CARLAGEN4D: A MULTI-AGENT FRAMEWORK FOR TEXT-DRIVEN 4D SCENE GENERATION IN CARLA

Anonymous authors
Paper under double-blind review

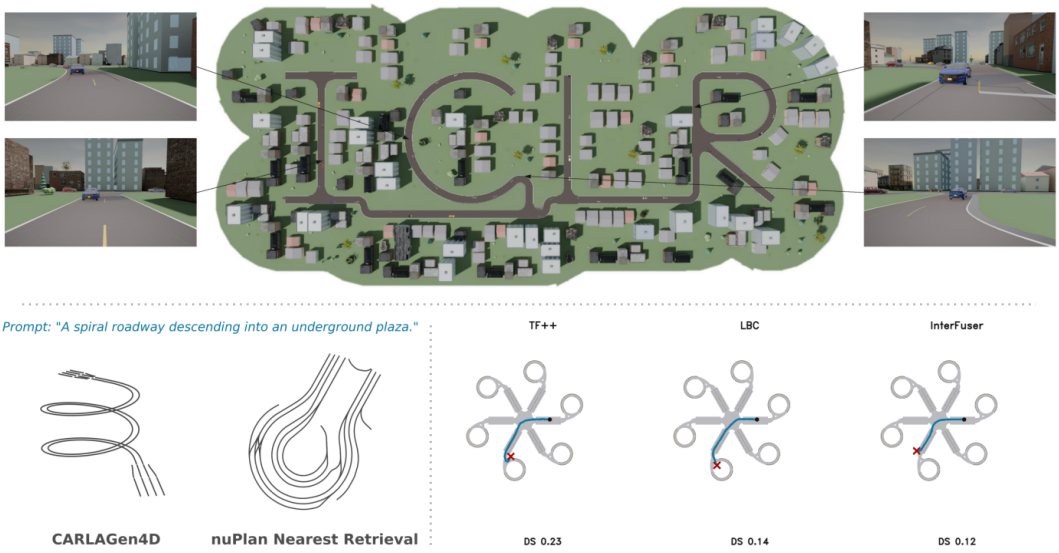


Figure 1: **Top : A drivable CARLA town written from a single text prompt.** CARLAGEN4D was asked for a map whose roads spell the word "ICLR". **Bottom Left :** The closest sample retrieved from nuPlan, illustrating the diversity of the generated maps. **Bottom Right :** An example generated scenario for exposing the failure modes of different driving policies.

ABSTRACT

Autonomous driving research heavily relies on high-fidelity simulators such as CARLA, where evaluation is almost exclusively performed on a limited set of predefined, artist-designed towns. Prior work in simulation-based evaluation largely depends on these fixed maps, constraining scenario diversity and limiting the ability to test generalization to novel environments and edge cases. As a result, current simulation-based evaluation pipelines struggle to scale with the complexity and variability required for robust autonomous driving validation. To address this gap, we present an end-to-end framework for text-to-4D simulation generation that directly enables controllable, scenario-driven evaluation of AD systems. Given a natural language prompt describing a testing scenario (e.g., unusual road topology, adversarial traffic behavior), our multi-agent system generates a complete, temporally consistent simulation, jointly synthesizing road topology, background assets, and dynamic agent behaviors from scratch. This removes reliance on fixed maps and enables on-demand creation of targeted evaluation scenarios. Since direct map generation is prohibitively expensive in token space, we design a domain-specific language that enables efficient map synthesis. We further benchmark driving policies across diverse generated road layouts and traffic scenarios, revealing their sensitivity to map configuration and scenario diversity. Beyond evaluation, the generated scenes support a range of downstream applications, including RGB-to-BEV perception, scene narration and captioning, and transfer to real-world imagery.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

1 INTRODUCTION

Autonomous driving systems need to be evaluated across a broad range of traffic scenarios, from routine highway driving to rare and safety-critical edge cases. High-fidelity simulators such as CARLA (Dosovitskiy et al., 2017) are widely used for this purpose, providing photorealistic rendering, realistic vehicle dynamics, and controllable sensor suites. However, their scalability is limited by a basic constraint: most simulator-based benchmarks still depend on a fixed set of hand-authored town layouts (Dosovitskiy et al., 2017; Caesar et al., 2021). Creating these maps is time-consuming, and the resulting layouts cover only the scenarios that were anticipated during their design. They also cannot easily be expanded when new evaluation cases are needed. Consequently, a model that performs well on existing towns may still struggle with road topologies, junction geometries, or traffic patterns that are not represented in those environments.

