DrPlanner: Diagnosis and Repair of Motion Planners for Automated Vehicles Using Large Language Models

Yuanfei Lin[1], *Student Member, IEEE,* Chenran Li[2], Mingyu Ding[2], *Student Member, IEEE,* Masayoshi Tomizuka[2], *Life Fellow, IEEE*, Wei Zhan[2], *Member, IEEE,* and Matthias Althoff[1], *Member, IEEE*

***Abstract*—Motion planners are essential for the safe operation of automated vehicles across various scenarios. However, no motion planning algorithm has achieved perfection in the literature, and improving its performance is often time-consuming and labor-intensive. To tackle the aforementioned issues, we present `DrPlanner`, the first framework designed to automatically diagnose and repair motion planners using large language models. Initially, we generate a structured description of the planner and its planned trajectories from both natural and programming languages. Leveraging the profound capabilities of large language models, our framework returns repaired planners with detailed diagnostic descriptions. Furthermore, our framework advances iteratively with continuous feedback from the evaluation of the repaired outcomes. Our approach is validated using both search- and sampling-based motion planners for automated vehicles; experimental results highlight the need for demonstrations in the prompt and show the ability of our framework to effectively identify and rectify elusive issues.**

***Index Terms*—Integrated planning and learning, motion and path planning, intelligent transportation systems, large language models, automated software repair.**

I. Introduction

MOTION planners for automated vehicles are responsible for computing safe, physically feasible, and comfortable motions [1]. A major challenge is the excessive manual effort required to tune motion planners, which entails diagnosing the planner based on a variety of critical test scenarios and evaluation metrics. To address this, we establish a framework that leverages the remarkable emergent abilities of large language models (LLMs) [2]–[4] to automatically provide and apply diagnostic solutions for a motion planner of automated vehicles, as illustrated in Fig. 1.

Manuscript received: March 12, 2024; Revised June 08, 2024; Accepted July 24, 2024. This paper was recommended for publication by Editor H. Kurniawati upon evaluation of the Associate Editor and Reviewers' comments. The authors gratefully acknowledge partial financial support by the German Federal Ministry for Digital and Transport (BMDV) for the project KoSi, by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) for the project SFB 1608, under Grant 501798263, and by the Berkeley DeepDrive. The work was developed during Y. Lin's visit to the University of California, Berkeley. *(Corresponding author: Wei Zhan.)*

[1] Y. Lin and M. Althoff are with the School of Computation, Information and Technology, Technical University of Munich, 85748 Garching, Germany. (e-mail: `yuanfei.lin@tum.de`; `althoff@tum.de`).

[2] C. Li, M. Ding, M. Tomizuka, and W. Zhan are with the Department of Mechanical Engineering, University of California, Berkeley, CA 94720, USA. (e-mail: `chenran_li@berkeley.edu`; `myding@berkeley.edu`; `tomizuka@berkeley.edu`; `wzhan@berkeley.edu`).

Digital Object Identifier (DOI): see top of this page.

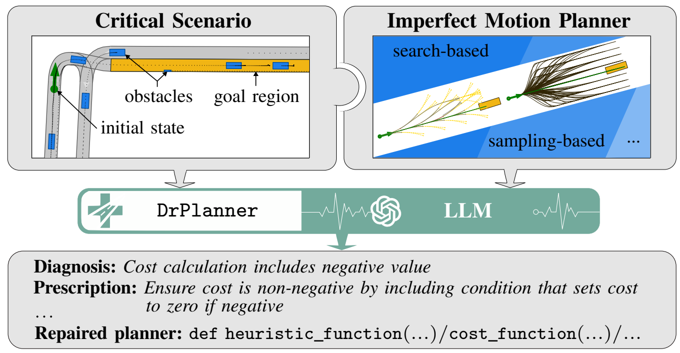


Fig. 1: An example usage of `DrPlanner`: In a critical scenario, our imperfect motion planner plans a trajectory. The description of the trajectory and the planner is then fed into `DrPlanner`. By harnessing the strengths of LLMs, we adeptly diagnose and repair the deficiencies within the planner.

A. Related Work

Although many motion planning algorithms can tackle a diverse range of tasks, they often face issues related to probabilistic completeness, computational complexity, or real-time constraints in finding the optimal solution [1], [5]–[8]. Besides, guaranteeing safety, rule compliance, and social compatibility of motion planners remains a challenge [9]–[12]. To provide an overview of how one can improve and repair such planners, we first survey methods from automated software repair, followed by summarizing contributions based on LLMs.

1) Automated Software Repair: With the increasing complexity and size of software, automatic debugging and repair techniques have been developed to reduce the extensive manual effort required to fix faults and to improve quality [13]. For instance, human-designed templates are used to repair certain types of bugs in code [14]–[18], but their effectiveness is often limited to hard-coded patterns. To overcome these limitations, deep-learning-based approaches utilize neural machine translation [19] to learn from existing patches, treating the repaired code as a translation of the buggy one [20]–[23]. However, the performance of these approaches is limited by the quality and quantity of the training data as well as its representation format [24]. As LLMs have shown emergent abilities in solving programming tasks [25]–[29], they are applied for generating program patches [30]–[32], self-debugging [33], [34], and cleaning code [35]. Unlike simply maintaining functional equivalence, we aim to both rectify imperfections and boost the performance of the planning algorithms. Although the aspect of linking text with code aligns with [29] and

©2024 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

the focus on performance improvement with [36], our work uniquely addresses the challenges posed by the larger and more intricate codebases of motion planners. Another branch of work focuses on repairing the outcome of given software [37]–[39] or addressing specified diagnostic criteria [40].

2) Language Models for Motion Planning: With their indispensable role of common sense reasoning and generalization [41]–[43], LLMs have been applied in motion planning for autonomous driving to make high-level decisions [44]–[48], generate driving trajectories [49], [50] or provide control signals directly [51]–[53]. However, the refinement of motion planners themselves is still driven by the nuanced intuition of humans and by real traffic data. In this work, LLMs serve to bridge this gap by emulating human-like problem-solving strategies, offering strategic guidance in analyzing complex motion planners.

B. Contributions

In this work, we introduce `DrPlanner`, the first framework to autonomously diagnose and repair motion planners for automated vehicles, harnessing the power of LLMs that improve as they scale with additional data and model complexity. In particular, our contributions are:

1) establishing a structured and modular description for motion planners across both natural and programming language modalities to exploit the capabilities of LLMs for diagnosis and repair;
2) leveraging the in-context learning capabilities of LLMs by providing demonstrations to the model at the point where it infers diagnostic results; and
3) enhancing the understanding of underlying improvement mechanisms by generating continuous feedback in a closed-loop manner.

The remainder of this work is structured as follows: Sec. II lists necessary preliminaries. The proposed framework for diagnosing and repairing motion planners is described in Sec. III. We demonstrate the benefits of our approach in Sec. IV and conclude the paper in Sec. V.

