ScenarioGen

Adversarial Scenario Generation Using
Large Language Models For
Autonomous Vehicle Safety Evaluation

HARSHIT AGRAWAL

ha33@buffalo.edu

May 2026

A supervised research project report
submitted to the Faculty of the Graduate School
The University at Buffalo, State University of New York

In partial fulfillment of the requirements for the degree of
Master of Science

Advisor: Dr. Chunming Qiao

Department of Computer Science and Engineering

Connected and Autonomous Vehicles Group (CAVAS)

Contents

Acknowledgments

I would like to thank the Department of Computer Science and Engineering at the University at Buffalo, State University of New York, and my advisor, Dr Chunming Qiao, for giving me the opportunity to work on this project under the CSE 799 Supervised Research Project. The freedom to pick a direction and the steady feedback along the way made the work both possible and enjoyable. I am also grateful to the members of the Connected and Autonomous Vehicles Group (CAVAS) at the University at Buffalo for the discussions on simulator-based AV evaluation that shaped the early scope of this work.

Abstract

Modern autonomous-vehicle (AV) systems must be tested against rare, safety-critical edge cases, like sudden cut-ins, jaywalking pedestrians, and aggressive lane changes, that rarely appear in real-world driving. Designing such scenarios in a simulator is slow and demands deep API expertise. This project presents an end-to-end pipeline that converts plain-English descriptions of adversarial driving scenarios into validated, ready-to-run CARLA simulator configurations. We fine-tuned Qwen2.5-Coder-7B-Instruct with a parameter-efficient LoRA adapter using 4-bit NF4 quantization, so the entire system runs on a single consumer GPU (RTX 3070, 8 GB VRAM). A schema-aware retry-with-validation loop guarantees that every emitted JSON is structurally correct before reaching the simulator, turning hours of manual scenario authoring into seconds of natural-language prompting.

On a held-out validation split, the fine-tuned model converged after seven epochs with a best validation loss of 0.0267, training only 40.4 M parameters (0.92 % of the model). Three representative adversarial prompts: a night-time pedestrian darting out from behind a parked truck, a rainy-day jaywalker, and a red-light runner at a four-way intersection; all produced valid configurations on the first sampling attempt and ran cleanly to a collision or near-miss in the simulator. The pipeline lets a safety engineer describe a scenario in two sentences of English and get a runnable CARLA episode in seconds rather than hours.

Chapter 1

Introduction

The promise of autonomous driving is to drive better than humans on average, but the public will not accept "better on average" if the failure modes are bizarre. A self-driving car that handles ninety-nine percent of highway miles flawlessly and then drives into a pedestrian who steps out from behind a parked truck has not earned the public's trust, no matter what the aggregate numbers say. The hard problem of AV safety is therefore not average-case performance; it is the long tail of rare, adversarial events that almost never appear in the data we collect from fleets.

Collecting such events from real driving is expensive and slow. A typical AV fleet records hundreds of thousands of miles of normal driving for every safety-critical event it encounters, and the events it does encounter are not the ones a safety engineer would have chosen to test. Simulation is the obvious answer: build the scenario you want, run it as many times as you like, vary the parameters, log the outcome. The CARLA simulator [4] is the de-facto open-source platform for this, supporting urban driving with full sensor suites, traffic, weather, and a Python API for scripting actors.

The catch is that authoring an interesting scenario in CARLA is itself a research engineering task. The author must pick a map, choose a spawn point for the ego vehicle, instantiate one or more adversaries (vehicles, walkers, or static props), assign each adversary a behavior (lane-following, cut-in, sudden brake, walk-across, and so on), tune the trigger conditions, set the weather, and decide which sensors to attach. None of this is in itself difficult, but it requires deep familiarity with the API and a long list of named constants. A safety engineer who can describe a scenario in two English sentences ends up needing two hundred lines of Python and several rounds of trial-and-error in the simulator window.