This motivates *text-to-4D scene generation* for autonomous driving simulation, where a natural-language prompt is used to generate a complete and temporally consistent CARLA scene. For instance, a prompt such as *"a busy T-junction with a slow-moving bus blocking the left lane and a cyclist weaving between cars"* can describe the road layout, surrounding environment, and behavior of the actors in the scene. This makes it possible to create rare scenarios on demand, test driving policies on road layouts they have not seen before, and describe complex traffic interactions without manually building maps or writing scripts. Figure 1 shows how far this control can extend: a prompt describing a road network that spells a word is used to generate a populated, drivable town in CARLA.

This problem is substantially harder than prior text-to-scenario settings because it requires *jointly* solving three coupled generation problems: map topology, scene dressing, and multi-agent behavior. These components must remain geometrically consistent, physically plausible, and semantically faithful to the prompt, while also being executable inside the simulator. Recent progress in large language and vision-language models (OpenAI, 2026; Anthropic, 2026; Qwen Team, 2026) has made them increasingly capable of code generation, visual reasoning, and interaction with external tools, but their use for procedural generation of complete driving simulations remains largely unexplored. We leverage these capabilities through a set of specialized agents, each operating on a different part of the scene. A road-generation agent produces a compact program for the road topology and uses compiler feedback to iteratively repair the map. Given the resulting topology, background agents populate the available space with semantically appropriate assets, while foreground agents place actors and generate their trajectories using the road graph and closed-loop simulation for validation. We instantiate this framework as CARLAGEN4D and evaluate it both for generating diverse road topologies and for producing simulation data for training driving models.

Contributions. We present CARLAGEN4D, a multi-agent framework for text-to-4D scene generation in driving simulation. Our main contributions are:

1. **A domain-specific language for map synthesis.** We introduce a compact representation that produces cleaner maps at lower cost: $2.6\times$ fewer output tokens and $2.1\times$ faster than raw XML, while improving map quality.
2. **A compiler-in-the-loop topology agent.** The agent iteratively refines its program using compiler feedback about roads and junctions, and only finishes when the map compiles successfully and the intended routes are drivable.
3. **Evaluation on generated topologies.** We benchmark various driving policies on novel generated road layouts and find driving scores drop by up to 1.26× on junction-heavy maps and up to 3.33× on elevated maps relative to simple layouts.
4. **Generated data for training.** We show that augmenting training with generated scenes improves open-loop displacement error and camera-to-BEV mIoU on held-out generated scenes, while maintaining or improving performance on existing towns.

2 RELATED WORK

2.1 SCENARIO GENERATION IN CARLA VIA LLMS AND AGENTIC PIPELINES

Several systems translate language prompts into executable Scenic (Fremont et al., 2019) or OpenSCENARIO programs that run inside CARLA (Dosovitskiy et al., 2017). ChatScene (Zhang et al.,

2024) retrieves Scenic snippets from a hand-built database keyed on geometry, behaviour, and location sub-prompts. TTSG (Ruan et al., 2024) stacks four LLM stages (analysis, retrieval, planning, ranking) and reports a reduced collision rate on SafeBench (Xu et al., 2022) critical scenarios, but it also assumes a fixed map and is largely limited to relatively simple traffic motions. ScenicNL (Elmaaroufi et al., 2024) converts crash reports into probabilistic Scenic programs, Text2Scenario (Cai et al., 2025) parses prompts via a hierarchical DSL corpus, and ScenGE (Liu et al., 2025) adds RAG and adversarial trajectory optimisation. All of these methods place actors and trajectories on top of pre-authored CARLA towns, and the underlying OpenDRIVE map is never generated.

2.2 Text-conditioned map generation

A second group produces the road layout itself rather than placing scenarios on a fixed town. OmniTester (Lu et al., 2024) uses a multimodal LLM to generate SUMO networks with a vision-language repair loop, and ML-SceGen (Xiao et al., 2025) builds scenarios at multiple levels of abstraction. ScenarioControl (Gao et al., 2026) generates vectorized lane graphs from text with a latent diffusion model, whereas LCTGen (Tan et al., 2023) retrieves its map from a library of real-world layouts and generates only the traffic on top of it, so its topologies are bounded by that library. Our setting instead targets novel road topologies. Our DSL separates topology from geometry: the model specifies the road structure, while the compiler resolves geometry, including elevation, and flags inconsistencies.