II. Preliminaries

A. Motion Planning for Automated Vehicles

We refer to the vehicle for which trajectories are planned as the *ego vehicle*. As illustrated in Fig. 2, motion planning algorithms are tasked with ensuring that the ego vehicle travels from an initial state to a goal region within a specified time [54]. The motion planner typically minimizes a given objective function $J(\chi)$, e.g., by penalizing the travel time or passenger discomfort [1, Sec. IV]. Simultaneously, the solution, denoted by χ, must satisfy common and safety-relevant requirements, such as being drivable, collision-free, and rule-compliant [38], [55]. Subsequently, we denote a motion planner by `M` and a motion planning problem by `P`.

B. Prompt Engineering for LLMs

The technique of using a textual string ℓ to instruct LLMs is referred to as *prompting* [56, Sec. 4]. This approach enables

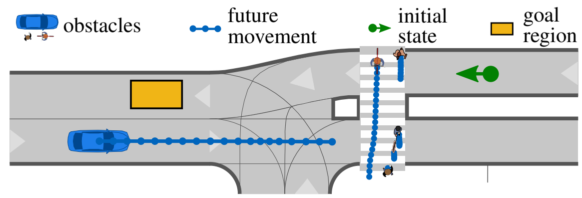


Fig. 2: Exemplary motion planning problem, where the ego vehicle needs to travel from its initial state to reach the goal region safely and efficiently.

LLMs to be pretrained on a massive amount of data [56, Sec. 3] and subsequently adapt to new use cases with few or no labeled data. To enhance the in-context learning capabilities, the prompt may include a few human-annotated examples of the task, known as *few-shot prompting* [2], or utilize chain-of-thought reasoning [41], [57]. We divide the input prompt ℓ into two components: the system prompt ℓ_{system}, which outlines the task for the LLMs, and the user prompt ℓ_{user}, providing context for the diagnostic task. The labels, manual inputs, and automatically generated content within the prompt are marked with angle brackets, square brackets, and curly brackets, respectively. The output consists of both a list of diagnosis-prescription pairs and patched programs, collectively denoted by $\boldsymbol{\ell}_{\text{dp}}$ and $\boldsymbol{p}_{\text{p}}$. It is important to consider that LLMs have a limit on the number of tokens they can process [58], which imposes a maximum length on the prompt and prevents us from including extensive code within a single prompt.

III. DrPlanner

This section presents our prompt engineering with a nuanced diagnostic description. We begin by introducing the overall algorithm, followed by a more detailed presentation.

A. Overall Algorithm

A general overview of using `DrPlanner` is presented in Fig. 3 and Alg. 1. Before initiating the process, the user fills in the placeholders enclosed in square brackets. For a given scenario, the motion planner `M` is first deployed to address the associated planning problem `P` (see line 1). Subsequently, the planned trajectory χ is evaluated using the objective function J (see line 2). Following this, a diagnostic description ℓ_{user} encompassing the diagnostic instructions, the description of the planner, the evaluation of the trajectory, and the few-shot examples are formulated (see line 3). This description, along with the system prompt ℓ_{system}, is fed into the LLM (see line 6). The structure of the input prompt is illustrated in the center of the framework in Fig. 3. Afterwards, the obtained patched programs are applied to the motion planner by integrating the modifications into the existing codebase (see line 7).

However, it is important to note that the output generated may include errors such as hallucinations and inaccurate analyses [59]. To mitigate these issues, we employ an iterative prompting strategy, repeatedly refining the process. The iteration is terminated when a notable improvement in the planner is observed, e.g., when the difference between the current best performance $J_{\min}$ and a target value J^* is smaller than a threshold $\epsilon \in \mathbb{R}_+$, or when the token limit of the LLM is reached (see lines 5-14). Finally, the repaired planner demonstrating the best improvement, if any, along with the

