the interaction between the LLM and the user in the LaMPilot framework. The process begins with the user providing an instruction (I), which the LLM uses to generate a program (P). The program interacts with the LaMPilot simulator, and the execution result is evaluated by a human. Depending on the level of satisfaction, the system can either commit the code to the Code Repository or provide feedback to the LLM for further improvement.

shown in Fig. 3, human feedback is infused into the learning process, transforming it from a static, open-loop system into a dynamic, evolving feedback loop. After an LLM generates a policy program (P), it can integrate human feedback (F) to receive context-specific guidance, enabling the LLM to refine its output. For example, a human might suggest adopting a more assertive strategy during an unprotected left turn in heavy cross-traffic. This advice helps the LLM adjust its policy to better align with human preferences and situational requirements.

This approach does not involve calculating gradients or modifying model weights, as these processes are computationally expensive. Instead, we introduce a *code repository* module, which functions as a vector database. This repository allows for the storage and retrieval of effective code snippets to be used in similar situations. Following the approach in [43], the LLM generates a function description, which is then converted into a vector to serve as a database key. This key is paired with the corresponding function code as its value. Once the human critic is satisfied with the policy, the code is committed to the repository for future reuse.

5. Experiments and Results

Our experimental setup is designed to establish baselines for the proposed LaMPilot benchmark. The main objectives are to assess the performance of code-generating LLMs in understanding human verbal instructions within driving scenarios. We also aim to evaluate the capability of these LLMs to generate code for vehicle control using pre-defined behavioral primitives. Furthermore, we investigate the effectiveness of memory-augmented LLMs in incorporating human feedback.

5.1. Settings

We set a 60-second time limit for each scenario. If a task is not completed within this time frame, the test case will be considered a failure, and the simulation will be terminated, resulting in a driving score of 0. For successful cases, the driving score is calculated using the following formula:

$$\text{Driving Score} = \frac{\alpha}{N_s}\sum_{i=1}^{N_s}\text{Score}_i - \beta \cdot P_{\text{collision}} \quad (10)$$

Here, N_s represents the number of successful test cases, α is the success rate (ranging from 0 to 1), β is the collision rate (also between 0 and 1), and $P_{\text{collision}}$ is the penalty factor for collisions (set at 500 in our experiments). The Score_i for each individual sample is calculated based on Eq. (9), with weights W_{TTC}, W_{SV}, and W_{TE} set to 0.5, 0.3, and 0.2 respectively. The TTC Score, SV Score, and TE Score are computed as the averages of the corresponding metrics of all successful test cases.

The in-context examples are created by an annotator proficient in Python. They are provided with API descriptions and given the opportunity to write and test their code using the validation set. For the few-shot setting, a set of three examples that are the same for all test cases is used. The API Docs are also the same across all test cases.

5.2. Results

In this section, we present the experiment results on the LaMPilot benchmark, summarizing the performance of various methods, including heuristic baselines, zero-shot and few-shot baselines, and human feedback baselines. These results are detailed in Tab. 2.

Rule-Based Methods We first evaluate the performance of rule-based approaches, specifically the IDM and MOBIL algorithms. Both of these rule-based methods achieve a zero collision rate in the LaMPilot benchmark. This outcome highlights their reliability and effectiveness as well-established driving policies. It is important to note that these methods operate independently of the provided instructions. Specifically, without considering human instructions, these methods achieve a success rate ranging from 15% to 20%. This provides a crucial reference point for assessing the effectiveness of LLM-based agents in following human instructions.

Human Performance Humans consistently outperform all baseline models in terms of various evaluation metrics. They maintain a zero collision rate and demonstrate a significant advantage in completion rate compared to the top-performing LLM-based agent with human feedback. This