2.3 Multi-agent LLM driving frameworks and text-to-4D generation

LLM-based agents have emerged on the planning side of autonomous driving (Mao et al., 2024; Wen et al., 2024; Xu et al., 2024; Hu et al., 2024; Hou et al., 2025; Wang et al., 2025), with some adopting multi-agent designs. DiLu (Wen et al., 2024) partitions decision-making across Reasoning, Reflection, and Memory modules; AgentsCoDriver (Hu et al., 2024) lets LLM-driven vehicles communicate and negotiate in collaborative driving. None of these systems generates the simulation environment itself. Generic text-to-4D generation (Singer et al., 2023; Bahmani et al., 2024) optimises a dynamic 3D representation with score distillation from text-to-video diffusion models, producing individual dynamic objects or small scenes, and does not address simulator integration or road networks.

3 Methodology

We present CarlaGen4D, a multi-agent framework for synthesizing complete 4D driving scenes in the CARLA simulator (Dosovitskiy et al., 2017) from a single natural-language prompt $\mathcal{T}$. We represent a scene $\mathcal{S}$ as $\mathcal{S} = \big(\mathcal{R}, \mathcal{B}, \mathcal{F}\big)$, where $\mathcal{R}$ denotes the road topology, $\mathcal{B}$ the static background (e.g., buildings, vegetation, and props), and $\mathcal{F}$ the dynamic actors and their time-indexed trajectories. Each component is generated by a specialized module. Road topology is generated first, since both static and dynamic elements are defined with respect to the resulting road network. Once the map is loaded, the background and foreground modules operate independently and can be executed concurrently. An overview of the pipeline is shown in Figure 2.

3.1 Road Topology Generation

The layout DSL Given $\mathcal{T}$, we first generate a road topology $\mathcal{R}$ that can be loaded by CARLA. Directly generating OpenDRIVE XML is verbose and makes it difficult to express and revise complex road layouts. We use a compact Python-embedded DSL $\mathcal{D}$ that captures the essential road structure while leaving low-level geometric details to the compiler. The DSL provides four basic objects: *cross-sections*, *roads*, *ports*, and *connections*.

A *cross-section* specifies an ordered set of lanes, with each lane defined by its type and width. A *road* assigns a cross-section to a reference line composed of straight, arc, and clothoid segments, and supports elevation profiles defined by a small set of knots. Each road exposes its two endpoints as *ports*. A port can either remain unconnected, forming a map boundary, or participate in a single connection. Connections are either *joints* or *junctions*. A joint connects two compatible ports that form a continuous road, while a junction connects multiple ports and generates the required connecting roads from the permitted movements. The resulting DSL remains close to the OpenDRIVE representation, making compilation largely a direct translation rather than a reconstruction of the intended topology.