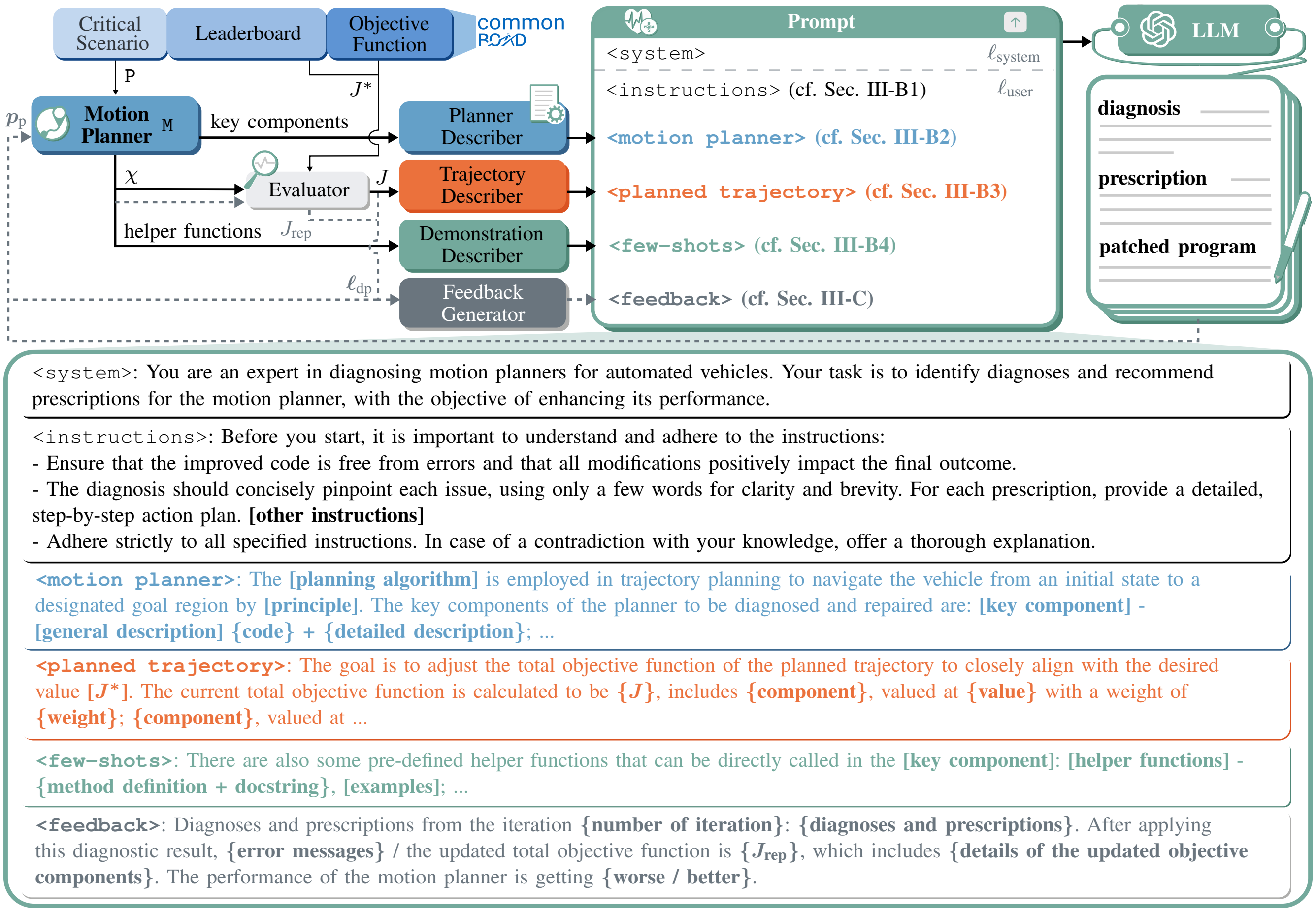


Fig. 3: Overview of the `DrPlanner` framework. The process starts with obtaining a planned trajectory for the planning problem with the given motion planner. Then, the planned trajectory is evaluated by the objective function. Afterwards, the description for the planner is generated and used to prompt an off-the-shelf LLM to generate the diagnoses and prescriptions for the planner, along with the patched programs. After applying the patches, the evaluation of the updated planner is incorporated back into the prompt as feedback to continuously enhance the diagnostic performance (marked by dashed arrows).

Algorithm 1 DIAGNOSEANDREPAIRPLANNER

Input: planning problem P, motion planner M, target value J^*, system prompt ℓ_{system}, LLM
Output: diagnoses and prescriptions ℓ^*_{dp}, repaired planner $\mathtt{M}^*_{\text{rep}}$

1: $\chi \leftarrow$ M.PLAN(P)
2: $J \leftarrow$ EVALUATE(χ)
3: $\ell_{\text{user}} \leftarrow$ DESCRIBE(M, J, J^*) ▷ Sec. III-B
4: $J_{\min} \leftarrow J$, $\ell^*_{\text{dp}} \leftarrow \emptyset$, $\mathtt{M}^*_{\text{rep}} \leftarrow \emptyset$
5: **while** not REACHTOKENLIMIT(LLM) and $J_{\min} - J^* > \epsilon$ **do**
6: (ℓ_{dp}, $\boldsymbol{p}_{\text{p}}$) $\leftarrow$ LLM.QUERY(ℓ_{system}, ℓ_{user}) ▷ Sec. III-C
7: $\mathtt{M}_{\text{rep}} \leftarrow$ REPAIR(M, $\boldsymbol{p}_{\text{p}}$)
8: $\chi \leftarrow \mathtt{M}_{\text{rep}}$.PLAN(P)
9: $J_{\text{rep}} \leftarrow$ EVALUATE(χ)
10: $\ell_{\text{user}} \leftarrow$ ADDFEEDBACK(ℓ_{user}, J, J_{rep}, ℓ_{dp}) ▷ Sec. III-C
11: **if** $J_{\text{rep}} < J_{\min}$ **then**
12: $J_{\min} \leftarrow J_{\text{rep}}$, $\ell^*_{\text{dp}} \leftarrow \ell_{\text{dp}}$, $\mathtt{M}^*_{\text{rep}} \leftarrow \mathtt{M}_{\text{rep}}$
13: **end if**
14: **end while**
15: **return** ℓ^*_{dp}, $\mathtt{M}^*_{\text{rep}}$

corresponding diagnoses and prescriptions, is returned (see line 15).

Another regime is to finetune the LLM to the given task. However, to date, there exists no open-source dataset containing input-output examples of motion planners. Additionally, finetuning usually only provides modest improvements in solving challenging and complex tasks compared to in-context learning [34], [35], [57]. Regardless of the approach, when deploying the repaired planners on roads, a safety layer is always required [12].

B. Diagnostic Description

As discussed in Sec. II-B, prompt design is challenging, particularly when considering the limited information about the diagnostic object in the pretrained LLM. To enhance conclusions, we design a structured and comprehensive description of the motion planner, emulating the process of a real doctor. Its overall skeleton is depicted in the lower part of Fig. 3. As we assume that the motion planner internally handles goal-reaching and drivability-checking of the trajectory in the scenario (cf. Sec. II-A), a detailed description of the scenario, motion planning problem, and trajectory states is omitted in the prompt. Alternatively, these tasks can be addressed by additional modules, such as those employing LLM-embedded agents (cf. Sec. I-A2).

1) Instructions: The instruction provides general guidance for the LLM, detailing the expected output and reasoning constraints. In addition, we can include the commonly used rule-of-thumb from expert knowledge. For instance, "*merely adjusting the weighting or coefficients is often cumbersome and not very effective*".

2) Motion Planner: The description of the motion planner begins with the selection and a brief introduction to the planning algorithm. This is followed by a general description of the key components that primarily affect the performance of the planner. To gain a better understanding of how the algorithm is practically implemented, we also include the code of the key components as an additional input modality. As mentioned in Sec. I-A1, the LLM is then able to generate repaired programs given corresponding instructions. Motivated by the chain of thought (cf. Sec. II-B), we incorporate existing explanations found within the docstrings of subfunctions to provide natural language summaries for the code blocks. The description adheres to the format of **{subfunction name}** followed by its **{docstring}**. For instance, an automatically generated **{detailed description}** is: "`self.calc_angle_to_goal` *returns the orientation of the goal with respect to current position; ...*" (cf. Fig. 6a).

3) Planned Trajectory: There are various measures to quantitatively evaluate the planned trajectory and track its improvement. These measures include the cost function [54], criticality measures [60], courtesy to other traffic participants [61], and degree of traffic rule compliance [11], [38]. To align the LLM with the desired behavior, we present not only the evaluation results for the selected measures but also incorporate the target value J^*, which can be, e.g., sourced from the motion planning benchmark leaderboard. In addition, the numerical data of the values and weights of the objective components is translated into a narrative description by mapping them to their corresponding placeholders.

4) Few-Shots: As it is not necessary for LLMs to have prior knowledge of the other part of the large-scale motion planner, we provide existing helper functions and their exemplary usage in the prompt. Furthermore, several human-annotated examples for improving the performance of the specific type of motion planner can be added here, with examples available in Fig. 4.

C. LLM Querying and Iterative Prompting

When querying the LLM, it is essential to specify the desired output format. To achieve this, one can guide the LLM by emphasizing the diagnoses, prescriptions, and key components of the planner (cf. Sec. II-A) in the prompt as desired responses or employ other third-party tools such as LangChain[1]. Consequently, the structured patched results can directly replace the original elements to repair the planner.

Motivated by how LLMs are utilized in improving technical systems [34], [45], [62], [63], we examine the repaired planner by executing it and then pass the evaluation result back to the LLM. In case of compilation or execution errors, the previous diagnostic result is combined with the information indicating where the error occurred and what it entails. Otherwise, the combination is made with a comparison of the performance between the updated planned trajectory and the original one.

[1]https://www.langchain.com/

IV. Evaluation

We evaluate our approach using the open-source motion planners from the CommonRoad platform [54], which are written in `Python`. As CommonRoad provides customizable challenges and annual competitions, where users can compete against each other on predefined benchmarks, we can continuously integrate enhancements into `DrPlanner` based on insights from a broad user base. Furthermore, we choose GPT-4-Turbo[2] as our LLM and use its function calling feature to generate structured outputs. It should be noted that our framework is not limited to GPT-4-Turbo and can be easily adapted for use with other LLMs by modifying the interface. The patched programs are then stringified in a JSON object and directly parsed to the motion planner, followed by execution through the `exec` function in `Python`. The token limit is set to $8,000$, the threshold ϵ is equal to 10, and we choose the sampling temperature of the LLM at 0.6 (cf. [26, Fig. 5]). Code and exemplary prompts are available at https://github.com/CommonRoad/drplanner.

A. Setup

1) Search-Based Motion Planner[3]: We adapt the anytime A* search algorithm using lattice-based graphs [64]. This implementation features a time-limited search cut-off and employs a cost function and an estimated cost to the goal, namely, a *heuristic function*, to guide the search process. The graph is constructed with *motion primitives*—short trajectories generated offline through a forward simulation of a given vehicle model. The number of explored nodes in the graph is denoted as N_n. Motion primitives are typically referenced by IDs encoded with configurable parameters[4]:

$$\texttt{MP} = \texttt{"V\_}\boxed{v_{\min}}\texttt{\_}\boxed{v_{\max}}\texttt{\_Vstep\_}\boxed{\Delta v}\texttt{\_SA\_}\boxed{\delta_{\min}}\texttt{\_}\boxed{\delta_{\max}}\texttt{\_}$$
$$\texttt{SAstep\_}\boxed{\Delta\delta}\texttt{\_T\_}\boxed{\tau}\texttt{\_Model\_}\boxed{m}\texttt{"},$$

where $v_{\min}$ and $v_{\max}$ are the sampling velocity limits, $\delta_{\min}$ and $\delta_{\max}$ are the sampling steering angle bounds, Δv and $\Delta\delta$ specify their respective step sizes, τ is the time duration of each motion primitive, and m is the model identifier of the ego vehicle. Therefore, the heuristic function and motion primitives constitute the key components. We provide the entire code block of the heuristic function along with descriptions of the involved subfunctions in natural language. In the description of motion primitives, the explanation includes the naming convention, followed by their ID.

2) Sampling-Based Motion Planner[5]: Similarly, we evaluate our approach on the sampling-based motion planner of [65], which computes jerk-optimal trajectories using polynomials to connect sampled end states with the initial state. From the set of feasible trajectory samples, the optimal trajectory is selected based on a *cost function*. Consequently, the cost function and *sampling configurations*, such as the sampling time horizon t_s, are the key components.

[2]ID `gpt-4-turbo-preview` in the API of OpenAI.

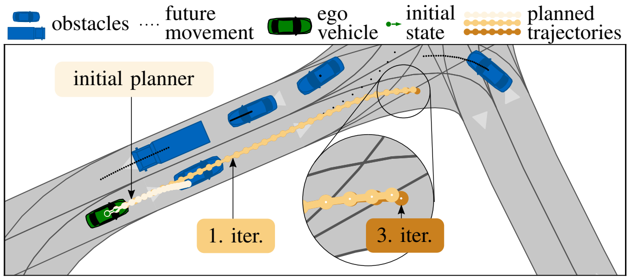
[3]https://commonroad.in.tum.de/tools/commonroad-search

[4]All parameters are given in SI units.

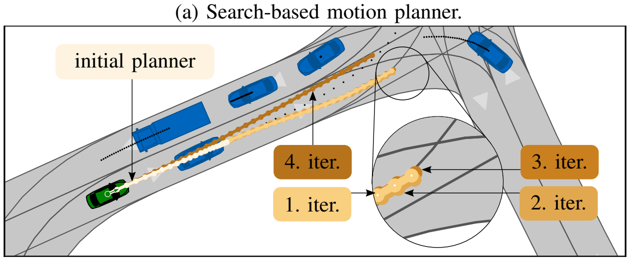
[5]https://commonroad.in.tum.de/tools/commonroad-reactive-planner

```
There are some pre-defined helper functions that can be directly called
in the heuristic function:
def calc_acceleration_cost(self, path: List[KSState]) -> float:
    """Returns the acceleration costs.""" ...
Examples:
(input)
def heuristic_function(self, node_current: PriorityNode) -> float:
    ...
    cost = angle_to_goal
    return cost
(output)
Diagnosis: the acceleration is not considered
Prescription: add the acceleration cost to the heuristic function
def heuristic_function(self, node_current: PriorityNode) -> float:
    acceleration_cost =
        self.calc_acceleration_cost(node_current.list_paths[-1])
    ...
    cost = angle_to_goal + acceleration_cost
    return cost
Feasible motion primitives with the same name format that you can
directly use:
"V_0.0_20.0_Vstep_1.0_SA_-1.066_1.066_SAstep_2.13_T_0.5_Model_BMW_320i",
"V_0.0_20.0_Vstep_2.0_SA_-1.066_1.066_SAstep_0.18_T_0.5_Model_BMW_320i",
...
```

Fig. 4: Snippet of the few-shot prompting used for the search-based planner.



(a) Search-based motion planner.



(b) Sampling-based motion planner.

Fig. 5: Critical intersection scenario[7] in which the ego vehicle needs to safely drive for 33 time steps. For clarity, the planned trajectories for the ego vehicle from different planners are marked with different colors and labels.

3) Measures of the Planned Trajectory: To evaluate the quality of the planned trajectory, we utilize the standardized objective function J_{SM1}[6] from CommonRoad [54, Sec. VI], which includes the cost for acceleration, steering angle, steering rate, distance and orientation offset to the centerline of the road, and velocity offset to the desired value.

4) Few-Shots: To gain a deeper insight into the planner, we include method definitions and docstrings for existing helper functions within the planner class. For instance, as shown in Fig. 4, we also provide a list of IDs corresponding to offline-generated motion primitives from which the LLM can select for the search-based planner.

B. Case Study

We choose an intersection scenario from the CommonRoad platform (cf. Fig. 5), which is generated by the scenario

[6]The objective function can be adapted or replaced as needed.
[7]CommonRoad-ID: DEU_Guetersloh-15_2_T-1

```
def heuristic_function(self, node_current: PriorityNode) -> float:
    path_last = node_current.list_paths[-1]
    angleToGoal =
        self.calc_angle_to_goal(path_last[-1])
    orientationToGoalDiff = self.calc_orientation_diff(angleToGoal,
        path_last[-1].orientation)
    cost_time = self.calc_time_cost(path_last)
    if self.reached_goal(node_current.list_paths[-1]):
        heur_time = 0.0
    if self.position_desired is None:
        heur_time = self.time_desired.start -
            node_current.list_paths[-1][-1].time_step
    else:
        velocity = node_current.list_paths[-1][-1].velocity
    if np.isclose(velocity, 0):
        heur_time = np.inf
    else:
        heur_time =
            self.calc_euclidean_distance(current_node=node_current) /
            velocity
    cost = 20 * orientationToGoalDiff + 0.5 * cost_time + heur_time
    if cost < 0:
        cost = 0
    return cost
```

```
MP = "V_0.0_20.0_Vstep_4.0_SA_-1.066_1.066_SAstep_0.18_T_0.5_Model_\
    BMW_320i"
```

(a) Search-based motion planner.

```
def evaluate(self, trajectory: TrajectorySample) -> float:
    cost == 0.0
    cost += CostFunction.steering_velocity_costs(trajectory)
    cost += CostFunction.path_length_costs(trajectory)
return cost
```

```
planning:
    dt: 0.1
    replanning_frequency: 3
    time_steps_computation: 20
```

(b) Sampling-based motion planner.

Fig. 6: Key components used in the initial planner.

TABLE I: Comparison of the planned trajectories before and after repair. The lowest values of the objective function are marked in bold.

Type	Item	Initial Planner	Repaired Planner 1. Iter.	2. Iter.	3. Iter.	4. Iter.
Search-based	J_{SM1}	4606.93	752.56	-	**4.65**	-
	N_n	11	9	-	9	-
Sampling-based	J_{SM1}	2614.76	169.49	305.42	**147.55**	197.58
	t_s	$2.0s$	$3.0s$	$2.5s$	$3.0s$	$3.0s$

factory for safety-critical traffic scenarios [60], [66]. In the urban environment, the motion planners are responsible for navigating the ego vehicle from the initial state for $3.3s$ without colliding with any obstacles. The time increment of the scenario is $0.1s$. In both planned trajectories by the initial planners configured as shown in Fig. 6, the ego vehicle brakes and steers slightly to the right, leading to high values of J_{SM1} (cf. Tab. I). The target value of J_{SM1} is extracted from the CommonRoad benchmark leaderboard[8] and is $J^*_{\mathrm{SM1}} = 0.16$.

The diagnostic results for the search-based planner using our approach are illustrated in Fig. 7. In the first iteration, the provided helper functions are automatically included in the heuristic function by the LLM (cf. Fig. 7a). Meanwhile, some hyperparameters are adjusted, such as the orientation weight and the heuristic for zero velocity, and new motion primitives are selected. Considering all the above factors, the repaired planner results in a decrease in J_{SM1} of the planned trajectory, particularly in the acceleration cost. Additionally, it

[8]https://commonroad.in.tum.de/solutions/ranking

allows the vehicle to travel further forward with fewer explored nodes in the search graph due to the coarser motion primitives applied (see Tab. I and Fig. 5a). In contrast, the diagnostic result from the second iteration leads to a `KeyError` (cf. Fig. 7b), indicating that the repaired heuristic function is not provided by the LLM. With the iterative prompting, the error message is incorporated as feedback into the prompt for the third iteration. As shown in Fig. 7c, our approach not only helps the LLM avoid the errors from previous iterations (cf. the diagnosis "*KeyError in heuristic function*") but also retains the previous diagnostic results that lead to a positive impact on the planner. As a result, the planner significantly improves its performance, with a substantial reduction in J_{SM1} from 752.56 to 4.65, achieved by further balancing the objective components (cf. Tab. I). Moreover, it can be observed from Fig. 7 that `DrPlanner` can provide fine-grained diagnoses and prescriptions based on both the prompt design and fundamental aspects of programming, such as aliasing (cf. lines 10, 13, 15 in Fig. 7c). The resulting patched programs align precisely with these diagnoses and prescriptions.

The initial configuration snippet of the sampling-based planner is shown in Fig. 6b. A similar repair pattern to the search-based planner can be observed in Tab. I and Fig. 5b. For brevity, we only show the diagnostic details for the third iteration in Fig. 9, which achieves the best performance among all iterations. The cost function improves through weight tuning and adding more items, and a larger t_s is selected, leading to a noticeable reduction in J_{SM1}.

C. Performance Evaluation

We further evaluate the performance of `DrPlanner` by analyzing 50 randomly selected critical CommonRoad scenarios, along with 50 A*-search-based motion planners in various setups from the CommonRoad challenges. The former is benchmarked against the search-based planner configured as shown in Fig. 6a. The latter evaluation utilizes the scenario illustrated in Fig. 5 and employs the pass@k metric. We use its unbiased version as proposed in [26, Sec. 2.1], defined as the probability that at least one of the top $k \in \mathbb{N}_+$ generated code samples for a problem passes the given tests. Here, we use a decrease of J_{SM1} for the returned planner as the criterion for passing. As a baseline, the performance of `DrPlanner` is compared with a genetic approach [14], where the program of the heuristic function is repaired to minimize J_{SM1}. The solution space consists of 10 chromosomes, and the process runs for 100 generations. Additionally, we conduct ablation studies to examine the impact of omitting two specific components within the framework across different planners: few-shots and feedback. For each study, we execute the framework 10 times to collect solution samples.

Fig. 8 and Tab. II present the results of the performance evaluation. Overall, `DrPlanner` effectively diagnoses and repairs motion planners under various setups, outperforming the baseline approach in all metrics, with a pass rate of 98.0% at $k = 10$ and an average reduction of 54.5% in J_{SM1}. The benchmark results in Fig. 8 further indicate robust performance across diverse scenarios, with an average J_{SM1} decrease of

```
1  def heuristic_function(self, node_current: PriorityNode) -> float:
6  +   acceleration_cost = self.calc_acceleration_cost(path_last)
7  +   path_efficiency = self.calc_path_efficiency(path_last)
8  +   steering_angle_cost = self.calc_steering_angle_cost(path_last)
9  +   steering_velocity_cost = self.calc_steering_velocity_cost(path_last)
10     if self.reached_goal(node_current.list_paths[-1]):
11 -       heur_time = 0.0
11 +       return 0
16     if np.isclose(velocity, 0):
17 -       heur_time = np.inf
17 +       heur_time = 10  # Instead of np.inf, assign a high but finite
   cost
20 -   cost = 20 * orientationToGoalDiff + 0.5 * cost_time + heur_time
20 +   cost = 10 * orientationToGoalDiff + 0.5 * cost_time +
           heur_time + acceleration_cost + path_efficiency +
           steering_angle_cost + steering_velocity_cost
```

MP = "V_0.0_20.0_Vstep_~~4.0~~2.0_SA_-1.066_1.066_SAstep_0.18_T_0.5_Model_\ BMW_320i"

Diagnosis	Prescription
Orientation weight too high	Adjust the weight for orientation to goal difference
Missing acceleration cost	Include acceleration cost in heuristic
Missing path efficiency	Include path efficiency in heuristic
Missing steering angle cost	Include steering angle cost in heuristic
Missing steering velocity cost	Include steering velocity cost in heuristic
Infinite heuristic for zero velocity	Handle zero velocity case appropriately
Sparse motion primitives	Recommend motion primitives with higher branching factor

(a) 1. iteration.

```
KeyError: 'repaired_heuristic_function'
```

(b) 2. iteration.

```
1  def heuristic_function(self, node_current: PriorityNode) -> float:
6  +   acceleration_cost = self.calc_acceleration_cost(path_last)
7  +   path_efficiency = self.calc_path_efficiency(path_last)
8  +   steering_angle_cost = self.calc_steering_angle_cost(path_last)
9  +   steering_velocity_cost = self.calc_steering_velocity_cost(path_last)
10 -   if self.reached_goal(node_current.list_paths[-1]):
10 +   if self.reached_goal(path_last):
11         heur_time = 0.0
12 -   if self.position_desired is None:
12 +   elif self.position_desired is None:
13 -       heur_time = self.time_desired.start -
               node_current.list_paths[-1][-1].time_step
13 +       heur_time = self.time_desired.start - path_last[-1].time_step
14     else:
15 -       velocity = node_current.list_paths[-1][-1].velocity
15 +       velocity = path_last[-1].velocity
16     if np.isclose(velocity, 0):
17 -       heur_time = np.inf
17 +       heur_time = 1e6  # A large but not infinite cost
20 -   cost = 20 * orientationToGoalDiff + 0.5 * cost_time + heur_time
20 +   cost = 10 * orientationToGoalDiff + 0.5 * cost_time +
           heur_time + acceleration_cost + path_efficiency +
           steering_angle_cost + steering_velocity_cost
```

MP = "V_0.0_20.0_Vstep_~~4.0~~2.0_SA_-1.066_1.066_SAstep_0.18_T_0.5_Model\ _BMW_320i"

Diagnosis	Prescription
Orientation weight excessive	Decrease orientation weight in heuristic
Heuristic excludes costs	Incorporate all costs into heuristic
Zero velocity infinite cost	Refine zero velocity case handling
Motion primitives high branching	Optimize motion primitives branching
KeyError in heuristic function	Ensure correct key for improved heuristic

(c) 3. iteration.

Fig. 7: Diagnostic and repair result for the search-based motion planner in Fig. 6a. The identical program patches in the first and third iteration are highlighted with black borders in (c). For the second iteration, we omit the diagnoses and prescriptions since it leads to an error.

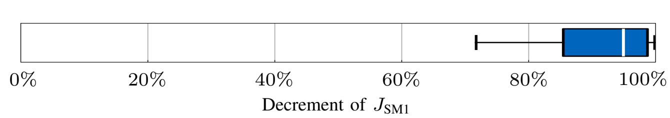


Fig. 8: Benchmarked reduction of J_{SM1} across scenarios using `DrPlanner`. For better visibility, outliers in the box plot are not shown.

```
def evaluate(self, trajectory: TrajectorySample) -> float:
-    cost += CostFunction.steering_velocity_costs(trajectory)
+    cost += CostFunction.acceleration_costs(trajectory) * 25
+    cost += CostFunction.steering_velocity_costs(trajectory) * 25
+    cost += CostFunction.longitudinal_jerk_costs(trajectory) * 10
```

`planning:`
`    time_steps_computation:` ~~20~~30

Diagnosis	Prescription
Acceleration cost weight too high	Reduce the weight of the acceleration cost
Steering velocity cost weight too high	Consider reducing its weight to better balance the cost function
Inclusion of jerk costs	To add nuance to the trajectory evaluation, include jerk costs in the cost function
Planning horizon too short	Increase the planning horizon

Fig. 9: Diagnostic and repair result of the third iteration for the sampling-based motion planner in Fig. 6b.

TABLE II: Performance evaluation and ablation studies across planners on the design of `DrPlanner`. Values in bold denote the best performance.

Method	**pass@k** $k=1\uparrow$	$k=5\uparrow$	$k=10\uparrow$	**Decrement of J_{SM1}** Avg. $\uparrow$	Std. Dev.
Genetic [14]	0.8%	0.4%	7.7%	0.1%	1.6%
w/o Few-Shots	0.0%	0.0%	0.0%	0.0%	0.0%
w/o Feedback	45.4%	86.2%	92.0%	49.6%	36.3%
`DrPlanner`	**68.0%**	**95.1%**	**98.0%**	**54.5%**	34.9%

90.76%. Note that, similar to the case study in Sec. IV-B, the value of J_{SM1} does not converge with the iterations due to diagnostic inaccuracies. However, the average number of iterations required to observe its first decrease is 1.4.

Moreover, the ablation studies demonstrate that both the few-shot learning (cf. Sec. III-B4) and the iterative prompting (cf. Sec. III-C) play crucial roles in enhancing the effectiveness of `DrPlanner`. In particular, the few-shots prompting is more effective since the LLM is intrinsically unaware of the other supportive components of the planner, e.g., the available motion primitives. Additionally, since the initial planners are not buggy but underperforming, the results without using few-shots show that they cannot be easily improved with only the descriptions of the planner and the planned trajectory. Likewise, this applies to the baseline, which only optimizes the existing code that already works.

V. Conclusion

We present the first framework for diagnosing and repairing motion planners for automated vehicles that leverages both common sense and domain-specific knowledge about causal mechanisms in LLMs. Through a modular and iterative prompt design, our approach automates the generation of descriptions for the planner and continuously enhances diagnostic performance. The major limitation of our approach is that the improvement of the planner cannot be guaranteed. However, as the capabilities of LLMs advance, we anticipate the paradigm to enhance significantly over time. Future work will involve conducting additional tests across various application domains and developing datasets by monitoring user submissions over time. Additionally, we plan to extend the few-shot component with a memory module to leverage experiential learning. We encourage researchers using `DrPlanner` to refine their motion planners and contribute towards establishing a large-scale framework that encompasses a variety of planner types for diagnostic and repair tasks.

Acknowledgment

The authors kindly thank Sebastian Illing for implementing the experiments for the sampling-based planner.

References

[1] B. Paden, M. Čáp, S. Z. Yong, D. Yershov, and E. Frazzoli, "A survey of motion planning and control techniques for self-driving urban vehicles," vol. 1, no. 1, pp. 33–55, 2016.

[2] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," *Proc. of the Advances in Neural Info. Processing Syst.*, vol. 33, pp. 1877–1901, 2020.

[3] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, *et al.*, "Training language models to follow instructions with human feedback," *Proc. of the Advances in Neural Info. Processing Syst.*, vol. 35, pp. 27 730–27 744, 2022.

[4] OpenAI, "GPT-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[5] M. Zucker, N. Ratliff, A. D. Dragan, M. Pivtoraiko, M. Klingensmith, C. M. Dellin, J. A. Bagnell, and S. S. Srinivasa, "CHOMP: Covariant Hamiltonian optimization for motion planning," *Int. J. of Robot. Research*, vol. 32, no. 9-10, pp. 1164–1193, 2013.

[6] T. Gu, J. M. Dolan, and J.-W. Lee, "Runtime-bounded tunable motion planning for autonomous driving," in *Proc. of the IEEE Intell. Veh. Symp.*, 2016, pp. 1301–1306.

[7] S. Aradi, "Survey of deep reinforcement learning for motion planning of autonomous vehicles," *IEEE Trans. on Intell. Transp. Syst.*, vol. 23, no. 2, pp. 740–759, 2022.

[8] S. Liu and P. Liu, "Benchmarking and optimization of robot motion planning with motion planning pipeline," *Int. J. of Advanced Manufacturing Technology*, vol. 118, pp. 949–961, 2022.

[9] H. Krasowski, X. Wang, and M. Althoff, "Safe reinforcement learning for autonomous lane changing using set-based prediction," in *Proc. of the IEEE Int. Conf. on Intell. Transp. Syst.*, 2020, pp. 1–7.

[10] L. Wang, L. Sun, M. Tomizuka, and W. Zhan, "Socially-compatible behavior design of autonomous vehicles with verification on real human data," *IEEE Robot. and Automation Letters*, vol. 6, no. 2, pp. 3421–3428, 2021.

[11] Y. Lin, H. Li, and M. Althoff, "Model predictive robustness of signal temporal logic predicates," *IEEE Robotics and Automation Letters*, vol. 08, no. 12, pp. 8050 – 8057, 2023.

[12] N. Mehdipour, M. Althoff, R. D. Tebbens, and C. Belta, "Formal methods to comply with rules of the road in autonomous driving: State of the art and grand challenges," *Automatica*, vol. 152, no. 110692, 2023.

[13] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Trans. on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2019.

[14] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Trans. on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[15] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proc. of the IEEE Int. Conf. on Software Engineering*, 2013, pp. 802–811.

[16] X. Liu and H. Zhong, "Mining StackOverflow for program repair," in *Proc. of the IEEE Int. Conf. on Software analysis, Evolution and Reengineering*, 2018, pp. 118–129.

[17] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "TBar: Revisiting template-based automated program repair," in *Proc. of the ACM Int. Symp. on Software Testing and Analysis*, 2019, pp. 31–42.

[18] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, "FixMiner: Mining relevant fix patterns for automated program repair," *Empirical Software Engineering*, vol. 25, pp. 1980–2024, 2020.

[19] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *Int. Conf. on Learning Representations*, 2015.

[20] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "CoCoNut: combining context-aware neural translation models using ensemble for program repair," in *Proc. of the ACM Int. Symp. on Software Testing and Analysis*, 2020, pp. 101–114.

[21] Q. Zhu, Z. Sun, Y.-A. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair," in *Proc. of the ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering*, 2021, pp. 341–353.

[22] N. Jiang, T. Lutellier, and L. Tan, "CURE: Code-aware neural machine translation for automatic program repair," in *Proc. of the IEEE/ACM Int. Conf. on Software Engineering*, 2021, pp. 1161–1173.

[23] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *Proc. of the IEEE/ACM Int. Cof. on Software Engineering*, 2022, pp. 1506–1518.

[24] C. S. Xia and L. Zhang, "Less training, more repairing please: Revisiting automated program repair via zero-shot learning," in *Proc. of the ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering*, 2022, pp. 959–971.

[25] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, *et al.*, "CodeBERT: A pre-trained model for programming and natural languages," in *Proc. of the Conf. on Emperical Methods in Natural Language Processing*, 2020, pp. 1536–1547.

[26] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[27] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[28] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-T. Yih, L. Zettlemoyer, and M. Lewis, "InCoder: A generative model for code infilling and synthesis," in *Proc. of the Int. Conf. on Learning Representations*, 2023.

[29] J. Liang, W. Huang, F. Xia, P. Xu, K. Hausman, B. Ichter, P. Florence, and A. Zeng, "Code as policies: Language model programs for embodied control," in *Proc. of the IEEE Int. Conf. on Robot. and Autom.*, 2023, pp. 9493–9500.

[30] S. D. Kolak, R. Martins, C. Le Goues, and V. J. Hellendoorn, "Patch generation with language models: Feasibility and scaling behavior," in *Proc. of the Int. Conf. on Learning Representations: Deep Learning for Code Workshop*, 2022.

[31] J. A. Prenner, H. Babii, and R. Robbes, "Can OpenAI's Codex fix bugs? an evaluation on QuixBugs," in *Proc. of the Int. Workshop on Automated Program Repair*, 2022, pp. 69–75.

[32] C. S. Xia, Y. Wei, and L. Zhang, "Practical program repair in the era of large pre-trained language models," in *Proc. of the Int. Conf. on Software Engineering*, 2023, pp. 1482–1494.

[33] N. Shinn, F. Cassano, A. Gopinath, K. R. Narasimhan, and S. Yao, "Reflexion: Language agents with verbal reinforcement learning," *Proc. of the Advances in Neural Info. Processing Syst.*, no. 377, pp. 8634–8652, 2023.

[34] X. Chen, M. Lin, N. Schärli, and D. Zhou, "Teaching large language models to self-debug," in *Proc. of the Int. Conf. on Learning Representations*, 2023.

[35] N. Jain, T. Zhang, W.-L. Chiang, J. E. Gonzalez, K. Sen, and I. Stoica, "LLM-assisted code cleaning for training accurate code generators," in *Proc. of the Int. Conf. on Learning Representations*, 2024.

[36] A. Madaan, A. Shypula, U. Alon, M. Hashemi, P. Ranganathan, Y. Yang, G. Neubig, and A. Yazdanbakhsh, "Learning performance-improving code edits," in *Proc. of the Int. Conf. on Learning Representations*, 2024.

[37] Y. Lin, S. Maierhofer, and M. Althoff, "Sampling-based trajectory repairing for autonomous vehicles," in *Proc. of the IEEE Int. Conf. on Intell. Transp. Syst.*, 2021, pp. 572–579.

[38] Y. Lin and M. Althoff, "Rule-compliant trajectory repairing using satisfiability modulo theories," in *Proc. of the IEEE Intell. Veh. Symp.*, 2022, pp. 449–456.

[39] S. Maierhofer, Y. Ballnath, and M. Althoff, "Map verification and repairing using formalized map specifications," in *Proc. of the IEEE Int. Conf. on Int. Transp. Syst.*, 2023, pp. 1277–1284.

[40] A. Pacheck and H. Kress-Gazit, "Physically feasible repair of reactive, linear temporal logic-based, high-level tasks," *IEEE Trans. on Robot.*, vol. 39, no. 6, pp. 4653–4670, 2023.

[41] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," *Proc. of the Advances in Neural Info. Processing Syst.*, vol. 35, pp. 22 199–22 213, 2022.

[42] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "ReAct: Synergizing reasoning and acting in language models," in *Proc. of the Int. Conf. on Learning Representations*, 2023.

[43] E. Kıcıman, R. Ness, A. Sharma, and C. Tan, "Causal reasoning and large language models: Opening a new frontier for causality," *arXiv preprint arXiv:2305.00050*, 2023.

[44] H. Sha, Y. Mu, Y. Jiang, L. Chen, C. Xu, P. Luo, S. E. Li, M. Tomizuka, W. Zhan, and M. Ding, "LanguageMPC: Large language models as decision makers for autonomous driving," *arXiv preprint arXiv:2310.03026*, 2023.

[45] L. Wen, D. Fu, X. Li, X. Cai, T. Ma, P. Cai, M. Dou, B. Shi, L. He, and Y. Qiao, "DiLu: A knowledge-driven approach to autonomous driving with large language models," in *Proc. of the Int. Conf. on Learning Representations*, 2024.

[46] W. Wang, J. Xie, C. Hu, H. Zou, J. Fan, W. Tong, Y. Wen, S. Wu, H. Deng, Z. Li, *et al.*, "DriveMLM: Aligning multi-modal large language models with behavioral planning states for autonomous driving," *arXiv preprint arXiv:2312.09245*, 2023.

[47] C. Sima, K. Renz, K. Chitta, L. Chen, H. Zhang, C. Xie, P. Luo, A. Geiger, and H. Li, "DriveLM: Driving with graph visual question answering," in *Proc. of the European Conf. on Computer Vision*, 2024.

[48] C. Cui, Y. Ma, X. Cao, W. Ye, and Z. Wang, "Drive as you speak: Enabling human-like interaction with large language models in autonomous vehicles," in *Proc. of the IEEE/CVF Winter Conf. on Applications of Computer Vision*, 2024, pp. 902–909.

[49] J. Mao, Y. Qian, H. Zhao, and Y. Wang, "GPT-driver: Learning to drive with GPT," *arXiv preprint arXiv:2310.01415*, 2023.

[50] J. Mao, J. Ye, Y. Qian, M. Pavone, and Y. Wang, "A language agent for autonomous driving," *arXiv preprint arXiv:2311.10813*, 2023.

[51] Z. Xu, Y. Zhang, E. Xie, Z. Zhao, Y. Guo, K. K. Wong, Z. Li, and H. Zhao, "DriveGPT4: Interpretable end-to-end autonomous driving via large language model," *arXiv preprint arXiv:2310.01412*, 2023.

[52] L. Chen, O. Sinavski, J. Hünermann, A. Karnsund, A. J. Willmott, D. Birch, D. Maund, and J. Shotton, "Driving with LLMs: Fusing object-level vector modality for explainable autonomous driving," in *Proc. of the IEEE Int. Conf. on Robot. and Autom.*, 2024, pp. 14 093–14 100.

[53] H. Shao, Y. Hu, L. Wang, S. L. Waslander, Y. Liu, and H. Li, "LMDrive: Closed-loop end-to-end driving with large language models," in *Proc. of the IEEE/CVF Conf. on Computer Vision and Pattern Recognition*, 2024.

[54] M. Althoff, M. Koschi, and S. Manzinger, "CommonRoad: Composable benchmarks for motion planning on roads," in *Proc. of the IEEE Intell. Veh. Symp.*, 2017, pp. 719–726.

[55] C. Pek, V. Rusinov, S. Manzinger, M. C. Üste, and M. Althoff, "CommonRoad Drivability Checker: Simplifying the development and validation of motion planning algorithms," in *Proc. of the IEEE Intell. Veh. Symp.*, 2020, pp. 1013–1020.

[56] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–35, 2023.

[57] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou, *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Proc. of the Advances in Neural Info. Processing Syst.*, vol. 35, pp. 24 824–24 837, 2022.

[58] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Proc. of the Advances in Neural Info. Processing Syst.*, vol. 30, pp. 5998–6008, 2017.

[59] J. Yang, H. Jin, R. Tang, X. Han, Q. Feng, H. Jiang, B. Yin, and X. Hu, "Harnessing the power of LLMs in practice: A survey on ChatGPT and beyond," *ACM Trans. Knowl. Discov. Data*, vol. 18, no. 6, pp. 1–32, 2024.

[60] Y. Lin and M. Althoff, "CommonRoad-CriMe: A toolbox for criticality measures of autonomous vehicles," in *Proc. of the IEEE Intell. Veh. Symp.*, 2023, pp. 1–8.

[61] W. Schwarting, A. Pierson, J. Alonso-Mora, S. Karaman, and D. Rus, "Social behavior for autonomous vehicles," *Proc. of the National Academy of Sciences*, vol. 116, no. 50, pp. 24 972–24 978, 2019.

[62] Z. Liu, A. Bahety, and S. Song, "REFLECT: Summarizing robot experiences for failure explanation and correction," in *Proc. of the Conferene on Robot Learning*, 2023.

[63] M. Skreta, N. Yoshikawa, S. Arellano-Rubach, Z. Ji, L. B. Kristensen, K. Darvish, A. Aspuru-Guzik, F. Shkurti, and A. Garg, "Errors are useful prompts: Instruction guided task programming with verifier-assisted iterative prompting," *arXiv preprint arXiv:2303.14100*, 2023.

[64] M. Pivtoraiko and A. Kelly, "Kinodynamic motion planning with state lattice motion primitives," in *Proc. of the IEEE/RSJ Int. Conf. on Intell. Robots and Syst.*, 2011, pp. 2172–2179.

[65] M. Werling, J. Ziegler, S. Kammel, and S. Thrun, "Optimal trajectory generation for dynamic street scenarios in a Frénet frame," in *Proc. of the IEEE Int. Conf. on Robot. and Autom.*, 2010, pp. 987–993.

[66] M. Klischat, E. I. Liu, F. Holtke, and M. Althoff, "Scenario factory: Creating safety-critical traffic scenarios for automated vehicles," in *Proc. of the IEEE Int. Conf. on Intell. Transp. Syst.*, 2020, pp. 1–7.