Automating Real-to-Sim Traffic Scene Generation with Large Language Models

Alex Tianyi Xu

CMU-CS-25-106

April 2025

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Chenyan Xiong, Chair
Reid Simmons

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Keywords: Large Language Models, Self-Driving, Code Generation

Abstract

Simulation-based evaluation of autonomous driving (AD) offers a scalable and reproducible alternative to real-world testing, yet current scenario generation methods often prioritize coverage over realism. This thesis presents an exploration in enabling open-source models to automatically generate realistic traffic scenarios from natural language descriptions of real-world crashes. I conducted a series of experiments to investigate the effectiveness of different inference-time methods in this domain, and proposed a framework for leveraging these approaches to create a dataset that can be used to finetune open-source models with fewer parameters. I found that open-source models can effectively learn from synthetic data generated by closed-source LLMs in the simulator code generation domain: an open-source model finetuned on this new dataset achieves a 87.3% success rate in generating syntactically correct scenarios while also achieving a higher ROUGE-L F1 score for high-level behavioral alignment with the description. This work demonstrates the feasibility of LLM-assisted scenario reconstruction at scale and lays the foundation for open, realistic, and automated evaluation pipelines for AD algorithms.

Acknowledgments

I would like to thank my advisors, Professor Chenyan Xiong and Professor Reid Simmons, for their guidance and support throughout this project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

The evaluation of autonomous driving (AD) is an active interdisciplinary area of research that lies at the intersection of computer vision, reinforcement learning, and probabilistic methods [2, 6, 23, 41]. The impact and significance of this research topic is apparent, as there are already many companies working on deploying self-driving cars in the real world. Although real-world evaluation of autonomous driving algorithms is ideal, the high cost incurred by such evaluation methods makes such methods prohibitive to reproduce and build on. As a result, more researchers have turned to simulation testing as the standard testing environment [6, 28, 30, 40]. There are already many publicly available simulator environments in the driving domain [14, 22, 29].

Despite the existence of such tools, it remains an open challenge to generate safety-critical scenarios in these simulators to effectively test the self-driving algorithms' behaviors in a wide range of traffic situations. There are two main aspects that we need to consider when evaluating a set of generated scenarios: *coverage* and *realism*. A good set of test scenarios should be comprehensive in the variety of cases that it covers, so that one can ensure a self-driving algorithm is safe in all scenarios it will encounter. For the assessment of a self-driving algorithm to be grounded in reality, it must also consist only of scenarios that can and will happen in the real world, ideally at the same regularity as these types of situations in day-to-day driving.

Existing traffic scenario generation methods tend to focus much more on coverage than realism: There is a trend to generate the hardest possible scenarios to trouble self-driving algorithms. While this is a well-justified effort in terms of efficiency (more coverage in less samples) in the comparison of the existing algorithms, these scenario generation methods struggle to show how well a self-driving car would perform if it was deployed on the road today.

With the recent advances in the in-context learning capabilities of large language models (LLMs), new opportunities have emerged in many different research areas and traffic scenario generation is no exception. Several recent works have attempted to generate traffic scenarios based on simple natural language descriptions of a scene [16, 38, 51]. One particular capability of LLMs stands out as the key to this task: code generation. Existing simulators like CARLA [14] come with scenario generation tools (e.g. scenario editors) that can be used to create custom scenarios. These tools are often based on domain-specific languages (DSLs), such as Scenic [19],

which are designed to be efficient representations for the simulator domain. However, these DSLs are not always intuitive and can be difficult and time consuming to learn for non-experts. It is even more consuming to manually write the code to translate a large number of driving scenarios into DSL code. LLMs have shown great promise in generating code in a variety of programming languages, including DSLs. This opens up the possibility of using LLMs to generate traffic scenarios for simulators such as CARLA by simply providing a natural language description of the desired scenario.

While existing approaches have shown promise in generating traffic scenarios using LLMs, they often rely on a limited set of templates or curated snippets. This can lead to a lack of diversity and realism in the generated scenarios. Additionally, these approaches rely on proprietary language models, which can limit reproducibility as there are frequent changes and removal of feature support in their APIs.

1.2 Research Goal

Inspired by the need for a scenario generation method that is well grounded in real world situations, this work aims to develop an automated and reproducible framework to generate faithful reconstructions of real-life traffic accidents using natural language descriptions of crashes from reports. We find that there are publicly available datasets consisting of crash reports from across the U.S. Thus, our goal is to leverage LLM's reasoning capabilities to infer the scenario structure directly from this data. In this work, I aim to answer the following research questions.

1. How effective are the in-context learning capabilities of LLMs for generating code for domain-specific languages like Scenic?
2. How can we leverage open-source LLMs to generate code for traffic scenarios in simulators like CARLA?
3. How can we ensure that the generated scenarios are diverse and realistic, while still being faithful to the original natural language descriptions?

To answer these questions, I first investigated the effectiveness of different inference-time methods in the domain of code generation for traffic scenarios. After finding that state-of-the-art methods in this space are not enough for producing syntactically correct code and often computationally expensive, I then proposed a framework for leveraging these approaches to create a synthetic dataset with some syntactic filtering. This synthetic dataset was then used to finetune open-source models with fewer parameters. Finally, I evaluated the generated scenarios using a variety of metrics for both syntax and semantics.

Comparing the performance of these fine-tuned models to that of the closed-sourced model on unseen samples, I find that open-source models can effectively learn the syntax from the dataset created without losing its adherence to the natural language description: an open-source model finetuned on this new dataset achieves a 87.3% success rate in generating code that was able to construct a valid scenario in Scenic. The model also achieved a high ROUGE-L F1 score of 83.6% for matching high-level behaviors called in the definition of the scenario, which shows that the generated scenarios are more faithful to the original natural language descriptions. Compared to the best in-context learning approach I tested for our setting, which only had 37.1%

success rate in constructing a scenario, this is a significant improvement. This work provides a solid platform on which more domain-specific tuning of these open-source models can be built in the future to make the generated scenarios even more realistic.

Chapter 2

Related Work

Our work primarily builds upon and extends the work in two domains: AD evaluation, and LLM code generation. Below, we summarize related efforts in different areas in these two domains, including the data, methods, tools, and benchmarks used in these fields.

2.1 Autonomous Driving Evaluation

The evaluation of AD algorithms is a critical area of research, especially in the context of safety-critical applications. The process of evaluating AD systems involves simulating various driving scenarios to assess the performance and robustness of the algorithms. This section provides an overview of the methodologies and tools used in AD evaluation, with a focus on scenario generation techniques.

2.1.1 Crash Reconstruction

A previous technical report by Waymo [39] had recreated realistic driving scenarios by manually reconstructing real-world crashes on a small scale. The study evaluated the performance of Waymo Driver in 107 fatal collisions between 2008 and 2017 in Chandler, AZ. They first obtained police reports, scene diagrams, and photos from the Arizona Department of Transportation and selected cases that were compatible with the simulation tools. A third-party engineering firm was then hired to reconstruct these collisions in their custom physics-based simulator based on the pre-crash kinematics. The quality check for the generated scenarios was performed by human visual verification. This approach allowed for counterfactual evaluation, where the AD algorithm replaces the human driver to assess whether it could have avoided or mitigated the crash as both the driver causing the accident and as the driver attempting to avoid collision. The engineers also observed that pre-crash alignment was important when creating the counterfactual simulation, as the AD algorithm could slow down the vehicle way before reaching the crash site (to adhere to traffic rules or due to some preference of the algorithm), making the scenario no longer safety-critical.

Inspired by the crash reconstruction from this report, we wanted to create an automated process to reconstruct real-world crashes. This automation would allow for a much more efficient

and scalable approach to generating realistic driving scenarios for AD evaluation.

2.1.2 Scenario Generation Methodologies

The problem of generating safety-critical scenarios for testing AD algorithms has been a popular research area with many different lines of work. According to a recent literature survey [13], the process of creating safety-critical AD testing scenarios is traditionally divided into three paradigms: data-driven, adversarial, and knowledge-based methods. In this section, we will first focus on non-LLM-based methods, and leave the discussion of LLM-based scenario generation frameworks for section 2.2.4.

Creating a test scenario usually involves specifying a static environment to set the scene (detailing road shape, traffic signs, etc.), configuring the initial conditions of dynamic objects, and describing behaviors of those dynamic objects over time. I will now present several different categories of methods that achieve this goal.

Data-Driven Scenario Generation

Data-driven methods utilize real-world driving data to generate scenarios either by directly sampling critical scenarios or by learning the underlying distribution of such scenarios.

A common direct sampling technique is *data replay*, where selected sequences from road test logs are re-simulated in controlled environments. Variants of this method have proposed clustering for more efficient testing [26, 27]. Companies like Waymo, Baidu, and Uber also augment the data with random perturbations on their physics model parameters to improve the diversity of the generated scenarios [17, 32]. Performing such perturbations on important parameters can cause more scenarios to be safety-critical.

Density estimation methods aim to learn the distribution of critical scenarios. Probabilistic graphical models and deep learning models have been used to generate natural scenarios [44, 45] and can be combined with sampling and clustering techniques [46, 47] to select more safety-critical generations. More recently, deep generative models like variational autoencoders have also been used [8]. These models encode scenarios into low-dimensional latent spaces and are able to generate new scenarios by sampling in the latent space. For example, CMTS [10] proposes interpolating between collision scenarios and normal scenarios in the latent space for data augmentation.

One of the key limitations of data-driven approaches is their dependence on available data. The data used in these works must be collected through real-world car deployment and safety-critical scenarios are very rare, so the data collection process is not cost-effective. Moreover, high-dimensional data, such as LiDAR or videos, make model learning and interpretability challenging.

Adversarial Scenario Generation

Adversarial methods are designed to exploit the weaknesses of AD algorithms and expose their failure modes by actively searching for risky scenarios. These methods seek to maximize danger metrics on the generated scenario by optimizing the dynamic behaviors of non-ego agents based

on these objectives. Some methods in this category [9] optimize over the initial conditions of the scenario while others train a policy model to control the dynamic objects throughout the entire scenario. Reinforcement learning (RL) is commonly used to control agents that interact adversarially with the ego vehicle. Algorithms such as DDPG [31] and A2C [35] are used to train agents who intentionally violate traffic norms or block critical paths, often with the goal of maximizing the likelihood of collision.

Adversarial approaches have shown success in surfacing AD algorithms' weaknesses but are often limited in diversity and realism. They tend to focus on narrow failure cases and require extensive computational resources to explore scenario spaces thoroughly.

Knowledge-Based Generation

Knowledge-based approaches integrate domain knowledge like expert-defined constraints and rules into the generation process [13, 36]. Rule-based generation methods require manually-designed equations and conditions that can be outlined in natural language or formal specification [20, 34]. Many existing simulators also come with test suites where the scenarios are manually designed.

A more recent trend in knowledge-based generation is *knowledge-guided learning*, where domain knowledge (e.g., causal relationships, right-of-way rules) is embedded in the representation space [11, 12]. Some works in this space model scenario generation as a constraint optimization problem to maximize risk metrics while satisfying predefined constraints [3, 25, 37].

The main advantage of knowledge-based methods is their interpretability and the ability to incorporate expert knowledge. However, they often require significant manual effort to define rules and constraints, which can limit scalability and diversity.

2.1.3 Tools and Platforms

A wide range of simulation platforms and benchmarking tools support scenario generation and AD evaluation. Here, I will summarize several notable platforms that are widely used.

CARLA [14] is a widely used open-source 3D simulator built on Unreal Engine. It offers rich assets for urban driving, realistic sensors (RGB, LiDAR, segmentation), and extensible interfaces for vehicle and pedestrian control. As such, it is considered a comprehensive simulator in this space [30]. The CARLA Scenario Runner supports scripting scenarios in Python or using OpenSCENARIO standards [4].

Two other open-source simulators in this space are MetaDrive [29] and Waymax [22]. They are considered driving policy simulators as they offer executable traffic scenarios that focus on the evaluation of a driving policy. They also allow importing real-world trajectories data from open trajectory datasets. Unfortunately, these simulators do not have built-in scenario editors, which makes them less convenient for scenario generation.

Scenic [19] is a domain-specific language that is compatible with multiple simulators. It is most commonly used with the CARLA simulator. The Scenic language is an efficient representation of spatial and behavioral constraints in scenario generations. Scenic is particularly useful for generating diverse and complex scenarios, as it allows users to define high-level scene descriptions that can be automatically translated into executable code.

SafeBench [48] is a unified platform built on CARLA for benchmarking AD algorithms under safety-critical scenarios. It provides a suite of pre-defined routes, scenario definitions based on NHTSA typologies, and evaluation metrics for safety, functionality, and smoothness. Metrics include collision rate, time-to-collision, lane invasion, and off-road distance. SafeBench can be used to compare RL-based agents across scenario generation strategies, offering a standardized evaluation pipeline.

For my work, I will be using CARLA as the simulator and Scenic as the DSL for scenario generation, following the same setting as ScenicNL [16]. That is, I will be training an LLM to generate valid Scenic code that will describe a scenario. This simplifies the task, as the LLM only has to invoke default behaviors from Scenic (such as turning and changing lanes) without coming up with the low-level controls. As Scenic is a probabilistic language, during execution, scenes (containing specific values from the defined distributions) will be sampled from this code and only ones that are consistent with all the constraints will be kept and given to CARLA to simulate. SafeBench presents a promising tool to evaluate AD algorithms on the generated scenarios in the future, but that is not within the scope of this report.

2.1.4 Public Data

Traffic Datasets

Most existing datasets used in AD evaluation consist of sensor data from the real-world deployment of vehicles. One of the most widely used datasets is the Waymo Open Dataset [42], which contains over 1,000 hours of driving data collected in urban environments. It includes high-resolution LiDAR and camera data, as well as annotations for objects, lanes, and traffic lights. The dataset is designed to support vision tasks associated with AD, including object detection, tracking, and semantic segmentation.

Accident Data

Several public datasets collected by the NHTSA provide real-world crash data that can be used for scenario reconstruction. The NHTSA SCI (Special Crash Investigations) dataset includes detailed reports of unique traffic incidents, with top-down scene diagrams, witness statements, and pre/post-crash kinematics. While fewer in number (330 cases), these reports are rich in content and useful for modeling rare safety-critical events. The Crash Investigation Sampling System (CISS) dataset [50] includes over 10,000 crash reports, albeit with briefer descriptions and simpler scene diagrams. Both datasets offer structured summaries and schematic representations of pre-crash conditions, making them promising for LLM-based reconstruction.

The US Accidents dataset provides environmental context (weather, lighting, road type) for over a million incidents, though its natural language descriptions are templated and less informative. This dataset may be able to complement reconstruction efforts in the future by providing context or conditioning variables during scene generation.

For this work, I chose to use the CISS dataset as it contains a large number of crash reports with informative summaries. The size of the dataset will allow me to create a sizable amount of

synthetic data for training. The CISS dataset also contains a variety of crash types, which will help ensure that the generated scenarios are diverse and realistic.

2.2 LLM Code Generation

The recent advances in LLMs have shown great promise in performing many language-based tasks with human-level capability. One active area of research is to enable LLMs to generate code with high accuracy based on a description of the problem. There have been a variety of techniques proposed to improve the performance of LLMs in code generation tasks, including prompting strategies, and finetuning techniques. In this section, I will summarize the relevant work in this area.

2.2.1 Prompting Strategies

The most common inference-time techniques to boost the coding performance of pre-trained LLMs is prompt engineering. Zero-shot prompting describes the requirements of the task and relies mainly on the model's pre-trained knowledge. In few-shot prompting, the model is provided with a few examples of input-output pairs to guide its generation. This approach has been shown to improve the quality of generated code by providing context and structure.

Some other prompting techniques build on these basic frameworks. For example, Chain-of-Thought (CoT) prompting [43] encourages the model to generate intermediate reasoning steps before arriving at the final code. This technique has been shown to improve performance on reasoning and code generation tasks by breaking them down into smaller, more manageable steps. Tree-of-Thought (ToT) prompting takes this a step further and considers many possible paths of reasoning in parallel, allowing the model to explore multiple avenues of thought before converging on a final solution [49]. This approach has been shown to improve performance on complex tasks that require multi-step reasoning.

2.2.2 Code Refinement

Various prompting strategies have also been proposed in LLM-based code repair. For example, self-debug [7] is an approach that iteratively refines the generated code by incorporating the execution feedback. The model first generates a code snippet and is then prompted with feedback to improve the initial code. The feedback can be given at 4 different levels of granularity: (1) simple result of execution (correct/incorrect), (2) unit test feedback (execution results of failed cases and any error messages), (3) code explanation (LLM explanation of generated code), (4) execution trace (generated by LLM reasoning line-by-line). The feedback process can also be iterative and can lead to more accurate and reliable code generation.

2.2.3 Low-Resource Programming Languages

Code generation for programming languages with limited data is a difficult task that has gained a lot of attention in recent years, as LLM code generation in high-resource programming languages

like Python is becoming increasingly popular. This limitation in the quantity of data makes it hard for language models to generalize and produce correct code for these languages based on problem descriptions.

A recent study [21] compares three standard techniques often used to adapt a code LLM for LRPL code generation: fine-tuning, few-shot prompting, and task-oriented pre-training followed by fine-tuning. The authors found that for larger models like DeepSeekCoder-33B, in-context learning approaches tended to perform better as the fine-tuned models easily overfit on the fine-tuning data. On smaller models, the fine-tuning approaches performed better. The inclusion of a tailored pre-training objective had very minor effects on the performance compared to only having finetuning. They concluded that overall, few-shot translation examples are a safer bet when trying to improve LRPL code generation.

An even more extreme case of this data scarcity limitation is code generation for domain-specific languages (DSLs). These languages were created or optimized for specific problem domains and often offer a higher-level of abstraction and efficiency compared to LRPLs [24]. Examples include Ansible and Verilog. As there is often even less data for DSLs, different inference-time techniques like retrieval and iterative feedback have also been applied to improve code generation on these less well-known languages [1, 15, 33].

2.2.4 LLM-Based Scenario Generation

The advances in LLM code generation outlined previously have also enabled AD scenario generation using LLMs.

ChatScene [51] was one of the first systems to apply LLMs to this problem. It prompts GPT-4 to generate high-level natural language descriptions of risky scenarios, extracts semantic components (e.g., road geometry, actor behavior), and retrieves corresponding Scenic code snippets from a curated database. The Scenic language provides a composable DSL for specifying spatial and behavioral constraints. These scripts are executed in CARLA, and the resulting scenes are evaluated using SafeBench. To overcome LLM hallucination and API inconsistency, ChatScene uses a retrieval-augmented generation approach, combining few-shot prompting with a manually-verified snippet database. Despite showing improved scene diversity and criticality, ChatScene is limited by its reliance on synthetic prompts and hand-engineered data.

Text-to-Traffic Scene Generation (TTSG) [38] extends ChatScene by discarding fixed routes and agent configurations. Instead, it prompts GPT-4o to reason about the road conditions required for a scenario, then retrieves the appropriate road segments and agent types from a large database. Scene construction is done using a custom renderer rather than Scenic. While TTSG improves scenario flexibility and realism, it still starts from brief, artificial textual prompts rather than real-world incident data.

ScenicNL [16] is an LLM-based framework designed to translate natural language descriptions from autonomous vehicle crashes into probabilistic scenario programs using the Scenic language. This approach enables the modeling of uncertain and variable aspects of real-world incidents. By leveraging a combination of large language models, prompting strategies, constrained decoding, and simulation tools, ScenicNL [16] attempts to generate semantically and syntactically correct Scenic code, providing a promising path for automated scenario generation from real-world crash reports.

For this work, I studied the effectiveness of in-context learning methods like the one used in ScenicNL [16]. After finding that such approaches struggle to generate syntactically correct code, I devised a pipeline for turning the valid generations from these prompting techniques into training data for finetuning a smaller model. Furthermore, I relied on unconstrained generation instead of constrained decoding to improve the diversity of the generated scenarios. This approach allows for more flexible and creative generations, enabling the model to produce a wider range of scenarios that may not be captured by fixed templates or snippets.

Chapter 3

Method

From the existing literature from Section 2.2.4, one can observe that there is a strong tendency for LLM-based scenario generation methods to rely on closed-source models like GPT-4. This dependence may be justified by the state-of-the-art in-context learning performance of these closed-source models, as well as the under-exploration of code generation in this domain. However, the limitations of closed-source models are also significant. For instance, many of these methods struggle to generate syntactically correct code without relying on templates or even directly combining human-written snippets. Frequent updates to the closed-source API may cause some features to become unavailable and render certain methods unreproducible.

To address this limitation, I try to leverage open-source models for LLM generation. The method proposed in this report only uses the most basic completion functionality of GPT-4o to use its in-context learning ability in the creation of synthetic data, which can be used to train open-source models. Moreover, instead of relying on a static snippet database or templated methods like constrained decoding, we explore using LLM for unconstrained generation for more diverse and flexible program synthesis as well as better scalability.

3.1 Overview

The new framework I propose can be split into the following parts: data processing, prompting, finetuning, and evaluation. The data processing stage obtained the crash report summaries from CISS data collected by the NHTSA and filtered them for scenario generation. The prompting stage used the crash report summaries to prompt a closed-source LLM to generate Scenic code using few shot examples. At this stage, I will also analyze the performance of the existing in-context learning methods on our task. The valid code generations were then selected to create a dataset that can be used to train an open-source model to output code in the Scenic format. Finally, to evaluate the generated scenarios, we use several code formatting metrics as well as text-based semantic similarity metrics to assess the generated scenarios.

3.2 Data Preprocessing

The first step in our framework is to process the crash report summaries from the CISS dataset. To do this, we need to consider the ability of the simulator to recreate the scenarios. Since the Scenic language and the CARLA simulator have built-in behavior definitions that are only available for drivable lanes inside the road limits, we first filter out single vehicle accidents as most of these are collisions between the ego vehicle and an obstacle. We then used a simple web scraper to get the remaining accident summaries from the NHTSA Crash Viewer website. After filtering out all the cases that did not have a brief accident summary, 10,727 cases remained. At this point, I reserve 10% of these cases to use as the test set. The remaining examples would be used for prompt tuning and creation of synthetic data for training.

3.3 Synthetic Data Generation

The next step in our framework is to use the crash report summaries to prompt a closed-source LLM to generate Scenic code. The purpose of this generation is to create pairs of crash descriptions and compilable Scenic code so that open-source models can be trained with this data. I decided to use GPT-4o as closed-source model for the code generation process as it is the latest instruction-following LLM and had similar characteristics as GPT-3.5-Instruct, which is frequently used in previous state-of-the-art LLM-based scenario generation frameworks [16, 38, 51].

3.3.1 Prompting Strategies

I first studied the performance of various prompting techniques on a randomly selected subsample of the test set, which was created in the data preprocessing in Section 3.2. This prompt subset contained 100 cases. I compared a modified version of ScenicNL [16] against different variants of common in-context learning prompting methods. Here is a description of the prompting strategies we used:

1. **Zero-shot**: I provide the model with a crash description and ask it to generate Scenic code without any examples. This is the simplest form of prompting and serves as a baseline for comparison.
2. **Few-shot**: I provide the model with a few examples of crash descriptions and their corresponding Scenic code. This allows the model to learn from the examples and generate more accurate code.
3. **Few-shot with CoT**: Typically, CoT in code generation refers to adding a planning process in each example before writing the code that implements this plan. The code also usually contains comments to inform the LLM of notable intermediate steps. Since Scenic code for the purpose of reconstruction broadly follows a fixed structure, I provide a walkthrough of the structure of the snippets directly following the task description as this basic structure applies to all examples. The model is then provided with a few examples of crash reconstructions. Each of these examples contains the crash description, a plan of action

for reconstruction, and the Scenic code with comments specifying intermediate steps and pointing out any deviation from the generic structure given at the start.

4. **Modified ScenicNL** [16]: With the removal of the logit bias feature from the OpenAI API, constrained decoding approaches are no longer available through open-source libraries like LMQL [5]. As a result, the original ScenicNL pipeline is no longer reproducible. We make the minimal modification of excluding constrained decoding from their pipeline and keep everything else as implemented in the original ScenicNL paper. The remaining parts of the approach includes the ToT reasoning for understanding the different components of the described accident, and retrieves the most suitable examples in the few-shot prompt.

3.3.2 Description Simplification

While examining the descriptions from the data collected, I observed that some descriptions contain precise lane information and post-crash dynamics, which are distracting information in the code generation process. Here's an example of such a description:

```
V1 was traveling westbound on a five lane, two way roadway approaching
 a light controlled intersection. V2 was traveling westbound on a five
 lane, two way roadway approaching the same light controlled
intersection. At the intersection the front of V1 contacted the left
side of V2. V2 continued on and contacted a pole with its right side
before coming to rest.
```

There are a few problems that could be caused by these details at the synthetic data generation stage. First, since Scenic code can only describe scenarios on the default towns in CARLA, there are limited options for lane types and road shapes. Thus, if a strict condition (such as finding a 7-lane road) is imposed in the Scenic code, the simulator will not be able to recreate the scenario. In these cases, recreating a simpler version of the scene is better than providing no valid reconstruction at all, so simplifying the description could help the LLM come up with a simpler reconstruction for complicated scenes. Second, the CARLA simulator does not support the specification of post-crash dynamics, so we cannot use this information in the code generation process.

Thus, before prompting the LLM to generate code, we first ask it to remove detailed lane information and post-crash dynamics by prompting GPT-4o to rewrite the description using a simple zero-shot prompt. No prompt engineering was necessary, as this initial prompt provided results that aligned with expectations. The full prompt is provided in Listing A.1 in the appendix.

The following is the simplified version of the previous description.

```
V1 was traveling westbound on a roadway approaching a light-controlled
 intersection. V2 was also traveling westbound on the same roadway
approaching the intersection. At the intersection, the front of V1
contacted the left side of V2.
```

An analysis of the effect of this simplification is presented in Section 4.1.2.

3.3.3 Code Correction Techniques

Using the code outputs from the best-performing prompting strategy, I also experimented with two different code correction methods: Self-Debug and a simple domain-specific strategy for CARLA.

1. **Self-Debug**: Self-debugging was implemented here with unit test feedback. Since there are no inputs or test cases to the Scenic snippet, we only consider the scene construction correctness as it is deterministic in the Scenic execution process. More specifically, we try to invoke the Scenic Python API to construct a scenario from the generated code and check for errors. If any errors are found, the model is prompted with the original task description, accident description, the generated code, and the error message to generate a new version of the code to fix the error. In the experiments, I kept all self-debugging to a single iteration.
2. **Map Replacement**: A simple strategy for improving Scenic code is to try the proposed code on other maps. The map definition part of the Scenic language is very templated and can only be specified by changing two global parameters in Scenic: `map` and `carla_map`. These parameters specify the OpenDrive file to load the CARLA map into Scenic during scenario construction. To replace the map, I simply search for these statements and replace the town number in these statements to get a modified version of the snippet. I then attempted to construct scenarios for these modified versions to see if it can succeed in scenario construction. I hypothesize that this method would work well on code that imposed strict restrictions on the road configuration and helps mitigate the effect of CARLA's limited default map pool.

3.3.4 Final Synthetic Dataset

In the end, to generate Scenic code for the synthetic dataset, I decided to use the few-shot CoT prompt with map replacement on the simplified description. The main reason for this choice is that this prompt performed the best in generating code that led to successful scenario *constructions*, and that it has a manageable API cost. The reason for choosing the construction metric is that it is the strictest yet deterministic syntax checker in this setting. This will ensure that the generated code is syntactically correct while ensuring that we don't miss examples that failed the generation step as a result of sampling stochasticity. Finally, map replacement also helped me maximize the amount of code that could be used in fine-tuning. The final synthetic dataset contained 5345 examples.

3.4 Finetuning

3.4.1 Training Details

To train the model, I used the Qwen-2.5 Coder-1.5B model as the base model as it has been shown to perform well on code generation tasks with Python. Since Scenic is a DSL based on Python, this is a suitable base model to fine-tune.

During the training process, I got rid of the few-shot prompt used in the synthetic data generation and instead opted for a simple zero-shot prompt consisting of only the crash description, which is followed by the generated Scenic code as a response. This is because the base coding LLM is not well suited for in-context learning, unlike other instruction-tuned open-source models. Each response is also followed by a special token that indicates the end of the code generation. This is important as it allows the model to learn where each generated snippet should end.

For the hyperparameters for fine-tuning, I mostly followed the default supervised fine-tuning optimizer hyperparameters in Open-R1, which specifies a learning rate of 2e-5 and a warm-up phase. The loss function I use here is the cross-entropy loss on next-word prediction, which is commonly used for finetuning language models. I modified the maximum sequence length to 2048 as the generated code snippets were generally quite short. I also altered the batch size to train on fewer GPUs.

I finetuned the model for 5 epochs since the dataset is quite small. I also divided the created dataset into a training set and a validation set, following a standard 80:20 split. I then plotted the loss curves across the training steps to check for overfitting and decide on the checkpoint to use for the final evaluation on the test set. I also evaluated the model every 500 training steps on both syntactic and semantic metrics on the validation set to see how quickly the model learns to perform well on each metric.

3.5 Evaluation

3.5.1 Terminology

First, we need to differentiate the terminology used in the definition of the evaluation metric. This set of terminology is consistent with the documentation of the Scenic language. A typical Scenic script contains definitions of behaviors, road configurations, some constants for speeds and distances, where the cars should be spawned, and the *requirements* that should be met in a generated scene. Here are the definitions of the terms that are relevant for specifying the syntactic metrics:

- **AST**: An abstract syntax tree (AST) is a tree representation of the structure of the Scenic code. It is generated by the Scenic syntax checker when parsing the code.
- **Scenario**: A scenario is a framework that defines the broad structure of a scene. It mainly revolves around the definition of sampling distributions, which can include the constraints on the vehicle speeds and roads. The map specified is also checked in this step to ensure that the domains defined for sampling are non-empty.
- **Scene**: A scene is an instance sampled from a Scenic scenario using rejection sampling based on the initial configuration requirements defined in the scenario. This sampling process rejects any generation that does not satisfy all specified *requirements*.

This is a brief overview of the Scenic execution process, summarized according to the information from the Scenic documentation [18]

3.5.2 Metrics

Syntax

For evaluation, we use a series of fine-grained syntactic metrics that measure the correctness of the code format in increasing difficulty:

1. **Compile Success Rate**: A Scenic snippet is considered compilable if the Scenic syntax checker can successfully parse the code and generate an *AST*.
2. **Scenario Construction Success Rate**: A Scenic snippet is considered constructable if the CARLA simulator can construct a *scenario*.
3. **Generation Success Rate**: A Scenic snippet is considered generatable if the CARLA simulator can sample a *scene* from the defined scenario within a certain iterations of rejection sampling (experiments in this report use the default value of 2000 iterations).

Here are a few examples to clarify each of these metrics.

1. **Compile Failure**: Here is an example of the relevant part of a Scenic snippet that does fails to compile.

```
...
require (distance to intersection) <= 30
require (distance from v2 to intersection) <= 30
terminate when ego collides with v2
```

   This fails because n the last line, `collides with` is not a Scenic keyword that can be used in a termination condition.

2. **Construct Failure**: Here is an example of a snippet that fails construction. Most of the snippets that fail this stage are caused by hallucinations of non-existent properties or functions.

```
...
## SELECTING ROAD AND INTERSECTION
dividedRoads = filter(lambda i: i.isPhysicallySeparated, network.
   allRoads)
intersection = Uniform(*network.intersections)
...
```

   The error here is that Road objects have no attribute named `isPhysicallySeparated`.

3. **Generate Failure**: Here is an example of code that fails to generate a valid scene.

```
...
## SELECTING ROAD AND INTERSECTION
threeLaneRoads = filter(lambda r: len(r.laneGroups) == 3, network
   .allRoads)
fiveLaneRoads = filter(lambda r: len(r.laneGroups) == 5, network.
   allRoads)
intersection = Uniform(*filter(lambda i: i.is4Way, network.
   intersections))
```

```
v1_start_lane = Uniform(*intersection.incomingLanes)
v2_start_lane = Uniform(*intersection.incomingLanes)

require v1_start_lane in fiveLaneRoads
require v2_start_lane in threeLaneRoads
...
```

This snippet starts by selecting roads by the number of lanes it should have and all four way intersections. It then samples two lanes from these roads and an intersection uniformly at random and requires that the two lanes should meet at the sampled intersection. Considering a large map with many intersections and roads, this is highly unlikely, which is why none of the 2000 samples met this requirement.

I would like to acknowledge that these are not perfect metrics for syntactic correctness as there could be cases where the code was syntactically valid but fail construction due to a limitation of CARLA's default map pool or cases that fail generation due to chance. However, it is difficult to distinguish if the fault of failure lies in the code generation or a limitation of the tools without examining at the code manually, and this classification is the closest I could get to breaking down the mistakes.

Semantics

One simple way to measure if the scenario generated at least makes some sense for the description given is by counting the number of vehicles in the scene. This can serve as one behavioral metric that I would expect the model to be correct on most of the time.

I also propose a text-based similarity metric to evaluate the adherence of the generated code to the original description. The idea is to compare the sequence of invoked high-level behaviors in the code against the expected behaviors inferred from the scenario description.

The vehicle behaviors defined in each Scenic snippet can be broken down into a sequence of high-level behaviors, such as `TurnBehavior`, and `FollowLaneBehavior`, according to the order in which these behaviors are activated in the script. We can extract a similar sequence from the scenario description using an LLM as a synthetic labeler. In this case, GPT-4o was used as the synthetic labeler

Then, to compare these two behavior sequences, I use ROUGE-L, a commonly used metric to measure sequence similarity. Let the sequence from the generated code be $\hat{y}$ and the reference sequence from the crash description be y. ROUGE-L is based on the length of the longest common subsequence (LCS) between the two sequences and is defined as follows:

$$P = \frac{\text{LCS}(\hat{y}, y)}{|\hat{y}|} \quad \text{(Precision)}$$

$$R = \frac{\text{LCS}(\hat{y}, y)}{|y|} \quad \text{(Recall)}$$

$$F_1 = \frac{2PR}{P + R} \quad \text{(F1 score)}$$

This metric provides a quantitative measure of behavioral fidelity. A higher ROUGE-L score indicates a greater alignment between the intended behaviors described in the natural language input and the behaviors realized in the Scenic program. The F1 metric here penalizes both omissions (low recall) and irrelevant additions (low precision), making it a good indicator of overall behavioral faithfulness. Finally, since a scene can have multiple vehicles, I take the average of the ROUGE-L scores across all the vehicles.

Chapter 4

Results

4.1 Prompting

4.1.1 Prompting Strategies

As previously described, I first performed prompt engineering on a set of 100 cases randomly chosen from the test set. Table 4.1 shows the success rates for compilation, scenario construction, and generation by applying each of these prompting strategies. Here, I tried using both the original descriptions and the simplified descriptions from Section 3.3.2, as well as different phrasings of each prompting strategy, and report the best results for each strategy.

Table 4.1: Scenic code generation performance of various prompting methods.

Prompting Method	**Compile (%)**	**Construction (%)**	**Generate (%)**	**API Cost ($)**
Zero-shot	9	0	0	0.636
Few-shot	77	39	22	1.192
Few-shot with CoT	**90**	**51**	**40**	1.913
Modified ScenicNL [16]	78	40	19	10.85

Zero-shot prompting was unable to generate syntactically correct code effectively, only passing compilation 9% of the time. This is not surprising as the model has no examples to learn from. It is also consistent with the trend from the ScenicNL [16] paper, where the zero-shot prompting method had 1% syntactic correctness. The zero-shot method was also unable to generate any valid code that could be executed in the simulator.

The few-shot prompting method was able to generate code with a compile success rate of 77%, which is significantly higher than the zero-shot method. The few-shot with CoT method further improves all three metrics by more than 10% each. This shows that the planning step and walkthrough of the Scenic code structure was able to help the model learn the code syntax better. However, the construction and generation success rates are still quite low, indicating that there is still room for improvement in the code generation process.

The modified ScenicNL [16] method performed similarly to the few-shot prompting strategies. The reduction in performance compared to the results reported in the original ScenicNL paper [16] might be due to several reasons. First, the original paper only reported results on crash cases that they classified as easy, meaning there is only one dynamic vehicle from the description. This is different from the cases obtained from the CISS dataset as most of the CISS cases contain multiple dynamic vehicles. It could also be because of the removal of constrained decoding. If so, it would mean that the ScenicNL method depends heavily on this templated form of generation, limiting the expressiveness of the resulting generated code.

Another important detail to notice is in the last column of Table 4.1: the API cost of the modified ScenicNL method costs is more than 5 times that of the other methods! The higher cost is associated with the Tree-of-Thought reasoning that it uses, which asks the LLM to generate multiple answers to reasoning questions in the planning stage in order to synthesize the final output, significantly driving up the number of tokens in the LLM output.

4.1.2 Effect of Description Simplification

To understand the effect of the description simplification step, I also performed an ablation on this using the few-shot CoT method. Table 4.2 shows the effect of this simplification.

Simplification	Compile	Construct	Generate	Total API Cost
Yes	90%	51%	40%	$1.913
No	87%	36%	19%	$1.968

Table 4.2: Effect of simplification on various scenario generation stages and API cost.

Note that both the generation success rate and the construction success rate increased significantly when using simplified descriptions, suggesting that the additional information in the original prompts were indeed distracting during code generation. The simplification also lowered the API cost slightly due to a shorter input length. This increase in the construction success rate led me to use the simplified descriptions to generate the code in the final dataset.

4.1.3 Code Correction

Table 4.3 shows the improvements that were obtained by applying self-debug and map replacement on the outputs of the few-shot prompting method with chain of thought explanations. This experiment was performed on the same 100 cases as the search for the best prompting method. The self-debugging method was applied to the code generations that either failed compilation or scene construction, while the map replacement method was only applied to the code generations that failed scenario construction due to an empty domain error, which is often associated with strict conditions on the road specification in the Scenic code. The results are shown in Table 4.3.

Here, both methods improve the construction and generation success rates slightly – coincidentally by the same amount. Note that map replacement only targets the construction success rate, as it can only fix issues related to limitation of the default map used. Interestingly, while

Table 4.3: Effectiveness of self-debug and map replacement on the code generation formatting.

Prompting Method	**Compile (%)**		**Construction (%)**		**Generate (%)**	
	Acc.	Δ ↑	Acc.	Δ ↑		
Few-shot with CoT	90		51		40	
+ Self-Debug	94	+4	56	+5	42	+2
+ Map Replacement	90	+0	56	+5	42	+2

self-debugging improved construction success rate by 5% (when taking the better generation from the initial and fixed versions of the code), it also sometimes edited samples unsuccessfully. In another 5% of the cases, the self-debugging method caused code that initially passed compilation to fail compilation after correction. This is likely due to the fact that the self-debugging method is not deterministic and can sometimes generate code that is syntactically incorrect.

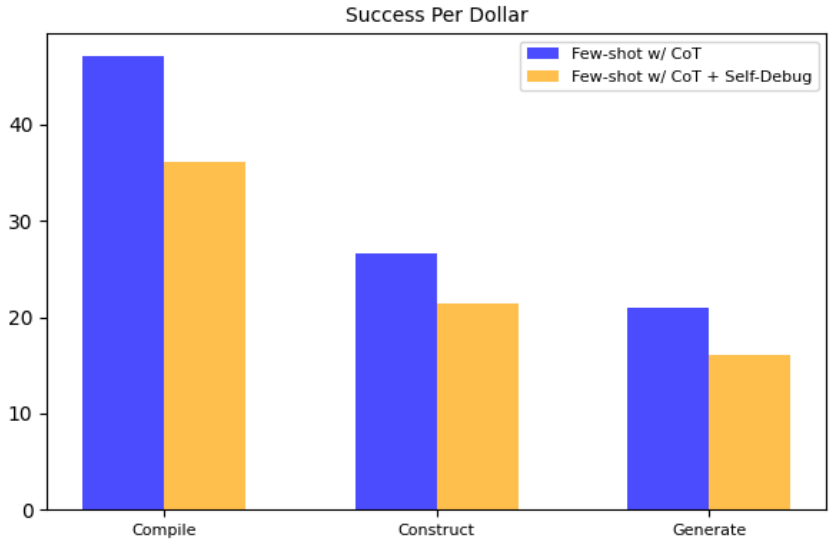


Figure 4.1: Cost efficiency of different prompting methods.

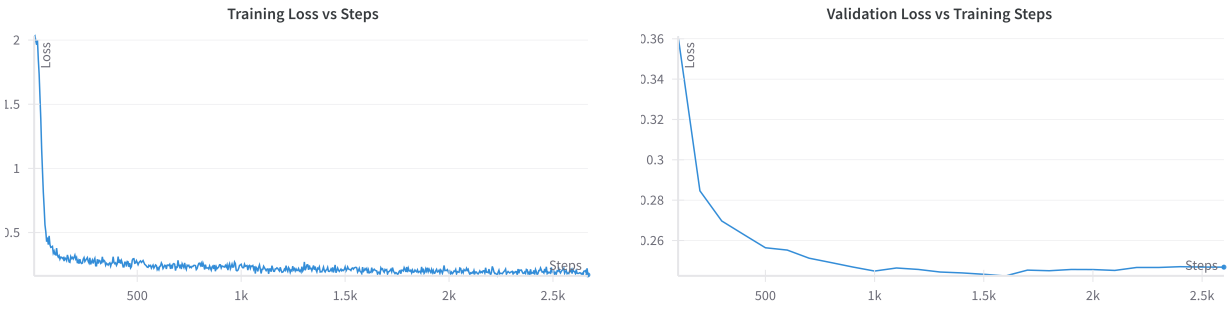
Another dimension that one should consider when applying self-debugging is the cost efficiency, which I measure here by the number of successful Scenic snippets one can expect to get from the method by spending a dollar on the OpenAI API. Since the code correction prompt often contains a long context, it is important to consider whether attempting to self-debug is even worth it. Figure 4.1 is a cost efficiency chart that showing the cost efficiency on the baseline approach of few-shot CoT along with the addition of self-debugging.

Here, we can see that the self-debugging method decreases the cost efficiency of the code generation process significantly while only increasing the success rates very slightly. On the other hand, map replacement delivers similar levels of construction success rate increase by simply replacing the map, without any additional calls to the OpenAI API. This is a strong indication that self-debugging is not worth the cost in this case, especially considering the fact

that map replacement can provide similar improvements at no additional API cost.

4.2 Finetuning

The finetuning process was performed for 5 epochs, and the training and validation cross-entropy loss curves are shown in Figure 4.2. The training loss curve shows a steady decrease in loss over the course of training, indicating that the model is learning to generate Scenic code from the crash report summaries. The validation loss curve also shows a steady decrease, indicating that the model is generalizing well to unseen data. There are some signs of overfitting past 1500 steps, and at 1000 steps, the loss has essentially converged, so I chose to use the model checkpoint at 1000 steps as the fine-tuned model to be used when evaluating on the test set.



(a) Training loss curve

(b) Validation loss curve

Figure 4.2: Training and validation loss curves during the finetuning of Qwen2.5-Coder-1.5B.

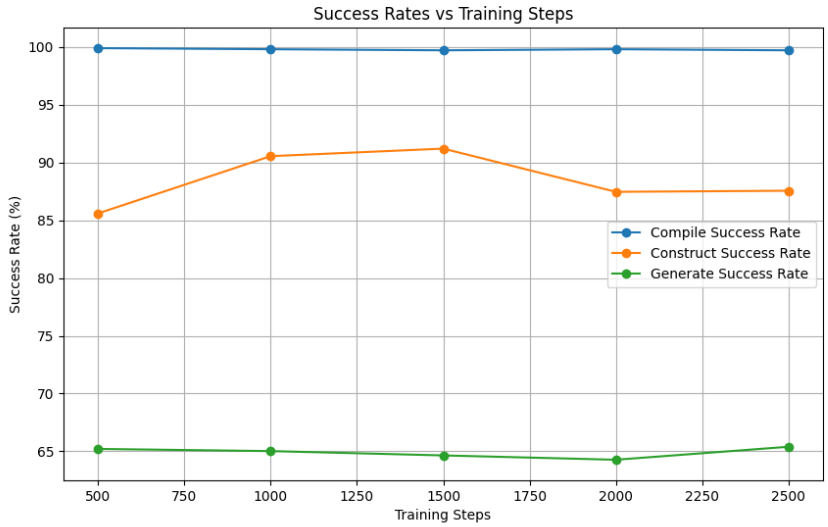


Figure 4.3: How the syntactic metrics on the validation set changed across training steps during finetuning.

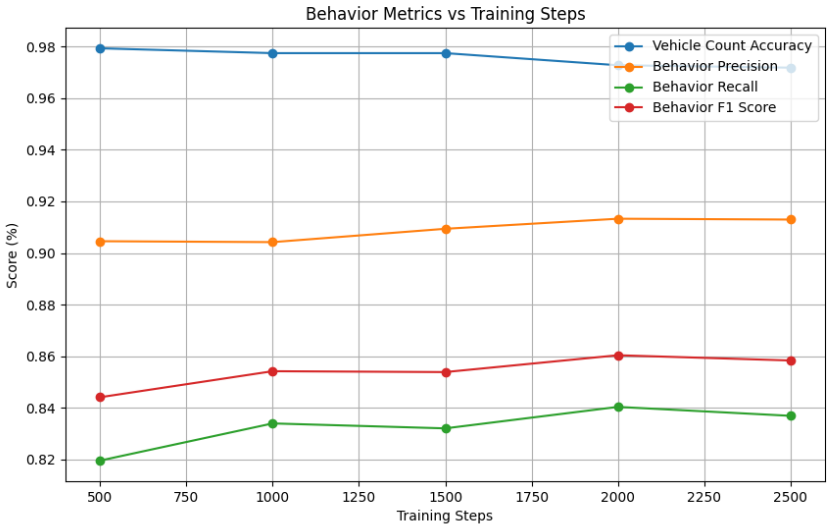


Figure 4.4: How the behavioral semantic metrics on the validation set changed across training steps during finetuning.

To better understand how the model's syntactic and semantic capabilities changed over the fine-tuning process, I evaluated the model on syntactic and semantic metrics every 500 training steps. As shown in Figure 4.3, the compile success rate is consistently high, indicating that the model was able to generate code to effectively use the Scenic language after the first 500 steps. The generation success rate remained roughly the same past 500 steps as well. Generating code that consistently passed scenario construction took the longest to learn as there is still a significant increase from 500 steps to 1000 steps.

From the plot of the behavioral metrics in Figure 4.4, I saw that the model was also able to align itself with the high-level behaviors that GPT selected from description. Examining the F1 score, most of that adjustment occurred within the first 1000 steps. The vehicle count accuracy was close to 100% even at the start, which is what I expected, although this metric did tend to decrease after 1500 steps, indicating that the model might be overfitting on the training set.

Method	**Compile**	**Construct**	**Generate**	**Vehicle Count**	**Behavior F1**
In-Context	87.3%	37.1%	22.4%	92.1%	81.6%
Fine-Tuned	99.9%	87.3%	58.7%	97.1%	83.2%

Table 4.4: Comparison of the best performing in-context learning method and the fine-tuned model across both syntactic and semantic metrics on the held-out test set.

I also evaluated the finetuned model on the full test set of unseen crash descriptions. Table 4.4 compares the fine-tuned model to the in-context learning method I used for synthetic data generation (few-shot CoT prompting) on the full test set. The finetuned model was able to achieve a compile success rate of 99.9%, a scenario construction success rate of 87.3%, and

a scene generation success rate of 58.7%, significantly improving the syntactic correctness of the generated code from in-context learning approaches. At the same time, the finetuned models also outputted code that aligned slightly better with the vehicle behaviors from the original crash description. This shows that the finetuned model is able to generate Scenic code that are highly accurate to the original scene the and is able to construct scenarios that can be executed in the CARLA simulator, beating the previous in-context learning methods that rely on closed-source models.

4.2.1 Qualitative Evaluation

To qualitatively evaluate the generated scenarios, I ran the generated code snippets from the finetuned model (after all 5 training epochs) in the CARLA simulator and observed the vehicle behaviors and interactions. The goal of this evaluation is to assess the realism of the generated scenarios. I used a set of 100 randomly selected scenarios from the finetuning dataset and ran them in the CARLA simulator. I observed that the generated scenarios were able to recreate most of the pre-crash conditions described in the crash reports, and the vehicle behaviors were mostly consistent with the descriptions.

One important limitation of the current generations is the timing of the vehicle interactions. The generated scenarios often have vehicles that do not crash into each other even though both are following the correct trajectory. This is likely because it is difficult to model the timing of the vehicle interactions in the Scenic language. Even for a human expert, writing a good Scenic code snippet that captures the timing of the vehicle interactions is a difficult task that requires a lot of tuning and testing. To address this limitation, additional steps can be taken to tune the open-source model to better capture the timing of the vehicle interactions. One possible approach is to use reinforcement learning to optimize the timing of the vehicle interactions in the generated scenarios, which will be discussed in Section 5.1.

Figure 4.5: Simulation frames showing the progression of two vehicles missing each other at the intersection using code generated from the description in Example 1.

Figure 4.6: Simulation frames showing the progression of two vehicles successfully colliding at the intersection using code generated from the description in Example 1.

Examples

A few examples of the generated scenarios are presented here:

1. Here's the description of the first example.

   ```
   V1 was traveling westbound in the number 2 lane of a 2 lane
   roadway approaching a four way intersection. V2 was traveling
   northbound in the number 2 lane approaching the same intersection
   . Both V1 and V2 entered the intersection and the front of V2
   contacted the left of V1. After impact both vehicles rotated
   toward the northwest intersection apex and the left side of V1
   contacted the right side of V2.
   ```

   Figures 4.5 and 4.6 show the progression of two scenes sampled from a scenario constructed from this description. The first figure shows a miss between the two vehicles, while the second figure shows a collision between the two vehicles. This is a difficult situation to reconstruct as it requires the programmer to constrain the generation conditions enough to consistently cause a collision where one car runs into the side of another. The generated code from the fine-tuned model is not yet able to generate such collisions consistently.

2. Here's another relatively simple example.

   ```
   V1 and V2 were heading eastbound. V2 slowed down in roadway due
   to traffic. The front end of V1 impacted the back end of V2.
   ```

   This example is a simple rear-end collision. For this example, the LLM generated a scenario that led to collisions more consistently. Figure 4.7 shows an example of a generated scene.

As we can see from these examples, the generated scenarios were mostly able to recreate the pre-crash conditions described in the crash reports, but the script does not necessarily lead to a crash. This is an area for future work and improvement.

Figure 4.7: Simulation frames showing the progression of a rear-end collision using code generated from Example 2.

Chapter 5

Conclusion

In this report, I presented a framework for generating realistic traffic scenarios based on real-world crash reports. The framework consists of four main components: data processing, prompting, finetuning, and evaluation. I used the NHTSA CISS dataset to collect crash report summaries and simplified them for scenario generation. Then, I experimented with a variety of prompting strategies using GPT-4o and found that in-context learning is insufficient for this problem. Using the best-performing prompt as well as code correction techniques, I constructed a finetuning dataset from the generated code snippets and trained an open-source model to generate Scenic code. The finetuned model was able to construct valid scenarios with a success rate of 87.3%, outperforming closed-source models significantly in learning the code syntax while also maintaining faithfulness to the behavior of the vehicles described in the original crash report. Finally, I also observed that while each vehicle's behavior in the final simulation adhered to the description, the current fine-tuned models were not always able to cause a collision as a result of bad timing, which brings us to future work that could be done to address this limitation.

5.1 Future Work

While the finetuning of open-source models showed promise for syntactically correct crash scene code generation, there are still many areas for improvement in LLM-based scenario generation methods, as I briefly discussed in the qualitative observations.

First, more comprehensive semantic evaluation metrics are needed to determine the quality of the generated scenes without human labeling. Currently, the semantic evaluation metric is text-based, which means that it only examines the behaviors defined in the code without verifying the resulting simulation. One caveat of Scenic behavior definitions is that not all parts of the calls to the high-level behaviors are necessarily executed in the final simulation. For example, if a script defines a long list of behaviors but the scene terminates after 10 seconds, the car may have only performed the first few maneuvers. This means that the behavior sequence may not accurately reflect the actual behavior of the vehicles in the simulation. This is a limitation of the current evaluation metric, which could be addressed by adding validation of vehicle behaviors from reading the simulation logs.

One of the most significant limitations in the current framework is that the generated scenes

do not necessarily result in a collision. One possible way to address this in the future is to use reinforcement learning to optimize the timing of the vehicle interactions in the generated scenarios. This would involve defining a rule-based reward that provides rewards based on the existence and correctness of a collision. In this case, the correctness of the collision could mean the actors that are involved in the collision, as well as the direction of the collision. This correctness metric is important to prevent reward hacking, which is a common problem in reinforcement learning. If the only metric was the collision rate, the model could learn an easy way to cause a collision like always making one car collide into the back of another. This would not be a realistic scenario, as it does not reconstruct the original crash. Thus, it is important to define a reward function that captures collision correctness and combine the existing metrics like the behavioral alignment score as rewards during reinforcement learning as well.

Finally, our method also relies on the assumption that descriptions contain sufficient semantic cues to infer the details needed to reconstruct the crash. However, in practice, some crash report summaries may omit important contextual or temporal details. This leads to an oversimplification of the scenario, resulting in an unrealistic reconstruction. Future work could address these limitations by incorporating multi-modal signals such as the scene diagram from the crash report, or more detailed descriptions of pre-crash conditions as supervision or input in training to create more realistic scenes.

Appendix A

Full Prompt Templates

This appendix includes the prompt templates used for synthetic code generation and description simplification, as well as the format of examples used with the prompts.

Listing A.1: Prompt for description simplification

```
Please simplify the following description of a crash scenario. You
should remove any information regarding the number of lanes in the
road. You should also remove any information regarding what happened
after the first moment of any vehicle-to-vehicle collision.

Description: <DESCRIPTION>

Simplified:
```

Listing A.2: Zero-Shot Prompt

```
Scenic is a probabilistic programming language based on Python that is
 used for modeling the environments of autonomous cars. A Scenic
program defines a distribution over scenes, configurations of physical
 objects and agents. Scenic can also define (probabilistic) policies
for dynamic agents, allowing modeling scenarios where agents take
actions over time in response to the state of the world. We use CARLA
to render the scenes and simulate the agents. Please generate a Scenic
 program for the scenario in the following description:
<DESCRIPTION>
```

Listing A.3: Few-Shot Prompt

```
We want to write a Scenic program to reconstruct a scenario based on a
 text description.

Scenic is a probabilistic programming language based on Python that is
 used for modeling the environments of autonomous cars. A Scenic
program defines a distribution over scenes, configurations of physical
 objects and agents. Scenic can also define (probabilistic) policies
```

```
for dynamic agents, allowing modeling scenarios where agents take
actions over time in response to the state of the world. We use CARLA
to render the scenes and simulate the agents.

Here are some example of a Scenic programs based on crash descriptions
.
<EXAMPLE 1>

<EXAMPLE 2>

<EXAMPLE 3>

------------------------

Now that you have seen how to write a Scenic program, write one for
the following description.
Description: <DESCRIPTION>
```

Listing A.4: Few-Shot Prompt with CoT

```
Scenic is a probabilistic programming language based on Python that is
 used for modeling the environments of autonomous cars. A Scenic
program defines a distribution over scenes, configurations of physical
 objects and agents. Scenic can also define (probabilistic) policies
for dynamic agents, allowing modeling scenarios where agents take
actions over time in response to the state of the world. We use CARLA
to render the scenes and simulate the agents.

Here's the structure of a Scenic program.

First, we need to decide the town in which this scenario is suitable
for reconstruction. We will always work with the CARLA model as we are
 using the CARLA simulator. Here is a description of all maps on CARLA
:
Town07 - imitates a rural community.
Town06 - contains large, 4-6 lane roads and special junctions like the
 Michigan Left.
Town05 - urban environment containing raised highway and large
multilane roads and junctions.
Town04 - small town with a multi-lane road circumnavigates it in a "
figure of 8".
Town03 - downtown urban area containing a roundabout, underpasses and
overpasses.
Town02 - small town with numerous T-junctions.
Town01 - another small town with numerous T-junctions.

We will then define some constants for the properties of the vehicles,
```

```
 including model and behavioral properties. One can choose to set a
specific value or sample from one of the following distributions:
Range(low, high) - Uniform distribution over the range [low, high]
DiscreteRange(low, high) - Uniform distribution over the discret
integer range [low, high]
Normal(mean, std) - Normal distribution with mean and standard
deviation
TruncatedNormal(mean, stdDev, low, high) - Normal distribution with
mean and standard deviation truncated to the range [low, high]
Uniform(value, ...) - Uniform distribution over the values provided
Discrete([value: weight, ...]) - Discrete distribution over the values
 provided with the given weights

Next, we can define the behavior of each vehicle, which can be a
combination of the default behaviors. The default behaviors are as
follows:
behavior ConstantThrottleBehavior(x : float):
behavior DriveAvoidingCollisions(target_speed : float = 25,
avoidance_threshold : float = 10):
    # Drive at a target speed, avoiding collisions with other vehicles
    # Throttle is off and braking is applied if the distance to the
nearest vehicle is less
    # than the avoidance threshold
behavior AccelerateForwardBehavior(): # Accelerate forward with
throttle set to 0.5
behavior FollowLaneBehavior(target_speed : float = 10, laneToFollow :
Lane = None, is_oppositeTraffic : bool = False):
    # Follow's the lane on which the vehicle is at, unless the
laneToFollow is specified.
    # Once the vehicle reaches an intersection, by default, the
vehicle will take the straight route.
    # If straight route is not available, then any availble turn route
 will be taken, uniformly randomly.
    # If turning at the intersection, the vehicle will slow down to
make the turn, safely.
    # This behavior does not terminate. A recommended use of the
behavior is to accompany it with condition,
    # e.g. do FollowLaneBehavior() until ...
    # :param target_speed: Its unit is in m/s. By default, it is set
to 10 m/s
    # :param laneToFollow: If the lane to follow is different from the
 lane that the vehicle is on, this parameter can be used to specify
that lane. By default, this variable will be set to None, which means
that the vehicle will follow the lane that it is currently on.
behavior FollowTrajectoryBehavior(target_speed : float = 10,
trajectory : List[Lane] = None, turn_speed : float = None):
```

```
    # Follows the given trajectory. The behavior terminates once the
end of the trajectory is reached.
    # :param target_speed: Its unit is in m/s. By default, it is set
to 10 m/s
    # :param trajectory: It is a list of sequential lanes to track,
from the lane that the vehicle is initially on to the lane it should
end up on.
behavior TurnBehavior(trajectory : List[Lane] = None, target_speed :
float = 6):
    # This behavior uses a controller specifically tuned for turning
at an intersection.
    # This behavior is only operational within an intersection, it
will terminate if the vehicle is outside of an intersection.
behavior LaneChangeBehavior(laneSectionToSwitchTo : Lane,
is_oppositeTraffic : bool = False, target_speed : float = 10):
    # is_oppositeTraffic should be specified as True only if the
laneSectionToSwitch to has
    # the opposite traffic direction to the initial lane from which
the vehicle started LaneChangeBehavior
Here is the syntax for defining a new behavior:
behavior <name>(<arguments>):
    <statement>+
If you need to use a try statement, here's the format for that:
try:
    <statement>+
[interrupt when <boolean>:
    <statement>+]*
[except <exception> [as <name>]:
    <statement>+]*

Next, we need to think more specifically about the setting where this
scene could take place based on the description. We also obtain all
the necessary variables for placing the objects in the scene, such as
the starting intersection or lanes, and spawn points. If there is an
intersection, we should generally start there, otherwise, we can
filter for roads. The filter statement should be simple, like all four
-way intersections, all bi-directional roads, etc. We will also
specify constraints at this stage by adding require statements for the
 starting configuration of the scene. These requirements will capture
details in the starting configuration that cannot be specified when
filtering and sampling. The require statements are formatted as:
require <boolean>
where <boolean> is a Python condition that evaluates to a boolean.
Note that some Python operators may not work in this statement. Check
the examples for statements that work.
```

```
Please limit the classes and properties used here to the ones listed
below:
ManeuverType: Enum class with values STRAIGHT, LEFT_TURN, RIGHT_TURN,
U_TURN
Maneuver properties: type (ManeuverType) startLane (Lane) endLane (
Lane) connectingLane (Lane | None) intersection (Intersection | None)
conflictingManeuvers (Tuple[Maneuver]) reverseManeuvers (Tuple[
Maneuver])
NetworkElement properties: name (str) uid (str) id (str | None)
network (Network) vehicleTypes (FrozenSet[VehicleType]) speedLimit (
float | None)
LinearElement properties: centerline (PolylineRegion) leftEdge (
PolylineRegion) rightEdge (PolylineRegion) successor (Union[
NetworkElement, None]) predecessor (Union[NetworkElement, None])
Road properties: lanes (Tuple[Lane]) forwardLanes (LaneGroup | None)
backwardLanes (LaneGroup | None) laneGroups (Tuple[LaneGroup])
sections (Tuple[RoadSection]) crossings (Tuple[PedestrianCrossing])
sidewalks (Tuple[Sidewalk]) sidewalkRegion (PolygonalRegion)
LaneGroup properties: road (Road) lanes (Tuple[Lane]) curb (
PolylineRegion) sidewalk (Sidewalk | None) bikeLane (Lane | None)
shoulder (Shoulder | None) opposite (LaneGroup | None)
Lane properties: group (LaneGroup) road (Road) sections (Tuple[
LaneSection]) adjacentLanes (Tuple[Lane]) maneuvers (Tuple[Maneuver])
RoadSection properties: lanes (Tuple[LaneSection]) forwardLanes (Tuple
[LaneSection]) backwardLanes (Tuple[LaneSection]) lanesByOpenDriveID (
Dict[LaneSection])
LaneSection properties: lane (Lane) group (LaneGroup) road (Road)
isForward (bool) adjacentLanes (Tuple[LaneSection]) laneToLeft (
LaneSection | None) laneToRight (LaneSection | None) fasterLane (
LaneSection | None) slowerLane (LaneSection | None)
Intersection properties: roads (Tuple[Road]) incomingLanes (Tuple[Lane
]) outgoingLanes (Tuple[Lane]) maneuvers (Tuple[Maneuver]) signals (
Tuple[Signal]) crossings (Tuple[PedestrianCrossing]) is3Way (bool)
is4Way (bool) isSignalized (bool)
Signal properties: openDriveID (int) country (str) type (str)
isTrafficLight (bool)
Network properties: elements (Dict[str, NetworkElement]) roads (Tuple[
Road]) connectingRoads (Tuple[Road]) allRoads (Tuple[Road]) laneGroups
 (Tuple[LaneGroup]) lanes (Tuple[Lane]) intersections (Tuple[
Intersection]) crossings (Tuple[PedestrianCrossing]) sidewalks (Tuple[
Sidewalk]) shoulders (Tuple[Shoulder]) roadSections (Tuple[RoadSection
]) laneSections (Tuple[LaneSection]) driveOnLeft (bool) tolerance (
float) drivableRegion (PolygonalRegion) walkableRegion (
PolygonalRegion) roadRegion (PolygonalRegion) laneRegion (
PolygonalRegion) intersectionRegion (PolygonalRegion) crossingRegion (
PolygonalRegion) sidewalkRegion (PolygonalRegion) curbRegion (
```

```
PolylineRegion) shoulderRegion (PolygonalRegion) roadDirection (
VectorField) pickledExt (str)

After that, we can define anything else we require for the vehicle
behaviors that depend on the starting roads/lanes, like trajectories
for turning. Then, we can finally spawn the cars into our scene, and
specify any additional distance constraints between the spawned
vehicles that weren't covered previously. Note that we should always
spawn one of the cars as ego because some of the constraints require
an ego vehicle to be specified. Here, we assign v1 to be the ego
vehicle and the distance requirements that don't specify from are
based on the position of the ego car.

Finally, we want the simulation to go on for at most 10 seconds, so we
 will always specify:
terminate after 10 seconds
Note: you do not need any other termination condition as they are very
 difficult to specify.

Here are some example of a Scenic programs based on crash descriptions
.
<EXAMPLE 1>

<EXAMPLE 2>

<EXAMPLE 3>

-----------------------

Now that you have seen how to write a Scenic program, write one for
the following description.
Description: <DESCRIPTION>
```

Listing A.5: An example of a few-shot CoT example

```
Description: V1 and V2 were both traveling west on a two-way road. V1
attempted to pass V2 on the left. V2, a medium/heavy truck, was making
 a left turn, and the front of V1 impacted the left side of V2.
Plan: V1 will go through FollowLaneBehavior and LaneChangeBehavior to
overtake the slower V2. V2 will need FollowTrajectoryBehavior and
FollowLaneBehavior to perform a left turn and continue along the road.
 The scene needs to take place at a 4-way intersection to guarantee
that there will be a road to turn left to. Both vehicles should start
in the leftmost lane in their direction with V1 slightly behind V2.
Scenic Program:
'''python
## SET MAP AND MODEL
```

```
param map = localPath('../CARLA/CarlaUE4/Content/Carla/Maps/OpenDrive/
Town04.xodr')
param carla_map = 'Town04'
model scenic.simulators.carla.model

## CONSTANTS
V1_MODEL = "vehicle.lincoln.mkz_2017"
V2_MODEL = "vehicle.carlamotors.carlacola"
V1_SPEED = 30   # in m/s
V2_SPEED = 7    # in m/s
OVERTAKE_DIST = 20
INIT_DISTANCE = 20

## DEFINING BEHAVIORS
behavior V1Behavior(speed, leftLaneSec):
    laneChangeCompleted = False
    try:
        do FollowLaneBehavior(speed)
    interrupt when withinDistanceToAnyCars(self, OVERTAKE_DIST) and
not laneChangeCompleted:
        do LaneChangeBehavior(leftLaneSec, is_oppositeTraffic=True,
target_speed=speed)
        laneChangeCompleted = True

behavior V2Behavior(speed, trajectory):
    do FollowTrajectoryBehavior(speed, trajectory)
    do FollowLaneBehavior(speed)

## SELECTING ROAD AND INTERSECTION
fourWayIntersections = filter(lambda i: i.is4Way, network.
intersections)
intersection = Uniform(*fourWayIntersections)
v1_start_lane = Uniform(*intersection.incomingLanes)
require v1_start_lane == v1_start_lane.group.lanes[-1]  # V1 should
start in the leftmost lane in its direction
v2_start_lane = v1_start_lane   # should start in the same lane
v2_spawn_pt = new OrientedPoint on v2_start_lane.centerline     #
first make sure that V2 has not passed the light
v1_spawn_pt = new OrientedPoint following roadDirection from
v2_spawn_pt for -INIT_DISTANCE     # V1 should be behind

## SPAWNING CARS
left_turn_maneuvers = filter(lambda m: m.type == ManeuverType.
LEFT_TURN, v2_start_lane.maneuvers)
v2_maneuver = Uniform(*left_turn_maneuvers)
v2_trajectory = [v2_maneuver.startLane, v2_maneuver.connectingLane,
```

```
v2_maneuver.endLane]
v1_lane_section = network.laneSectionAt(v1_spawn_pt)
leftLaneSec = v1_lane_section.laneToLeft    # overtake lane
ego = new Car at v1_spawn_pt,
    with blueprint V1_MODEL,
    with behavior V1Behavior(V1_SPEED, leftLaneSec)
v2 = new Car at v2_spawn_pt,
    with blueprint V2_MODEL,
    with behavior V2Behavior(V2_SPEED, v2_trajectory)

## TERMINATION CONDITION
require (distance from v2 to intersection) <= 10
terminate after 10 seconds
```

Bibliography

[1] Seif Abukhalaf, Mohammad Hamdaqa, and Foutse Khomh. On codex prompt engineering for ocl generation: an empirical study. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 148–157. IEEE, 2023. 2.2.3

[2] Hesham Alghodhaifi and Sridhar Lakshmanan. Autonomous vehicle evaluation: A comprehensive survey on modeling and simulation approaches. *Ieee Access*, 9:151531–151566, 2021. 1.1

[3] Matthias Althoff and Sebastian Lutz. Automatic generation of safety-critical test scenarios for collision avoidance of road vehicles. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 1326–1333. IEEE, 2018. 2.1.2

[4] Association for Standardization of Automation and Measuring Systems (ASAM). ASAM OpenSCENARIO Standard Version 2.0.0. `https://www.asam.net/standards/detail/openscenario/v200/`, 2023. Accessed: 2025-04-28. 2.1.3

[5] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Prompting is programming: A query language for large language models. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1946–1969, 2023. 4

[6] Qianwen Chao, Huikun Bi, Weizi Li, Tianlu Mao, Zhaoqi Wang, Ming C Lin, and Zhigang Deng. A survey on visual traffic simulation: Models, evaluations, and applications in autonomous driving. In *Computer Graphics Forum*, volume 39, pages 287–308. Wiley Online Library, 2020. 1.1

[7] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023. 2.2.2

[8] Wenhao Ding, Wenshuo Wang, and Ding Zhao. A new multi-vehicle trajectory generator to simulate vehicle-to-vehicle encounters. *arXiv preprint arXiv:1809.05680*, 2018. 2.1.2

[9] Wenhao Ding, Baiming Chen, Minjun Xu, and Ding Zhao. Learning to collide: An adaptive safety-critical scenarios generating method. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2243–2250. IEEE, 2020. 2.1.2

[10] Wenhao Ding, Mengdi Xu, and Ding Zhao. Cmts: A conditional multiple trajectory synthesizer for generating safety-critical driving scenarios. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4314–4321. IEEE, 2020. 2.1.2

[11] Wenhao Ding, Bo Li, Kim Ji Eun, and Ding Zhao. Semantically controllable scene generation with guidance of explicit knowledge. *arXiv preprint arXiv:2106.04066*, 2021. 2.1.2

[12] Wenhao Ding, Haohong Lin, Bo Li, and Ding Zhao. Causalaf: Causal autoregressive flow for goal-directed safety-critical scenes generation. *arXiv preprint arXiv:2110.13939*, 4(4.4), 2021. 2.1.2

[13] Wenhao Ding, Chejian Xu, Mansur Arief, Haohong Lin, Bo Li, and Ding Zhao. A survey on safety-critical driving scenario generation—a methodological perspective. *IEEE Transactions on Intelligent Transportation Systems*, 24(7):6971–6988, 2023. 2.1.2, 2.1.2

[14] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. Carla: An open urban driving simulator. In *Conference on robot learning*, pages 1–16. PMLR, 2017. 1.1, 2.1.3

[15] Avik Dutta, Mukul Singh, Gust Verbruggen, Sumit Gulwani, and Vu Le. Rar: Retrieval-augmented retrieval for code generation in low resource languages. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 21506–21515, 2024. 2.2.3

[16] Karim Elmaaroufi, Devan Shanker, Ana Cismaru, Marcell Vazquez-Chanlatte, Alberto Sangiovanni-Vincentelli, Matei Zaharia, and Sanjit A Seshia. Scenicnl: generating probabilistic scenario programs from natural language. *arXiv preprint arXiv:2405.03709*, 2024. 1.1, 2.1.3, 2.2.4, 3.3, 3.3.1, 4, 4.1, 4.1.1

[17] Jin Fang, Dingfu Zhou, Feilong Yan, Tongtong Zhao, Feihu Zhang, Yu Ma, Liang Wang, and Ruigang Yang. Augmented lidar simulator for autonomous driving. *IEEE Robotics and Automation Letters*, 5(2):1931–1938, 2020. 2.1.2

[18] Daniel J. Fremont. How scenic is compiled. `https://docs.scenic-lang.org/en/latest/internals/compilation_overview.html`, 2025. Accessed: 2025-04-28. 3.5.1

[19] Daniel J Fremont, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L Sangiovanni-Vincentelli, and Sanjit A Seshia. Scenic: a language for scenario specification and scene generation. In *Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation*, pages 63–78, 2019. 1.1, 2.1.3

[20] Daniel J Fremont, Edward Kim, Yash Vardhan Pant, Sanjit A Seshia, Atul Acharya, Xantha Bruso, Paul Wells, Steve Lemke, Qiang Lu, and Shalin Mehta. Formal scenario-based testing of autonomous vehicles: From simulation to the real world. In *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*, pages 1–8. IEEE, 2020. 2.1.2

[21] Alessandro Giagnorio, Alberto Martin-Lopez, and Gabriele Bavota. Enhancing code generation for low-resource languages: No silver bullet. *arXiv preprint arXiv:2501.19085*, 2025. 2.2.3

[22] Cole Gulino, Justin Fu, Wenjie Luo, George Tucker, Eli Bronstein, Yiren Lu, Jean Harb, Xinlei Pan, Yan Wang, Xiangyu Chen, et al. Waymax: An accelerated, data-driven simulator for large-scale autonomous driving research. *Advances in Neural Information Processing Systems*, 36:7730–7742, 2023. 1.1, 2.1.3

[23] Yijie Hou, Chengshun Wang, Junhong Wang, Xiangyang Xue, Xiaolong Luke Zhang, Jun

Zhu, Dongliang Wang, and Siming Chen. Visual evaluation for autonomous driving. *IEEE Transactions on Visualization and Computer Graphics*, 28(1):1030–1039, 2021. 1.1

[24] Sathvik Joel, Jie JW Wu, and Fatemeh H Fard. A survey on llm-based code generation for low-resource and domain-specific programming languages. *arXiv preprint arXiv:2410.03981*, 2024. 2.2.3

[25] Moritz Klischat and Matthias Althoff. Generating critical test scenarios for automated vehicles with evolutionary algorithms. In *2019 IEEE Intelligent Vehicles Symposium (IV)*, pages 2352–2358. IEEE, 2019. 2.1.2

[26] Friedrich Kruber, Jonas Wurst, and Michael Botsch. An unsupervised random forest clustering technique for automatic traffic scenario categorization. In *2018 21st International conference on intelligent transportation systems (ITSC)*, pages 2811–2818. IEEE, 2018. 2.1.2

[27] Friedrich Kruber, Jonas Wurst, Eduardo Sánchez Morales, Samarjit Chakraborty, and Michael Botsch. Unsupervised and supervised learning with the random forest algorithm for traffic scenario clustering and classification. In *2019 IEEE Intelligent Vehicles Symposium (IV)*, pages 2463–2470. IEEE, 2019. 2.1.2

[28] Changwen Li, Joseph Sifakis, Qiang Wang, Rongjie Yan, and Jian Zhang. Simulation-based validation for autonomous driving systems. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 842–853, 2023. 1.1

[29] Quanyi Li, Zhenghao Peng, Lan Feng, Qihang Zhang, Zhenghai Xue, and Bolei Zhou. Metadrive: Composing diverse driving scenarios for generalizable reinforcement learning. *IEEE transactions on pattern analysis and machine intelligence*, 45(3):3461–3475, 2022. 1.1, 2.1.3

[30] Yueyuan Li, Wei Yuan, Songan Zhang, Weihao Yan, Qiyuan Shen, Chunxiang Wang, and Ming Yang. Choose your simulator wisely: A review on open-source simulators for autonomous driving. *IEEE Transactions on Intelligent Vehicles*, 2024. 1.1, 2.1.3

[31] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015. 2.1.2

[32] Sivabalan Manivasagam, Shenlong Wang, Kelvin Wong, Wenyuan Zeng, Mikita Sazanovich, Shuhan Tan, Bin Yang, Wei-Chiu Ma, and Raquel Urtasun. Lidarsim: Realistic lidar simulation by leveraging the real world. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11167–11176, 2020. 2.1.2

[33] Yunwei Mao, You You, Xiaosi Tan, Yongming Huang, Xiaohu You, and Chuan Zhang. Flag: Formula-llm-based auto-generator for baseband hardware. In *2024 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2024. 2.2.3

[34] Till Menzel, Gerrit Bagschik, and Markus Maurer. Scenarios for development, test and validation of automated vehicles. In *2018 IEEE intelligent vehicles symposium (IV)*, pages 1821–1827. IEEE, 2018. 2.1.2

[35] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lilli-

crap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PmLR, 2016. 2.1.2

[36] Ashish Rana and Avleen Malhi. Building safer autonomous agents by leveraging risky driving behavior knowledge. In *2021 International Conference on Communications, Computing, Cybersecurity, and Informatics (CCCI)*, pages 1–6. IEEE, 2021. 2.1.2

[37] Elias Rocklage, Heiko Kraft, Abdullah Karatas, and Jörg Seewig. Automated scenario generation for regression testing of autonomous vehicles. In *2017 ieee 20th international conference on intelligent transportation systems (itsc)*, pages 476–483. IEEE, 2017. 2.1.2

[38] Bo-Kai Ruan, Hao-Tang Tsui, Yung-Hui Li, and Hong-Han Shuai. Traffic scene generation from natural language description for autonomous vehicles with large language model. *arXiv preprint arXiv:2409.09575*, 2024. 1.1, 2.2.4, 3.3

[39] John M Scanlon, Kristofer D Kusano, Tom Daniel, Christopher Alderson, Alexander Ogle, and Trent Victor. Waymo simulated driving behavior in reconstructed fatal crashes within an autonomous vehicle operating domain. *Accident Analysis & Prevention*, 163:106454, 2021. 2.1.1

[40] Tong Duy Son, Ajinkya Bhave, and Herman Van der Auweraer. Simulation-based testing framework for autonomous driving development. In *2019 IEEE International Conference on Mechatronics (ICM)*, volume 1, pages 576–583. IEEE, 2019. 1.1

[41] Marco Stang, Daniel Grimm, Moritz Gaiser, and Eric Sax. Evaluation of deep reinforcement learning algorithms for autonomous driving. In *2020 IEEE intelligent vehicles symposium (IV)*, pages 1576–1582. IEEE, 2020. 1.1

[42] Pei Sun, Henrik Kretzschmar, Xerxes Dotiwalla, Aurelien Chouard, Vijaysai Patnaik, Paul Tsui, James Guo, Yin Zhou, Yuning Chai, Benjamin Caine, et al. Scalability in perception for autonomous driving: Waymo open dataset. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 2446–2454, 2020. 2.1.4

[43] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022. 2.2.1

[44] Tim A Wheeler and Mykel J Kochenderfer. Factor graph scene distributions for automotive safety analysis. In *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*, pages 1035–1040. IEEE, 2016. 2.1.2

[45] Tim A Wheeler, Mykel J Kochenderfer, and Philipp Robbel. Initial scene configurations for highway traffic propagation. In *2015 IEEE 18th International Conference on Intelligent Transportation Systems*, pages 279–284. IEEE, 2015. 2.1.2

[46] Tim Allan Wheeler and Mykel J Kochenderfer. Critical factor graph situation clusters for accelerated automotive safety validation. In *2019 IEEE Intelligent Vehicles Symposium (IV)*, pages 2133–2139. IEEE, 2019. 2.1.2

[47] Blake Wulfe, Sunil Chintakindi, Sou-Cheng T Choi, Rory Hartong-Redden, Anuradha Kodali, and Mykel J Kochenderfer. Real-time prediction of intermediate-horizon automotive

collision risk. *arXiv preprint arXiv:1802.01532*, 2018. 2.1.2

[48] Chejian Xu, Wenhao Ding, Weijie Lyu, Zuxin Liu, Shuai Wang, Yihan He, Hanjiang Hu, Ding Zhao, and Bo Li. Safebench: A benchmarking platform for safety evaluation of autonomous vehicles. *Advances in Neural Information Processing Systems*, 35:25667–25682, 2022. 2.1.3

[49] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822, 2023. 2.2.1

[50] Fan Zhang, Eun Young Noh, Rajesh Subramanian, and Chou-Lin Chen. Crash investigation sampling system: Sample design and weighting. Technical report, 2019. 2.1.4

[51] Jiawei Zhang, Chejian Xu, and Bo Li. Chatscene: Knowledge-enabled safety-critical scenario generation for autonomous vehicles. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 15459–15469, 2024. 1.1, 2.2.4, 3.3