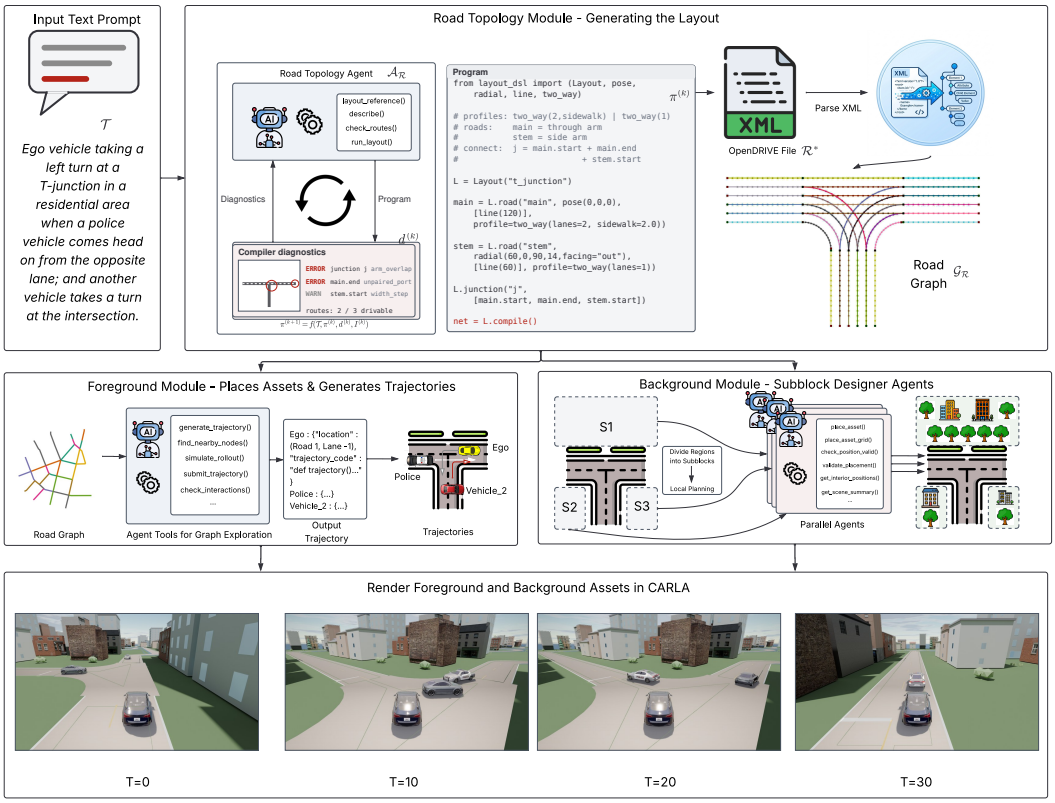


Figure 2: **Overview of CARLAGEN4D.** A text prompt specifies the road topology, foreground, and background modules. Each stage is checked by an executable verifier: compiler and route checks for roads, interaction and closed-loop rollout checks for foreground, and boundary and overlap checks for background. The verified outputs are then combined into a single CARLA scene.

Figure 8 shows representative examples, with the complete syntax and OpenDRIVE mapping provided in Appendix A.

Road Topology Agent The Road Topology Agent $\mathcal{A}_{\mathcal{R}}$ generates a DSL program from $\mathcal{T}$ and iteratively refines it until the resulting map is drivable. We use the compiler as the feedback mechanism rather than a separate critic. Compilation returns structured diagnostics for invalid joints, incompatible lanes, overlapping junction arms, and unconnected ports, together with a rendered map. The agent can also inspect port poses, lane intervals, and the drivable status of specified movements. Termination is allowed only when compilation succeeds and the required routes are drivable. The system prompt and tool definitions are provided in Appendix B.

Before generating the program, the agent specifies the intended decomposition in terms of cross-sections, roads, and connections. At iteration k, the program $\pi^{(k)}$ produces an image $I^{(k)}$, an OpenDRIVE map $\mathcal{R}^{(k)}$, and diagnostics $d^{(k)}$, which are used to construct the next program:

$$\pi^{(k+1)} = f\big(\mathcal{T}, \pi^{(k)}, d^{(k)}, I^{(k)}\big). \tag{1}$$

The process terminates when the agent produces a valid map with passing routes, or when the iteration or token budget is exhausted. In the latter case, we retain the last successfully compiled map. The loaded map $\mathcal{R}^*$ is represented as a directed graph $\mathcal{G}_{\mathcal{R}}$ over lane endpoints, providing the road structure used by the foreground and background agents for subsequent scene generation.

3.2 Foreground Asset Placement and Trajectory Generation

Given $\mathcal{G}_{\mathcal{R}}$ and $\mathcal{T}$, the foreground module generates $\mathcal{F} = \{(a_i, \tau_i)\}_{i=1}^{N}$ where a_i specifies the actor type and CARLA blueprint, and τ_i its trajectory. Unlike previous text-to-scenario systems such as TTSG (Ruan et al., 2024), which operate on fixed towns, our foreground agent is conditioned on the



Figure 3: **Generated towns.** Aerial renders of generated road layouts. Additional examples are provided in Appendix H.2.

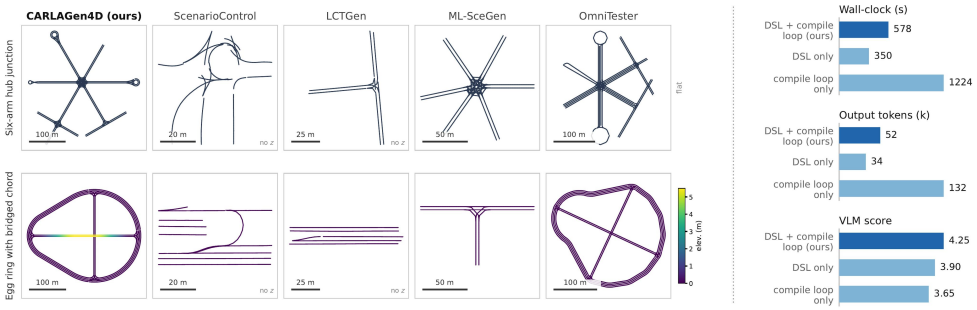


Figure 4: **Left: Qualitative comparison of text-to-map generation.** Each column shows a different method. Additional examples are provided in Appendix H.2. **Right: Ablation of our map generation pipeline.** We compare against two variants: one without our DSL and one without the compile loop. Our full method produces higher-quality maps while requiring substantially fewer tokens.

generated road graph, allowing scenarios to be instantiated on a topology that supports the requested interactions. This removes a key bottleneck of fixed-map approaches: across our prompt set, all generated scenarios can be executed, compared with only 30.8% for TTSG, with most TTSG failures arising because its fixed town lacks a suitable road or site for the requested scenario (Appendix G). A placement agent determines the actors and their initial positions using queries to the road graph. Actors are placed relative to the road network rather than absolute coordinates to ensure valid lane placement. A second agent jointly generates the trajectories for all N actors. It produces a Python function for each actor that constructs its path using map-aware primitives such as lane following and route planning. Importantly, the foreground agent does not only generate plausible trajectory descriptions, but executes and verifies them in CARLA, and the plan is accepted only if the rollout completes without collisions, rule violations, or blocked actors. The tool interfaces, trajectory helpers, and motion specification are described in Appendix C.1.

3.3 Background Asset Placement

Static assets are placed on *plots*, non-drivable regions defined by enclosed city blocks and roadside area. This also provides placement regions for road layouts without enclosed blocks, such as corridors and grade-separated interchanges. Each of the resulting M plots is assigned a Designer Agent, which populates it with buildings, vegetation, and other structures from a semantic asset catalogue. Candidate assets are validated during placement by checking their bounding boxes against the plot boundary and existing assets. Agents prioritize the plot perimeter and operate independently across plots. Each plot is also assigned a ground polygon to avoid exposed terrain. The catalogue and placement tools are described in Appendix D. Figure 3 shows examples of generated scenes. The final scene $\mathcal{S} = (\mathcal{R}^*, \mathcal{B}, \mathcal{F})$ is then instantiated in CARLA.

Table 1: **Text-to-map generation over** 250 **prompts**, pooled over the four sets. Tier B is marked *n/a* for methods whose output format cannot express it. Connectivity, layout diversity and the per-set breakdown are given in Table 7.

	Tier A (all methods)				**Tier B** (OpenDRIVE)	
Method	Success	Ext.→ 1	SPG↑	VLM↑	Load	Elev.↑
ScenarioControl (Gao et al., 2026)	**100.0%**	0.28	0.041	2.36	n/a	n/a
LCTGen (Tan et al., 2023)	99.6%	0.43	0.009	1.75	n/a	n/a
ML-SceGen (Xiao et al., 2025)	95.6%	0.56	0.106	2.93	100%	0.0%
OmniTester (Lu et al., 2024)	97.2%	**1.20**	0.160	4.02	100%	28.0%
Ours	98.4%	1.28	**0.196**	**4.54**	100%	**69.4%**

Table 2: **Driving policies on held-out generated maps.** DS is driving score, RC is route completion, Col. is total collisions

Agent	DS↑	RC↑	Col.↓	Easy	Junction	Elevation
Roach (Zhang et al., 2021)	**0.683**	**0.740**	4	0.689	0.547	**0.813**
CILRS (Codevilla et al., 2019)	0.586	0.708	8	0.676	0.536	0.546
LBC (Chen et al., 2019)	0.415	0.521	10	0.401	0.391	0.454
InterFuser (Shao et al., 2022)	0.296	0.322	**3**	0.410	0.355	0.123
TF++ (Jaeger et al., 2023)	0.669	0.726	4	**0.738**	**0.631**	0.639
UniAD (Hu et al., 2023)	0.348	0.470	9	0.284	0.462	0.297
VAD (Jiang et al., 2023)	0.241	0.372	14	0.189	0.302	0.233

4 EXPERIMENTS

We design our evaluation around three questions. **(Q1)** Are the generated maps correct, loadable and diverse, and how do they compare to existing text-to-map methods and to the road layouts already available in simulation and in real-world driving datasets? **(Q2)** Do the resulting simulations function as a *benchmark*, exposing failures of modern driving policies on topologies they have never seen? **(Q3)** Does training on these simulations *improve* downstream models like driving policies, camera-to-BEV perception, and driving captioners?

4.1 SETUP

We evaluate on 250 prompts in four sets: *base* prompts covering the usual layouts spanning CARLA towns; *novel* prompts describing layouts absent from any CARLA town, such as brick-bond tessellations, turbine interchanges and corkscrew ramps; *ordinary* prompts asking for the plain grid and collector networks that dominate real roads, so the comparison is not limited to unusual geometries; and *complex* prompts combining multiple constraints simultaneously. Appendix H.1 lists the prompt suite, the held-out maps and the collection settings.

4.2 Q1: MAP GENERATION QUALITY AND DIVERSITY

Baselines. We compare against four text-conditioned scenario/map generators that span the design space: **ScenarioControl** (Gao et al., 2026), a latent diffusion model over vectorized scenes conditioned on text; **LCTGen** (Tan et al., 2023), which retrieves from a library of cached real-world (Waymo (Sun et al., 2020)) map templates; **ML-SceGen** (Xiao et al., 2025), an LLM-to-symbolic-solver pipeline emitting OpenDRIVE; and **OmniTester** (Lu et al., 2024), an LLM-to-SUMO pipeline with a vision-language self-repair loop.

Metrics. We report a *tiered* evaluation where Tier A evaluates the raw lane graph. Generation Success is the fraction of prompts producing a valid map artifact, while Spatial Extent (Ext.) is the ratio of realized to requested extent for prompts specifying a target size. Shuffled Perturbation Gap (SPG) (Gao et al., 2026) measures prompt fidelity by comparing prompt-map similarity with that under random reassignment. VLM is the mean score from a vision-language judge across four 0-5 dimensions, using both rendered views and parsed map attributes. Tier B evaluates simulator-ready maps: Load is the fraction successfully loaded by CARLA, and Elevation is the fraction of elevation-specified prompts producing non-zero grade. Definitions and formulas are in Appendix F.2.

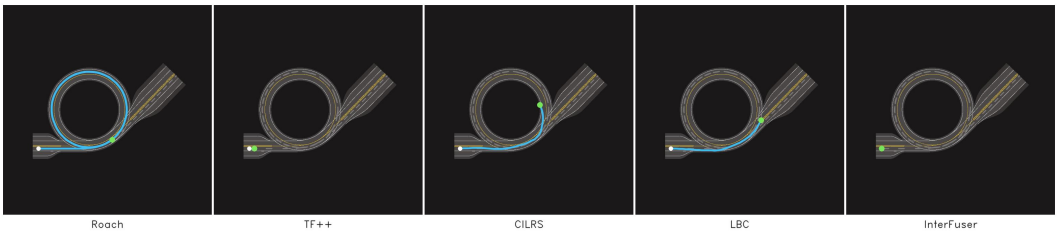


Figure 5: **Novel topologies distinguish privileged from sensorimotor driving.** Executed trajectories against the planned route on a corkscrew ramp. The white marker is the start, the green marker where each agent stopped. Roach reads a top-down raster and completes the loop; the sensorimotor agents must infer the layout and all often stop midway or deviate.

Table 3: **Generated scenes as training data.** Open-loop L_2 (m, averaged over $1/2/3$ s) for the driving policy and mIoU for camera-to-BEV perception, each evaluated on held-out hand-authored towns and on held-out generated scenes. Mean $\pm$ std over three seeds.

	Driving policy $L_2 \downarrow$		Camera-to-BEV mIoU↑	
Training data	hand-authored	CARLAGEN4D	hand-authored	CARLAGEN4D
CARLAGEN4D only (control)	6.62	3.48	0.082	0.407
Hand-authored towns only	4.90 ± 0.10	7.79 ± 0.16	0.157 ± 0.012	0.110 ± 0.003
+ CARLAGEN4D (matched)	4.42 ± 0.26	3.10 ± 0.11	$\mathbf{0.183} \pm 0.006$	0.364 ± 0.001
+ CARLAGEN4D (all data)	$\mathbf{4.10 \pm 0.12}$	$\mathbf{3.09 \pm 0.04}$	0.172 ± 0.002	$\mathbf{0.409} \pm 0.007$

Results. The main differences across methods come from prompt adherence. Functional and VLM evaluations show the same overall ranking, with our method performing best or comparable on both. ML-SceGen performs well on ordinary prompts but struggles with complex ones that require more specific combinations of constraints (additional results in Appendix H.2). Among OpenDRIVE-based methods, elevation shows the clearest differences. OmniTester is the strongest baseline overall, trailing on extent, elevation, and prompt adherence. Qualitative comparison is shown in Figure 4. Scores from an additional VLM judge which preserves our ranking, are reported in Appendix F.3.

Figure 4 shows the ablations for the DSL and the compiler loop where we see that the full approach achieves a $1.16\times$ higher VLM score, while using $2.1\times$ less wall-clock time and $2.6\times$ fewer output tokens than the compile-loop-only baseline. Detailed metrics for the ablation are listed in Appendix F.4. For diversity, roughly one-third of our road patches have no nuPlan patch among their 10 nearest neighbors. Additional details and comparisons with CARLA towns are provided in Appendix I.

4.3 Q2: GENERATED TOPOLOGIES AS A POLICY BENCHMARK

Protocol. We evaluate seven driving policies, a privileged reinforcement-learning expert Roach (Zhang et al., 2021), and six sensorimotor agents through a common evaluation interface. Fifteen held-out maps are grouped into *easy*, *junction* and *elevation* categories and we report driving score (DS), route completion (RC) and collision count under the standard CARLA definitions.

Results. The privileged expert (Zhang et al., 2021) and TF++ (Jaeger et al., 2023) achieve similar overall performance, while the other agents perform lower. The category-wise results show that the policies respond differently to harder map structures. Relative to the Easy split, Junction and Elevation scores drop for several sensorimotor policies, with the largest drop on Elevation for InterFuser (Shao et al., 2022) (0.410 to 0.123) and CILRS (Codevilla et al., 2019) (0.676 to 0.546). These differences suggest that generated topologies expose failures less visible in simpler CARLA towns.

Figure 5 shows a representative failure on a corkscrew ramp. The privileged agent completes the route, while the sensorimotor agents often stop midway or deviate from the road. This highlights a limitation of camera-based agents in reasoning about three-dimensional road geometry. In a top-down raster, the spiral looks like a regular curved road, while camera-based agents have to infer how the road continues as it rises with increasing depth. Such cases are absent from standard flat-town benchmarks and therefore remain difficult to evaluate.

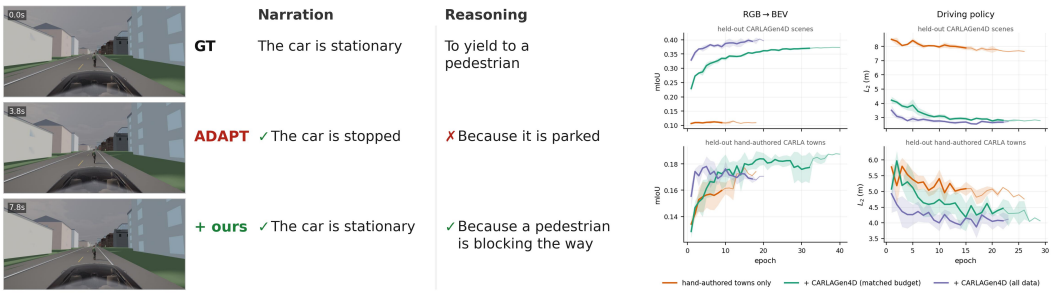


Figure 6: **Left: driving captions before and after fine-tuning.** Three frames of an 8 s clip with the reference caption, the released ADAPT checkpoint and our fine-tune; the ego holds for a crossing pedestrian. **Right: validation curves for both joint-training studies**, mean over 3 seeds, shaded ± 1 std. *Top:* held-out generated scenes. *Bottom:* held-out hand-authored towns.

4.4 Q3: Generated simulations as training data

Sections 4.2 and 4.3 show that CarlaGen4D produces diverse, drivable maps that reveal failure cases not captured by existing benchmarks. We next evaluate whether these maps are useful as training data. We add CarlaGen4D scenes to training sets for three downstream tasks with different architectures, objectives, and evaluation metrics. