UNIVERSITÀ DEGLI STUDI DI GENOVA

POLYTECHNIC SCHOOL

DITEN

Department of Naval, Electrical, Electronic and Telecommunications Engineering

MASTER'S DEGREE

IN ENGINEERING TECHNOLOGY FOR STRATEGY (AND SECURITY)

ScenWeave: Retrieval-Augmented Generation of Executable CARLA Scenarios

Supervisors:

Prof. Francesco Bellotti

Prof. Riccardo Berta

Co-supervisor:

Dott. Alessandro Pighetti

Candidate:

Giorgia Baiardo

July 2026

ScenWeave: Generazione Retrieval-Augmented di Scenari CARLA Eseguibili

Sommario

L'addestramento e la validazione di modelli per il riconoscimento di manovre nella guida autonoma richiedono grandi volumi di dati etichettati per eventi statisticamente rari nella guida naturalistica, frenate brusche, tagli di corsia, cambi di corsia improvvisi. Questa tesi presenta ScenWeave, una pipeline end-to-end che trasforma automaticamente dati di guida reali in scenari OpenSCENARIO 1.0 validati ed eseguibili per il simulatore CARLA, eliminando lo sforzo di ingegnerizzazione manuale tradizionalmente necessario per tradurre registrazioni di guida grezze in scenari pronti per la simulazione.

ScenWeave riceve in ingresso registrazioni di guida reali dal progetto di guida naturalistica MOOVE, file JSON per scenari di intersezione urbana e file binari NumPy per classi di manovre autostradali, e produce file .xosc direttamente eseguibili attraverso una pipeline a sei stadi: estrazione dei parametri, selezione del punto di spawn, corrispondenza corpus con retrieval-augmented generation, generazione di codice tramite LLM locale, esecuzione e validazione XSD automatizzata con auto-correzione deterministica. Il meccanismo di Retrieval-Augmented Generation (RAG) utilizza la corrispondenza deterministica per parole chiave su un corpus specializzato di 16 script Python (R@1 = 1,0), e un modello linguistico di grandi dimensioni installato localmente (Qwen2.5-Coder-7b, servito tramite Ollama) adatta lo script di riferimento recuperato ai parametri estratti attraverso un prompt strutturato in sei sezioni e un ciclo di auto-correzione a tre tentativi.

Un benchmark sistematico condotto su cinque LLM orientati al codice su 11 classi di manovre autostradali mostra che il modello di produzione selezionato raggiunge il 100% di success rate e il 100% di first-attempt success rate, con zero allucinazioni o errori di sintassi. Questi risultati, corroborati da un benchmark trasversale dedicato ai livelli di specificità del dominio [5], dimostrano che un'architettura RAG progettata con cura può ridurre sostanzialmente il divario tra LLM locali e cloud su un task di generazione di codice ultra-specifico del dominio, abilitando una generazione di scenari completamente offline e rispettosa della privacy senza perdita misurabile di correttezza.

La tesi documenta inoltre un database di spawn point per CARLA Town04, generato tramite uno script dedicato e successivamente verificato manualmente, con funzioni di scoring context-aware per scenari sia urbani che autostradali, e una pipeline di canonicalizzazione che converte i dati grezzi sia urbani (JSON) che autostradali (NPY) in schemi JSON unificati progettati per facilitare il recupero dei parametri tra i domini. Le limitazioni e una roadmap strutturata per estensioni future verso un sistema di data augmentation a ciclo chiuso per il riconoscimento di manovre sono discusse esplicitamente.

Parole chiave: Retrieval-Augmented Generation, Large Language Models, Generazione di Scenari, CARLA, OpenSCENARIO, Guida Autonoma, Dati di Guida Naturale.

ScenWeave: Retrieval-Augmented Generation of Executable CARLA Scenarios

Abstract

Training and validating manoeuvre recognition models for autonomous driving requires large volumes of labelled data for events that are statistically rare in naturalistic driving, sudden braking, cut-ins, abrupt lane changes. This thesis presents ScenWeave, an end-to-end pipeline that automatically transforms real-world driving data into validated, executable OpenSCENARIO 1.0 scenarios for the CARLA simulator, removing the manual engineering effort traditionally required to translate raw driving recordings into simulation-ready scenarios.

ScenWeave takes as input real driving records from the MOOVE naturalistic driving project, JSON files for urban intersection scenarios and NumPy binary files for highway manoeuvre classes and produces directly executable .xosc files through a six-stage pipeline: parameter extraction, spawn point selection, retrieval-augmented corpus matching, local LLM-based code generation, execution, and automated XSD validation with deterministic auto-fix. The retrieval-augmented generation (RAG) mechanism uses deterministic keyword matching against a domain-specific corpus of 16 Python scripts (R@1 = 1.0), and a locally deployed large language model (Qwen2.5-Coder-7b, served via Ollama) adapts the retrieved reference script to the extracted parameters through a structured six-section prompt and a three-attempt self-healing retry loop.

A systematic benchmark conducted on five code LLMs across 11 highway manoeuvre classes shows that the selected production model achieves 100% success rate and 100% first-attempt success rate, with zero hallucinations or syntax errors. These results, corroborated by a dedicated cross-domain benchmark on software specificity levels [5], demonstrate that a carefully designed RAG architecture can substantially close the gap between local and cloud-hosted LLMs on an ultra-domain-specific code generation task, enabling fully offline, privacy-preserving scenario generation with no measurable loss of correctness.

The thesis further documents a spawn point database for CARLA Town04, generated through a dedicated script and subsequently manually verified, with context-aware scoring functions for both urban and highway scenarios, and a canonicalization pipeline that converts both urban (JSON) and highway (NPY) raw data into unified JSON schemas designed to facilitate parameter retrieval across domains. Limitations and a structured roadmap for future extensions towards a closed-loop data augmentation system for manoeuvre recognition are discussed explicitly.

Keywords: Retrieval-Augmented Generation, Large Language Models, Scenario Generation, CARLA, OpenSCENARIO, Autonomous Driving, Naturalistic Driving Data.

Acknowledgements

Desidero ringraziare il Prof. Francesco Bellotti e il Prof. Riccardo Berta per avermi guidata durante tutto il percorso di tesi e anche nella magistrale, per la disponibilità costante, per le conoscenze che mi avete trasmesso e che mi hanno appassionato e per i consigli che hanno reso questo lavoro più solido e consapevole.

Un ringraziamento speciale va ad Alessandro Pighetti, che mi ha affiancata quotidianamente durante il tirocinio presso l'ELIOS Lab: la sua pazienza e la sua disponibilità a confrontarsi su ogni aspetto del progetto sono stati fondamentali per portare il mio lavoro al punto in cui è oggi. Volevo ringraziare anche Luca Lazzaroni per il supporto che mi ha fornito, Matteo Fresta per l'aiuto che mi ha offerto e per l'improvvisata torta di compleanno, e il mio "maestro" Pietro Firpo. Ringrazio l'ELIOS Lab nel suo insieme per aver creato un ambiente di lavoro stimolante, dove il tirocinio è stata un'esperienza di ricerca vera, che ha dato origine a questa tesi.

Voglio ringraziare anche i miei compagni di avventura di questa magistrale, Luca Benvenuto, Francesco Romaggi e Simone Incerti: abbiamo condiviso molto e senza di voi sarebbe stato tutto molto più difficile, quindi grazie.

Ringrazio la mia famiglia. Mia madre, che mi ha supportata durante gli anni dell'università e mi è sempre stata accanto: ancora oggi mi accompagna e mi incoraggia a intraprendere strade nuove che, senza il suo supporto, probabilmente non avrei mai considerato.
E mio fratello: ora è il tuo momento di iniziare questo percorso. Si rivelerà estremamente difficile ma anche estremamente gratificante, gli amici che farai in questi anni e le conoscenze che acquisirai ti formeranno e ti faranno crescere. Come sempre, se hai bisogno, per te ci sono.

Grazie a Filippo, per essere la spalla su cui piangere ma anche quella persona con cui posso condividere il bello e il brutto.

Un pensiero va anche ai miei amici. Eleonora, Elisa e Sammy: siete le mie più care amiche e vi ringrazio per non lasciarmi mai da sola, so che siete sempre lì per me e questo vale moltissimo. Angly, Sandro ed Ema: le conversazioni, i momenti di sconforto condivisi e le piccole celebrazioni lungo la strada hanno reso tutto più leggero.

Sono felice di essere riuscita a raggiungere un traguardo che fino a qualche anno fa ritenevo impossibile, o comunque molto lontano. Ripensando alla fatica fatta per arrivare fino a qui, mi rendo conto che è stato anche un percorso bellissimo. Sono felice delle scelte che ho fatto, perché mi hanno portata a essere la persona che sono oggi. Ho ancora tanto da imparare, tante strade da percorrere, tante difficoltà da affrontare. Perciò, come direbbe un personaggio molto conosciuto, e so che in molti capiranno la citazione, anche se un po' banale e scontata, “last but not least, I want to thank me”.

Table of Contents

1 Introduction

Autonomous and driver-assistance systems are only as reliable as the data used to train and validate them. Yet the events that matter most for safety, a vehicle braking hard because the car ahead stops suddenly, another car cutting into the lane without warning, an abrupt lane change at high speed are, by their very nature, rare. A fleet of vehicles can drive millions of kilometres and still record only a handful of examples of each of these manoeuvres. Without enough real examples, it is difficult to train a system to recognise these situations reliably, or to test whether it responds safely when they occur. Simulation offers a way around this scarcity: once a critical event has been captured once, it can in principle be reproduced as many times as needed, with controlled variations, inside a driving simulator. The obstacle has never been the simulator itself, but the translation step, turning a short segment of raw sensor data (speeds, positions, timings) into an executable, standards-compliant simulation file has traditionally required an engineer to hand-write the scenario logic, one manoeuvre type at a time. This manual effort does not scale to the volume of data needed to meaningfully close the gap for rare events. This thesis presents ScenWeave, a system that automates this translation end to end. Given a real driving recording, ScenWeave uses a locally run artificial intelligence model, guided by real, verified examples rather than left to invent a solution from scratch, to write the code that reproduces the manoeuvre inside the CARLA driving simulator. The result is a validated, ready-to-use scenario file, produced automatically and without any of the original driving data ever leaving the laboratory. Rather than a general-purpose scenario generator, ScenWeave is designed as a scenario compiler: a system that reliably translates a real, measured driving event into an executable simulation, at a scale and speed manual engineering cannot match. Tested at production scale, ScenWeave converts real driving recordings into working simulation scenarios with a 100% success rate, producing hundreds of scenarios per hour, entirely offline.

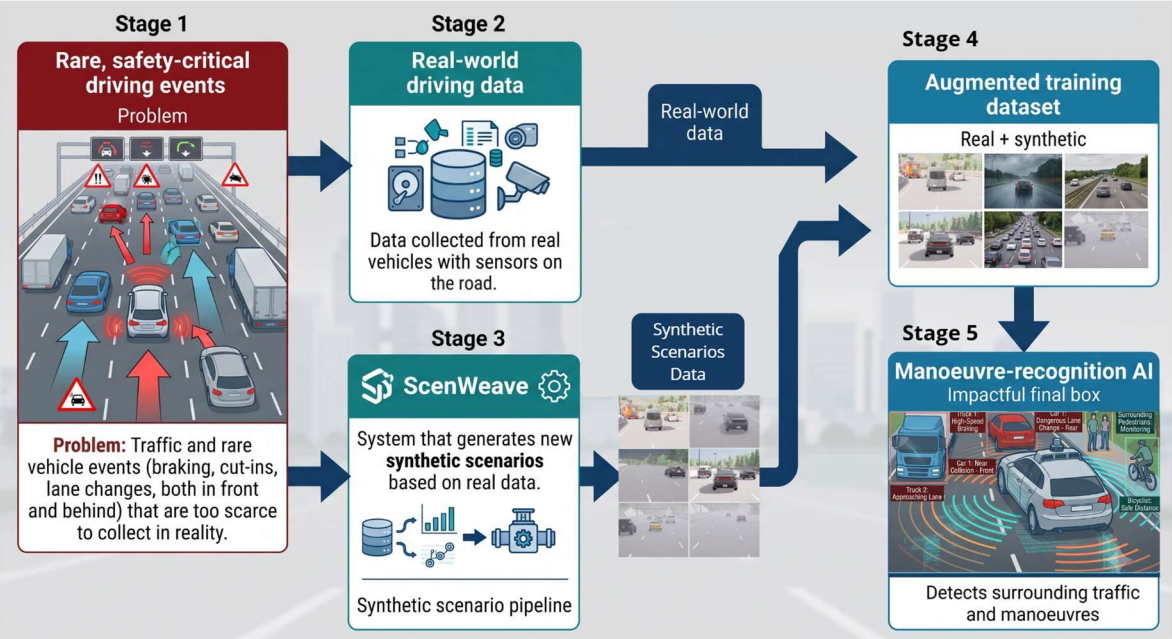

Figure 1.1 - From rare real-world events to an augmented dataset for manoeuvre-recognition AI

The result is a practical, privacy-preserving way to augment the real-world data available for training and validating driving-related AI systems, supplementing (rather than replacing) real recordings, particularly for manoeuvres involving traffic both ahead of and behind the vehicle.

1.1 Research Questions

Realising this vision requires more than a working prototype: it requires establishing that automated, AI-driven scenario generation is technically sound, that it can be delivered with tools that respect data privacy and cost constraints, and that it can grow to cover new driving situations without being rebuilt from scratch each time. These three requirements translate directly into the research questions that structure this thesis:

- RQ1: Is it possible to generate valid and semantically correct OpenSCENARIO (XML) files using a RAG and LLM system?
- RQ2: Among the LLMs that can be run locally via Ollama, which one offers the best trade-off between reliability and efficiency for this task?

- RQ3: Is the system generalisable to new manoeuvre classes or new corpus entries without requiring structural redesign of the pipeline?

RQ1 is addressed through the production-scale validation run, which evaluates the end-to-end pipeline across hundreds of real scenario instances spanning both highway and urban manoeuvre classes, measuring both structural validity and semantic correctness where applicable. RQ2 is addressed through the comparative benchmark of five candidate LLMs across multiple domain specificity levels, evaluating success rate, first-attempt success rate, and generation latency. RQ3 is addressed through a concrete extension episode encountered during development, in which the urban corpus scripts were corrected to align with the highway construction pattern without modification to the retrieval, generation, or validation logic.

1.2 Original Contributions

This thesis makes three original contributions to the problem of automated scenario generation from real-world driving data. Each contribution addresses a distinct component of the end-to-end pipeline. Here are the original contributions:

1 - End-to-end pipeline itself is a complete system that takes real driving data, in either JSON or NPY format, and produces validated, executable OpenSCENARIO scenarios in CARLA, with no manual parameter specification required at any stage.

2 - A domain-specific RAG corpus of 16 Python scripts implementing canonical scenario generation functions for 11 highway and 5 urban manoeuvre classes, built on the scenariogeneration API. Each script acts as a structural reference that the LLM adapts to new parameters, rather than generating code from scratch.

3 - Automated XSD validator, a domain-aware component that checks every generated .xosc file against the OpenSCENARIO 1.0 schema and applies five deterministic auto-fix rules to correct known LLM error patterns before execution.

Together, these contributions constitute the first end-to-end pipeline that takes real-world naturalistic driving data in raw domain-specific formats and produces validated, executable OpenSCENARIO scenarios using a local LLM in a RAG architecture, with no cloud API dependency and no manual parameter specification.

1.3 Design Alternatives Considered

The design choices made in ScenWeave reflect a specific set of constraints and priorities. Understanding the alternatives that were considered and rejected is important for situating the contribution correctly. Here are the alternative approaches considered.

- An early prototype of the pipeline asked the LLM to *generate OpenSCENARIO XML directly*, without using the scenariogeneration library as an intermediary. This approach was abandoned after observing systematic XML structure errors: the LLM would generate syntactically valid XML that nonetheless failed XSD validation due to incorrect element ordering, missing required attributes, or namespace errors. The scenariogeneration library encodes the OpenSCENARIO schema constraints in its Python API, making it significantly easier for the LLM to generate correct output by following the library's method signatures rather than constructing raw XML.

- Using a *cloud-hosted model* would almost certainly improve generation quality, as GPT-4 has demonstrated strong performance on code generation tasks across all specificity levels. However, the MOOVE dataset is confidential, and sending driving data to an external API would raise data privacy concerns. The choice of a local model via Ollama ensures that no scenario parameters leave the laboratory network. An additional practical consideration is cost: generating thousands of scenarios for a training dataset would incur significant API costs with a cloud model, while a local model runs at zero marginal cost once deployed.

- An alternative to RAG would be to *fine-tune a base model* on the scenariogeneration corpus. Fine-tuning would eliminate the need for retrieval at inference time and could produce higher-quality output for well-covered manoeuvre classes. However, it requires a significantly larger corpus (the current 16-script corpus is insufficient for fine-tuning), specialised training infrastructure, and re-training every time the corpus or the scenariogeneration library version changes. RAG is more flexible and requires only adding a new script to the corpus to extend coverage to a new manoeuvre class. A natural extension of this trade-off becomes available once the RAG pipeline has produced a sufficiently large and diverse set of valid .xosc files: these generated scenarios could themselves serve as fine-tuning data for a base model, this time

trained directly on OpenSCENARIO XML output rather than on the intermediate Python representation. Such a model could potentially revisit the direct XML generation approach discussed above, which was initially abandoned due to systematic structural errors, now made viable by a corpus of validated, structurally correct examples rather than relying on the model's general-purpose understanding of the XML schema. This is noted here as a forward-looking direction rather than a path pursued in this thesis and is discussed further as future work.

- The prior laboratory pipeline used *deterministic template filling* given a manoeuvre class, a hard-coded function would populate a fixed scenario template with the extracted parameters. This approach is robust and fast but inflexible: each new manoeuvre class requires writing a new template function, and the output is always structurally identical regardless of the input parameters. ScenWeave's LLM-based generation introduces structural variability across runs, which is desirable for dataset diversity.

1.4 Document Structure

The remainder of this thesis is organised as follows:

- Chapter 2 provides the background necessary to follow the rest of the thesis: scenario-based simulation for synthetic data generation, the OpenSCENARIO and OpenDRIVE standards, the CARLA simulator, large language models for code generation, Retrieval-Augmented Generation, and the prior laboratory pipeline that ScenWeave extends.

- Chapter 3 reviews related work, surveying existing LLM-based scenario generation systems, RAG approaches for domain-specific code generation, and prior use of naturalistic driving data for scenario generation, closing with a systematic analysis of the research gap ScenWeave addresses.

- Chapter 4 describes the two input datasets in detail: the structure and semantics of the urban (JSON) and highway (NPY) driving data, the manoeuvre taxonomy used throughout the thesis, and the canonicalization pipelines that convert raw data into a unified format.

- Chapter 5 presents the system design and implementation, walking through every stage of the pipeline: the parameter extractor, the SpawnMatcher module, RAG-based retrieval, LLM-based generation, and the XSD validator.
- Chapter 6 reports the experimental evaluation: a comparative benchmark across five LLMs on the highway manoeuvre classes, an analysis of observed failure modes, latency profiling, and a production-scale validation run spanning both the highway and urban domains.
- Chapter 7 discusses the results, answering the three research questions directly, examining the trade-offs between local and cloud-hosted LLMs, and assessing the generalisability of the approach.
- Chapter 8 concludes the thesis with a roadmap for future work, including extending ScenWeave to larger corpora, additional CARLA maps, and multi-actor scenarios.

2 Background

A fundamental challenge in developing and validating systems for manoeuvre recognition in autonomous driving is the scarcity of labelled data for rare but critical events. Manoeuvres such as sudden braking, cut-in by a neighbouring vehicle, or abrupt lane changes occur infrequently in naturalistic driving datasets: even large-scale data collection campaigns spanning millions of kilometres may yield only a few hundred instances of each class. This statistical rarity makes it difficult to train robust classifiers or evaluate their performance with statistical confidence. Simulation addresses this problem by allowing controlled reproduction of any scenario with arbitrary frequency. Given a parametric description of a manoeuvre, the ego vehicle speed, the NPC position and velocity, the road geometry, a simulator can instantiate that scenario on demand and produce labelled data for the manoeuvre recognition system. The key challenge is translating real-world observations into parametric scenario descriptions that are faithful to the original event and varied enough to cover the relevant distribution of the manoeuvre class. ScenWeave operates in this context. Its input is a set of real driving recordings from the MOOVE naturalistic driving project, annotated with manoeuvre class labels. Its output is a set of executable simulation scenarios that reproduce the kinematic profile of the real event in CARLA, with controlled variation of the spawn point to produce a diverse dataset. The scenarios are not intended for AV testing (where the focus is on finding failure modes) but for dataset augmentation for manoeuvre recognition models.

2.1 OpenSCENARIO and OpenDRIVE

OpenSCENARIO [15] is an open standard developed by ASAM (Association for Standardisation of Automation and Measuring Systems) for the description of dynamic driving scenarios. Version 1.0, which is the version used in ScenWeave, defines an XML schema for specifying scenario actors, their initial states, their behaviours, and the conditions that trigger state transitions. The format is simulator-agnostic: the same .xosc file can in principle be loaded by any simulator that implements the standard, making scenarios portable across simulation platforms.

The top-level structure of an OpenSCENARIO 1.0 file is as follows:

Table 2.1 - Top-level structure of an OpenSCENARIO 1.0 file

Element	Description
FileHeader	Metadata: description, author, revMajor/revMinor, date
ParameterDeclarations	Named parameters used throughout the file (e.g. InitSpeed, BrakeTime)
CatalogReferences	References to vehicle, driver, and environment catalogues
RoadNetwork	Reference to the OpenDRIVE map file (.xodr) defining the road geometry
Entities	List of scenario actors (ego vehicle, NPC vehicles). Each has a bounding box and controller.
Storyboard > Init	Initial state of all entities: spawn position (WorldPosition or LanePosition), initial speed
Storyboard > Story > Act > ManeuverGroup	Sequence of Actions and Conditions defining the scenario behaviour. Each ManeuverGroup is assigned to one entity.
Storyboard > StopTrigger	Condition that terminates the scenario (e.g. SimulationTimeCondition)

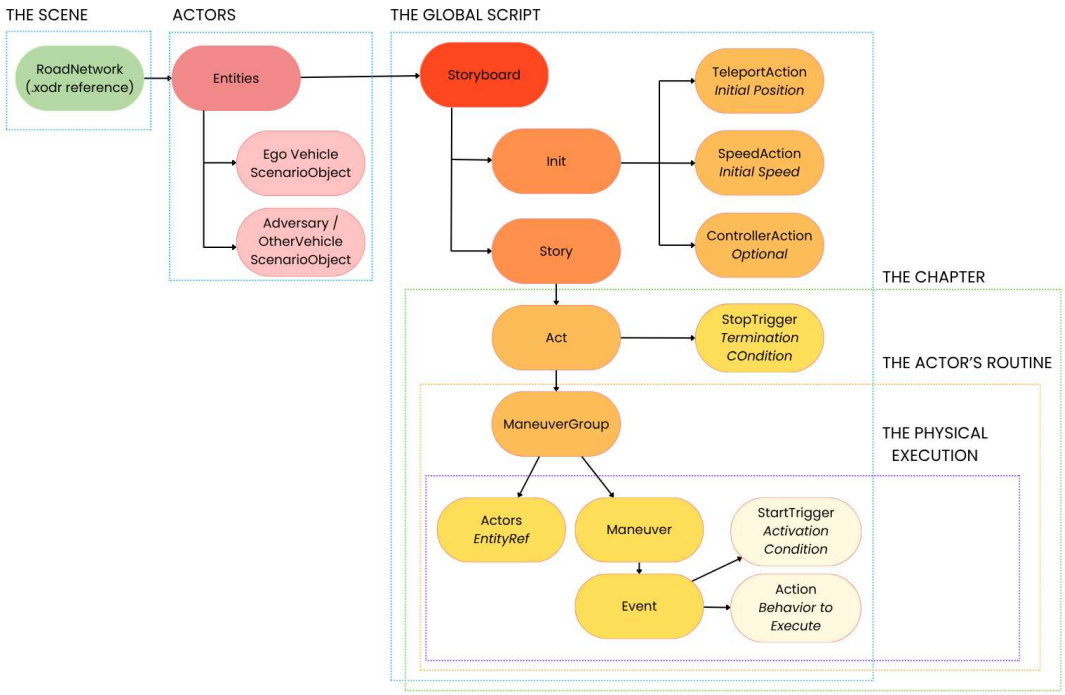

Figure 2.1 - Structure of an OpenSCENARIO 1.0 file

The Storyboard is the core of the scenario. It defines what happens after the entities are spawned: which vehicle accelerates, which brakes, when the traffic light changes state, and when the scenario ends. Actions are triggered by Conditions, which can be spatial (entity reaches a position), temporal (simulation time exceeds a threshold), or state-based (entity speed falls below a value). This event-driven structure is what makes OpenSCENARIO expressive enough to describe complex multi-actor manoeuvres without requiring a frame-by-frame trajectory specification. ScenWeave generates .xosc files conforming to OpenSCENARIO 1.0 and validates them against the ASAM XSD schema. The scenariogeneration Python library is used as the code-level interface: rather than constructing XML directly, corpus scripts call the library API to build the scenario object graph, which is then serialised to XML. This separation of concerns, Python logic in the corpus scripts, XML in the output, is what enables the LLM to generate correct scenarios without needing to know the XML syntax of OpenSCENARIO.

OpenDRIVE [16] is a complementary standard that defines road network geometry. A CARLA map is associated with an OpenDRIVE file that encodes road topology, lane structure, curvature, and junction connectivity. Spawn points in ScenWeave are referenced using two coordinate systems depending on the scenario domain. Urban scenarios use WorldPosition coordinates (x, y, z, heading) expressed in the CARLA world frame. Highway scenarios use LanePosition references (roadId, laneId, s-offset) derived from the OpenDRIVE map, which are more robust to road topology changes and correctly position the ego vehicle within a specific lane. The coordinate conversion from CARLA to OpenSCENARIO requires negating the y coordinate and heading angle, because CARLA uses a left-handed coordinate system while OpenSCENARIO uses a right-handed system. The SpawnMatcher module handles this conversion automatically before returning spawn coordinates to the pipeline.

2.2 CARLA Simulator and ScenarioRunner

CARLA (Car Learning to Act) [4] is an open-source simulator built on the Unreal Engine, designed specifically for autonomous driving research. It provides a Python API for controlling vehicles, sensors, and the environment; a set of pre-built maps (Town01–Town15) with varying road topology; and a physics engine that models vehicle dynamics,

pedestrian behaviour, and weather conditions. CARLA version 0.9.14 is used in ScenWeave, chosen for its stability and compatibility with the scenariogeneration library. The CARLA architecture is client-server: the simulator runs as a server process and exposes a TCP socket interface. Python client scripts connect to this interface to spawn actors, set their properties, and step the simulation. This architecture allows ScenWeave to run the scenario generation pipeline independently of the simulator and only connect to CARLA for the final execution step.

ScenarioRunner is the official CARLA tool for executing OpenSCENARIO files. It parses the .xosc file, instantiates all entities in the CARLA world, executes the storyboard actions according to the defined triggers, evaluates pass/fail criteria, and terminates the simulation when the stop trigger fires. ScenarioRunner is the downstream consumer of ScenWeave's output: a .xosc file that ScenarioRunner can load and execute without modification is the definition of a successful scenario generation. The validation module in ScenWeave (validate_xosc.py) uses the same XSD schema that ScenarioRunner uses internally, ensuring that validation results are predictive of execution success.

2.3 Large Language Models for Code Generation

Large Language Models (LLMs) are neural language models trained on large corpora of text and code, capable of generating syntactically correct and semantically meaningful code given a natural language or structured prompt. Their applicability to code generation has been extensively evaluated on benchmarks such as HumanEval [1] and MBPP [2] which measure performance on generic programming tasks well-represented in any training corpus. Zan et al. [9] survey this space comprehensively, cataloguing the range of LLM-based approaches to code generation and their respective strengths and limitations. For domain-specific code generation, where the target API is absent or underrepresented in the training corpus, the picture is more nuanced. General-purpose LLMs tend to hallucinate plausible-but-incorrect API calls, constructing function names and argument signatures that are syntactically valid Python but do not correspond to any real method in the target library. This hallucination pattern is precisely the failure mode that ScenWeave is designed to prevent through its RAG-based architecture. ScenWeave uses Qwen2.5-Coder-7b, a 7-billion parameter model from the Qwen2.5-Coder family, specialised for code generation

tasks through domain-specific pre-training and instruction tuning. The model is served locally via Ollama, which provides a standardised HTTP interface for local LLM inference without requiring cloud API access. The selection of this specific model was made on the basis of the benchmark reported in Baiardo et al. [5], which evaluates five models under identical RAG conditions and establishes that Qwen2.5-Coder-7b achieves the best accuracy-latency trade-off on the scenariogeneration API: 100% success rate on all 11 highway manoeuvre classes at approximately 25 seconds per scenario on the lab hardware.

Key LLM inference parameters used in ScenWeave:

- Temperature = 0.1: low randomness, reflecting a deliberate design choice for the current single-reference corpus structure; as the corpus is extended with multiple reference variants per class, a higher temperature could be explored to encourage more diverse recombination across references.
- Max tokens = 4096: sufficient for the longest corpus scripts (~200 lines of Python).
- Top-p = 0.9: nucleus sampling applied after temperature scaling.
- No system-level fine-tuning or LoRA adaptation: the model is used off-the-shelf with prompt engineering only.

ScenWeave makes use of a second, distinct LLM for a different purpose within the data preparation stage. During urban data canonicalization, a larger model (Qwen2.5:14b, also served locally via Ollama) is used to generate short natural-language descriptions of each scenario instance from its extracted parameters. This is a descriptive metadata synthesis task rather than a code generation task and operates entirely offline prior to the scenario generation pipeline described in Chapter 5.

2.4 Retrieval-Augmented Generation

Retrieval-Augmented Generation (RAG) [3] is a framework that augments LLM generation with relevant documents retrieved from an external corpus at inference time. Rather than relying solely on the parametric knowledge encoded in the model weights, RAG conditions generation on retrieved examples or documents that are relevant to the current query. This approach is particularly effective for domain-specific tasks where the relevant knowledge is too specialised to be reliably memorised during training. The standard RAG pipeline consists

of three stages: indexing (the corpus is embedded and stored in a vector database), retrieval (a query embedding is compared to corpus embeddings to identify the top-k most similar documents), and augmented generation (the retrieved documents are inserted into the prompt alongside the query). ScenWeave adopts a simplified but highly effective variant of RAG where the retrieval stage uses deterministic keyword matching rather than embedding similarity. This design choice is motivated by the structure of the corpus: each script has a unique filename that encodes the manoeuvre class tag, making the retrieval problem equivalent to a dictionary lookup. The result is R@1 = 1.0 on all scenario classes, perfect retrieval precision, with zero computational overhead from embedding inference.

The RAG components in ScenWeave are:

Table 2.2 - RAG components of the ScenWeave pipeline

Component	**Role in ScenWeave**
Corpus	16 Python scripts implementing canonical scenario generation functions using the scenariogeneration API
Retrieval	Keyword match on manoeuvre class tag against corpus filenames. R@1 = 1.0 on all classes. Semantic embedding fallback (BAAI/bge-large-en-v1.5) as last resort.
Augmentation	Retrieved script injected into structured six-section prompt as reference. LLM instructed to adapt the reference implementation to new parameters, understanding their role within the script's logic rather than performing mechanical substitution.
Generation	Local LLM (Qwen2.5-Coder-7b via Ollama) produces adapted Python code. Self-healing loop re-injects error tracebacks on failure.

The augmentation strategy is the most critical design decision in the ScenWeave RAG pipeline. Rather than inserting the retrieved script as additional context and asking the LLM to generate a new scenario from scratch, ScenWeave instructs the LLM to reproduce the entire retrieved script and adapt only the numeric parameter values to match the extracted scenario data. This framing gives the model a strong structural anchor to reason from: the LLM still has to understand what each parameter represents, where it belongs within the script's logic, and how it interacts with the surrounding code, but it does so by adapting a

known-correct implementation rather than generating from an open-ended specification. This substantially reduces the risk of hallucinated API calls while still requiring genuine comprehension of the reference script's structure.

2.5 Previous Generation Pipeline

ScenWeave was developed as an evolution of an existing pipeline developed at the ELIOS Lab, University of Genova. The prior pipeline generated driving scenarios for CARLA using deterministic Python scripts: given a set of scenario parameters, a hard-coded function would produce a CARLA-specific Python file that, when executed, spawned the appropriate vehicles and reproduced the manoeuvre. This approach was functional but had three significant limitations that ScenWeave is designed to address.

Table 2.3 - Comparison between the prior laboratory pipeline and ScenWeave

Property	**Prior lab pipeline**	**ScenWeave**
Output format	CARLA Python scripts (proprietary)	OpenSCENARIO 1.0 (.xosc), portable standard
Generation logic	Deterministic, hard-coded	RAG and LLM, data-driven
New manoeuvre support	Requires rewriting code	Add one script to corpus

The first limitation was the output format. CARLA-specific Python scripts are not portable: they depend on the CARLA Python API version and cannot be directly loaded by other simulators or shared with the broader scenario generation community. OpenSCENARIO 1.0, by contrast, is a validated standard with a formal XSD schema, making it verifiable, tool-agnostic, and compatible with any simulator that implements the standard.

The second limitation was the deterministic generation logic. The prior pipeline produced the same output for the same input, with no mechanism for generating diverse variations of the same scenario. ScenWeave introduces diversity through two mechanisms: the spawn point rotation (round-robin across the top-N candidates from the SpawnMatcher) and the LLM-based generation (which, even at low temperature, produces slight structural variations across runs).

The third limitation was extensibility. Adding a new manoeuvre class to the prior pipeline required writing a new deterministic generation function from scratch. In ScenWeave, adding a new class requires only adding one Python script to the RAG corpus, a task that is substantially simpler and does not require modifying any existing pipeline code.

3 Related Work

3.1 Automated Driving Scenario Generation

The automated generation of driving scenarios for simulation has been an active research area for over a decade, driven by the impossibility of collecting sufficient real-world data to cover the long tail of rare but safety-critical events. Early approaches relied on combinatorial methods and rule-based systems that enumerated scenario parameters within predefined ranges, producing large volumes of scenarios but with limited semantic coherence and no grounding in real-world driving behaviour. More recent work has shifted towards data-driven and learning-based methods.

TrafficGen [11] learns to generate diverse and realistic traffic scenarios from naturalistic driving datasets by modelling agent behaviour distributions on HD maps. While it produces plausible scenarios grounded in real data, the output is a trajectory distribution rather than an executable simulation file, requiring additional steps to instantiate scenarios in a simulator.

Scenic [12] introduced a domain-specific language (DSL) for probabilistic scenario specification. Scenic programs are human-readable and executable in CARLA, but authoring them requires expert knowledge of the DSL syntax and manual parameter specification. ScenWeave differs fundamentally: rather than asking a human to write a scenario description, it takes raw numerical driving data and produces executable code automatically, with no manual specification of parameters.

3.2 LLM-Based Scenario Generation for Simulation

The availability of large language models with strong code generation capabilities has opened a new direction in scenario generation: using LLMs to translate high-level descriptions or structured representations into executable simulation scripts. Several systems have been proposed in this space, all sharing the characteristic of using cloud-hosted LLMs and text or structured knowledge as the primary input.

ChatScene [13] is the most widely cited work in LLM-driven safety-critical scenario generation. It uses a knowledge graph derived from traffic accident reports to guide GPT-4 in generating scenarios for CARLA. The knowledge graph provides structured safety-critical knowledge that reduces hallucination and improves scenario diversity. ChatScene generates scenarios in the form of CARLA Python scripts and evaluates them using reinforcement learning agents. Its primary limitation for the ScenWeave use case is that the input is a knowledge graph derived from accident reports rather than real naturalistic driving data, and the output is CARLA-specific Python rather than the portable OpenSCENARIO standard.

Talk2Traffic [14] extends the LLM-based generation paradigm to multimodal inputs, allowing users to specify scenarios through text, speech, or hand-drawn sketches. A multimodal LLM interprets these inputs and produces structured representations, which are then translated into executable Scenic code using a retrieval-augmented generation mechanism. Notably, Talk2Traffic is one of the few prior works to explicitly adopt RAG for scenario generation, using it to reduce hallucinations in the Scenic DSL syntax. However, the retrieval corpus consists of Scenic code templates rather than domain-specific Python scripts, and the input remains user-provided text rather than real driving data. ScenWeave shares the RAG mechanism but applies it to a fundamentally different input modality: structured numerical parameters extracted from a naturalistic driving dataset.

A complementary direction translates *Operational Design Domain (ODD)* [24] specifications into CARLA-executable scenarios. One recent system decomposes ODD descriptions into environmental, scenery, and dynamic elements using a Tree of Thoughts prompting strategy and generates ScenarioRunner scripts compatible with CARLA. The method is evaluated on scenario diversity and syntactic correctness. Its key difference from ScenWeave is the input: ODD descriptions are abstract, human-authored safety specifications, while ScenWeave starts from concrete kinematic measurements of real driving events. Several recent works integrate LLM-based scenario generation into closed-loop evaluation pipelines, where generated scenarios are executed by a full-stack autonomy system and performance feedback is used to iteratively refine the generation. These systems typically use chain-of-thought reasoning and progressive difficulty scaling. While they achieve impressive coverage of edge cases, they rely on GPT-4 or equivalent cloud models, do not support offline deployment, and do not use real-world driving data as the parametric

source. ScenWeave is designed from the outset for local deployment with no cloud API dependency.

TARGET [23] addresses the difficulty of generating syntactically correct OpenSCENARIO or Scenic code directly from LLMs by introducing a simplified intermediate DSL. The LLM generates scenarios in the simplified DSL, which is then compiled to executable code. The approach reduces hallucination rates on complex DSL syntax and achieves high correctness on the Autoware platform. ScenWeave takes a different approach to the same problem: instead of simplifying the target language, it provides the LLM with a complete working script as a structural reference and instructs it to adapt that implementation to new parameters, requiring the model to understand the role of each parameter within the script's logic, rather than mastering the API syntax independently.

3.3 RAG for Domain-Specific Code Generation

Retrieval-Augmented Generation was originally proposed for knowledge-intensive NLP tasks [3] and has since been widely adopted for code generation tasks involving domain-specific APIs absent from the model training corpus. The core insight is that injecting a retrieved example of the target API at inference time substantially reduces hallucination of non-existent methods or incorrect signatures.

RepoCoder [8] applies iterative retrieval from repository-level code to improve code completion, showing that retrieval from domain-relevant code is more effective than generic retrieval. CodeT5+ [7] demonstrates that code-specific pre-training combined with retrieval at inference time significantly outperforms general-purpose LLMs on domain-specific tasks. These results motivate the corpus-based retrieval approach adopted in ScenWeave.

Dedicated benchmarks such as *DS-1000* [6] further demonstrate that data-science-specific code generation remains challenging even for large models without domain-specific retrieval.

The closest work to ScenWeave in the RAG-for-code space is the *NAMD-Agent system* [25], which uses RAG over molecular dynamics simulation scripts to enable LLMs to generate correct NAMD configuration files, an API similarly absent from training data. The analogy

to ScenWeave is direct: both systems apply RAG to a scientific simulation API that is too specialised to be reliably known by any general-purpose LLM.

The benchmark presented in Baiardo et al. [5] directly evaluates the effectiveness of RAG for the scenariogeneration API used in ScenWeave. The benchmark compares five code LLMs across three domain specificity levels and demonstrates that keyword-based retrieval with a structured six-section prompt enables four out of five models to achieve 100% success rate on the domain-specific L3 level, an API almost certainly absent from their training data. This result validates the RAG-based approach of ScenWeave and provides empirical grounding for the model selection (Qwen2.5-Coder-7b) adopted in the production pipeline.

3.4 Naturalistic Driving Data for Scenario Generation

Several works have explored the use of real-world driving data as a source for simulation scenarios, motivated by the same data gap that ScenWeave addresses. The MOOVE project, from which the ScenWeave dataset is derived, collected approximately one million kilometres of naturalistic driving data across 15 European countries. Similar large-scale naturalistic datasets include *D2-City* [10] and various dashcam datasets, but these are primarily used for perception training rather than scenario generation. The Scenic language DSL includes constructs for replaying real-world trajectories in simulation, and several works have explored extracting scenario parameters from crash reports or incident databases. However, none of the surveyed works implement an end-to-end pipeline that takes raw numerical driving data in a domain-specific format (JSON or NPY) and produces validated, executable OpenSCENARIO files automatically, without manual parameter specification or human-in-the-loop editing.

3.5 Comparative Analysis and Research Gap

The table below summarises the key characteristics of the most relevant prior systems in relation to ScenWeave along five dimensions: input modality, output format, LLM type, use of real-world driving data, and direct executability in CARLA.

Table 3.1 - Comparative analysis of LLM-based scenario generation systems

System	Input	Output	LLM	Real-world data
ChatScene (2024)	Knowledge graph and text	CARLA scripts	GPT-4 (cloud)	No
Talk2Traffic (2025)	Text / speech / sketch	Scenic code	MLLM (cloud)	No
ODD→CARLA (2025)	ODD description	ScenarioRunner scripts	GPT and ToT (cloud)	No
TARGET (2023)	Traffic rules	Custom DSL → executable	GPT-4 (cloud)	No
ScenWeave (this work)	Real driving data (JSON/NPY)	OpenSCENARIO (.xosc)	Local LLM (Qwen2.5-Coder-7b)	Yes

The comparison reveals a consistent pattern across all prior work: existing LLM-based scenario generation systems take high-level human-authored inputs (text descriptions, ODD specifications, knowledge graphs) and produce executable code using cloud-hosted LLMs. None of the surveyed systems start from raw numerical driving data in a domain-specific binary or JSON format, and none use a local LLM deployed without cloud API dependency. Rather than natural language or structured knowledge graphs, ScenWeave takes parametric data extracted directly from real driving recordings (NPY binary files and JSON records from the MOOVE project). This grounding in real kinematic measurements ensures that generated scenarios are statistically representative of actual driving behaviour rather than plausible-but-synthetic constructions. ScenWeave uses a local LLM (Qwen2.5-Coder-7b via Ollama) with no dependency on cloud APIs, making it suitable for environments with data privacy constraints or no internet connectivity. rather than semantic embedding similarity, ScenWeave uses deterministic keyword retrieval that achieves R@1 = 1.0 on all scenario classes, eliminating retrieval noise as a source of generation failures.

The closest prior work in terms of using RAG for scenario code generation is Talk2Traffic, which uses a retrieval mechanism over Scenic templates. However, Talk2Traffic retrieves from a corpus of human-authored templates keyed on natural language descriptions, while

ScenWeave retrieves from a corpus of domain-specific Python scripts keyed on maneuver class identifiers derived from real driving taxonomy. This distinction is significant: keyword retrieval on domain labels is both simpler and more reliable than semantic retrieval on natural language descriptions when the query is a structured numerical parameter set rather than a human-written sentence.

4 Dataset and Analysis

This chapter describes the two datasets used as input to the ScenWeave pipeline: the highway driving scenarios dataset derived from NPY binary files and the urban driving scenarios dataset from JSON records. This chapter provides a comparative analysis of the two formats, describes the data canonicalization pipelines for both the highway and urban domains and contextualises the dataset against publicly available driving datasets.

4.1 Highway Driving Scenarios Dataset

This section provides an overview of the parametric data structure underpinning the synthetic highway driving scenario dataset generated using the CARLA simulator. The dataset was produced as part of the research framework described in the PhD thesis by Cossu M. [22]. Each scenario instance is parameterised by a set of kinematic and contextual variables sampled from multivariate distributions derived from real-world driving data (MOOVE project, ~1 million km across 15 European countries). The sampled values are stored in NumPy binary files (.npy), one file per scenario class, used to instantiate concrete simulation episodes in CARLA. All scenarios take place on highway road layouts and represent interactions between the ego vehicle (EV) and one or more surrounding actors. The dataset covers 12 highway driving scenario classes organised into 6 functional groups. Each class has a corresponding .npy parameter file. The semantic meaning of each scenario class in terms of real-world driving behaviour is summarised in Table 4.1.

Table 4.1 - Semantic description of highway manoeuvre classes

Manoeuvre	**Meaning**	**n. Features**
BRAKE	The ego vehicle brakes on its own, e.g. because the vehicle ahead is slowing; no other vehicle actively interacts	8
BR_BK	Same as Brake, but occurring with surrounding traffic present, which may influence how the ego brakes	8
CINF	A vehicle from an adjacent lane merges into the ego's lane ahead of it, reducing the following distance	8
CINB	A vehicle from an adjacent lane merges into the ego's lane behind it, closely following the ego	8
COUTF	A vehicle previously ahead of the ego leaves its lane for an adjacent one, clearing the path in front	8
COUTB	A vehicle previously behind the ego leaves its lane for an adjacent one, no longer following it	8
LCF	The ego vehicle changes lane while another vehicle is present ahead	5
LCB	The ego vehicle changes lane while another vehicle is present behind	5
FLF	The ego vehicle follows another vehicle ahead at a close, steady distance (ordinary car-following)	11
FLB	Similar to Flf, but with a larger or more variable following distance	11
FRF	The ego vehicle drives with no relevant vehicle ahead to follow or react to	25
FRB	The ego vehicle drives with no relevant vehicle behind influencing its behaviour	25

The *Follow Front and Follow Behind* groups are each divided into four kinematic sub-classes defined by the MOOVE project taxonomy.

Table 4.2 - Follow scenario kinematic sub-classes (B1–B4)

Sub-class	Definition
B1	The ego vehicle gets farther away from the lead vehicle, due to the leader accelerating or the ego slowing down.
B2	The ego vehicle gets closer to the lead vehicle, typically because the ego speed is higher than the leader's.
B3	The ego vehicle maintains a speed similar to the lead vehicle, keeping a roughly constant following distance (neither closing in nor falling back).
B4	The ego vehicle follows the lead vehicle at a distance greater than 50 metres, regardless of relative speed.

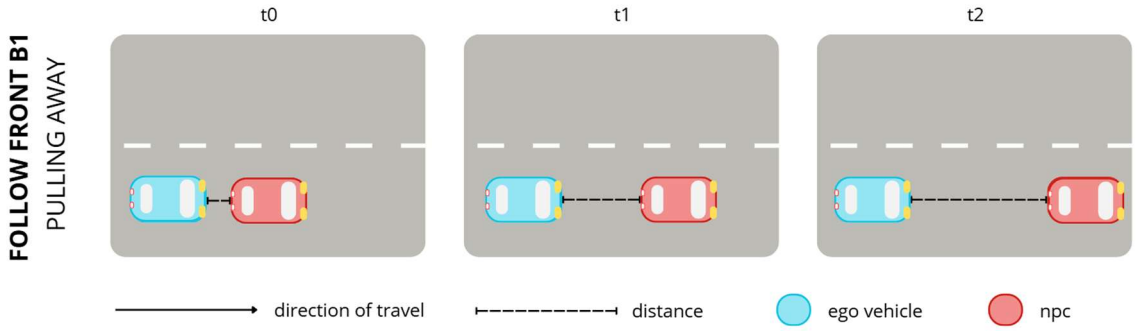

Figure 4.1 – Manoeuvre FLF B1

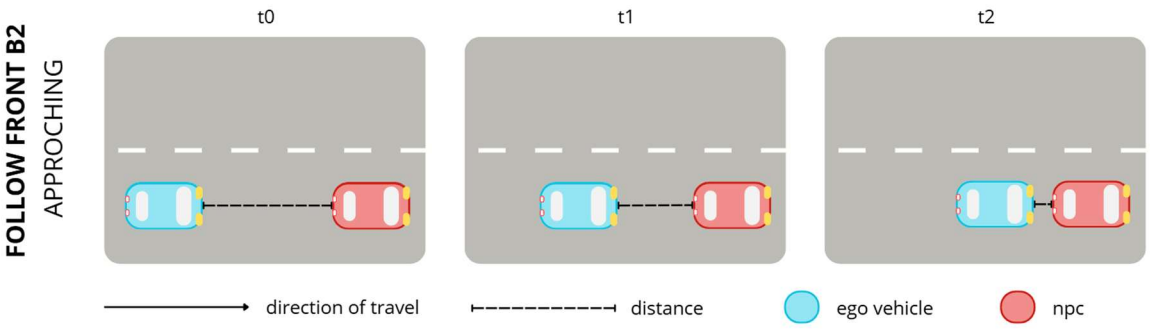

Figure 4.2 – Manoeuvre FLF B2

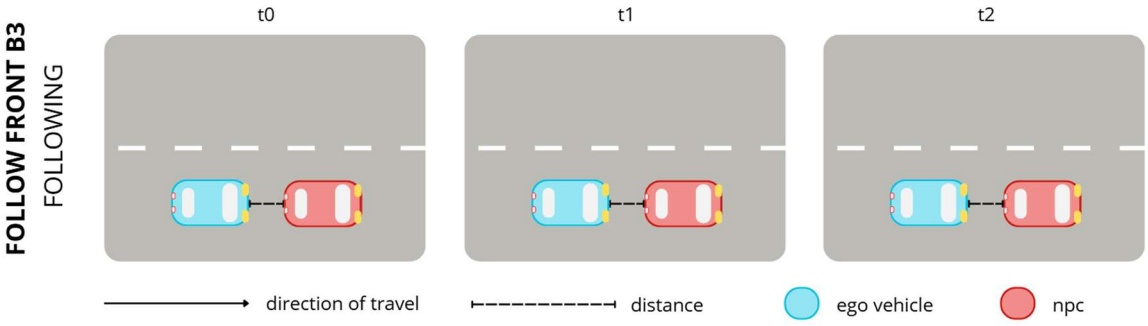

Figure 4.3 – Manoeuvre FLF B3

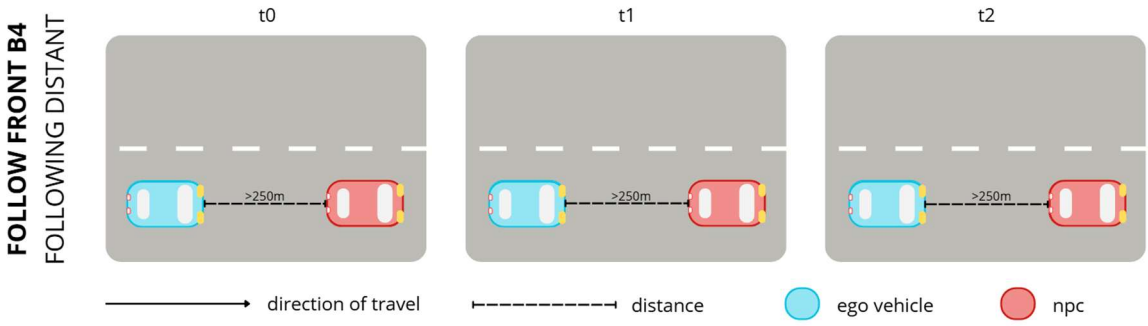

Figure 4.4 – Manoeuvre FLF B4

Common parameter conventions:

- Speed units: all ego and inter-vehicle speed fields are stored in km/h, except VEL_CENTER, VEL_RIGHT, VEL_LEFT in Free Ride Front (FRF) which are in m/s.
- Distance units: all positional fields (POS, DIST, POSX_*) are in metres.
- Sentinel values: in Free Ride files, absent surrounding vehicles are flagged with −1 (position) or −3.6/−1.0 (velocity).
- Data types: all values stored as strings (dtype object). Numeric fields cast to float at runtime.
- RIGHT field: present as 'Yes'/'No' (opposite-direction traffic) in most files; in FRF it stores driving direction ('Right'/'Left').

BRAKE, BR_BK, CINF, CINB, COUTF, and COUTB share the same 8-features structure. Key differences are summarised below.

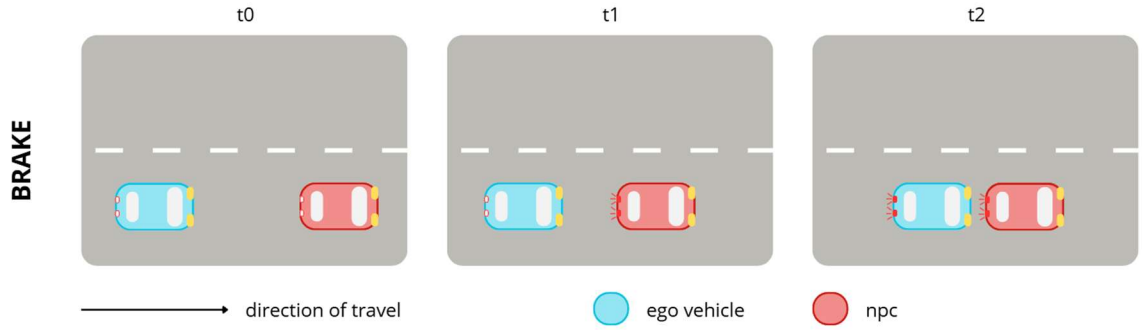

Figure 4.5 – Manoeuvre BRAKE and BR_BK

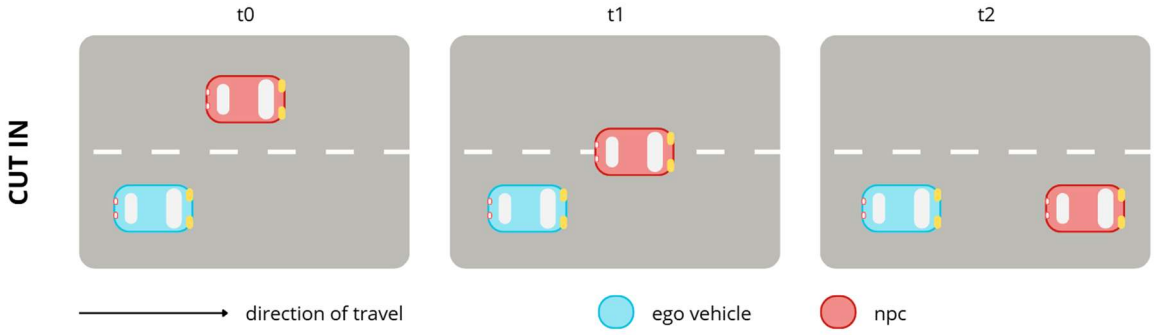

Figure 4.6 – Manoeuvre CUT IN FRONT

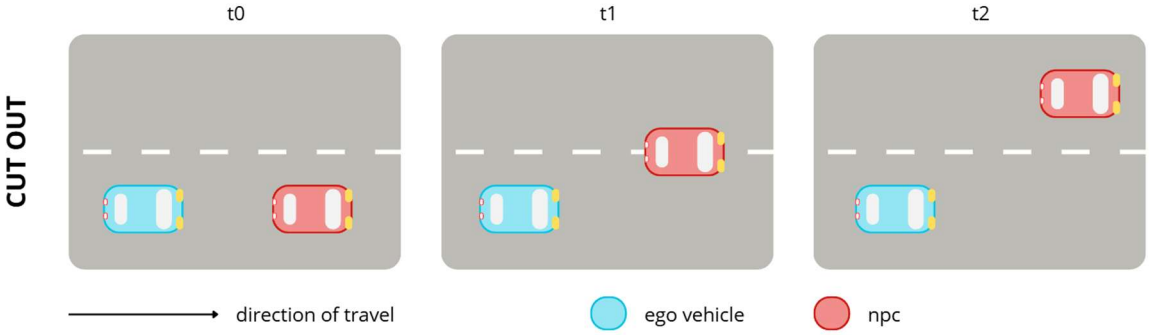

Figure 4.7 – Manoeuvre CUT OUT FRONT

Table 4.3 - NPC speed field conventions and POS sign for Brake, Cut-In and Cut-Out scenarios

Scenario	NPC speed field	POS sign	Notable specifics
BRAKE	LEAD_VEL (absolute)	Positive (NPC ahead)	BRAKE field (index 5): normalised braking intensity 0–1.
CINF	REL (relative)	Positive (NPC ahead)	LEAD_VEL = EGO_VEL + REL. Negative REL = NPC faster than ego.
CINB	REL (relative)	Negative (NPC behind)	LEAD_VEL = EGO_VEL + REL. Negative REL = NPC slower than ego.
COUTF	REL (relative)	Positive (NPC ahead)	REL and DELTA very small (~0.1–0.9 km/h): NPC nearly at same speed.
COUTB	REL (relative)	Negative (NPC behind)	LEAD_VEL = EGO_VEL − REL.

FLF and FLB share an identical 11-features structure. In FLF the other vehicle is ahead of the ego (as we can see in figure 4.1, 4.2, 4.3, 4.4); in FLB it is behind. Example rows and sub-class examples are provided in the detailed field documentation (Appendix B).

Table 4.4 - Field structure of Follow Front and Follow Behind scenarios (11-field schema)

Idx	Features	Unit	Example	Description
0–1	EGO_VEL_0/1	km/h	10.9 / 4.0	Initial and final ego speed
2–3	LEAD_VEL_0/1	km/h	17.0 / 13.0	Initial and final speed of other vehicle. Delta = Lead_VEL_1 − Lead_VEL_0
4–5	POS_0/1	m	11.0 / 10.0	Initial and final inter-vehicle distance
6	DIST	m	20.0	Distance travelled by other vehicle during scenario
7	DUR	s	2.0	Duration of the scenario
8	RIGHT	,	'Yes'	Presence of opposite-direction traffic
9	BEHAVIOUR	,	'B1'–'B4'	Sub-class label (MOOVE taxonomy)
10	TYPE_VEHICLE	,	'Car'	Type of other vehicle

Free Ride is the most complex scenario class. The ego drives freely while up to three surrounding vehicles (centre lane ahead, right lane, left lane) may or may not be present. FRF and FRB share the same 25-features structure with two differences: DUR is at index 9 in FRB and index 21 in FRF; the RIGHT field stores opposite-direction traffic ('Yes'/'No') in FRB and driving direction ('Right'/'Left') in FRF. Absent vehicles are flagged with −1 (position) or −3.6/−1.0 (velocity).

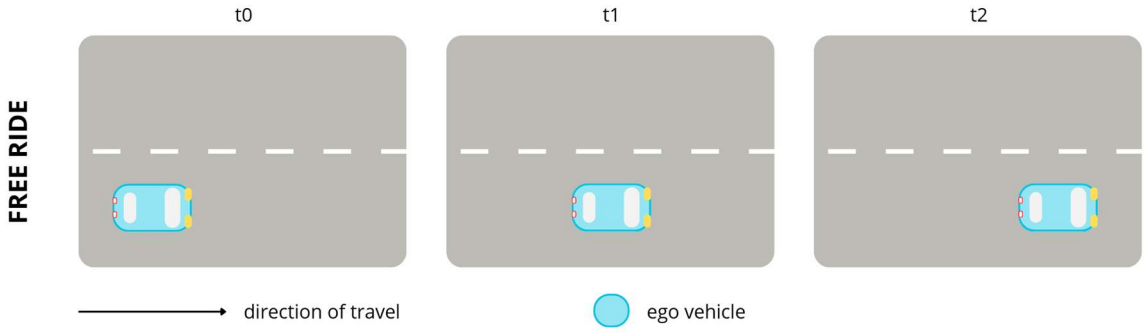

Figure 4.8 – Manoeuvre FREE RIDE

LCF and LCB are the simplest scenario class with only 5 features. Only ego kinematics and lane change direction are stored; surrounding traffic is generated procedurally at runtime. Example values show ego speeds around 125–130 km/h, consistent with high-speed highway lane changes.

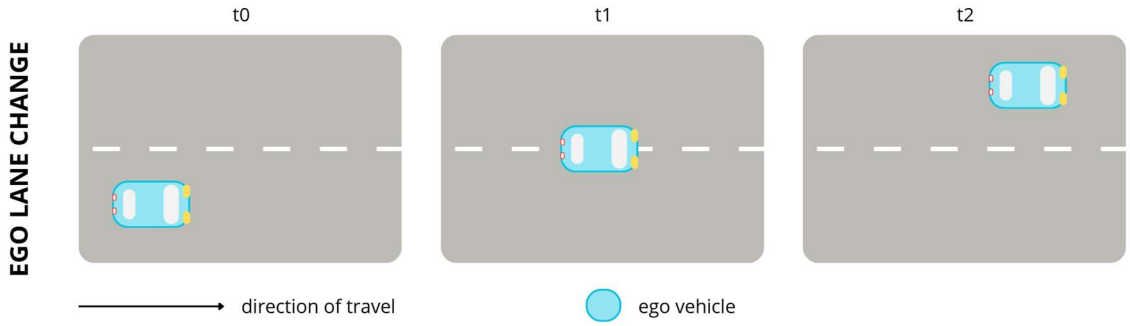

Figure 4.9 – Manoeuvre EGO LINE CHANGE

Table 4.5 - Field structure of Lane Change scenarios (5-field schema)

Idx	Features	Unit	Example	Description
0	EGO_VEL_0	km/h	125.4	Initial ego speed at start of lane change
1	EGO_VEL_1	km/h	129.5	Final ego speed. DELTA = EGO_VEL_1 − EGO_VEL_0
2	DUR	s	3.9	Duration of the lane change manoeuvre
3	OTHER_DIR_TRAFFIC	,	'Yes'	Presence of opposite-direction traffic (not used in scenario logic)
4	DIRECTION	,	'Left'	Direction of the ego lane change

4.2 Urban Driving Scenarios Dataset

The urban driving scenarios dataset consists of real-world driving data collected under the MOOVE project. Each scenario instance is stored as a structured JSON file capturing a temporal segment of ego vehicle behaviour, road infrastructure, surrounding traffic, and environmental conditions. The dataset currently covers five manoeuvre classes: one baseline in-lane driving class (U1_1) and four intersection-related classes (U4_1 through U4_4), all involving traffic-light-controlled crossroads. All instances were recorded in France (ISO country code 250).

Table 4.6 - Urban manoeuvre classes and corresponding ego behaviour at traffic-light-controlled intersections

Code	Name	TL state	Ego behaviour
U1_1	In-Lane Driving	N/A	Constant speed, no intersection
U4_1	Ego Stop at TFL	Red or Orange	Decelerates and stops at crossroad
U4_2	Ego Pass TFL	Green (or late Orange)	Maintains speed, passes intersection
U4_3	Ego Straight through Green TFL	Green	Passes intersection going straight
U4_4	Ego Right Turn at Green TFL	Green	Passes intersection turning right

Each manoeuvre class has a dedicated folder. Each folder contains N JSON files, one per scenario instance. The filename corresponds to the begin_time of the segment (e.g. 1726.4.json), expressed in seconds from the start of the recording session.

Each JSON file contains four top-level objects: scenario_code (manoeuvre type), segment (temporal boundaries and recording metadata), environment.infrastructure (waypoints with road geometry and signage), attributes (country code), and key_times (array of sampled frames, each with ego_vehicle, temporal_climatic, positioning, and obstacles sub-objects).

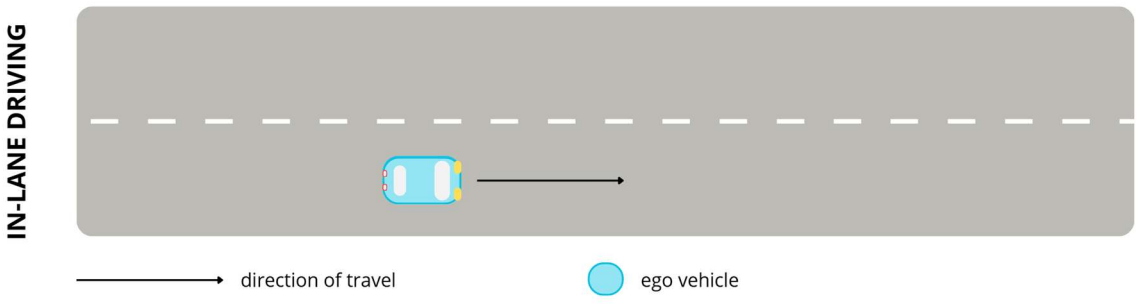

Figure 4.10 – Manoeuvre U1_1 IN-LINE DRIVING

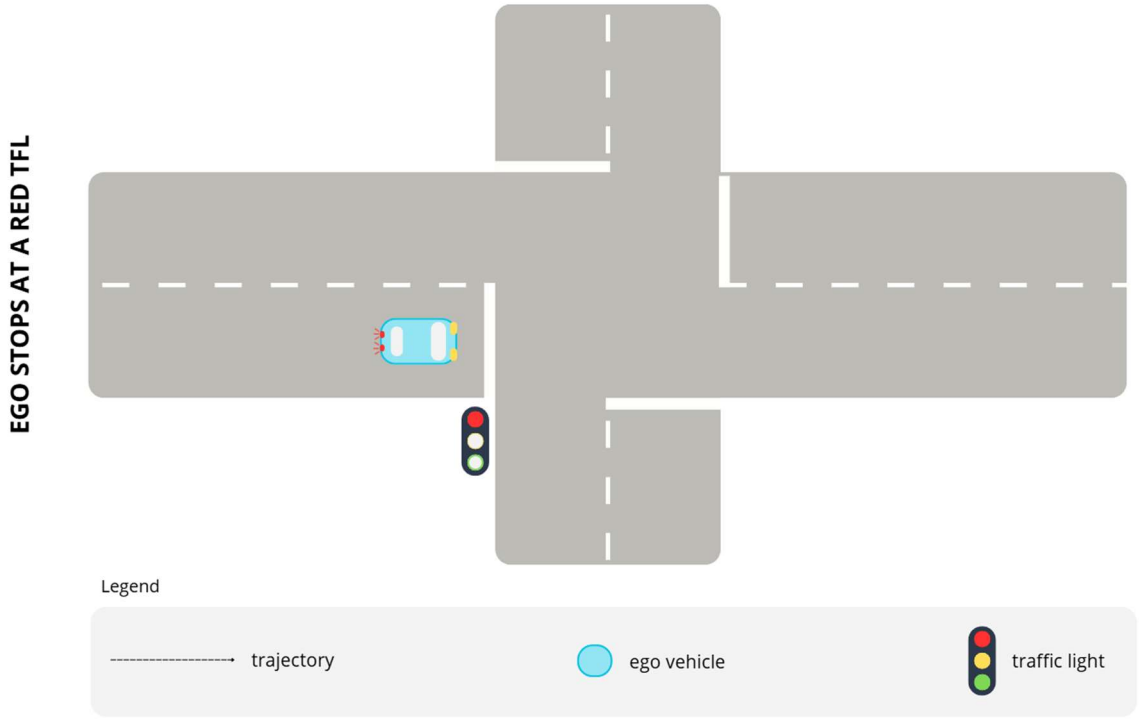

Figure 4.11 – Manoeuvre U4_1 EGO STOP AT TFL (red TFL)

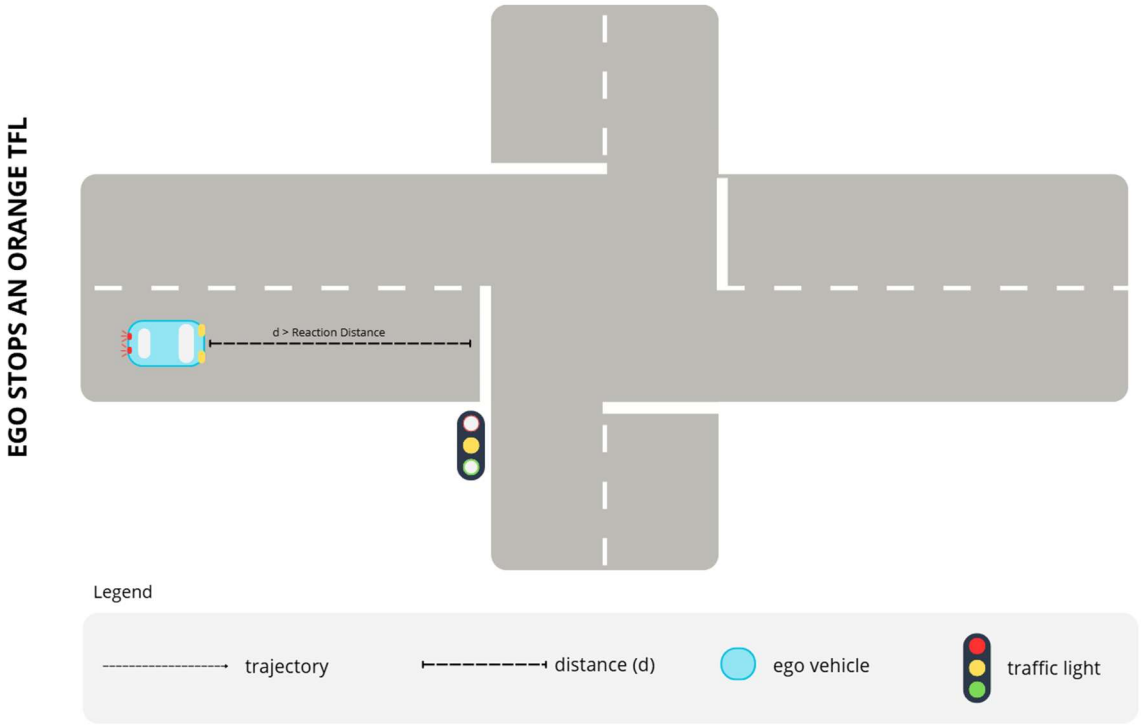

Figure 4.12 – Manoeuvre U4_1 EGO STOP AT TFL (orange TFL)

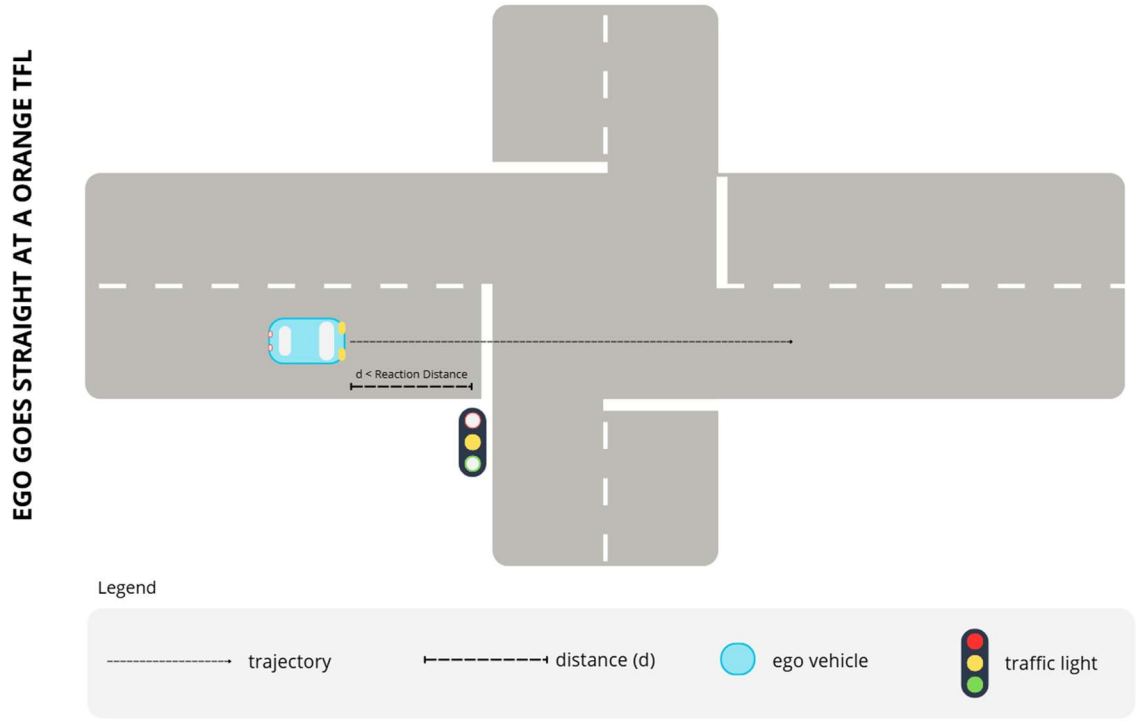

Figure 4.13 – Manoeuvre U4_2 EGO PASS TFL (orange TFL)

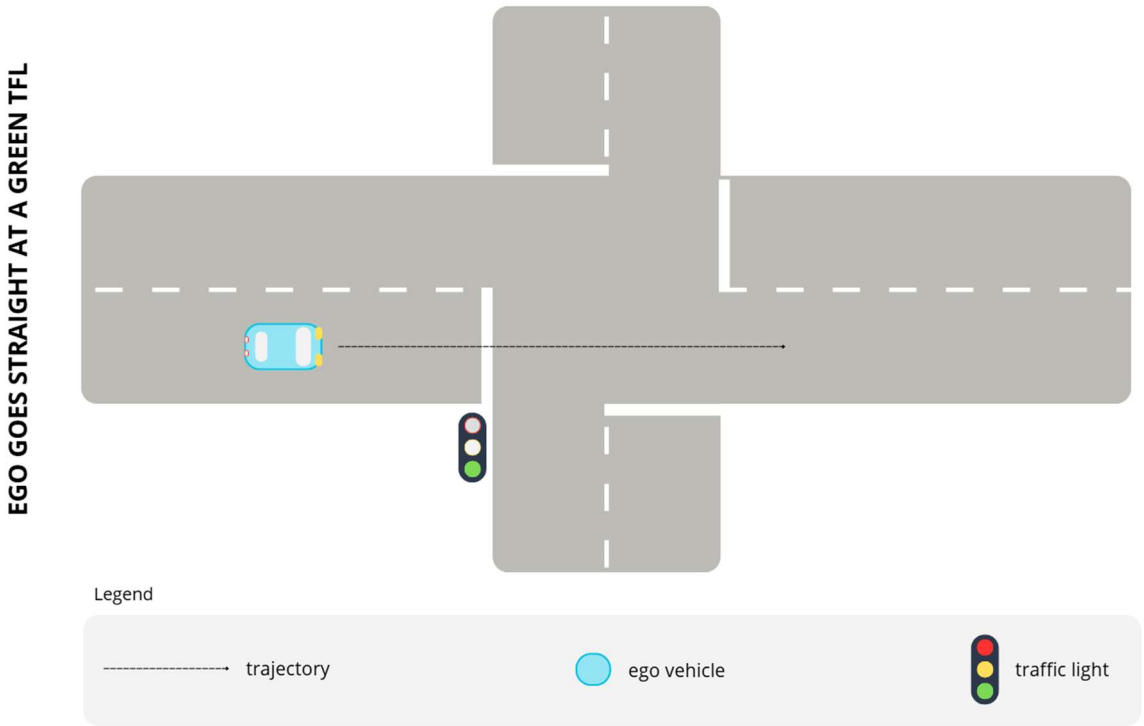

Figure 4.14 – Manoeuvre U4_3 EGO PASS STRAIGHT THROUGH GREEN TFL

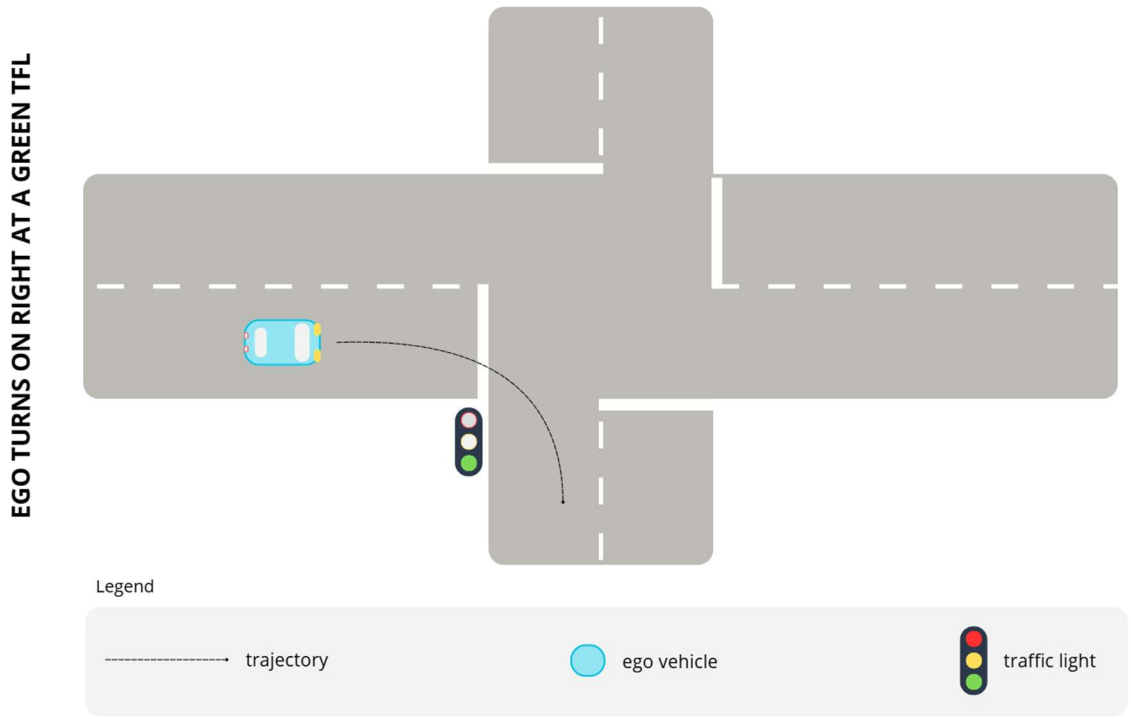

Figure 4.15 – Manoeuvre U4_4 EGO RIGHT TURN AT GREEN TFL

Table 4.7 - Key ego vehicle fields in the urban ADScene JSON format

Field	Unit	Example	Description
ego_speed	km/h	70.36	Instantaneous ego speed
longitudinal_acceleration_wheels	m/s²	0.117	Positive = acceleration, negative = braking
ego_brake	,	0	Brake pedal state. 0 = not braking
steer_wheel_angle	deg	0.0	Positive = right, negative = left
ego_lane	,	1	Current lane index of ego vehicle
dist_to_marking_left/right	m	−1.66 / 1.55	Distance to lane markings (negative = marking to that side)
turning_signal_state	,	0	0=inactive, 1=right, 2=left (RT_TurningSignalState)
latitude / longitude	deg	48.7744 / 2.1836	GPS position (WGS84)

The environment.infrastructure.way_points array describes road geometry and signage along the ego trajectory. Each waypoint is located at a longitudinal distance (position_longi_from_0, in metres) and carries infrastructure elements including SpeedLimit, RoadType, LaneWidth, FrontCurvature, RearCurvature, Slope, TrafficSignType, and GuardRail. Traffic light states are encoded as TrafficSignType codes: 167 = Green, 170 = Orange, 172 = Red. The current instances use RoadType = 2 (Country Road / Secondary Road), reflecting the naturalistic nature of the MOOVE collection.

Each frame may contain zero or more obstacles detected by the ego vehicle sensor suite. Key fields include: pos_x (longitudinal distance, positive = ahead), pos_y (lateral offset, negative = right), obstacle_lane, obstacle_relative_lane (0 = same lane as ego), absolute_velocity_x, relative_velocity_x, time_to_collision, mobile_object_classification (1=Car, 3=Truck, 4=Pedestrian), and blinker state.

While all five classes share the same JSON schema, each is characterised by a distinct combination of field values. U4_1 (stop) shows ego_brake > 0 and ego_speed → 0 in final frames, with TrafficSignType codes 170–172 in infrastructure. U4_4 (right turn) shows turning_signal_state = 1 and positive steer_wheel_angle during the manoeuvre. U1_1 shows no traffic sign events and constant ego_speed across all key_times.

4.3 Comparative Analysis: Urban vs Highway

The urban and highway datasets share the same origin (MOOVE project) and the same goal (providing real-world kinematic data for simulation), but differ substantially in format, richness, and the information available for scenario generation. The table below summarises the key differences across eight dimensions.

Table 4.8 - Comparative analysis of urban and highway dataset properties

Property	**Urban (U1_1, U4_x)**	**Highway (BRAKE, CIN…)**	**Notes**
Input format	JSON (structured text)	NPY binary → JSON	Highway requires canonicalisation step
Sampling rate	Event-based key_times frames	Bin-based per-manoeuvre samples	Urban: variable frame count; Highway: fixed fields per record
Ego kinematics	Speed, acceleration, steer, lane, GPS	Speed (begin/end), duration	Urban has richer ego state; Highway focuses on kinematic endpoints
NPC representation	Obstacle list (pos, vel, classification)	Single NPC: relative speed, distance, direction	Urban: multiple obstacles per frame; Highway: one principal actor
Traffic light	Encoded in infrastructure waypoints (TrafficSignType)	Not present	Urban: TFL state drives scenario class; Highway: no TFL
Weather	Explicit (precipitation, fog, day_period)	Not stored, assigned statically	Urban: extracted from temporal_climatic; Highway: static Day_Clear
Road geometry	Waypoints with curvature, lane width, slope	Derived from Town04 spawn DB	Urban: real road geometry from MOOVE; Highway: CARLA map geometry
Scenario classes	5 (U1_1, U4_1–U4_4)	12 (BRAKE, CINF/B, COUTF/B, FLF/B B1–B4, FRF/B, LCF/B)	Highway covers more classes due to kinematic subclasses
Country	France (code 250)	15 European countries	Highway dataset has broader geographic coverage

The most significant difference for the ScenWeave pipeline is the NPC representation. Urban JSON files provide a full obstacle list with per-frame position and velocity measurements for all detected vehicles, allowing the extractor to reconstruct the dynamic interaction between the ego and surrounding actors. Highway NPY files, by contrast, store only the kinematic endpoints of the principal interaction (initial and final speed, gap, duration)

without frame-level trajectories. This means that the extractor for highway scenarios must work with summary statistics rather than continuous measurements, and the LLM must infer the trajectory shape from these endpoints.

A secondary difference is weather representation. Urban scenarios carry explicit weather fields in the temporal_climatic sub-object (precipitation code, fog, temperature, day period), which are mapped to CARLA weather presets by the urban extractor. Highway scenarios have no weather information in the NPY files; a static Day_Clear weather preset is assigned to all highway scenarios. This is a known limitation: if the original highway recording was made in rain, the generated scenario will not reflect that.

A third difference, less immediately visible but architecturally significant, concerns the statistical structure of the data itself. Empirical inspection of the highway dataset reveals that fields within the same manoeuvre record are not statistically independent: pairs such as initial and final ego speed, or initial and final inter-vehicle gap, exhibit strong linear dependencies reflecting the physical continuity of the recorded event. Similarly, some fields are effectively constant across all instances of a class (e.g. the direction of opposite-direction traffic in certain highway classes), while others exhibit genuine variability that constitutes a legitimate source of scenario diversity (e.g. the DIRECTION field in Lane Change scenarios). This structure, correlated variables, fixed constraints, and true degrees of freedom, is present in both domains but is more directly observable in the highway dataset due to its compact, fixed-schema representation. Figure 4.16 illustrates this point for the LCF manoeuvre class, where the initial and final ego speed fields (EGO_VEL_0 and EGO_VEL_1) exhibit a strong linear dependency across the full dataset, confirming that the two fields encode coupled kinematic structure rather than independent quantities.

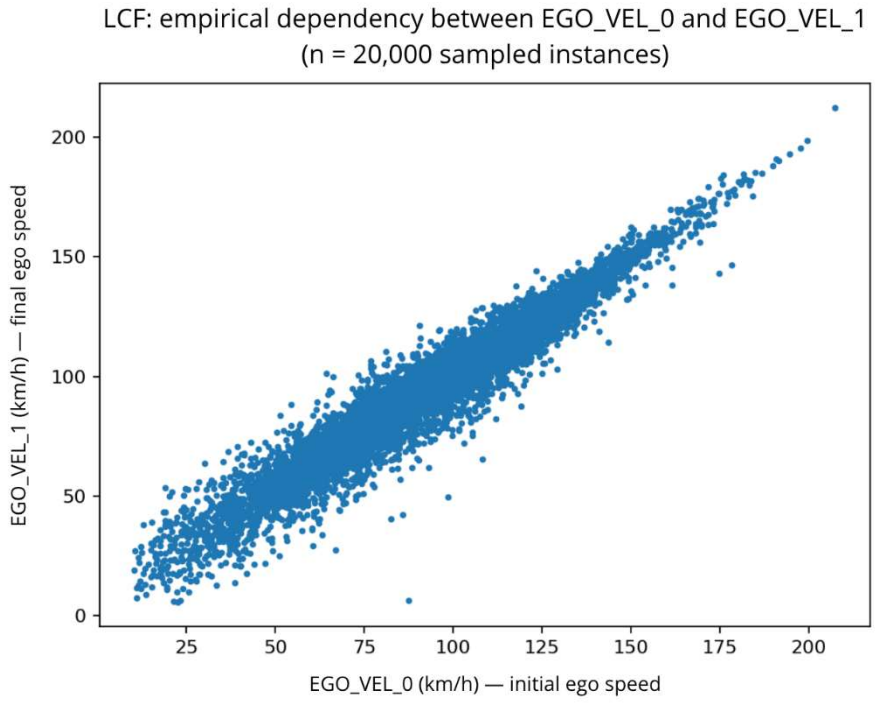

Figure 4.16 - LCF manoeuvre: empirical dependency between initial ego speed (EGO_VEL_0) and final ego speed (EGO_VEL_1) across 20,000 sampled instances.

This observation has a direct consequence for the generation strategy adopted in ScenWeave. Rather than sampling or synthesising individual parameters independently, the pipeline extracts all parameters for a given scenario instance from a single source record and instructs the LLM to substitute them into a complete reference script as a unit. This joint-extraction strategy preserves the empirically observed dependencies between fields: because all extracted parameters originate from the same real driving event, any statistical relationships between them are automatically maintained in the generated scenario, a property that independent parameter sampling would violate.

4.4 Data Canonicalization

The two input domains require distinct canonicalization pipelines, reflecting the different formats and levels of pre-processing of their source data.

The *highway* NPY binary files cannot be used directly by the ScenWeave extraction stage, which expects a structured dictionary with named fields. A canonicalization step, implemented in npy_to_json.py, converts each NPY file into a JSON array of records, one

per scenario instance. This conversion is called in the ScenWeave codebase. Unlike the urban dataset, the highway NPY files arrive already quality-filtered upstream: scenario instances below a minimum duration threshold and instances with implausible kinematic values (e.g. clearly erroneous speeds) were removed prior to delivery, as part of the data preparation conducted for the source PhD thesis. Canonicalization for highway data is therefore a purely structural transformation: it renames and reorganises fields into a named-field JSON representation, without performing any additional quality filtering or semantic interpretation.

The canonicalization process applies the following transformations to each NPY row:

- Field naming: each array index is mapped to a named field using the per-class field schema documented. The resulting dict uses descriptive names (e.g. ego_vel_kmh, npc_distance_m) rather than array indices.
- Sentinel value preservation: sentinel values (−1, −3.6) are preserved in the JSON output. The downstream extractor handles sentinel detection and default substitution, keeping the canonicalization step free of domain logic.
- Sub-class handling: for Follow scenarios, the BEHAVIOUR field (index 9) is extracted and stored as behavior_type. The maneuver_class is set to the base class (FLF or FLB) while the sub-class is encoded separately.
- Scenario ID generation: each record receives a unique scenario_id of the form {maneuver_class}_{binning_version}_{file_stem}_row{row_index}, enabling traceability from a generated scenario back to the source NPY record.

The output JSON schema for each highway record is:

Table 4.9 - JSON schema for highway canonicalized records

Field	Type	Description
schema_version	string	Canonicalisation schema version (e.g. "phase1.v1")
scenario_id	string	Unique identifier, see above
domain	string	"highway", all records are highway domain
maneuver_class	string	MOOVE taxonomy label (e.g. BR_BK, CINF, LCF, FLF)
maneuver_tag	string	Pipeline-internal tag matching corpus script name (e.g. brake, cut_in)
has_npc	bool	Whether the scenario involves an NPC actor
npc_position	string	"front" or "behind", NPC position relative to ego at scenario start
binning_version	string	Identifier of the binning/discretisation pass that produced the source NPY row
source	object	{ file, row_index }, traceability to the originating NPY file and row
params	object	Flat dict of raw kinematic parameters extracted from the NPY record. Field names and units match the NPY file schema (e.g. ego_vel_kmh, lead_vel_kmh, npc_distance_m, duration_s).
behavior_type	string	B1–B4 sub-class label (Follow scenarios only)

Unlike the highway dataset, the urban driving records are received directly from the MOOVE collection pipeline in their raw ADScene format, without any prior quality filtering. No minimum-duration or plausibility filter is applied at any stage of the urban pipeline: every available recorded instance is retained and canonicalized, a deliberate choice driven by the comparatively small number of available urban instances per manoeuvre class. Therefore, the urban canonicalization step performs not only structural reorganisation but

also a substantial amount of semantic interpretation that has no counterpart in the highway pipeline.

The *urban* canonicalization step is implemented in adscene_converter.py, which processes raw per-instance ADScene JSON files (organised in one folder per manoeuvre class) and produces one consolidated JSON file per manoeuvre class, each containing an array of canonicalized scenario records. The conversion applies the following transformations:

- Reference table decoding: all categorical fields encoded as raw numeric codes in the ADScene format (road type, traffic sign type, mobile object classification, precipitation, turning signal state, blinker state, and others) are decoded into human-readable labels using a set of reference tables derived from the ADScene Data Model. This includes a comprehensive traffic sign type table covering road type, weather, and infrastructure codes, used both for the standalone Appendix A.6 enumerations and actively during canonicalisation.
- Actor role classification: each detected obstacle is assigned a semantic role (lead_vehicle, adversary, oncoming, pedestrian, or background) based on its relative position, relative velocity, lateral motion, and turning signal state at scenario start. This classification has no equivalent in the highway pipeline, where the single NPC's role is implicit in the manoeuvre class itself.
- Trajectory construction: both the ego vehicle and each tracked NPC have their full trajectory reconstructed across all sampled frames, expressed as longitudinal and lateral offsets relative to their own position at scenario start (t, ds, dl, relative velocity, and yaw rate where applicable). This preserves frame-level dynamics that are entirely absent from the highway schema, which only retains kinematic endpoints.
- Time-step regularity detection: the sampling interval between consecutive frames is computed and checked for regularity, since urban key_times frames are not guaranteed to be evenly spaced; both the median step and an explicit regularity flag are recorded.

- Criticality metric extraction: where time-to-collision data is available in the source obstacles, a minimum TTC value is extracted and recorded as a criticality indicator for the scenario instance.
- Natural-language description generation: a two-sentence English description of the scenario is automatically generated using a locally deployed LLM (Qwen2.5:14b via Ollama), prompted with the manoeuvre type, domain, weather, ego speed, and NPC summary. This is a second, distinct use of an LLM within the ScenWeave pipeline, separate from the Python code generation role that will be described in the next chapter: here the LLM is used for descriptive metadata synthesis rather than code synthesis.
- Scenario ID generation: each record receives a scenario_id combining the manoeuvre name, a formatted timestamp derived from the source recording, and the source filename, mirroring the traceability function of the highway scenario_id.

The output schema for each urban record is substantially richer than the highway schema, reflecting the larger amount of information available in the source ADScene format and the semantic enrichment performed during conversion:

Table 4.10 - Output schema for urban canonicalized records (adscene_converter.py)

Top-level field	**Description**
scenario_id	Unique identifier combining manoeuvre, timestamp, and source filename
domain	Road type classification derived from infrastructure data (e.g. "country_road", "urban")
version	Schema version string
semantics	{ main_maneuver, description, tags, criticality_metric }; manoeuvre label, LLM-generated description, and TTC-based criticality
metadata	Source file, segment type, recording campaign and provider, duration, time-step regularity, number of sampled frames

environment	Decoded weather, temperature, precipitation, road type, lane count, speed limit, lane width, junction/guard-rail/emergency-lane status
infrastructure	Detected traffic signs with longitudinal position, and a road geometry profile (lane width, curvature, slope) sampled along the segment
initial_state	Ego kinematic state and classified actor list at scenario start, all relative to ego
dynamic_content	Full ego and NPC trajectories across sampled frames, aggregated parametric summary, and an events placeholder for future event-segmentation logic
final_state	Ego and actor state at the end of the segment
perturbation	Noise configuration (dl_noise_std, v_noise_std, t_noise_std) for downstream variant generation, mirroring the urban extractor's extract_with_variants() mechanism

Note: unlike the highway dataset, no quality filter is applied during urban canonicalisation; every recorded instance is retained regardless of duration or kinematic plausibility. This is a deliberate consequence of the limited number of available urban instances per manoeuvre class and is discussed as a limitation.

4.5 Comparison with Public Datasets

The MOOVE dataset used in ScenWeave is a confidential industrial dataset not publicly available. To contextualise its properties and validate that the extracted scenario parameters are representative of real-world driving conditions, it is useful to compare it against publicly available naturalistic driving datasets that have been used in the autonomous driving research community.

Table 4.11 - Comparison of MOOVE with publicly available naturalistic driving datasets

Dataset	Scale	Format	Manoeuvre labels	Relevance to ScenWeave
MOOVE (this work)	~1M km, 15 countries	JSON + NPY	Yes (12 highway and 5 urban classes)	Primary input dataset
nuScenes	1000 scenes, Boston + Singapore	JSON + LIDAR/camera	No (trajectory only)	Comparable urban ego kinematics; no manoeuvre labels
Waymo Open Dataset	1950 scenes, various US cities	TFRecord + LIDAR	No (trajectory only)	Comparable highway speeds and NPC interactions
KITTI [21]	Raw sensor data, Karlsruhe	Binary + images	No	Comparable highway lane-level data; no scenario labels
highD	60,000 vehicles, German highways	CSV	No (trajectory only)	Comparable highway manoeuvres; speed/gap distributions similar to MOOVE highway
inD	11,500 vehicles, urban intersections	CSV	No (trajectory only)	Comparable urban intersection approach behaviour

For *highway scenarios*, the most relevant public comparison dataset is *highD* [17], which was collected on German highway sections using drone cameras and contains over 60,000 vehicle trajectories. The MOOVE highway dataset and highD share comparable speed distributions for highway scenarios: ego speeds in the 100–130 km/h range, inter-vehicle gaps of 10–30 metres in following scenarios, and lane change durations of 3–5 seconds. This overlap validates that the MOOVE parameter distributions are representative of European highway driving conditions.

The *Waymo Open Dataset* [20] provides another point of comparison for NPC interaction distances and relative velocities in following and cut-in scenarios. While the Waymo dataset is US-based and uses different road geometry, the kinematic interaction parameters (time

headway, relative speed at cut-in onset) are comparable to the MOOVE values, suggesting that scenarios generated from MOOVE data would be recognisable as realistic by models trained on other naturalistic datasets.

For *urban scenarios,* the most relevant comparison is the *inD* dataset [18], which contains trajectory data from German urban intersections filmed by drones. The inD dataset covers approach and crossing behaviour at signalised and unsignalised intersections, which is structurally similar to the U4_1–U4_4 scenario classes in MOOVE. Ego approach speeds in MOOVE urban scenarios (approximately 50–70 km/h) are consistent with inD and nuScenes [19] urban intersection data. The traffic light state encoding (red/orange/green → decelerate/maintain speed) is a standard representation shared across all urban intersection datasets.

5 System Design and Implementation

ScenWeave is structured as a sequential six-stage pipeline that transforms a real-world driving scenario record into a validated, executable OpenSCENARIO file ready for CARLA. Each stage has a single well-defined responsibility and communicates with the next through a flat Python dictionary (the params dict) or a file path.

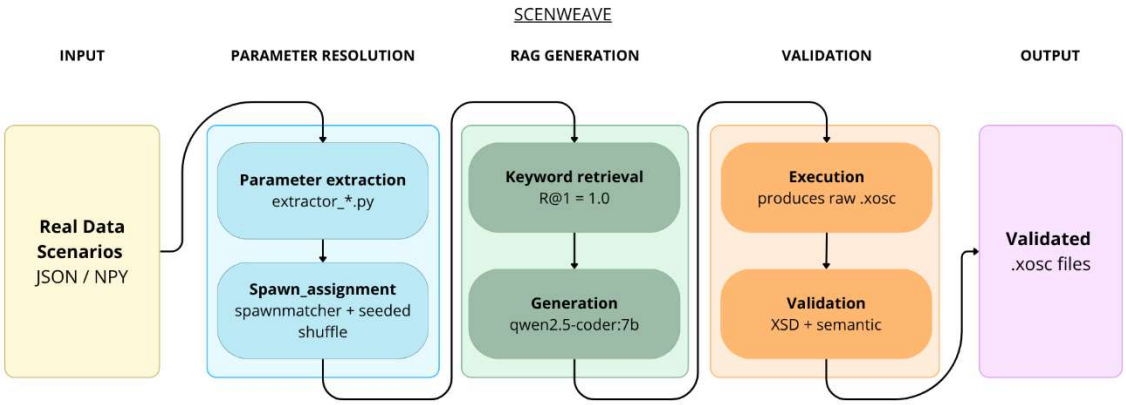

Figure 5.1 - End-to-end architecture of the ScenWeave pipeline organised into three macro-stages.

The pipeline supports two operating modes. In the standard mode, the LLM (qwen2.5-coder:7b via Ollama) generates a new Python script conditioned on the retrieved corpus script and the extracted parameters. In the --no-llm mode, the retrieved corpus script is executed directly with the extracted parameters, bypassing LLM inference entirely. The latter mode is deterministic and guarantees correctness at the cost of diversity. Batch processing is supported through run_batch(), which iterates over all scenario JSON files in the data/real_scenarios/ directory (urban and highway) and writes a batch_summary.json with per-scenario status, spawn_id, and fallback flag. While Figure 5.1 details the internal architecture of the six-stage pipeline, it is also useful to zoom out and view the system end to end: from a raw driving recording to a labelled dataset ready for downstream use. Figure 5.2 provides this bird's-eye view, situating the six stages described above within the broader context of scenario execution in CARLA and final dataset assembly.

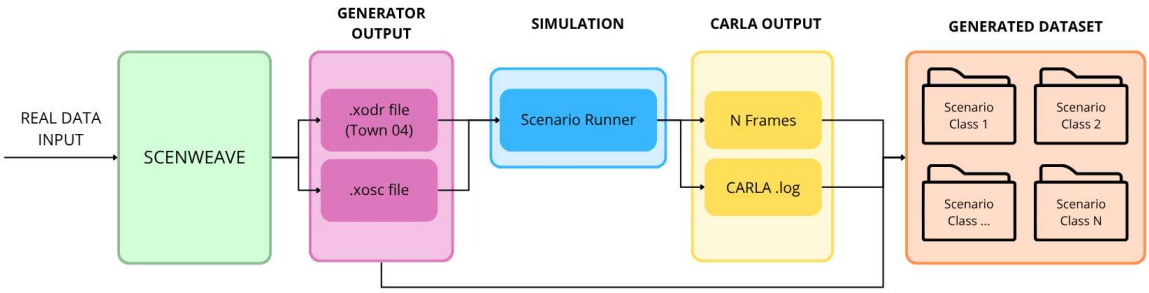

Figure 5.2 - Overview of the ScenWeave system: from a real driving recording to scenario execution in CARLA

5.1 Parameter Extractor

The extraction stage reads a canonical scenario record and produces a flat params dictionary containing all values needed by the downstream stages. Two separate extractor modules handle the structural differences between urban and highway data. The *urban extractor* processes the JSON format produced by the MOOVE pipeline. It reads the ego trajectory, infrastructure signs, environment block, and actor list to produce the following key parameters:

Table 5.1 - Key parameters extracted by the urban extractor (extractor_urban.py)

Parameter	Unit	Description
init_speed_ms	m/s	Ego speed at segment start, converted from km/h
brake_time_s	s	Duration of deceleration phase, computed from ego trajectory (first frame where v ≤ 0.1 m/s)
yellow_delay_s	s	Time for ego to reach ~5 m before TFL, clamped to [1.0, 8.0] s. Used to schedule TFL state change.
drive_distance_m	m	Total distance covered by ego during the scenario
tfl_ds_m	m	Distance from spawn point to traffic light, extracted from infrastructure signs
duration_s / timeout_s	s	Scenario duration and 2.5× generous timeout for CARLA
speed_limit_kmh	km/h	Posted speed limit from environment block
weather_key	,	One of: Day_Clear, Day_Cloudy, Day_Rain, Night_Clear → mapped to CARLA weather params
perturbation_*	,	Gaussian noise std on speed, lateral offset, timing, used for variant generation

The brake_time_s parameter is computed dynamically by scanning the ego trajectory array for the first frame where velocity drops below 0.1 m/s. If the ego never fully stops within the segment, the total trajectory span is used as a conservative estimate. The yellow_delay_s parameter estimates the time for the ego to travel from its spawn point to approximately 5 metres before the traffic light, clamped to [1.0, 8.0] seconds to avoid unrealistic trigger delays. The extractor also supports variant generation through extract_with_variants(), which applies Gaussian noise on initial speed and timing parameters to produce N perturbated copies of the base scenario. This mechanism is the primary source of scenario diversity when the --no-llm mode is used.

The *highway extractor* reads NPY→JSON records produced by npy_to_json.py. It dispatches each record to a maneuver-specific sub-extractor based on the maneuver_class field. All speeds are converted from km/h to m/s; sentinel values (−1.0, −3.6) are mapped to configurable defaults.

Table 5.2 - Extraction logic by highway manoeuvre class (extractor_highway.py)

Maneuver class	**Extractor logic**
BRAKE / BR_BK	ego speed, lead speed, braking duration, delta velocity, NPC distance and intensity. lead_end = max(lead_ms + delta_ms, 0)
CINF / CINB	ego speed, cutter speed (from relative), delta, duration, NPC distance. Sign of distance determines NPC position (front/behind).
COUTF / COUTB	Same structure as CIN. COUTF: ego final speed slightly increased (×1.05). Sentinel values −1/−3.6 map to defaults.
LCF / LCB	Ego initial and final speed, lane change direction, duration. No NPC fields.
FLF / FLB	Ego and lead vehicle initial/final speeds, gap begin/end, distance traveled, behavior type (B1–B4).
FRF / FRB	Ego speeds and distance. Three surrounding vehicle slots (center, right, left): present flag, velocity, longitudinal position. Absent vehicles flagged with −1/−3.6.

All extractors append a shared set of fields to the params dict: scenario_id, maneuver_class, domain, static weather parameters, and spawn coordinate placeholders (ego_x/y/z/h) that are overwritten by the SpawnMatcher in the next stage.

5.2 SpawnMatcher

The SpawnMatcher selects the most suitable ego vehicle spawn point from a script-generated, manually verified database of CARLA Town04 spawn points. It exposes a single public method find(params, top_n) that returns a ranked list of spawn candidates with all coordinates ready for OpenSCENARIO injection.

The database is stored as Town04_Spawnpoints_NEW.json and contains one entry per spawn point. Each entry carries: a transform block (x, y, z, yaw in CARLA coordinates), a surroundings block (speed limit, traffic lights with distance, stop line distance, highway layout flag), a geometry block (road_id, lane_id, s, road length), a features block (is_straight, is_curve, is_intersection, is_intersection_w_traffic_light), a context label (urban or highway), and an annotations block with fields assigned during database construction and subsequent manual verification (suitable_for, priority, manually_reviewed).

The matcher applies a two-phase selection: first, hard filters eliminate incompatible spawn points (score = −1); then, a scoring function ranks the surviving candidates. Separate scoring functions are used for urban and highway domains.

Table 5.3 - SpawnMatcher scoring criteria and filters for urban and highway domains

Domain	Scoring criterion	Points
Urban scoring		
Urban	Context match (urban spawn)	30 (base)
Urban	Speed limit proximity: max(0, 20 − |spawn_limit − scenario_limit|)	0–20
Urban	TFL distance proximity: max(0, 30 − |spawn_tfl_d − tfl_ds_m|)	0–30
Urban	Priority annotation × 10	0–30
Urban	Manually reviewed flag	+10
Urban	Stop line distance present	+15
Urban hard filters (score = −1 if violated)		
Urban	Spawn is on a junction	REJECT
Urban	No traffic lights at spawn	REJECT
Urban	Maneuver not in suitable_for list	REJECT

Highway scoring		
Highway	Context match (highway spawn)	30 (base)
Highway	Speed limit proximity: max(0, 20 − |Δspeed| × 0.5)	0–20
Highway	Straight road feature	+15
Highway	Highway layout annotation	+10
Highway	Space ahead > 100 m	+15
Highway	Priority annotation × 10	0–30
Highway hard filters (score = −1 if violated)		
Highway	Spawn is on a junction	REJECT
Highway	Spawn context ≠ highway	REJECT
Highway	Spawn has intersection or TFL feature	REJECT

Coordinate conversion: CARLA uses a left-handed coordinate system, while OpenSCENARIO uses a right-handed system. The matcher applies the transformation y_osc = −y_carla and heading_osc = −radians(yaw_carla) before returning spawn coordinates. When processing N scenarios of the same maneuver class, the pipeline calls matcher.find(params, top_n=spawn_pool_size) once per scenario and selects the candidate at index scenario_index % len(candidates). With a pool of 5 candidates, scenarios 0–4 use ranks 0–4 respectively, scenario 5 wraps back to rank 0, and so on. This ensures that N executions of the same maneuver type use different spawn points, producing spatial diversity in the generated dataset without requiring manual intervention.

5.3 RAG Corpus, Retrieval and LLM Generation

The corpus consists of 16 active Python scripts, 11 for highway and 5 for urban scenarios, plus a shared utility module corpus_utils.py. Each script implements a generate (params) function that takes the params dict and produces a valid OpenSCENARIO .xosc file using the scenariogeneration library. The corpus covers all the manoeuvre classes:

Each script opens with a docstring that describes the scenario in natural language. This docstring serves as the semantic anchor for retrieval. The script body implements the complete scenario logic using the scenariogeneration API, including ego spawn, NPC spawn (where applicable), traffic light control (urban only), storyboard actions, and timing

conditions. All scripts construct the final xosc. Scenario object through a shared helper, make_scenario(), defined once in corpus_utils.py and imported by every script in the corpus, ensuring a single, consistent construction pattern across the entire corpus rather than per-script boilerplate. Retrieval in ScenWeave is based on keyword matching rather than embedding similarity. The retrieve_keyword() function implements a three-level fallback hierarchy:

Table 5.4 - Keyword retrieval fallback hierarchy in retrieve_keyword()

Method	**Trigger condition**	**Mechanism**
Keyword retrieval (primary)	Always, maneuver class tag available	Exact substring match between maneuver_class and corpus script filename. e.g. "cinf" matches gen_hw_cinf.py. R@1 = 1.0 on all 16 classes (11 highway + 5 urban).
Tag fallback	No exact filename match	MANEUVER_TAGS dict maps maneuver_tag to corpus script. e.g. "cut_in" → gen_hw_cinf.py.
Semantic fallback	No keyword or tag match	Embedding similarity using BAAI/bge-large-en-v1.5. Query = "{maneuver} speed {v}ms distance {d}m". Last resort only.

The primary mechanism achieves R@1 = 1.0 on all 11 highway scenario classes and all 5 urban classes. This is possible because each corpus script has a unique, unambiguous filename that contains the maneuver class tag (e.g. `gen_hw_cinf.py` for cut-in front). Semantic retrieval is loaded lazily and only invoked as a last resort when no keyword match is found, in practice this never occurs on the current corpus. The LLM used for code generation is qwen2.5-coder:7b served locally via Ollama at http://localhost:11434/api/generate. The model was selected on the basis of a systematic benchmark of five code LLMs across three domain specificity levels, reported in Baiardo et al. [5]. The benchmark evaluates Qwen2.5-Coder-7b, DeepSeek-Coder-6.7b, CodeLlama-7b, Qwen2.5-Coder-14b, and CodeLlama-13b under identical RAG conditions on generic Python algorithms (L1), data science libraries (L2), and the scenariogeneration API used in ScenWeave (L3). Qwen2.5-Coder-7b achieves 100% success rate on L1 and L3 and 91% on L2, with the lowest average latency in the 7B cohort (5.5s/task on L1, 6s on L2, 24s on L3).

Crucially, it is the only model that achieves 100% first-attempt success rate on L3, meaning it generates correct code without requiring the self-healing retry loop on all 11 highway manoeuvre classes. The benchmark also establishes that model family is a stronger predictor of RAG code generation success than parameter count, and that 7B variants are Pareto-dominant over their 14B counterparts, matching accuracy at half the latency. The model is queried with temperature = 0.1 and a maximum token budget of 4096 tokens. The low temperature setting reflects a deliberate design choice for the current stage of the pipeline: with a compact, single-reference corpus per manoeuvre class, minimising variability in the generated code structure keeps generation close to a reliable, well-understood pattern. As the corpus is extended with multiple reference variants per manoeuvre class, a higher temperature could be explored to encourage more creative recombination across references, rather than close adherence to a single one.

Note: this first-attempt success figure refers specifically to the comparative benchmark of Baiardo et al., conducted on 11 highway manoeuvre classes with 5 instances each.

The prompt is structured in six sections:

Table 5.5 - Six-section prompt structure used for LLM code generation

Prompt section	**Content**
System instruction	You output only Python code. No explanations. Start directly with import statements.
Task framing	You are given a complete, working Python script for a CARLA {maneuver} scenario.
Constraints	OUTPUT THE ENTIRE SCRIPT BELOW, replacing ONLY the numeric parameter values with those in the JSON. DO NOT simplify, summarize, or stub any function.
Reference script	Full code of the retrieved corpus script (gen_hw_*.py or gen_ur_*.py)
Parameters JSON	Flat JSON of scenario params (speeds, distances, timing). Metadata fields excluded.
Output cue	Output the complete Python script starting with imports

The key design decision in the prompt is the explicit instruction to reproduce the entire reference script, modifying only the numeric parameter values to match the extracted scenario data. This framing gives the LLM a strong structural anchor to reason from: the

model still has to understand what each parameter represents, where it belongs within the script's logic, and how it interacts with the surrounding code, but it does so by adapting a known-correct implementation rather than generating a scenario from an open-ended specification. This substantially reduces the risk of hallucinated API calls while still requiring genuine comprehension of the reference script's structure. The LLM is explicitly instructed not to simplify, stub, or omit any function from the reference script. The raw LLM output is processed by extract_python_code() before execution. This function: (1) strips BOS/EOS tokens and thinking blocks (<think>...</think>); (2) extracts fenced Python code blocks; (3) falls back to heuristic line scanning if no fenced block is found. A truncation warning is emitted if the number of function definitions exceeds the number of return statements.

Before execution, exec_script() applies a set of deterministic corrections to known LLM hallucination patterns:

- Fake module imports (scenario_builder, scenariogeneration.core) → commented out
- Hallucinated API names (xosc.ManeuverGroupSequence → xosc.ManeuverGroup, DynamicsShapes.Sigmoid → DynamicsShapes.sinusoidal)
- Missing enum dot notation (e.g. DynamicsShapeslinear → DynamicsShapes.linear)
- Incorrect xosc.Scenario() constructor calls → reconstructed with correct keyword arguments
- Unicode special characters and box-drawing characters → stripped
- Spurious __main__ blocks → truncated before exec()

When the generated code fails to execute, the pipeline implements a three-attempt self-healing loop before falling back to direct corpus execution:

Table 5.6 - Self-healing retry loop: attempt types and behaviour

Attempt	Prompt type	Behaviour
1	Standard prompt	Full corpus script and params. LLM must output complete generate() function.
2	Error-aware prompt	Previous code and error traceback injected. LLM instructed to reproduce reference script verbatim, replacing only numeric values.
3	Error-aware prompt	Same as attempt 2. If still failing, fallback to direct corpus execution.
Fallback	No LLM	Retrieved corpus script executed directly with extracted params. Guarantees correctness, no diversity.

The error-aware prompt for attempts 2 and 3 injects the full Python traceback from the previous failure and strengthens the instruction to reproduce the reference script verbatim. This mechanism is effective against simple errors (wrong argument count, missing import) but not against systematic API misconceptions, for example, DeepSeek-Coder-6.7b consistently hallucinated a non-existent Priority attribute on LCB scenarios across all three retry attempts, a failure mode that error feedback alone cannot override [5]. A related but distinct failure mode, not specific to any one LLM, was identified during corpus development itself rather than during generation. The five urban scripts originally constructed their final xosc.Scenario object through a direct, multi-argument call to xosc.Scenario(name=..., author=..., parameters=..., entities=..., storyboard=..., roadnetwork=..., catalog=..., osc_minor_version=0), spread across several lines. This construction pattern consistently confused the LLM during generation, producing a systematic indentation error across all five urban scripts. The highway scripts, by contrast, had already adopted the shared make_scenario() helper from corpus_utils.py and never exhibited this failure. The fix consisted of editing each of the five urban scripts individually to call make_scenario() in place of the direct constructor call, aligning the urban side of the corpus with the construction pattern already standard on the highway side. No changes to the retrieval mechanism, the prompt structure, or the validator were required. This episode is discussed further as evidence for RQ3 in Chapter 7.

5.4 XSD Validator and Auto-Fix

The final stage of the pipeline is validate_xosc.py, which validates the generated .xosc file against the ASAM OpenSCENARIO 1.0 XSD schema and applies a set of deterministic fixes for known structural errors produced by the scenariogeneration library or the LLM. The validator first searches for the XSD file in the local ScenarioRunner installation (srunner/openscenario/0.9.x/OpenSCENARIO.xsd). If not found, it falls back to a local cache and finally downloads the schema from the esmini GitHub mirror. This ensures the validator works both in the development environment and in offline deployments.

Five deterministic fix rules are applied in sequence before XSD validation:

Table 5.7 - Deterministic auto-fix rules applied before XSD validation

Fix name	**Description**
precipitationIntensity → intensity	scenariogeneration emits precipitationIntensity= but the target schema requires intensity=. Replaced via string substitution.
Missing <TimeOfDay>	The schema requires TimeOfDay inside Environment. Default inserted (2020-01-01 12:00:00) if absent.
Missing <Precipitation>	The schema requires Precipitation inside Weather. Default dry/0.0 inserted if absent.
Spurious </TrafficSignalStateAction>	Orphaned closing tags generated by some LLM outputs. Removed via regex.
delay= expression → 0	Arithmetic expressions (e.g. delay="t+2") and $parameter references in Condition.delay are reset to 0. The schema does not allow expressions.

In addition to XSD validation, the validator performs domain-aware structural checks and semantic consistency checks. Structural checks verify the presence of mandatory XML elements; semantic checks verify that the generated scenario structure is consistent with the expected behaviour of the maneuver class (e.g. a BRAKE scenario must have an NPC and no lane change action).

Table 5.8 - Domain-aware structural and semantic checks performed by the XSD validator

Check	Domain	Condition
NO_ROOT	Both	Missing <OpenSCENARIO> root element
NO_STORY	Both	Missing <Story> block
NO_TELEPORT	Both	Missing TeleportAction, ego not spawned
NO_MANEUVER	Both	Missing ManeuverGroup
NO_TRIGGER	Both	Missing StartTrigger
NO_WORLDPOS	Urban	Missing WorldPosition for ego spawn
NO_TFL_ACTION	Urban	Missing TrafficSignalStateAction
ZERO_SPAWN	Urban	WorldPosition at origin (0,0)
DEFAULT_SPAWN	Urban	WorldPosition matches known template default coordinates
NO_LANEPOS	Highway	Missing LanePosition for ego spawn
NO_SPEED_ACTION	Highway	Missing AbsoluteSpeedAction
NPC_MISSING	Highway	Multiple ManeuverGroups but only one ScenarioObject
ZERO_SPEED	Both	InitSpeed or CruiseSpeed ≤ 0
DELAY_PARAM / DELAY_EXPR	Both	Unresolved parameter or expression in delay= attribute

Domain detection is automatic: if the XML contains a <LanePosition> element for ego spawn, the scenario is classified as highway; otherwise as urban. This distinction drives the application of domain-specific checks (WorldPosition and TrafficSignalStateAction for urban; LanePosition and AbsoluteSpeedAction for highway). The validator returns a structured result dict with four fields: valid (bool), fixes_applied (list of applied fix names), errors (list of structural/semantic issues with severity and code), and domain (detected domain). The pipeline logs the validation result and marks the scenario as "invalid_xml" if any ERROR-severity issue is found after auto-fix.

6 Experiments and Evaluation

This chapter presents the experimental evaluation of the ScenWeave pipeline. It describes the experimental setup for the comparative LLM benchmark and defines the evaluation metrics used throughout the chapter, before reporting the comparative benchmark results across five LLMs and 11 highway manoeuvre classes, used to select the production model. It then analyses the observed failure modes and discusses the comparative performance of the evaluated models, reports the spawn point diversification behaviour observed during batch generation, and closes with a separate, larger-scale validation run covering both highway and urban domains with the selected production model, Qwen2.5-Coder-7b.

6.1 Experimental Setup and Evaluation Metrics

The comparative benchmark reported in this chapter constitutes the L3 (scenariogeneration API) component of the broader three-level benchmark subsequently reported in Baiardo et al. [5], which additionally evaluates the same five models on generic Python algorithms (L1) and data science libraries (L2). This chapter presents the L3 results with the additional detail relevant to production model selection for ScenWeave (per-class latency, error taxonomy, spawn diversification), while [5] focuses on the comparison across specificity levels. All experiments were run on a single machine to ensure full reproducibility and eliminate hardware variance as a confounding factor. The configuration is summarised below.

Table 6.1 - Experimental setup for the comparative LLM benchmark

Component	Configuration
GPU	NVIDIA RTX A4000, 16 GB VRAM
Server	eliosdell, Ubuntu 24.04, kernel 6.8.0
Simulator	CARLA 0.9.14
Scenario runner	ScenarioRunner (CARLA 0.9.14-compatible build)
LLM inference	Ollama, local HTTP API (localhost:11434)
Embedding model (fallback only)	BAAI/bge-large-en-v1.5
Scenario library	scenariogeneration (Python)

Models evaluated	Qwen2.5-Coder-7b, Qwen2.5-Coder-14b, DeepSeek-Coder-6.7b, CodeLlama-7b, CodeLlama-13b
Benchmark corpus	11 highway manoeuvre classes, 5 scenario instances per class per model (55 generations per model)
Self-healing budget	3 attempts per scenario; error traceback injected on retry
Temperature	0.1 (all models)

Each of the five evaluated models was run on the complete set of 11 highway manoeuvre classes, with 5 scenario instances per class, for a total of 55 generation attempts per model (275 total generation attempts across all models). For each scenario, the model received the structured six-section prompt described, containing the retrieved corpus script and the extracted real-data parameters for that specific instance.

Four metrics are used to characterise model performance:

- Success rate (SR): The fraction of scenarios for which the pipeline produces a structurally valid, executable .xosc file within the 3-attempt self-healing budget. A scenario is counted as successful if any of the three attempts succeeds.
- First-attempt success rate (1st%): The fraction of scenarios solved correctly on the first attempt, without requiring the self-healing retry loop. This metric is more informative than overall SR because it is not inflated by the retry mechanism; it reflects the model's raw ability to adapt the retrieved reference script to new parameters.
- Average latency (s/task): Wall-clock time per scenario, including LLM inference, code extraction, execution, and XSD validation. Reported as the average over the 5 instances of each manoeuvre class.
- Error taxonomy: Failed generation attempts are classified into four categories: syntax (Python syntax errors), api_hallucination (calls to non-existent methods or attributes), name (undefined name references), and import (missing or incorrect module imports).

Correctness is verified by the structural and semantic XML validation described.The scenario is considered correct if it passes XSD schema validation and all domain-aware structural checks with no ERROR-severity issues after auto-fix.

6.2 Benchmark Results

Four of the five evaluated models, Qwen2.5-Coder-7b, Qwen2.5-Coder-14b, CodeLlama-7b, and CodeLlama-13b, achieve a 100% success rate and a 100% first-attempt success rate across all 11 highway manoeuvre classes, meaning the self-healing retry loop is never invoked for these models. DeepSeek-Coder-6.7b is the only model that falls short of perfect reliability: its overall success rate is 93%, with failures concentrated on two classes, LCB (40% SR) and COUTF (80% SR), while all other classes reach 100%. Its first-attempt success rate is lower still, at 75% on average, while BRAKE, CINB, LCF, and FRB are solved correctly on the first try in every instance, the model requires retries substantially more often on CINF (40%), LCB (40%), and FLF (20%), and moderately more often on COUTF, COUTB, FLB, and FRF (80% each). Table 6.2 reports average generation latency for each model and class. Table 6.3 summarises the total error count by category for the models that exhibited non-zero error rates.

Table 6.2 - Average Generation Latency (s/scenario) by Model and Scenario Class

Scenario	DSCoder 6.7b	Qwen Coder:7b	CL 7b	Qwen Coder:14b	CL 13b
BRAKE	25s	26s	**22s**	51s	42s
CINF	49s	24s	**21s**	47s	40s
CINB	25s	25s	**22s**	48s	40s
COUTF	43s	27s	**25s**	53s	46s
COUTB	46s	23s	**21s**	46s	39s
LCF	22s	28s	**27s**	55s	49s
LCB	56s	27s	**27s**	54s	49s
FLF	50s	25s	**23s**	49s	42s
FLB	30s	24s	**22s**	46s	40s
FRF	22s	18s	**15s**	36s	28s
FRB	20s	19s	**15s**	39s	28s

Between the two models with equally clean records, Qwen2.5-Coder-7b and CodeLlama-7b, model selection favours first-attempt reliability over raw speed: CodeLlama-7b is marginally faster per scenario, but Qwen2.5-Coder-7b's 100% first-attempt success rate, confirmed independently in the cross-domain benchmark of Baiardo et al. [5], makes it the

more robust choice for unattended, large-scale generation, where consistency matters more than a few seconds per scenario.

Table 6.3, Error Taxonomy (total errors across 55 generations per model)

Error type	**DSCoder 6.7b**	**QwenCoder 7b / 14b**	**CodeLlama 7b / 13b**
syntax	11	0	0
api_hallucination	10	0	0
name	1	0	0
Total errors (55 generations)	22	0	0

This reliability is reflected in Qwen2.5-Coder-7b's latency profile, which is the lowest among all models that achieve non-zero success rate, ranging from 18s (FRF) to 28s (LCF) per scenario.

6.3 Failure Mode Analysis

DeepSeek-Coder-6.7b is the only model among the five evaluated candidates that does not achieve 100% SR. Its 22 total errors (11 syntax, 10 api_hallucination, 1 name) are concentrated on two scenario classes. On LCB (40% SR), the model consistently hallucinates a non-existent Priority attribute on the maneuver object across multiple retry attempts. On COUTF (80% SR), one syntax error and two API hallucinations account for the single failure. Across both classes, the failure mode is systematic rather than random: the same incorrect API surface is reproduced across retry attempts, indicating that the error-aware retry prompt is insufficient to override a confidently memorised but incorrect API pattern. The benchmark results support a clear conclusion regarding model selection strategy: model family is a stronger predictor of success on this domain-specific API than parameter count. Both Qwen2.5-Coder variants (7b and 14b) and both CodeLlama variants (7b and 13b) achieve 100% SR, while scaling from 7B to 14B parameters within the same family produces no improvement in success rate, only a near-doubling of latency (e.g. Qwen2.5-Coder-7b: 18–28s/task vs Qwen2.5-Coder-14b: 36–55s/task). This makes the smaller variants of each successful family Pareto-dominant over their larger counterparts for this task: equal accuracy at substantially lower computational cost. This finding is consistent with and further substantiated by the dedicated three-level benchmark reported in Baiardo et

al. [5], which evaluates the same model families across generic Python (L1), data science libraries (L2), and the scenariogeneration API (L3) under identical RAG conditions. That benchmark confirms Qwen2.5-Coder-7b as the model with the best overall efficiency-accuracy trade-off across all three specificity levels, motivating its selection as the default model for the ScenWeave production pipeline.

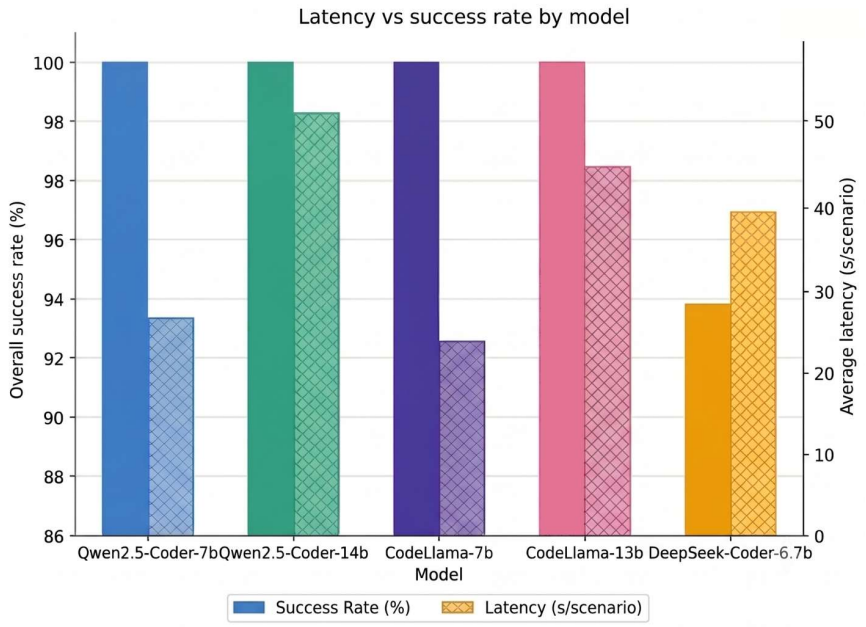

Figure 6.1 - Latency vs overall success rate for the five evaluated models.

Based on the combined evidence from this chapter and the dedicated LLM benchmark [5], Qwen2.5-Coder-7b is confirmed as the optimal choice for the ScenWeave production pipeline: 100% SR and 100% first-attempt SR across all 11 highway classes, the lowest latency among models with non-zero success rate, and no observed hallucination or syntax errors in 55 generation attempts. CodeLlama-7b is a viable alternative with comparable accuracy and latency, while DeepSeek-Coder-6.7b, despite its fast latency on successful classes, exhibits class-specific failure modes that would require additional corpus engineering or prompt refinement to fully resolve.

6.4 Spawn Point Diversification

The round-robin spawn diversification mechanism, implemented in assign_spawn(), was observed during batch generation across all 11 highway manoeuvre classes. For every manoeuvre class, the five generated scenario instances are assigned the same fixed sequence

of spawn points: spawn 298 (instance 1), spawn 300 (instance 2), spawn 26 (instance 3), spawn 27 (instance 4), and spawn 28 (instance 5). This result confirms that the round-robin mechanism successfully diversifies spawn point assignment within a manoeuvre class: five executions of the same scenario class use five distinct spawn points, avoiding the degenerate case where every instance of a class spawns at an identical location. However, the same fixed sequence is observed across different manoeuvre classes (BRAKE, CINF, LCF, and all other highway classes produce the identical spawn sequence), indicating that the spawn ranking itself does not vary by manoeuvre type. This behaviour is a direct consequence of the highway scoring function, which does not include any manoeuvre-specific scoring criterion: a spawn point is scored on context match, speed limit proximity, road straightness, and available space ahead, none of which depend on whether the scenario being generated is a BRAKE, a CINF, or an FLF. As a result, the top-N ranked candidates are identical across all highway manoeuvre classes, and the round-robin assignment produces the same sequence regardless of manoeuvre type. This is not considered a defect of the current implementation. On a straight highway segment, any spawn point satisfying the highway hard filters (non-junction, correct context, no intersection features) is kinematically suitable for any of the 11 manoeuvre classes, because scenario-specific behaviour is encoded entirely in the extracted parameters (speed, NPC distance, duration) rather than in the spawn point geometry itself. Manoeuvre-specific spawn diversification would only become necessary if the highway corpus were extended with scenarios that depend on specific road features not captured by the current scoring function, for example, manoeuvres requiring a curve, an on-ramp, or a multi-lane merge point. This is noted as a direction for future work, to be addressed if and when the corpus is extended with road-feature-dependent manoeuvre classes.

6.5 Full-Scale Results

While the previous benchmark compares five candidate LLMs on a controlled set of 11 highway manoeuvre classes with 5 instances each, it does not by itself demonstrate that the selected production model performs reliably at the scale and domain coverage required for actual dataset generation. This section reports a separate, larger-scale validation run of the production pipeline, using only the selected production model (Qwen2.5-Coder-7b) across both the highway and urban domains, on a substantially larger number of real scenario instances per class. The production run processed 678 real scenario instances drawn from

the MOOVE dataset, covering 12 highway manoeuvre tags, the 11 corpus-mapped classes plus a BR_BK revalidation of the BRAKE class, which shares the same extraction logic and corpus script, with 50 instances each, and all 5 urban manoeuvre classes (instance count per class limited by data availability). Unlike the comparative benchmark, instance counts per highway class are uniform (50 each), while urban instance counts vary according to the natural frequency of each manoeuvre type in the MOOVE collection, from 4 instances (turn_right_at_tfl_green) to 41 instances (pass_tfl_green_or_orange). The BRAKE class was validated twice during this run: once under its original maneuver tag (brake) and once after a data revision (br_bk), sharing the same extraction logic and corpus script. Both are treated as separate manoeuvre tags in this production-scale validation.

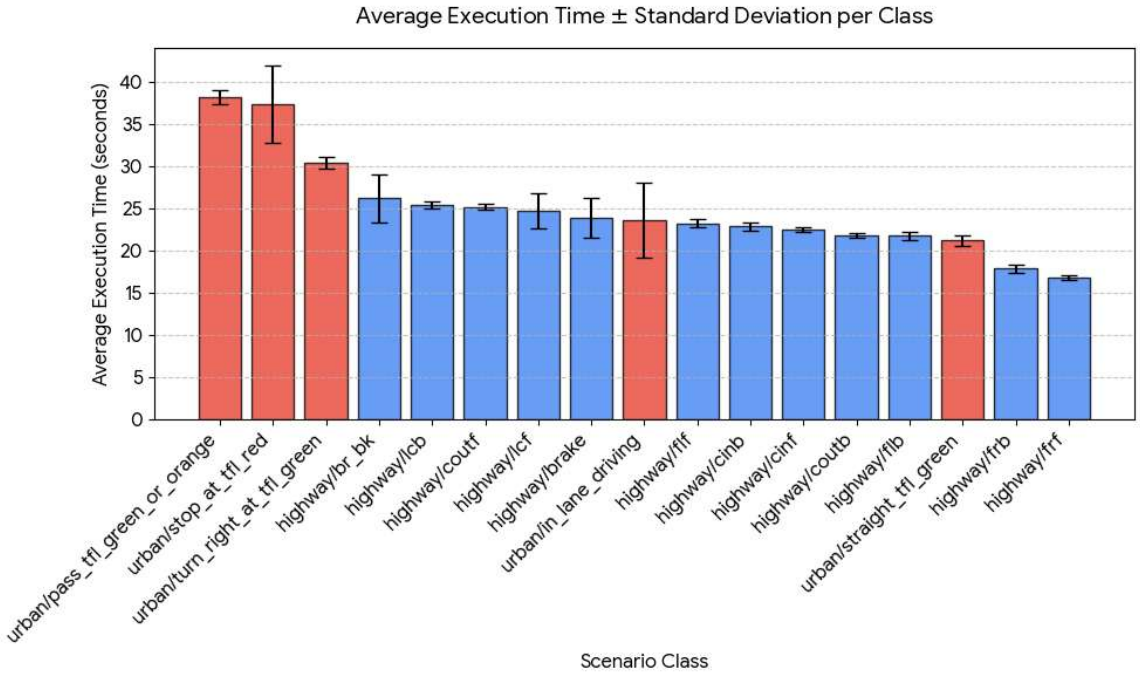

Figure 6.2 - Average generation latency (± standard deviation) per manoeuvre class across the production validation run.

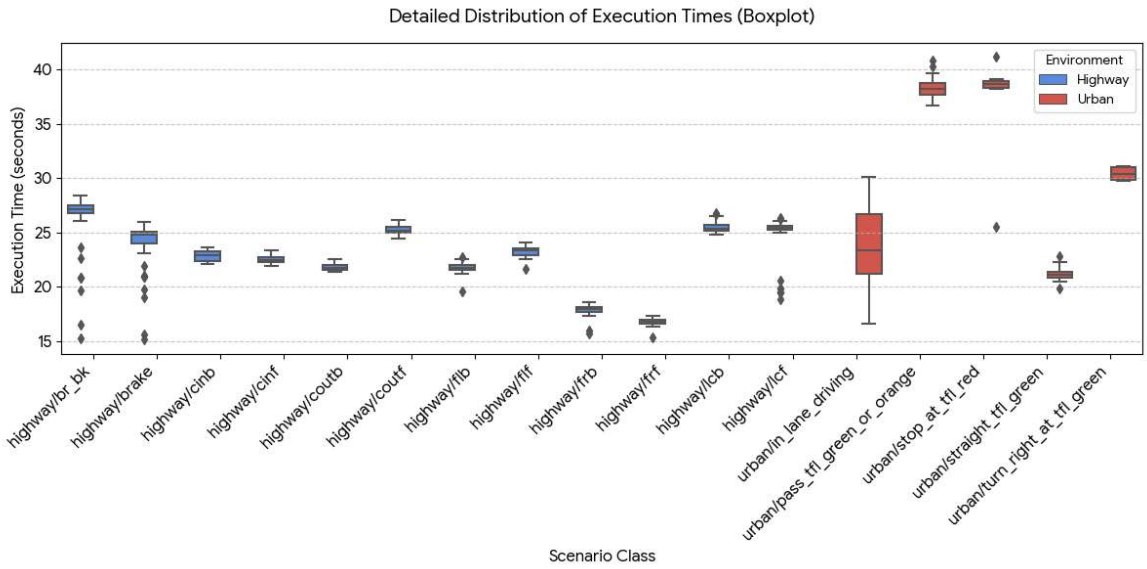

Figure 6.3 - Detailed distribution of execution times per manoeuvre class (boxplot), complementing the mean ± standard deviation summary in Figure 6.2

The full distribution, shown in Figure 6.3, confirms that execution time variance is low for highway classes and higher for the two most complex urban classes. Across all 678 scenario instances, the pipeline achieved a 100% success rate within the 3-attempt self-healing budget, with avg_attempts = 1 recorded for every manoeuvre class, indicating that the self-healing retry loop was never invoked: every scenario succeeded on its first generation attempt. This is consistent with the first-attempt success behaviour already observed for Qwen2.5-Coder-7b in the comparative benchmark and confirms it at substantially larger scale and across an additional domain not covered by that benchmark. Highway scenarios show 100% semantic correctness in addition to 100% structural success, confirming that the semantic consistency checks are satisfied throughout production-scale generation, not only on the smaller benchmark sample. Urban scenarios are reported with structural and XSD validation only, since the domain-aware semantic checks have not yet been extended to the urban domain. Generation latency for highway classes (16.8–26.2 s/scenario) is consistent with the range already observed in the comparative benchmark. Urban classes show generally higher latency (21.2–38.3 s/scenario), with pass_tfl_green_or_orange and stop_at_tfl_red notably slower (~37–38 s) than the other urban classes; this is consistent with these scenarios requiring more complex storyboard logic involving traffic light state transitions, compared to the simpler in_lane_driving and straight_tfl_green classes. This production-scale run provides direct evidence for RQ1: it demonstrates that the RAG and

LLM pipeline, using the production model selected via the comparative benchmark, generates valid and semantically correct OpenSCENARIO files at scale, across both domains, and not merely on the smaller controlled sample used for model comparison. The absence of any fallback to direct corpus execution (n_fallback = 0 for every class) further indicates that the LLM-based generation path, rather than the deterministic fallback, was responsible for all 678 successful scenarios. The unbalanced urban instance counts and the absence of urban semantic correctness checking remain acknowledged limitations of this validation. In addition to per-class success rate and latency, the practical throughput of the production pipeline is relevant for planning large-scale dataset generation campaigns. Figure 6.4 reports the cumulative number of scenarios generated over one hour of continuous, sequential execution, computed from the actual per-scenario generation times recorded during the production validation run, under three execution scenarios: a highway-only run, an urban-only run, and the actual mix of classes processed in this production run (highway classes followed by urban classes). In all three cases, execution is sequential, with no parallelisation across scenarios. Under continuous highway-only generation, the pipeline produces approximately 159 scenarios per hour, reflecting the comparatively lower average latency of highway classes (16.8–25.5 s/scenario). Under continuous urban-only generation, throughput drops to approximately 110 scenarios per hour, driven by the higher and more variable latency of urban classes, particularly pass_tfl_green_or_orange and stop_at_tfl_red (~37–38 s/scenario), which involve more complex traffic-light storyboard logic. The actual production run, which processes highway classes followed by urban classes in sequence, yields an intermediate throughput of approximately 151 scenarios per hour, consistent with its mix of the two domains. These figures illustrate that hourly throughput for a planned generation campaign depends directly on its class composition: a highway-heavy campaign approaches the upper bound of roughly 159 scenarios per hour, while a campaign weighted towards traffic-light urban classes approaches the lower bound of roughly 110 scenarios per hour. This throughput figure is reported for a single, non-parallelised pipeline instance. ScenWeave's batch processing via run_batch() currently executes scenarios sequentially; running multiple pipeline instances in parallel, each with its own Ollama inference process, would scale throughput roughly linearly with available GPU resources, though this was not evaluated in this thesis.

7 Discussion

7.1 Answers to the Research Questions

This section revisits the three research questions and answers each one directly, drawing on the experimental results presented in Chapter 6.

> **RQ1:** Is it possible to generate valid and semantically correct OpenSCENARIO (XML) files using a RAG and LLM system?

Yes, demonstrated at production scale. The production-scale validation run shows that the pipeline, using the selected production model (Qwen2.5-Coder-7b), achieves a 100% success rate across 678 real scenario instances spanning all 17 manoeuvre classes (12 highway, including the BR_BK/BRAKE pair sharing one corpus script, and 5 urban), with 100% semantic correctness on the highway domain, where the semantic checks described are currently implemented. Every scenario succeeded on its first generation attempt (avg_attempts = 1 throughout), with no reliance on the self-healing retry loop or fallback to direct corpus execution. This result holds across both domains and at a substantially larger scale than the comparative benchmark, providing direct evidence that the architecture generalises beyond a small controlled sample. The answer remains conditional on model choice: the comparative benchmark shows that DeepSeek-Coder-6.7b, while broadly competent, falls to 93% overall with class-specific failures, confirming that reliable, manual-intervention-free generation depends on selecting a model capable of structured, instruction-following code adaptation, not on the RAG architecture alone.

> **RQ2:** Among the LLMs that can be run locally via Ollama, which one offers the best trade-off between reliability and efficiency for this task?

Qwen2.5-Coder-7b. The comparative benchmark shows it is the only model achieving 100% first-attempt success rate across all 11 highway manoeuvre classes, meaning the self-healing retry loop is never invoked. Its latency (18–28 s/scenario) is the lowest among all models with non-zero success rate, and it matches the accuracy of its 14b variant at roughly half the latency, making the smaller variant Pareto-dominant. This finding is corroborated by the

dedicated cross-domain benchmark of Baiardo et al. [5], which establishes Qwen2.5-Coder-7b as the model with the best overall efficiency-accuracy trade-off across three domain specificity levels, not only the scenariogeneration API used in ScenWeave. CodeLlama-7b is a close second, with comparable accuracy and latency; DeepSeek-Coder-6.7b, despite competitive latency, exhibits systematic class-specific API hallucination that would require additional corpus engineering to resolve.

> **RQ3:** Is the system generalisable to new manoeuvre classes or new corpus entries without requiring structural redesign of the pipeline?

Yes, with direct evidence from a concrete extension episode encountered during development. The five urban corpus scripts originally failed systematically due to a construction pattern incompatible with reliable LLM generation; the fix consisted of editing each script individually to align it with the construction pattern already standard on the highway side, without any modification to the retrieval mechanism, the prompt structure, or the validator. This demonstrates that extending or correcting the corpus is a localised, per-script change rather than a structural one. This is further supported by the architectural analysis: adding a manoeuvre class from an entirely different taxonomy requires only a new extraction function, one corpus script, and an entry in the retrieval mapping, with no changes to the RAG retrieval, generation, or validation stages.

7.2 Limitations

This section introduces the high-level limitations of ScenWeave.

While the highway domain was validated on a uniform 50 instances per manoeuvre class, the urban domain reflects the natural scarcity of certain manoeuvre types in the MOOVE collection, with instance counts ranging from 4 (turn_right_at_tfl_green) to 41 (pass_tfl_green_or_orange). No quality filtering is applied during urban canonicalization, since doing so would further reduce the already limited number of available instances for the scarcest classes. While the production run achieved 100% success rate across all urban classes, results for the most data-scarce classes rest on a small number of instances and should be interpreted with appropriate caution. A further consequence of this asymmetry is the absence of semantic correctness checking for the urban domain: unlike highway

scenarios, which are validated against both structural and semantic checks, urban scenarios are currently verified through structural and XSD validation only. Extending domain-aware semantic checks to the urban domain and collecting additional urban instances for the scarcest manoeuvre classes, are identified as priority items for future work.

As noted in the RQ3 answer, this thesis does not report a controlled comparison between LLM-generated scenarios and the deterministic --no-llm fallback on the same set of input parameters. Such a comparison would quantify the added value of LLM-based generation, in terms of structural diversity or parameter fidelity, relative to its computational cost (18–55s/scenario vs near-instantaneous fallback execution). This comparison is methodologically straightforward to perform with the existing pipeline and is recommended as a near-term follow-up experiment.

The highway spawn scoring function does not currently differentiate between manoeuvre classes. This is not a limitation for the current 11-class highway corpus, where all manoeuvres are suitable for any straight, non-junction highway spawn point. It would become a limitation if the corpus were extended with manoeuvres that require specific road features (e.g. a curve for a high-speed lane change, an on ramp for a merge scenario), which the current scoring function does not capture.

Each manoeuvre class in the RAG corpus is represented by exactly one reference script. As discussed in the architectural rationale for the retrieval mechanism, this is sufficient for unambiguous keyword retrieval (R@1 = 1.0) but means the LLM has only one structural reference to adapt from per class. If multiple variants of the same manoeuvre were added to the corpus (e.g. BRAKE in different weather conditions or with multiple NPCs), the current keyword retrieval would need to be extended with a disambiguation mechanism to select the most appropriate variant.

The pipeline requires a GPU with at least 8 GB VRAM for Qwen2.5-Coder-7b inference via Ollama. This limits deployment to machines with compatible hardware.

The current corpus scripts generate scenarios with one ego vehicle and at most one explicitly scripted NPC (the adversary or lead vehicle defined by the manoeuvre class); multi-actor scenarios with complex interactions between multiple scripted NPCs are not supported at the .xosc generation level.

The production pipeline uses a single LLM (Qwen2.5-Coder-7b) as the default. While the benchmark in Chapter 6 and in Baiardo et al. [5] evaluates five models, the pipeline does not implement automatic model selection or ensembling across models based on manoeuvre class. Given that DeepSeek-Coder-6.7b shows class-specific failures while otherwise being competitive in latency, a class-aware model routing strategy could in principle improve overall efficiency, but this was not implemented or evaluated in this thesis.

7.3 Generalisability

The SpawnMatcher database is specific to CARLA Town04. Extending ScenWeave to another map (e.g. Town10 or Town06) would require constructing a new spawn point database with the same schema (transform, surroundings, geometry, features, context, annotations). The scoring functions are map-agnostic and would not require modification, since they operate on the annotated features rather than hard-coded coordinates. The verification effort, particularly confirming suitable_for labels for urban spawn points, is the main bottleneck for extending map coverage, as this step currently requires domain expertise and cannot be fully automated within the current pipeline design. The 12 highway and 5 urban manoeuvre classes used in ScenWeave are defined by the MOOVE project taxonomy. The pipeline architecture itself does not depend on this specific taxonomy: adding a new manoeuvre class from a different taxonomy requires (a) an extraction function that maps the new data format to the flat params dict structure, (b) one corpus script implementing the manoeuvre using the scenariogeneration API, and (c) an entry in the MANEUVER_TAGS mapping for retrieval. No changes to the RAG retrieval mechanism, the LLM generation stage, or the validator are required, demonstrating that the architecture generalises beyond the specific MOOVE taxonomy used in this thesis. The benchmark in Baiardo et al. [5] provides direct evidence on this question by evaluating the same RAG architecture, prompt structure, and self-healing loop on two additional domain specificity levels: generic Python algorithms (L1) and data science library code (L2). The finding that four out of five models achieve 100% SR on L3 (`scenariogeneration`), comparable to or better than their L1/L2 performance, suggests that the ScenWeave RAG methodology is not specific to the scenario generation domain and could be applied to other domain-specific Python APIs, provided a structured reference corpus of similar quality can be constructed.

8 Future Work

The limitations identified and the generalisability analysis point to a set of future extensions, ranging from near-term engineering work to longer-term research directions.

In the short term, three priorities stand out. Extending the domain-aware semantic checks to the urban domain would close the validation gap that currently leaves urban scenarios checked only structurally. Completing the Traffic Manager integration for background traffic, already under development, would allow classes such as BR_BK to reproduce the surrounding-traffic context that currently defines them only at the taxonomy level. Beyond these, generating scenarios at volume and evaluating them with an existing manoeuvre recognition model would provide the first genuine end-to-end quality signal, closing the loop between scenario generation and the recognition systems it is meant to train.

In the medium term, the priority is breadth rather than depth: extending ScenWeave to additional CARLA maps beyond Town04 requires only a new spawn point database, since the scoring functions are already map-agnostic ; and enriching the corpus with multiple reference variants per manoeuvre class, paired with a disambiguation mechanism at retrieval time, would allow greater scenario diversity and could support class-aware routing across models where one shows systematic weaknesses, as observed with DeepSeek-Coder-6.7b .

In the longer term, the most ambitious direction is generating CARLA-compatible maps that reproduce the actual road geometry recorded in the MOOVE data, allowing scenarios to be replayed in their original spatial context rather than adapted to Town04. This remains an open research problem, requiring a conversion pipeline from real-world road geometry to OpenDRIVE format. The long-term vision motivating this thesis is a closed loop in which real driving data is used to generate synthetic training scenarios, which in turn are used to train manoeuvre recognition models, whose performance gaps can then inform the collection or generation of additional targeted scenarios. ScenWeave addresses the translation step at the core of this loop, the automatic conversion from real driving records to executable simulation scenarios, and establishes that this step can be performed reliably with a local, privacy-preserving LLM pipeline. The remaining directions for closing the loop fully, including large-scale generation and validation with an existing recognition model and the

development of higher-fidelity scenario representations, are identified as the near- and medium-term extensions.

The central finding of this work is that a carefully designed RAG architecture, a small, well-curated corpus, a structured prompt that gives the model a concrete reference implementation to reason from and adapt, and a self-healing retry loop, can compensate for the absence of a domain-specific API in an LLM's training data to the point where a 7-billion-parameter local model matches the reliability expected of much larger systems. This result, validated both within ScenWeave and in the dedicated cross-domain benchmark of Baiardo et al. [5], suggests that the approach generalises beyond driving scenario generation to other domains where a structured code corpus can be assembled and a local, privacy-preserving deployment is preferred over a cloud-based alternative.

Bibliography

[1] Chen, M., Tworek, J., Jun, H., et al.: Evaluating Large Language Models Trained on Code. arXiv:2107.03374 (2021)

[2] Austin, J., Odena, A., Nye, M., et al.: Program Synthesis with Large Language Models. arXiv:2108.07732 (2021)

[3] Lewis, P., Perez, E., Piktus, A., et al.: Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In: Advances in Neural Information Processing Systems (NeurIPS), pp. 9459–9474 (2020)

[4] Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., Koltun, V.: CARLA: An Open Urban Driving Simulator. In: Conference on Robot Learning (CoRL), PMLR 78, pp. 1–16 (2017)

[5] Baiardo, G., Bellotti, F., Benvenuto, L.,Pighetti, A., Berta, R.: LLM-Assisted Software Development for Specialized Engineering Domains: The Role of Retrieval Across Domain Specificity Levels. Submitted to ApplePies 2026 (2026)

[6] Lai, Y., Li, C., Wang, Y., et al.: DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation. In: International Conference on Machine Learning (ICML) (2023)

[7] Wang, Y., Le, H., Gotmare, A. D., et al.: CodeT5+: Open Code Large Language Models for Code Understanding and Generation. In: Empirical Methods in Natural Language Processing (EMNLP) (2023)

[8] Zhang, F., Chen, B., Zhang, Y., et al.: RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. In: Empirical Methods in Natural Language Processing (EMNLP) (2023)

[9] Zan, D., Chen, B., Zhang, F., et al.: Large Language Models Meet NL2Code: A Survey. In: Association for Computational Linguistics (ACL), pp. 7443–7464 (2023).

[10] Scholtes, M., Westhofen, L., Turner, L. R., et al.: D2-City: A Large-Scale Dashcam Video Dataset of Diverse Traffic Scenarios. arXiv:1904.01975 (2019)

[11] Suo, X., Zhao, H., Tao, F., et al.: TrafficGen: Learning to Generate Diverse and Realistic Traffic Scenarios. In: IEEE International Conference on Robotics and Automation (ICRA), pp. 3923–3929 (2023)

[12] Fremont, D. J., Dreossi, T., Ghosh, S., Yue, X., Sangiovanni-Vincentelli, A. L., Seshia, S. A.: Scenic: A Language for Scenario Specification and Scene Generation. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (2019)

[13] Zhang, J., Xu, C., Li, B.: ChatScene: Knowledge-Enabled Safety-Critical Scenario Generation for Autonomous Vehicles. In: IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) (2024)

[14] Sheng, H., et al.: Talk2Traffic: Multimodal Scenario Generation for Autonomous Driving Simulation. In: CVPR Workshops (CVPRW) (2025)

[15] ASAM e.V.: OpenSCENARIO Standard, Version 1.0. Association for Standardization of Automation and Measuring Systems (2020)

[16] ASAM e.V.: OpenDRIVE Standard, Version 1.7. Association for Standardization of Automation and Measuring Systems (2021)

[17] Krajewski, R., Bock, J., Kloeker, L., Eckstein, L.: The highD Dataset: A Drone Dataset of Naturalistic Vehicle Trajectories on German Highways for Validation of Highly Automated Driving Systems. In: IEEE International Conference on Intelligent Transportation Systems (ITSC) (2018)

[18] Bock, J., Krajewski, R., Moers, T., Runde, S., Vater, L., Eckstein, L.: The inD Dataset: A Drone Dataset of Naturalistic Road User Trajectories at German Intersections. In: IEEE Intelligent Vehicles Symposium (IV) (2020)

[19] Caesar, H., Bankiti, V., Lang, A. H., et al.: nuScenes: A Multimodal Dataset for Autonomous Driving. In: IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) (2020)

[20] Sun, P., Kretzschmar, H., Dotiwalla, X., et al.: Scalability in Perception for Autonomous Driving: Waymo Open Dataset. In: IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) (2020)

[21] Geiger, A., Lenz, P., Stiller, C., Urtasun, R.: Vision Meets Robotics: The KITTI Dataset. International Journal of Robotics Research (2013).

[22] Cossu, M.: A Simulation-based Framework for Highway Driving Scenario Recognition. PhD Thesis, University of Genova (2025)

[23] Deng, Y., Tu, Z., Yao, J., Zhang, M., Zhang, T., Zheng, X.: TARGET: Automated Scenario Generation from Traffic Rules for Testing Autonomous Vehicles via Validated, LLM-Guided Knowledge Extraction. arXiv:2305.06018 (2023)

[24] Danso, A. A., Büker, U.: Automated Generation of Test Scenarios for Autonomous, Driving Using LLMs. Electronics 14(16), 3177 (2025)

[25] Chandrasekhar, A., Barati Farimani, A.: Automating MD Simulations for Proteins using Large Language Models: NAMD-Agent. arXiv:2507.07887 (2025).

Symbology

Abbreviations and acronyms

ADS - Autonomous Driving System

API - Application Programming Interface

ASAM - Association for Standardisation of Automation and Measuring Systems

CARLA - Car Learning to Act

DSL - Domain-Specific Language

EV - Ego Vehicle

GPU - Graphics Processing Unit

HD - High Definition

JSON - JavaScript Object Notation

LLM - Large Language Model

LoRA - Low-Rank Adaptation

MBPP - Mostly Basic Programming Problems

NPC - Non-Player Character

NPY - NumPy binary file format

ODD - Operational Design Domain

RAG - Retrieval-Augmented Generation

SR - Success Rate

TFL - Traffic Light

TTC - Time To Collision

VRAM - Video Random Access Memory

XSD - XML Schema Definition

.xosc - OpenSCENARIO scenario file format

.xodr - OpenDRIVE road network file format

Notation

R@1 - Retrieval precision at rank 1: the fraction of queries for which the correct document is returned as the top-ranked result

L1, L2, L3 - Domain specificity levels used in the benchmark of Baiardo et al. [5]: L1 = generic Python algorithms, L2 = data science libraries, L3 = scenariogeneration API

1st% - First-attempt success rate: fraction of scenarios solved correctly on the first generation attempt, without invoking the self-healing retry loop

s/scenario - Seconds per scenario: unit used to report average generation latency

Appendix

Appendix A - Urban JSON Schema

This appendix documents the complete JSON schema for urban scenario records, derived from the MOOVE project data format.

A.1 Top-level structure

Table A.1 - Top-level structure of the urban ADScene JSON format

Field	Type / Description
scenario_code	string - manoeuvre identifier (U1_1, U4_1–U4_4)
segment	object - { type, begin_time, end_time, record: { name, date_time, provider, campaign, input_data_version }, processing_version }
environment.infrastructure.way_points	array of { position_longi_from_0: float, infrastructure_elements: [{ type, value }] }
attributes	array of { type: "code_country", value: string }
key_times	array of frame objects (see A.2–A.5)

A.2 key_times[].ego_vehicle

Fields: `steer_wheel_angle`, `steer_wheel_angle_speed`, `yaw_rate`, `ego_brake`, `driver_will`, `lateral_acceleration`, `engine_speed`, `ego_speed`, `covered_distance_by_ego`, `longitudinal_acceleration_wheels`, `dist_to_road_boundaries_left/right`, `front_id`, `dist_to_marking_left/right`, `marking_type_left/right`, `latitude`, `longitude`, `turning_signal_state`, `ego_lane`.

A.3 key_times[].temporal_climatic

Fields: `day_period`, `outside_temperature`, `dazzled`, `fog`, `precipitation`.

A.4 key_times[].positioning

Fields: `latitude`, `longitude` (WGS84, duplicated from ego_vehicle for convenience).

A.5 key_times[].obstacles[]

Fields: `role`, `confidence`, `unique_id`, `pos_x`, `pos_y`, `obstacle_length`, `obstacle_width`, `relative_velocity_x/y`, `absolute_velocity_x/y`,

relative_acceleration_x, absolute_acceleration_x, time_between_vehicles, orientation, status_mobile, fixed_object_classification, mobile_object_classification, age, age_max, obstacle_lane, obstacle_relative_lane, blinker, in_front_Brake, lane_shift, time_to_collision.

A.6 Code tables (RT_* enumerations)

The following enumerations, defined in the ADScene_codes reference file, are used to decode categorical fields throughout the schema: RT_RoadType, RT_MarkingType, RT_TurningSignalState, RT_GuardRail, RT_Precipitation, RT_Blinker, RT_FixedObjectClassification, RT_MobileObjectClassification, RT_TrafficSignType, RT_SubTrafficSignType, RT_CountryCode, RT_LaneConfidence, RT_JunctionBifurcation, RT_NumberOfLanes, RT_PresenceOfEmergencyLane, RT_SlipRoad, RT_SpecificZone, RT_TollGate, RT_Tunnel, RT_DrivingModeLateral, RT_DrivingModeLongitudinal, RT_Car_Type. Selected values used in the example instances are: RoadType 2 = Country Road / Secondary Road; MarkingType 3 = Dashed line; Precipitation 0 = No Rainfall; MobileObjectClassification 1 = Car, 3 = Truck, 4 = Pedestrian; TrafficSignType 167 = Green Traffic Light, 170 = Orange Traffic Light, 172 = Red Traffic Light, 257 = Back of Sign; CountryCode 250 = France.

Appendix B - Highway JSON Schema

This appendix documents the canonical JSON schema produced by npy_to_json.py from the raw NPY binary files.

B.1 Record schema

Table B.1 - Highway JSON record schema

Field	Type	Description
schema_version	string	Canonicalisation schema version (e.g. "phase1.v1")
scenario_id	string	{maneuver_class}_{binning_version}_{file_stem}_row{row_index}
maneuver_class	string	MOOVE taxonomy code (BRAKE, CINF, CINB, COUTF, COUTB, LCF, LCB, FLF, FLB, FRF, FRB)
maneuver_tag	string	Pipeline-internal tag matching corpus script (e.g. brake, cut_in, lane_change)
domain	string	"highway"
Has_npc	bool	Whether the scenario involves an NPC actor
npc_position	string	"front" | "behind" | null
binning_version	string	Identifier of the binning/discretisation pass that produced the source NPY row
source	object	{ file, row_index } - traceability to the originating NPY file and row
behavior_type	string	"B1"–"B4" (Follow scenarios only, else null)
params	object	Flat dict of raw field values from the source NPY row, keyed by descriptive field names (see B.2)

B.2 params field naming by maneuver class

Table B.2 - params field naming by manoeuvre class

Class	params keys
BRAKE	ego_vel_kmh, lead_vel_kmh, duration_s, delta_vel_kmh, npc_distance_m, brake_intensity, other_dir_traffic, lead_vehicle_type
CINF / CINB	ego_vel_kmh, cutter_vel_kmh (relative), delta_vel_kmh, duration_s, npc_distance_m, other_dir_traffic, direction, lead_vehicle_type
COUTF / COUTB	ego_vel_kmh, cutter_vel_kmh (relative), delta_vel_kmh, duration_s, npc_distance_m, other_dir_traffic, direction, lead_vehicle_type
LCF / LCB	ego_vel_begin_kmh, ego_vel_end_kmh, duration_s, other_dir_traffic, direction
FLF / FLB (B1–B4)	ego_vel_begin_kmh, ego_vel_end_kmh, lead_vel_begin_kmh, lead_vel_end_kmh, npc_distance_begin_m, npc_distance_end_m, lead_dist_traveled_m, duration_s, other_dir_traffic, behavior_type, lead_vehicle_type

FRF / FRB	ego_vel_begin_kmh, ego_vel_end_kmh, ego_dist_traveled_m, duration_s, center_posx_begin_m, center_vel_begin_ms, right_posx_begin_m, right_vel_begin_ms, left_posx_begin_m, left_vel_begin_ms (analogous _end fields), other_dir_traffic / direction

B.3 Example record (BRAKE)


```
{ "schema_version": "phase1.v1",
  "scenario_id": "BR_BK_B0_BRAKE_BINS_VALUES_row000042",
  "maneuver_class": "BR_BK",
  "maneuver_tag": "brake",
  "domain": "highway",
  "has_npc": true,
  "npc_position": "front",
  "binning_version": "B0",
  "source": {
    "file": "BRAKE_BINS_VALUES.npy",
    "row_index": 42
  },
  "behavior_type": null,
  "params": {
    "ego_vel_kmh": "13.1",
    "lead_vel_kmh": "16.3",
    "duration_s": "0.6",
    "delta_vel_kmh": "-0.08",
    "npc_distance_m": "11.7",
    "brake_intensity": "0.11",
    "other_dir_traffic": "Yes",
    "lead_vehicle_type": "Car"
  }
}
```

Appendix C - Example Generated .xosc

This appendix shows an annotated excerpt of a validated OpenSCENARIO 1.0 file generated by ScenWeave for a highway BRAKE scenario, illustrating the top-level structure.

C.1 FileHeader and Entities

```
<?xml version="1.0" encoding="UTF-8"?> <OpenSCENARIO>  <FileHeader revMajor="1"
revMinor="0"                          description="ScenWeave BRAKE scenario"
author="ScenWeave Pipeline" date="2026-06-11"/>      <ParameterDeclarations/>
<CatalogLocations/>      <RoadNetwork>          <LogicFile filepath="Town04"/>
</RoadNetwork>    <Entities>      <ScenarioObject name="ego">         <Vehicle
name="vehicle.tesla.model3" .../>       </ScenarioObject>       <ScenarioObject
name="npc_lead">     <Vehicle name="vehicle.audi.a2" .../>    </ScenarioObject>
</Entities>
```

C.2 Storyboard Init - spawn and initial speed

Ego is spawned using `LanePosition` (highway domain), with road_id, lane_id, and s-offset supplied by the SpawnMatcher:

```
<Init>    <Actions>       <Private entityRef="ego">          <PrivateAction>
<TeleportAction>            <Position>              <LanePosition roadId="12"
laneId="-2" s="184.5" offset="0"/>         </Position>       </TeleportAction>
</PrivateAction>            <PrivateAction>               <LongitudinalAction>
<SpeedAction>              <SpeedActionDynamics dynamicsShape="step" value="0"
dynamicsDimension="time"/>                                  <SpeedActionTarget>
<AbsoluteTargetSpeed    value="3.6389"/>         <!--    13.1    km/h    -->
</SpeedActionTarget>            </SpeedAction>          </LongitudinalAction>
</PrivateAction>    </Private>   </Actions> </Init>
```

C.3 Storyboard - braking ManeuverGroup

The lead vehicle (npc_lead) decelerates after a start trigger condition based on simulation time, reproducing the extracted braking_intensity_g and duration_s parameters:

```
<Story  name="BrakeStory">      <Act  name="BrakeAct">          <ManeuverGroup
name="LeadBrakeGroup"   maximumExecutionCount="1">                    <Actors
selectTriggeringEntities="false">            <EntityRef entityRef="npc_lead"/>
</Actors>      <Maneuver name="BrakeManeuver">        <Event name="BrakeEvent"
priority="overwrite">                          <Action  name="BrakeAction">
<PrivateAction>             <LongitudinalAction>               <SpeedAction>
<SpeedActionDynamics                                  dynamicsShape="linear"
value="0.6" dynamicsDimension="time"/>                    <SpeedActionTarget>
<AbsoluteTargetSpeed value="0.0"/>                      </SpeedActionTarget>
</SpeedAction>              </LongitudinalAction>            </PrivateAction>
</Action>        <StartTrigger>          <ConditionGroup>           <Condition
name="StartCondition"            delay="0"             conditionEdge="rising">
<ByValueCondition>                      <SimulationTimeCondition value="1.0"
rule="greaterThan"/>              </ByValueCondition>            </Condition>
</ConditionGroup>          </StartTrigger>         </Event>       </Maneuver>
</ManeuverGroup>     <StartTrigger>       <ConditionGroup>         <Condition
name="ActStart" delay="0" conditionEdge="rising">           <ByValueCondition>
<SimulationTimeCondition            value="0"            rule="greaterThan"/>
```

```
</ByValueCondition>       </Condition>     </ConditionGroup>    </StartTrigger>
</Act> </Story>
```

C.4 StopTrigger

```
<StopTrigger>   <ConditionGroup>      <Condition name="EndCondition" delay="0"
conditionEdge="rising">      <ByValueCondition>        <SimulationTimeCondition
value="14.4" rule="greaterThan"/>         <!-- timeout_s = duration_s * 4.0 -->
</ByValueCondition>     </Condition>   </ConditionGroup> </StopTrigger>
```

Appendix D - Spawn Point Database Format

This appendix documents the format of `Town04_Spawnpoints_NEW.json`, the manually verified spawn point database used by the SpawnMatcher module.

D.1 File structure

Table D.1 - Spawn point database file structure

Field	Description
metadata.map	CARLA map identifier, e.g. "Carla/Maps/Town04"
spawn_points[]	Array of spawn point entries (see D.2)

D.2 Spawn point entry schema

Table D.2 - Spawn point entry schema

Field	Description
spawn_id	Unique integer identifier
context	"urban" | "highway" | "rural" | "divided_road"
transform	{ x, y, z, yaw } - CARLA world coordinates (left-handed, degrees)
geometry	{ road_id, lane_id, s, road_network: { current_road_length_m } } - OpenDRIVE references
surroundings	{ speed_limit_kmh, traffic_lights: [{ x, y, distance_m }], stop_line_distance_m, is_highway_layout }
features	{ is_junction, is_straight, is_curve, is_intersection, is_intersection_w_traffic_light }
annotations	{ suitable_for: [string], priority: int, manually_reviewed: bool } - assigned during database construction and subsequently verified manually

D.3 Example entry (urban, traffic-light-suitable)


```
{   "spawn_id": 142,   "context": "urban",   "transform": { "x": 312.4, "y": -
171.8, "z": 0.3, "yaw": 90.0 },   "geometry": {     "road_id": 38, "lane_id": -
1, "s": 12.6,       "road_network": { "current_road_length_m": 95.0 }     },
"surroundings": {     "speed_limit_kmh": 50,     "traffic_lights": [{ "x": 315.0,
"y": -140.2, "distance_m": 31.7 }],          "stop_line_distance_m": 28.4,
"is_highway_layout": false     },     "features": {       "is_junction": false,
"is_straight": true, "is_curve": false,          "is_intersection": true,
"is_intersection_w_traffic_light": true         },         "annotations": {
"suitable_for":              ["stop_at_tfl_red",             "run_amber_tfl",
"go_straight_green_tfl"],     "priority": 2,     "manually_reviewed": true   }
}
```

D.4 Example entry (highway, straight segment)


```
{   "spawn_id": 298,   "context": "highway",   "transform": { "x": -22.1, "y":
18.9, "z": 0.5, "yaw": 0.3 },   "geometry": {     "road_id": 5, "lane_id": -2,
"s": 412.0,        "road_network": { "current_road_length_m": 780.0 }     },
"surroundings": {        "speed_limit_kmh": 130,        "traffic_lights": [],
"stop_line_distance_m": null,     "is_highway_layout": true   },   "features":
{           "is_junction": false,  "is_straight": true,  "is_curve": false,
"is_intersection": false,  "is_intersection_w_traffic_light": false       },
"annotations": {            "suitable_for": [],            "priority": 1,
"manually_reviewed": true   } }
```


Spawn 298 corresponds to the first spawn point in the round-robin sequence observed in the highway batch generation results discussed. As a highway entry, annotations.suitable_for is empty, since the highway scoring function (_score_highway()) does not use manoeuvre-specific suitability filtering, relying instead on the context and features blocks as described.