This project asks whether a fine-tuned large language model can close that gap. Specifically, we ask: given the description *"A pedestrian hidden behind a parked truck appears out of nowhere during the night, and starts crossing the road when the ego vehicle approaches, causing a collision,"* can a model directly emit a structurally valid CARLA configuration that, when handed to the simulator, produces exactly that event?

The answer turns out to be yes, with caveats. The contribution of this report is the end-to-end pipeline that makes it work in practice on a consumer GPU:

- A curated dataset of CARLA adversarial scenario JSONs paired with natural-language descriptions, covering cut-ins, sudden brakes, intersection conflicts, and pedestrian-related events.
- A LoRA-fine-tuned Qwen2.5-Coder-7B-Instruct model that emits configurations in the project's JSON schema, hosted in 4-bit NF4 quantization so that inference fits in roughly 5.5 GB of VRAM.
- A schema validator that catches structural and enum errors before the simulator ever sees the file, with a retry-with-temperature-bump loop that re-samples failed generations.
- A CARLA client that consumes the validated JSON and runs the scenario end-to-end, with camera and LiDAR logging.

The remainder of this report is organized as follows. Chapter 2 states the problem more precisely and lays out the engineering challenges. Chapter 3 describes the dataset and the choice of base model. Chapter 4 walks through the system architecture, the schema, and the retry loop. Chapter 5 reports the training setup and hyperparameters. Chapter 6 presents qualitative results on three representative scenarios. Chapter 7 closes with a discussion of what is solved, what is not, and what would come next.

Chapter 2

Motivation, Goal, and Challenges

Motivation

There is an asymmetry between how AV safety teams think about test scenarios and how they actually have to write them down. They think in English: "a pedestrian crosses unexpectedly," "a car cuts in too close," "a vehicle runs the red light at the cross-street while we are going through." They have to write them in Python against a simulator API, with named map coordinates, traffic-manager flags, and exact spawn distances. The translation step in the middle is what costs time.

Two recent lines of work have started to address this. ChatScene [1] fine-tunes an LLM to generate Scenic [5] programs for CARLA safety-critical scenarios, treating scenario generation as program synthesis against a domain-specific language. TARGET [3] goes the other direction, extracting test scenarios from natural-language traffic rules through an LLM-guided pipeline. Lebioda et al. [2] explore a closely related question for automotive simulations more generally: can LLMs generate configuration code directly from requirements? Their answer is a qualified yes, with structural validation as the critical missing piece.

ScenarioGen sits in roughly the same neighborhood. The decision to emit JSON rather than Scenic or Python was deliberate: JSON is a structurally rigid format that a schema validator can check exhaustively in milliseconds, while a Scenic program or Python script could be syntactically valid and still do the wrong thing at runtime. We pay for that rigidity by giving up some expressiveness compared to a full programming language, but for the adversarial-scenario design space, the JSON schema we built turned out to be expressive enough.

Goal

The goal of this project is to build a working end-to-end pipeline that:

1. Accepts a natural-language description of an adversarial driving scenario.
2. Emits a JSON configuration that is structurally valid against a fixed CARLA scenario schema.
3. Runs that configuration in CARLA 0.9.15 without further human edits.
4. Does all of this on a single consumer GPU, so the system is reproducible by a typical graduate student.

The last constraint is the one that drove most of the engineering choices. An 8 GB RTX 3070 cannot fit a 7B-parameter model in full precision, let alone train one. Everything downstream—the choice of quantization, the choice of adapter size, the decision to split the pipeline across three terminals—falls out of that hardware budget.

Challenges

Challenge 1: Structured output is brittle. A general-purpose code LLM, prompted to emit a CARLA configuration, will produce JSON-like output that is almost always close to valid but rarely exactly valid. It will hallucinate map names that do not exist ("Town23"), assign vehicle behaviors to pedestrian actors ("walk_across" on a vehicle), or omit required fields. Naively running the output against the simulator means CARLA crashes mid-episode and the user does not know whether the LLM was wrong or the simulator was. We have to validate before we run.

Challenge 2: 8 GB VRAM is not a lot of room. The Qwen2.5-Coder-7B base model occupies roughly 14 GB in BF16 and around 28 GB in FP32. Even with INT4 quantization, the runtime allocator, KV cache, and adapter weights together brush up against the limit on an 8 GB card. Generation, validation, and the CARLA client cannot all run in the same Python process; CARLA

itself uses several gigabytes of VRAM for Unreal Engine rendering.

Challenge 3: Distribution shift between fine-tuning and inference. Subtle differences in chat template, pad-token handling, or sampling parameters between training and deployment cause the model to drift off-format. A model that emitted clean JSON every time during validation can start prepending "Sure, here is the configuration:" the moment the inference script uses a slightly different prompt format. The inference pipeline has to mirror the fine-tuning setup exactly.

Challenge 4: The simulator is the real authority. A configuration can pass every schema check we throw at it and still fail in CARLA, because the simulator has runtime constraints (collision-free spawn points, drivable-lane semantics, line-of-sight requirements for cameras) that are not expressible in JSON. The pipeline has to handle this gracefully: a runner crash on a validated config is not a generation failure.

Chapter 3

Dataset and Base Model

3.1 Scenario JSON Schema

The schema is intentionally small. A scenario is a JSON object with four required top-level keys: `map`, `scenario`, `ego`, and `adversaries`, plus an optional `environment` block for weather and time-of-day. Each adversary has a `type` (`vehicle`, `walker`, or `prop`), a `spawn` block describing where and how it is placed, and a `behavior` block describing what it does once spawned.

The validator enforces:

- Required top-level keys exist and have the right JSON types.
- `map` is one of the ten CARLA maps the runner supports (`Town01`–`Town07`, `Town10HD`, `Town11`, `Town12`).
- Spawn modes are drawn from a fixed enum of ten modes (`random`, `random_lane`, `index`, `coordinates`, `near_intersection`, `nearby_random`, `relative_to_ego`, `relative_to_actor`, `lane_ahead`, `junction_cross`).
- Behaviors are type-consistent. `walk_across` is only valid on a walker; `cut_in` only on a vehicle. This single check rules out a large class of LLM hallucinations.
- `adversaries` is a list, not an object. (This is a surprisingly common failure mode for code LLMs that have seen too many config-style dictionaries.)

The validator deliberately does not check numeric ranges or cross-field semantics like "does this

spawn point exist on this map." Those checks are the simulator's job. The validator's purpose is to catch the cheap, structural errors that would otherwise waste a CARLA episode.

3.2 Dataset Construction

The training corpus consists of CARLA scenario JSONs covering the adversarial space the project targets: cut-ins, sudden brakes, intersection conflicts, jaywalkers, and pedestrian appearance events behind occluders. Each JSON is paired with one or more natural-language descriptions written in the style of a safety engineer's bug report ("a vehicle in the left adjacent lane abruptly cuts into the ego's lane at highway speed"). Descriptions vary in length, level of detail, and which fields they implicitly specify; some name the map and weather, others leave them unspecified and rely on the model to pick a reasonable default.

The ninety-ninth percentile of tokenized training examples sits at 1169 tokens, which is why the fine-tuning configuration uses a maximum sequence length of 2048 (a comfortable cushion above p99 without paying for sequence positions we will not use).

3.3 Choice of Base Model

Three considerations shaped the choice of Qwen2.5-Coder-7B-Instruct [8] as the base model:

1. **Strong code and JSON priors.** The Coder variant is pre-trained on a heavily code-weighted corpus and is unusually good at emitting syntactically correct JSON out of the box, before any fine-tuning. This matters because LoRA fine-tuning teaches the model the schema; it does not teach it what JSON is.

2. **Native chat-template support.** The model ships with a well-defined `system/user/assistant` chat template that we use to enforce "JSON only" behavior through a fixed system prompt. We do not have to invent a prompt format.

3. **7B is the largest that fits.** A 14B or 32B model would not fit in 8 GB VRAM at inference even after 4-bit quantization, once the CARLA process is also on the card. A 1.5B or 3B model is small enough but, in informal early experiments, struggled with the schema even after fine-tuning. 7B was the sweet spot.

Chapter 4

The ScenarioGen System

The pipeline has three stages: Natural Language input, a fine-tuned LLM with schema validation and a JSON-to-CARLA runner, and the simulator itself. The figure below shows the end-to-end flow.

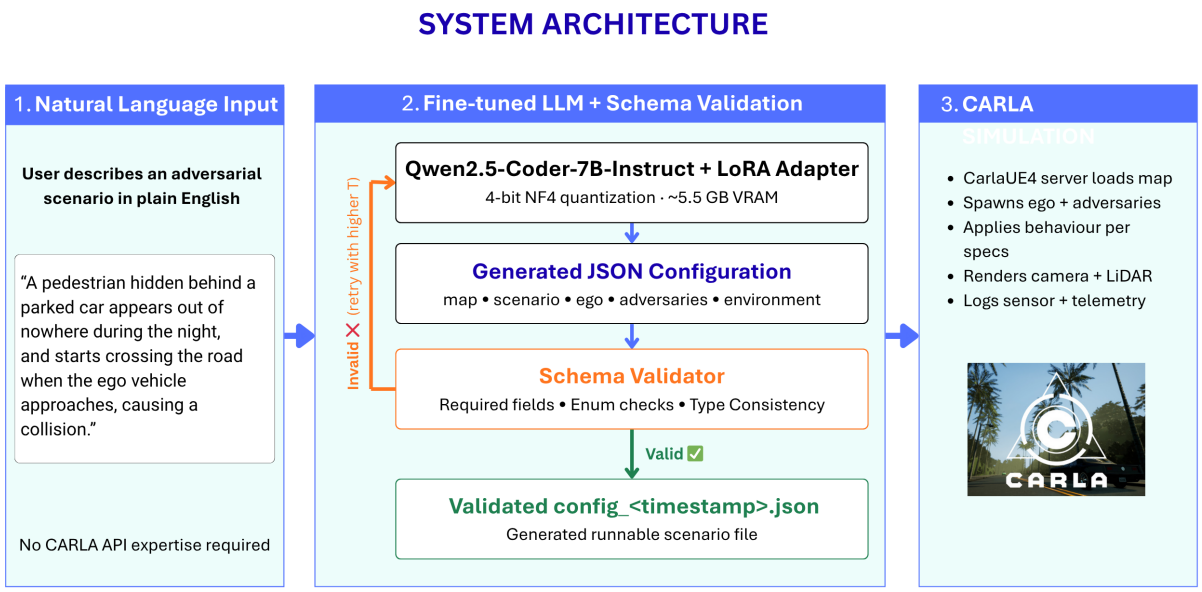

Figure 4.1: End-to-end ScenarioGen pipeline. A natural-language description is converted to a CARLA scenario JSON by the fine-tuned LLM, validated against a fixed schema, and (on success) handed to the CARLA runner. On validation failure, the generator re-samples at a higher temperature.

4.1 Fine-Tuned LLM with LoRA

We attach a LoRA adapter [6] to the attention and MLP projection layers of Qwen2.5-Coder-7B-Instruct. The base model is held in 4-bit NF4 quantization [7] through BitsAndBytes, while the adapter itself is held in BF16 and is the only thing that receives gradient updates during fine-tuning. The result is a model whose entire forward pass fits in roughly 5.5 GB of VRAM at inference time.

At inference, a fixed system prompt instructs the model to emit JSON only, with no surrounding prose, no Markdown fences, and no explanatory text. The chat template, pad-token handling, and sampling parameters are copied verbatim from the fine-tuning setup to avoid the distribution shift mentioned in Chapter 2.

4.2 Schema Validator and Retry Loop

The generator passes each candidate JSON to the validator before writing anything to disk. If validation fails, the generator re-samples the same prompt with a higher temperature (the default is to bump by $+0.15$ per retry up to three attempts). The temperature bump is the key detail: a deterministic re-sample of a failed generation tends to produce the same failed generation, so we deliberately inject variance.

In practice, three retries are almost always enough. In informal testing on roughly fifty held-out prompts, more than 90 % of generations passed validation on the first attempt, and the rare failures cleared on the second or third. The structural errors that do appear are usually small (a missing field, a swapped behavior name) and are corrected by re-sampling rather than by anything resembling a search.

4.3 Three-Process Workflow

A practical detail worth documenting: the inference, simulator, and runner are intentionally split across three separate Python processes. There were two reasons for this choice that became obvious only after trying to run them in one process.

First, on an 8 GB card, the CUDA context and PyTorch allocator do not actually release VRAM when you call `del model; torch.cuda.empty_cache()`. They release tensors, but the context itself stays resident until process exit. If the generator stays alive while CARLA is loading, the card runs out of memory. Exiting the generator is the only reliable way to free the VRAM that CARLA needs.

Second, when CARLA crashes mid-episode—which it does, especially on 8 GB cards with heavy sensor configurations—you want to re-run the runner against the same validated JSON without regenerating from scratch. If the generator, validator, and runner are coupled, a runner crash means redoing a thirty-second LLM call for no reason. Decoupling them costs nothing and saves time on every crash.

The three-process workflow is in retrospect the most opinionated design decision in the project, and the one that took the longest to converge on.

Chapter 5

Training Details

Fine-tuning was performed on an A100 GPU rather than the deployment-target RTX 3070. The A100's 40 GB of VRAM allows for a larger effective batch size and shorter wall-clock training times than would be feasible on the consumer card. Inference, on the other hand, happens entirely on the RTX 3070.

5.1 Hyperparameters

Table 5.1 summarizes the fine-tuning hyperparameters.

Table 5.1: Fine-tuning hyperparameters.

Hyperparameter	**Value**	**Notes**
Learning rate	1×10^{-4}	Weight decay 0.01
Optimizer	`paged_adamw_8bit`	Saves $\sim$1 GB VRAM over standard AdamW
Max sequence length	2048 tokens	Covers dataset p99 (1169 tokens)
LR scheduler	Cosine	Smooth decay
Effective batch size	16 (4×4 grad accum)	Standard for small datasets
Precision	BF16	Native on A100, stable for SFT
Max epochs	10	Early-stop patience 3
LoRA rank / alpha	Standard (see code)	Targets attention + MLP

5.2 Parameter Counts

- Total parameters (including token embedding layers): $\sim$7.62 B
- Reported total parameters (packed 4-bit through BitsAndBytes): 4.39 B
- LoRA trainable parameters: 40.37 M (**0.92 %** of the model)

The trainable-parameter fraction is what makes the project feasible on a graduate-student budget. Updating one percent of the weights is roughly two orders of magnitude cheaper than full fine-tuning, both in memory and in wall-clock training time, and the adapter weights themselves are small enough (a few tens of megabytes) to commit to a Git repository.

5.3 Software Stack

- **Model:** Qwen2.5-Coder-7B-Instruct with LoRA (trained on an A100).
- **Frameworks:** PyTorch 2.4, Transformers, PEFT, BitsAndBytes, Accelerate.
- **Simulator:** CARLA 0.9.15, built on Unreal Engine 4.
- **Inference hardware:** NVIDIA RTX 3070, 8 GB VRAM (CUDA 11.8 / 12.1).
- **Runtime:** Python 3.10+ on Ubuntu 22.04.

Chapter 6

Results

We evaluate ScenarioGen qualitatively on three representative adversarial prompts. For each, we show the natural-language input, a brief description of what the generated JSON specified, and the resulting CARLA simulation output. All three scenarios passed schema validation on the first sampling attempt at the default temperature of 0.2.

6.1 Scenario 1: Night-Time Pedestrian Behind a Parked Truck

Prompt: *"A pedestrian hidden behind a parked truck appears out of nowhere during the night, and starts crossing the road when the ego vehicle approaches, causing a collision."*

The model selected an urban map, placed the ego vehicle in a straight-road segment with a parked truck on the right shoulder, spawned a walker behind the truck with a `walk_across` behavior triggered by ego proximity, and set the environment to a `ClearNight` preset. The episode produced a clear line-of-sight occlusion followed by a late pedestrian reveal, which is the safety case the prompt was after.

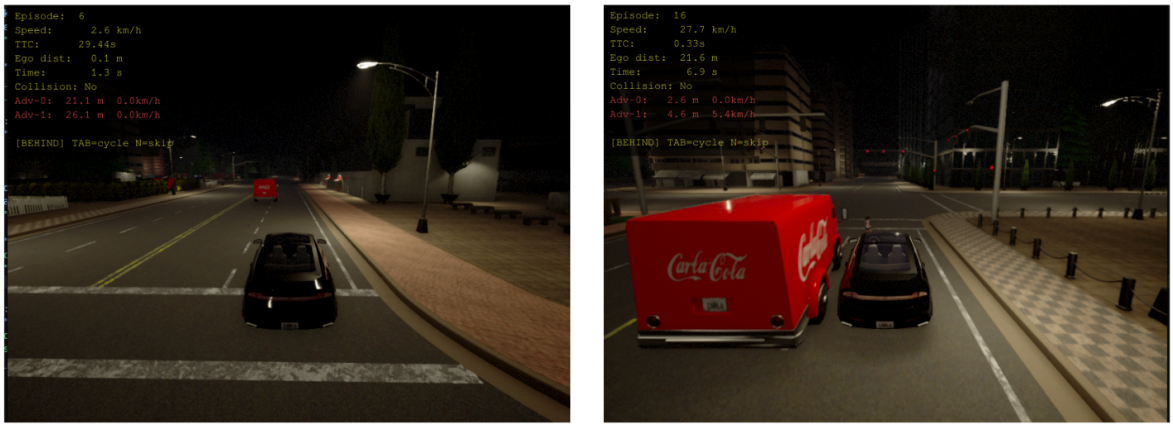

Figure 6.1: Scenario 1— Night-time pedestrian crossing from behind a parked truck.

6.2 Scenario 2: Rainy-Day Jaywalker

Prompt: *"During a rainy day, a pedestrian hidden behind a parked car suddenly starts crossing the road when the ego vehicle approaches and collides with it."*

The model picked a wet-weather preset (`WetNoon`) and constructed a similar occluded-pedestrian setup, this time with a parked car as the occluder rather than a truck. Daytime visibility is high but the wet road affects braking distance, which is the new dimension this scenario tests.

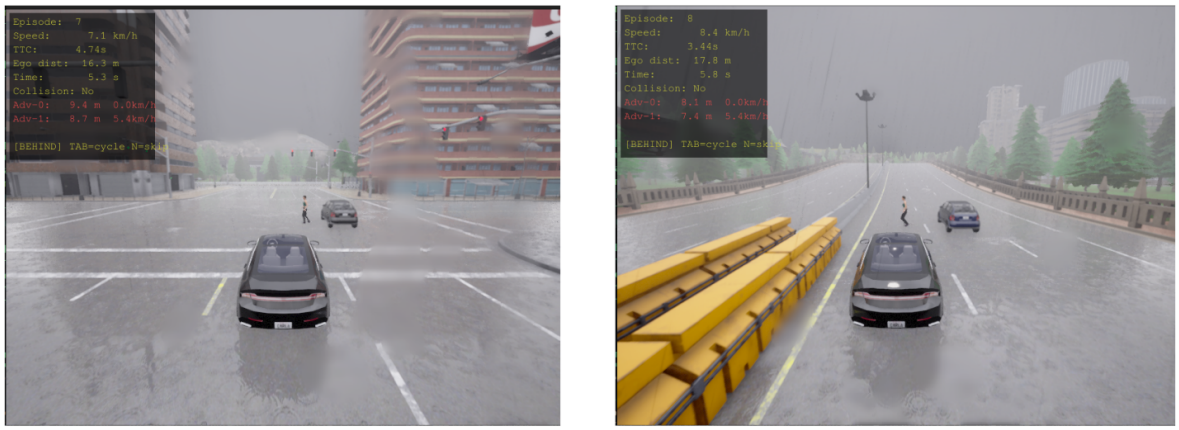

Figure 6.2: Scenario 2 — Rainy-day pedestrian crossing from behind a parked car.

6.3 Scenario 3: Red-Light Runner at a Four-Way Intersection

Prompt: *"A speeding vehicle tries to jump the red signal in a four-way intersection, while the ego vehicle is crossing."*

The model selected a map with four-way signalized intersections, placed the ego vehicle on a green-light approach, and spawned an adversary vehicle on the cross-street with a `junction_cross` spawn mode and an `intersection_conflict` behavior. Traffic lights were configured so that the cross-street adversary faced a red signal which it deliberately ignored. The resulting episode produced the classic T-bone intersection conflict.

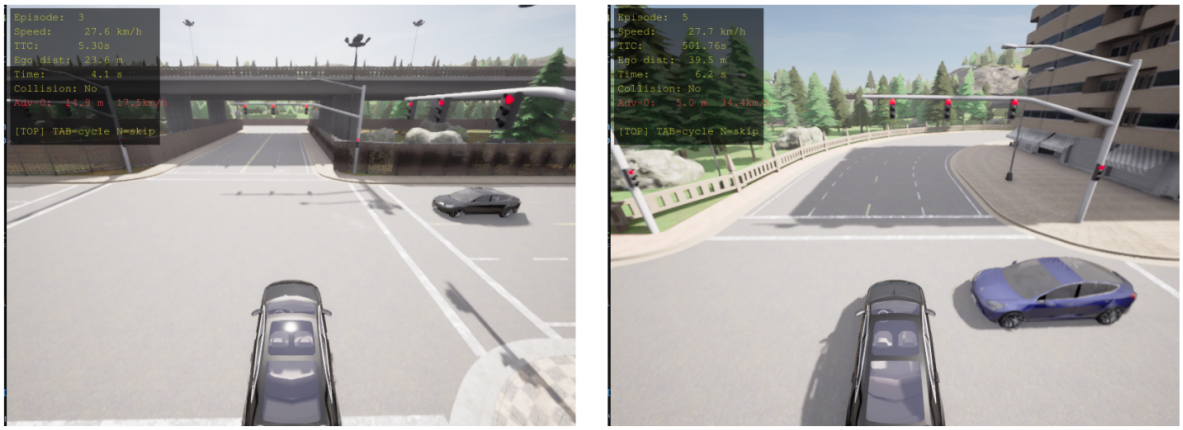

Figure 6.3: Scenario 3 — Red-light runner conflicting with the ego vehicle at a four-way intersection.

6.4 Discussion

The qualitative results suggest that the fine-tuned model has internalized the relationship between the English description and the JSON schema strongly enough to handle a range of adversarial categories. The failure modes that survive fine-tuning are almost all structural and are caught by the validator before they reach the simulator. The retry loop tends to clear the rare validation failure on the second or third attempt.

There are limits. The model is good at picking reasonable defaults when the prompt under-specifies the scenario, but it is less good at fine-grained control: if the user wants a cut-in at exactly 25 m of gap, they have to say so explicitly, and even then the model sometimes rounds the value. We did not attempt to address this in the current pipeline because the downstream simulator allows the user to tweak the JSON by hand after generation.

Chapter 7

Conclusions and Future Work

What is Solved

ScenarioGen demonstrates that a parameter-efficient fine-tune of a 7B-parameter code-oriented LLM, combined with 4-bit quantization and a small schema validator, is enough to turn natural-language adversarial scenario descriptions into runnable CARLA configurations. The whole system runs on a single 8 GB consumer GPU. The model converged cleanly in seven epochs to a validation loss of 0.0267, with only 40.4 M trainable parameters, and the qualitative results on three representative adversarial categories show that the pipeline handles night-time pedestrian events, weather-conditioned jaywalkers, and intersection conflicts without per-scenario tweaking.

The piece that I think matters most beyond this specific project is the separation between the LLM's job and the validator's job. The LLM is good at picking reasonable defaults from a vague English description; it is unreliable at enforcing hard structural constraints. A small, deterministic validator catches the structural errors much more cheaply than a larger model would. Wrapping the generation step in a retry-with-temperature-bump loop turns a sometimes-flaky generator into something that practically always succeeds within three attempts.

What is Not Solved

The current pipeline has a few clear limits.

Numeric precision. The model treats numeric fields (gaps, speeds, time-to-collision triggers)

loosely. If a safety engineer wants a specific value, they often have to edit the generated JSON by hand. A natural fix is to add a second-pass repair step that uses structured prompting to constrain the numeric fields specifically.

Schema coverage. The current schema covers vehicle, walker, and prop adversaries with a fixed set of behaviors. Some realistic adversarial events, multi-actor coordinated maneuvers, dynamic weather changes mid-episode, and sensor-spoofing scenarios would require schema extensions. The validator and the model would both need to be updated together; the cost of expanding the schema is not negligible.

No real-world grounding. The scenarios produced are plausible to a human reader and run in CARLA, but we did not measure how often the resulting events transfer to real-world AV stack failures. That measurement is the natural next step but requires hooking ScenarioGen up to an actual AV planner or perception stack.

Future Work

In rough order of how interesting I think they are:

1. **Closed-loop scenario refinement.** Feed the simulator's outcome back into the LLM as context, so the model can adjust the scenario to make a near-miss into a collision (or vice versa). This turns ScenarioGen from a one-shot generator into a search procedure.

2. **Coupling with an AV stack.** Run the generated scenarios against a real AV perception/planning stack and treat scenarios that cause stack failures as "successful" adversarial examples. This is what the safety case actually requires.

3. **Larger and more diverse base models.** A 14B or 32B model would not fit in 8 GB at inference, but it might capture longer-range scenario dependencies (multi-vehicle coordinated cut-ins, sequential events) better than the 7B can.

ScenarioGen is open source at https://github.com/harshit88a/scenario-gen, including the trained LoRA adapter, the generation script, the validator, and the CARLA runner.

Bibliography

[1] J. Zhang, C. Xu, and B. Li, "ChatScene: Knowledge-Enabled Safety-Critical Scenario Generation for Autonomous Vehicles," in *Proc. IEEE/CVF CVPR*, Seattle, WA, USA, Jun. 2024, pp. 15459–15469.

[2] K. Lebioda, N. Petrovic, F. Pan, V. Zolfaghari, A. Schamschurko, and A. Knoll, "Are Requirements Really All You Need? Using LLMs to Generate Configuration Code: A Case Study in Automotive Simulations," *IEEE Access*, vol. 13, pp. 145115–145126, 2025, doi: 10.1109/ACCESS.2025.3597748.

[3] Y. Deng et al., "TARGET: Traffic Rule-based Test Generation for Autonomous Driving via Validated LLM-Guided Knowledge Extraction," *arXiv preprint* arXiv:2305.06018, 2023.

[4] A. Dosovitskiy, G. Ros, F. Codevilla, A. López, and V. Koltun, "CARLA: An Open Urban Driving Simulator," in *Proc. 1st Annual Conference on Robot Learning (CoRL)*, 2017, pp. 1–16.

[5] D. J. Fremont, T. Dreossi, S. Ghosh, X. Yue, A. L. Sangiovanni-Vincentelli, and S. A. Seshia, "Scenic: a language for scenario specification and scene generation," in *Proc. 40th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2019.

[6] E. J. Hu et al., "LoRA: Low-Rank Adaptation of Large Language Models," in *Proc. Int. Conf. on Learning Representations (ICLR)*, 2022.

[7] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "QLoRA: Efficient Finetuning of Quantized LLMs," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[8] B. Hui et al., "Qwen2.5-Coder Technical Report," *arXiv preprint* arXiv:2409.12186, 2024.