Method	Learning	Collision Rate (%, ↓)	Completion Rate (%, ↑)	TTC Score (↑)	SV Score (↑)	TE Score (↑)	Driving Score (↑)
IDM [42]	-	0.0	20.4	92.8	80.9	71.0	17.3
MOBIL [17]	-	0.0	15.3	82.8	85.9	46.7	11.7
Human Driver	-	0.0	98.0	93.4	86.3	81.3	84.6
Llama 2 (llama-2-70b-chat) [41]	zero-shot	0.0	20.4	92.8	81.4	68.9	17.3
PaLM 2 (code-bison) [3]	zero-shot	3.1	35.7	78.6	76.5	78.8	12.8
ChatGPT (gpt-3.5-turbo) [28]	zero-shot	1.0	40.8	83.9	75.5	74.0	27.8
GPT-4 (gpt-4) [30]	zero-shot	4.1	72.4	61.3	70.5	74.0	28.5
GPT-4 Turbo (gpt-4-1106-preview) [29]	zero-shot	3.1	74.5	73.2	70.1	73.9	39.1
Llama 2 (llama-2-70b-chat) [41]	few-shot	1.0	63.3	68.3	71.4	73.8	39.6
PaLM 2 (code-bison) [3]	few-shot	3.1	71.4	73.4	69.7	72.0	36.6
ChatGPT (gpt-3.5-turbo) [28]	few-shot	4.1	83.7	70.9	73.7	77.7	41.2
GPT-4 (gpt-4) [30]	few-shot	1.0	84.7	69.4	72.0	76.7	55.9
GPT-4 Turbo (gpt-4-1106-preview) [29]	few-shot	1.0	80.6	69.1	71.5	74.9	52.6
Llama 2 (llama-2-70b-chat) [41]	human-fdbk	0.9	32.7	83.9	74.6	74.6	21.4
PaLM 2 (code-bison) [3]	human-fdbk	0.0	45.5	81.1	74.4	77.6	35.6
ChatGPT (gpt-3.5-turbo) [28]	human-fdbk	0.9	80.9	70.2	71.7	77.5	54.2
GPT-4 (gpt-4) [30]	human-fdbk	0.9	92.7	67.6	72.2	76.8	64.0
GPT-4 Turbo (gpt-4-1106-preview) [29]	human-fdbk	0.9	87.3	74.1	73.4	75.5	60.5

Table 2. Evaluation of Baselines on LaMPilot. TTC, SV, and TE scores denote time-to-collision, speed variance, and time efficiency scores, respectively. The completion rate and collision rate represent the percentage of successful and collision test cases, respectively. The driving score is calculated according to Eq. (10). ↓: Lower values are preferable. ↑: Higher values are preferable.

gap highlights potential areas for future research. It is worth noting that even human drivers do not achieve perfect scores, which emphasizes the complexities of specific scenarios in the LaMPilot benchmark. For example, making unprotected left turns in heavy traffic within the allocated time can be particularly challenging.

Large Language Models We assess the effectiveness of extensive pretraining on the ability of LLMs to reason with common sense in the context of autonomous driving and follow human instructions.

In the zero-shot setting, OpenAI's GPT models and PaLM 2 (without any fine-tuning and given only the API Docs, driving context, and instruction) achieve a significant performance advantage compared to rule-based methods. However, this also results in an increase in the collision rate (1%-4%), indicating that their generated code policies do not fully capture the complexities of driving tasks.

When provided with training examples that include demonstration code, all the evaluated LLMs exhibit notable improvements in completion rates. The integration of GPT models with human-in-the-loop learning further enhances their performance in driving tasks. Notably, GPT-4 achieves a state-of-the-art result with a completion rate of 92.7% and a minimal collision rate of 0.9%, compared to the human success rate of 98%, highlighting the great potential of LLMs in following instructions in the driving context.

6. Conclusion and Limitations

This paper introduces LaMPilot, a novel benchmark dataset comprising 4.9K scenes specifically designed to evaluate instruction-following tasks in autonomous driving. It is the first dataset of its kind aimed at assessing the capabilities of LLMs in this application. Various state-of-the-art LLMs have been tested on LaMPilot. The main focus is to explore the feasibility of using LLMs to generate code that adheres to driving instructions. The experiments showed that GPT-4, with human feedback, achieved an impressive task completion rate of 92.7% and a minimal collision rate of 0.9%.

While our framework represents a significant advancement, it also emphasizes the ongoing need for substantial improvements for LLMs to better support instruction following in driving tasks. Although the completion rates are promising, there is still a notable collision rate. It is important to note that the LLMs in our study were used without any specific fine-tuning or parameter adjustments. Nevertheless, LaMPilot includes a diverse range of scenarios that are well-suited for interactive learning, which presents a promising direction for future exploration.

In summary, our research establishes the foundation for further investigation in this field. We anticipate that our benchmark and framework will be valuable for researchers interested in leveraging the evolving capabilities of LLMs in autonomous driving.

Acknowledgment

This work is funded by the Digital Twin Roadmap of InfoTech Labs, Toyota Motor North America. The contents of this paper only reflect the views of the authors, who are responsible for the facts and the accuracy of the data presented herein. The contents do not necessarily reflect the official views of Toyota Motor North America.

References

[1] Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Daniel Ho, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Eric Jang, Rosario Jauregui Ruano, Kyle Jeffrey, Sally Jesmonth, Nikhil J. Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Kuang-Huei Lee, Sergey Levine, Yao Lu, Linda Luu, Carolina Parada, Peter Pastor, Jornell Quiambao, Kanishka Rao, Jarek Rettinghouse, Diego Reyes, Pierre Sermanet, Nicolas Sievers, Clayton Tan, Alexander Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Mengyuan Yan, and Andy Zeng. Do As I Can, Not As I Say: Grounding Language in Robotic Affordances, 2022. arXiv:2204.01691 [cs]. 2

[2] Jacob Andreas, Dan Klein, and Sergey Levine. Learning with Latent Language, 2017. arXiv:1711.00482 [cs]. 2

[3] Rohan Anil, Andrew M. Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, Eric Chu, Jonathan H. Clark, Laurent El Shafey, Yanping Huang, Kathy Meier-Hellstern, Gaurav Mishra, Erica Moreira, Mark Omernick, Kevin Robinson, Sebastian Ruder, Yi Tay, Kefan Xiao, Yuanzhong Xu, Yujing Zhang, Gustavo Hernandez Abrego, Junwhan Ahn, Jacob Austin, Paul Barham, Jan Botha, James Bradbury, Siddhartha Brahma, Kevin Brooks, Michele Catasta, Yong Cheng, Colin Cherry, Christopher A. Choquette-Choo, Aakanksha Chowdhery, Clément Crepy, Shachi Dave, Mostafa Dehghani, Sunipa Dev, Jacob Devlin, Mark Díaz, Nan Du, Ethan Dyer, Vlad Feinberg, Fangxiaoyu Feng, Vlad Fienber, Markus Freitag, Xavier Garcia, Sebastian Gehrmann, Lucas Gonzalez, Guy Gur-Ari, Steven Hand, Hadi Hashemi, Le Hou, Joshua Howland, Andrea Hu, Jeffrey Hui, Jeremy Hurwitz, Michael Isard, Abe Ittycheriah, Matthew Jagielski, Wenhao Jia, Kathleen Kenealy, Maxim Krikun, Sneha Kudugunta, Chang Lan, Katherine Lee, Benjamin Lee, Eric Li, Music Li, Wei Li, YaGuang Li, Jian Li, Hyeontaek Lim, Hanzhao Lin, Zhongtao Liu, Frederick Liu, Marcello Maggioni, Aroma Mahendru, Joshua Maynez, Vedant Misra, Maysam Moussalem, Zachary Nado, John Nham, Eric Ni, Andrew Nystrom, Alicia Parrish, Marie Pellat, Martin Polacek, Alex Polozov, Reiner Pope, Siyuan Qiao, Emily Reif, Bryan Richter, Parker Riley, Alex Castro Ros, Aurko Roy, Brennan Saeta, Rajkumar Samuel, Renee Shelby, Ambrose Slone, Daniel Smilkov, David R. So, Daniel Sohn, Simon Tokumine, Dasha Valter, Vijay Vasudevan, Kiran Vodrahalli, Xuezhi Wang, Pidong Wang, Zirui Wang, Tao Wang, John Wieting, Yuhuai Wu, Kelvin Xu, Yunhan Xu, Linting Xue, Pengcheng Yin, Jiahui Yu, Qiao Zhang, Steven Zheng, Ce Zheng, Weikang Zhou, Denny Zhou, Slav Petrov, and Yonghui Wu. PaLM 2 Technical Report, 2023. arXiv:2305.10403 [cs]. 6, 8

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. In *NeurIPS*, pages 1877–1901, 2020. 6

[5] Can Cui, Yunsheng Ma, Xu Cao, Wenqian Ye, and Ziran Wang. Receive, Reason, and React: Drive as You Say with Large Language Models in Autonomous Vehicles, 2023. arXiv:2310.08034 [cs]. 1

[6] Sanhita Das and Akhilesh Kumar Maurya. Defining time-to-collision thresholds by the type of lead vehicle in non-lane-based traffic environments. *IEEE Transactions on Intelligent Transportation Systems*, 21(12):4972–4982, 2020. 5

[7] Thierry Deruyttere, Simon Vandenhende, Dusan Grujicic, Luc Van Gool, and Marie Francine Moens. Talk2car: Taking control of your self-driving car. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 2088–2098, 2019. 3

[8] Vikrant Dewangan, Tushar Choudhary, Shivam Chandhok, Shubham Priyadarshan, Anushka Jain, Arun K. Singh, Siddharth Srivastava, Krishna Murthy Jatavallabhula, and K. Madhava Krishna. Talk2BEV: Language-enhanced Bird's-eye View Maps for Autonomous Driving, 2023. arXiv:2310.02251 [cs]. 3

[9] Xinpeng Ding, Jianhua Han, Hang Xu, Wei Zhang, and Xiaomeng Li. HiLM-D: Towards High-Resolution Understanding in Multimodal Large Language Models for Autonomous Driving, 2023. arXiv:2309.05186 [cs]. 3

[10] Daocheng Fu, Xin Li, Licheng Wen, Min Dou, Pinlong Cai, Botian Shi, and Yu Qiao. Drive Like a Human: Rethinking Autonomous Driving with Large Language Models, 2023. arXiv:2307.07162. 1, 5

[11] Prasoon Goyal, Scott Niekum, and Raymond J. Mooney. PixL2R: Guiding Reinforcement Learning Using Natural Language by Mapping Pixels to Rewards, 2020. arXiv:2007.15543 [cs, stat]. 2

[12] Wenlong Huang, Chen Wang, Ruohan Zhang, Yunzhu Li, Jiajun Wu, and Li Fei-Fei. VoxPoser: Composable 3D Value Maps for Robotic Manipulation with Language Models, 2023. arXiv:2307.05973 [cs]. 3

[13] Eric Jang, Alex Irpan, Mohi Khansari, Daniel Kappler, Frederik Ebert, Corey Lynch, Sergey Levine, and Chelsea Finn. BC-Z: Zero-Shot Task Generalization with Robotic Imitation Learning, 2022. arXiv:2202.02005 [cs]. 2

[14] Yiding Jiang, Shixiang Gu, Kevin Murphy, and Chelsea Finn. Language as an Abstraction for Hierarchical Deep Reinforcement Learning, 2019. arXiv:1906.07343 [cs, stat]. 2

[15] Bu Jin, Xinyu Liu, Yupeng Zheng, Pengfei Li, Hao Zhao, Tong Zhang, Yuhang Zheng, Guyue Zhou, and Jingjing Liu. Adapt: Action-aware driving caption transformer. *arXiv preprint arXiv:2302.00673*, 2023. 3

[16] Ye Jin, Xiaoxi Shen, Huiling Peng, Xiaoan Liu, Jingli Qin, Jiayang Li, Jintao Xie, Peizhong Gao, Guyue Zhou, and Jiangtao Gong. SurrealDriver: Designing Generative Driver Agent Simulation Framework in Urban Contexts based on Large Language Model, 2023. arXiv:2309.13193 [cs]. 6

[17] Arne Kesting, Martin Treiber, and Dirk Helbing. General Lane-Changing Model MOBIL for Car-Following Models. *Transportation Research Record*, 1999(1):86–94, 2007. 6, 8

[18] Edouard Leurent. An Environment for Autonomous Driving Decision-Making, 2018. 3

[19] Xiang Lorraine Li, Adhiguna Kuncoro, Cyprien de Masson d'Autume, Phil Blunsom, and Aida Nematzadeh. A systematic investigation of commonsense understanding in large language models. *CoRR*, abs/2111.00607, 2021. 1

[20] Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as Policies: Language Model Programs for Embodied Control. In *ICRA*, 2023. 1, 3

[21] Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. Chameleon: Plug-and-play compositional reasoning with large language models. *arXiv preprint arXiv:2304.09842*, 2023. 3

[22] Jelena Luketina, Nantas Nardelli, Gregory Farquhar, Jakob Foerster, Jacob Andreas, Edward Grefenstette, Shimon Whiteson, and Tim Rocktäschel. A Survey of Reinforcement Learning Informed by Natural Language, 2019. arXiv:1906.03926 [cs, stat]. 2

[23] Corey Lynch and Pierre Sermanet. Language Conditioned Imitation Learning over Unstructured Data, 2021. arXiv:2005.07648 [cs]. 2

[24] Aman Madaan, Shuyan Zhou, Uri Alon, Yiming Yang, and Graham Neubig. Language models of code are few-shot commonsense learners. *arXiv preprint arXiv:2210.07128*, 2022. 3

[25] Jiageng Mao, Yuxi Qian, Hang Zhao, and Yue Wang. GPT-Driver: Learning to Drive with GPT, 2023. arXiv:2310.01415. 1

[26] Dipendra Misra, John Langford, and Yoav Artzi. Mapping Instructions and Visual Observations to Actions with Reinforcement Learning, 2017. arXiv:1704.08795 [cs]. 2

[27] Suraj Nair, Eric Mitchell, Kevin Chen, Brian Ichter, Silvio Savarese, and Chelsea Finn. Learning Language-Conditioned Robot Behavior from Offline Data and Crowd-Sourced Annotation, 2021. arXiv:2109.01115 [cs]. 2

[28] OpenAI. ChatGPT, 2023. https://openai.com/blog/chatgpt. 6, 8

[29] OpenAI. GPT-4-Turbo, 2023. https://openai.com/blog/new-models-and-developer-products-announced-at-devday. 6, 8

[30] OpenAI. GPT-4 Technical Report, 2023. arXiv:2303.08774 [cs.CL]. 6, 8

[31] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In *NeurIPS*, pages 27730–27744, 2022. 6

[32] Hao Sha, Yao Mu, Yuxuan Jiang, Li Chen, Chenfeng Xu, Ping Luo, Shengbo Eben Li, Masayoshi Tomizuka, Wei Zhan, and Mingyu Ding. LanguageMPC: Large Language Models as Decision Makers for Autonomous Driving, 2023. arXiv:2310.03026. 1, 3

[33] Shai Shalev-Shwartz, Shaked Shammah, and Amnon Shashua. On a Formal Model of Safe and Scalable Self-driving Cars, 2017. 4

[34] Pratyusha Sharma, Balakumar Sundaralingam, Valts Blukis, Chris Paxton, Tucker Hermans, Antonio Torralba, Jacob Andreas, and Dieter Fox. Correcting Robot Plans with Natural Language Feedback. In *Robotics: Science and Systems XVIII*. Robotics: Science and Systems Foundation, 2022. 2

[35] Mohit Shridhar, Lucas Manuelli, and Dieter Fox. CLIPort: What and Where Pathways for Robotic Manipulation, 2021. arXiv:2109.12098 [cs]. 2

[36] Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. Progprompt: Generating situated robot task plans using large language models. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 11523–11530. IEEE, 2023. 3

[37] Mukul Singh, José Cambronero, Sumit Gulwani, Vu Le, Carina Negreanu, and Gust Verbruggen. CodeFusion: A Pre-trained Diffusion Model for Code Generation. In *EMNLP*, 2023. arXiv:2310.17680 [cs]. 6

[38] Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. Learning to summarize with human feedback. In *NeurIPS*, pages 3008–3021, 2020. 6

[39] Dídac Surís, Sachit Menon, and Carl Vondrick. Vipergpt: Visual inference via python execution for reasoning. *arXiv preprint arXiv:2303.08128*, 2023. 3

[40] Stefanie Tellex, Nakul Gopalan, Hadas Kress-Gazit, and Cynthia Matuszek. Robots That Use Language. *Annual Review of Control, Robotics, and Autonomous Systems*, 3 (1):25–55, 2020. _eprint: https://doi.org/10.1146/annurev-control-101119-071628. 2

[41] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open Foundation and Fine-Tuned Chat Models, 2023. arXiv:2307.09288 [cs]. 6, 8

[42] Martin Treiber, Ansgar Hennecke, and Dirk Helbing. Congested traffic states in empirical observations and microscopic simulations. *Physical Review E*, 62(2):1805–1824, 2000. 6, 8

[43] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An Open-Ended Embodied Agent with Large Language Models, 2023. arXiv:2305.16291. 2, 6, 7

[44] Qifan Wang, Jingang Wang, Xiaojun Quan, Fuli Feng, Zenglin Xu, Shaoliang Nie, Sinong Wang, Madian Khabsa, Hamed Firooz, and Dongfang Liu. Mustie: Multimodal structural transformer for web information extraction. In

Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 2405–2420, 2023. 3

[45] Xingyao Wang, Sha Li, and Heng Ji. Code4struct: Code generation for few-shot event structure prediction. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3640–3663, 2023. 3

[46] Xingyao Wang, Hao Peng, Reyhaneh Jabbarvand, and Heng Ji. Leti: Learning to generate from textual interactions. *arXiv preprint arXiv:2305.10314*, 2023. 3

[47] Zekun Wang, Ge Zhang, Kexin Yang, Ning Shi, Wangchunshu Zhou, Shaochun Hao, Guangzheng Xiong, Yizhi Li, Mong Yuan Sim, Xiuying Chen, et al. Interactive natural language processing. *arXiv preprint arXiv:2305.13246*, 2023. 3

[48] Wayve. LINGO-1: Exploring Natural Language for Autonomous Driving, 2023. 3

[49] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. Emergent Abilities of Large Language Models. *Transactions on Machine Learning Research*, 2022. 4

[50] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *NeurIPS*, 2022. 1

[51] Licheng Wen, Daocheng Fu, Xin Li, Xinyu Cai, Tao Ma, Pinlong Cai, Min Dou, Botian Shi, Liang He, and Yu Qiao. DiLu: A Knowledge-Driven Approach to Autonomous Driving with Large Language Models, 2023. arXiv:2309.16292. 1

[52] Terry Winograd. Procedures as a Representation for Data in a Computer Program for Understanding Natural Language. 1971. 2

[53] Chenfei Wu, Shengming Yin, Weizhen Qi, Xiaodong Wang, Zecheng Tang, and Nan Duan. Visual chatgpt: Talking, drawing and editing with visual foundation models. *arXiv preprint arXiv:2303.04671*, 2023. 3

[54] Zhenhua Xu, Yujia Zhang, Enze Xie, Zhen Zhao, Yong Guo, Kenneth K. Y. Wong, Zhenguo Li, and Hengshuang Zhao. DriveGPT4: Interpretable End-to-end Autonomous Driving via Large Language Model, 2023. arXiv:2310.01412. 1, 3

[55] Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. Large language models meet nl2code: A survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7443–7464, 2023. 3

[56] Andy Zeng, Maria Attarian, Brian Ichter, Krzysztof Choromanski, Adrian Wong, Stefan Welker, Federico Tombari, Aveek Purohit, Michael Ryoo, Vikas Sindhwani, Johnny Lee, Vincent Vanhoucke, and Pete Florence. Socratic Models: Composing Zero-Shot Multimodal Reasoning with Language, 2022. arXiv:2204.00598 [cs]. 3, 4

LaMPilot: An Open Benchmark Dataset for Autonomous Driving with Language Model Programs

Supplementary Material

7. API Documentation Prompts

Ego The Ego APIs are a fundamental component for the LLM's self-awareness within the vehicle. They provide an essential information interface that includes key aspects such as the ego vehicle's reference (get_ego_vehicle), the current target speed (get_target_speed), and the ability to communicate with users (say). This information is the cornerstone for all subsequent primitives.

```
def get_ego_vehicle() -> Vehicle:
    """
    Return the ego vehicle.
    """

def get_desired_time_headway() -> float:
    """
    Return the desired time headway,
    IDM parameter in seconds,
    the minimum possible time to the vehicle in front,
    default 1.5.
    """

def get_target_speed() -> float:
    """
    Return the target speed,
    IDM parameter in m/s,
    the velocity the vehicle would drive at
    in free traffic.
    """

def say(text: str):
    """Provide a text message to the passenger.
    >>> say("Making a left lane change.")
    """
```

Control The Control APIs empower LLMs by providing a higher-level abstraction for controlling the ego vehicle. This enables maneuvers like lane changes without the need for low-level, direct control operations such as steering and throttle manipulation. These APIs ensure that the LLM's instructions are effectively translated into executable actions on the road while reducing the risk of collisions caused by suboptimal code. Key functionalities within these APIs include the ability to set a target speed (set_target_speed), initiate lane changes (set_target_lane), and access a default behavior function (autopilot).

```
# Control APIs
def set_desired_time_headway(
        desired_time_headway: float):
    """
    >>> set_desired_time_headway(1.5)
    """

def set_target_speed(target_speed: float):
    """
    >>> set_target_speed(30)
    """

def set_target_lane(target_lane: Lane):
    """
    >>> set_target_lane(target_lane)
    """

def autopilot() -> List[float, float]:
    """
    Return a control command
    (acceleration, steering angle)
    according to the current policy.
    By default it follows the IDM model,
    it will stop the car if
    there is a stop sign or a red traffic light.
    """

def recover_from_stop():
    """
    Resume driving after stopping
    at a stop sign or a red traffic light.
    >>> recover_from_stop()
    ""
```

Perception Our Perception APIs provide LLMs with a crucial interface for acquiring essential information needed to support their decision-making process. These APIs offer a wide range of capabilities, including retrieving the positions of other vehicles (get_lane_of) and obtaining speed-related information (get_speed_of). Additionally, the Perception APIs can handle more complex functions, such as detecting geometric relationships between lanes and vehicles using methods like detect_front_vehicle_in and get_distance_between_vehicles, as well as identifying traffic-related objects like stop signs ahead (detect_stop_sign_ahead).

```
def get_speed_of(veh: Vehicle) -> float:
    """
    Return the speed of the vehicle in m/s.
    """

def get_lane_of(veh: Vehicle) -> Lane:
    """
    Return the lane of the vehicle.
    """

def detect_front_vehicle_in(
        lane: Lane, distance: float = 100) -> Vehicle:
    """
    Return the closest vehicle in front of
    the ego vehicle in the specified lane
    within the specified distance,
    None if no vehicle.
    """


def detect_rear_vehicle_in(
        lane: Lane, distance: float = 100) -> Vehicle:
    """
    Return the closest vehicle behind
    the ego vehicle in the specified lane
    within the specified distance,
    None if no vehicle.
    """

def get_distance_between_vehicles(
        veh1: Vehicle, veh2: Vehicle) -> float:
    """
    Return the distance between two vehicles in meters,
    positive if veh1 is in front of veh2,
    negative otherwise.
    """


def get_left_lane(veh: Vehicle) -> Lane:
    """
    Return the left lane of the vehicle,
    None if no lane is detected.
    """


def get_right_lane(veh: Vehicle) -> Lane:
    """
    Return the right lane of the vehicle,
    None if no lane is detected.
    """


def get_left_to_right_cross_traffic_lanes()
        -> List[Lane]:
    """
    Return the left to right cross traffic lanes
    of the ego vehicle,
    Empty list if no lane is detected.
    """


def get_right_to_left_cross_traffic_lanes()
        -> List[Lane]:
    """
    Return the right to left cross traffic lanes
    of the ego vehicle,
    Empty list if no lane is detected.
    """


def detect_stop_sign_ahead() -> float:
    """
    Return the distance to the stop sign ahead,
    -1 if no stop sign is detected.
    """
```

Planning The Planning APIs offer a collection of fundamental tools that include safety-related operations and path planning functions for routing purposes. For example, the function is_safe_enter provides a predefined safety protocol to evaluate the current safety conditions for entering an adjacent lane. The function turn_left_at_next_intersection serves as a path planning tool, generating a route composed of a sequence of lanes for executing a left turn at the upcoming intersection. This route calculation utilizes a standard search algorithm to ensure efficient and dependable path planning.

```
def is_safe_enter(
        lane: Lane, safe_decel: float = 5) -> bool:
    """
    Return True if the ego vehicle can safely
    enter the specified lane
    from the current position now,
    False otherwise.
    safe_decel is the maximum deceleration that
    both the ego vehicle and the
    rear vehicle in the target lane can afford.
    """

def turn_left_at_next_intersection():
    """
    Change the planned route to
    turn left at the next intersection.
    >>> turn_left_at_next_intersection()
    """

def turn_right_at_next_intersection():
    """
    Change the planned route to
    turn right at the next intersection.
    >>> turn_right_at_next_intersection()
    """

def go_straight_at_next_intersection():
    """
    Change the planned route to
    go straight at the next intersection.
    >>> go_straight_at_next_intersection()
    """
```

8. Evaluation

This section presents the set of criteria for evaluating the effectiveness of the Language Model Program (LMP) in executing tasks according to the provided instructions. An important aspect of this evaluation involves addressing code policy collisions. If a collision occurs, it is considered a failure, which negatively impacts the overall driving score. In contrast, the absence of a collision leads to the assessment of the program's success, based on a set of predefined criteria.

Algorithm 1 Distance Evaluation Criteria

Input: Desired distance (absolute or relative to the front vehicle)
Parameter: Time limit T, distance tolerance δ, time duration D for maintaining desired distance
Output: Returns True if the ego vehicle maintains the desired distance for duration D; False otherwise.

```
Initialize time counter $t = 0$,
Initialize last time of distance compliance $t_{\text{last}} = -1$
while $t \leq T$ do
    if Collision detected then
        return False {Immediate failure on collision}
    end if
    Capture current position of ego vehicle and front vehicle
    if Ego vehicle's distance to front vehicle is within $\delta$ of desired distance then
        if $t_{\text{last}} = -1$ then
            Set $t_{\text{last}} = t$ {Begin tracking maintained distance}
        end if
    else
        Set $t_{\text{last}} = -1$ {Distance criteria not met, reset timer}
    end if
    if $t_{\text{last}} > 0$ and $t - t_{\text{last}} > D$ then
        return True {Distance maintained for required duration}
    end if
    Update $t$ with the time elapsed since the instruction was given
end while
return False
```

The LaMPilot dataset instructions are categorized as follows (as shown in Tab. 1): 1) distance-related, 2) speed-related, 3) pulling over, 4) routing, 5) lane changing, and 6) overtaking. For each category, the pseudo-code of the criteria is given in Algorithm 1, Algorithm 2, Algorithm 3, Algorithm 4, Algorithm 5, and Algorithm 6 respectively.

Algorithm 2 Speed Evaluation Criteria

Input: Desired speed (absolute or relative to ego vehicle's speed)
Parameter: Time limit T, speed tolerance δ, maximum gap G_{max}, time duration D for maintaining desired speed
Output: Returns True if the ego vehicle maintains the desired speed for duration D; False otherwise.

```
Initialize time counter $t = 0$, last time of speed compliance $t_{\text{last}} = -1$
while $t \leq T$ do
    if Collision detected then
        return False {Immediate failure on collision}
    end if
    Capture current speed and position of ego vehicle
    Identify front vehicle if within $G_{\text{max}}$
    if Ego vehicle's speed is within $\delta$ of desired speed and (no front vehicle or front vehicle speed is appropriate) then
        if $t_{\text{last}} = -1$ then
            Set $t_{\text{last}} = t$ {Begin tracking maintained speed}
        end if
    else
        Set $t_{\text{last}} = -1$ {Speed criteria not met, reset timer}
    end if
    if $t_{\text{last}} > 0$ and $t - t_{\text{last}} > D$ then
        return True {Speed maintained for required duration}
    end if
    Update $t$ with the current time
end while
return False
```

Algorithm 3 Pulling Over Criteria

Input: Target parking position p^*
Parameter: Time limit T
Output: Returns True if the vehicle safely pulls over at p^* within the time limit; False otherwise.

```
Initialize time counter $t = 0$
while $t \leq T$ do
    if Collision detected then
        return False {Collision leads to immediate failure}
    end if
    Capture the current position $p$ of the ego vehicle
    if $p = p^*$ and vehicle is parked then
        return True
    end if
    Update $t$ to the time elapsed since the instruction was given
end while
return False
```

Algorithm 4 Routing Criteria

Input: Destination position p^*, designated route R
Parameter: Time limit T
Output: Returns True if the vehicle reaches destination p^* via route R within the time limit T, and False otherwise.

```
Initialize time counter $t = 0$
Initialize route adherence flag $f$ as False
while $t \leq T$ do
    if Collision detected then
        return False {Collision leads to immediate failure}
    end if
    Capture the current position $p$ of the ego vehicle
    Update $f$ based on the vehicle's adherence to route $R$
    if $p = p^*$ and $f$ is True then
        return True
    end if
    Update $t$ with the time elapsed since the instruction was given
end while
return False
```

Algorithm 5 Lane Changing Criteria

Input: Target lane l^*
Parameter: Time limit T, heading threshold ϵ
Output: Returns True for a safe and successful lane change within the time limit and heading threshold; False otherwise.

```
Initialize time counter $t = 0$
while $t \leq T$ do
    if Collision detected then
        return False {Collision results in immediate failure}
    end if
    Capture the current lane $l$ of the vehicle.
    Obtain the current vehicle heading $\theta$.
    Determine the target lane heading $\phi$ at the vehicle's current position.
    if $l = l^*$ and $|\theta - \phi| < \epsilon$ then
        return True
    end if
    Update $t$ to the time elapsed since the instruction was given
end while
return False
```

Algorithm 6 Overtaking Criteria

Input: Target vehicle v_t
Parameter: Time limit T, distance threshold ϵ
Output: Returns True if the ego vehicle v_e successfully overtakes v_t within T and maintains a safe distance greater than ϵ; False otherwise.

```
Initialize time counter $t = 0$
Identify the ego vehicle as $v_e$
while $t \leq T$ do
    Check if a collision occurred. Return False if so.
    Measure the current lane distance $d$ between $v_e$ and $v_t$, where $d > 0$ indicates $v_e$ is ahead of $v_t$
    if $d > \epsilon$ then
        return True
    end if
    Update $t$ to the time elapsed since the instruction was given
end while
return False
```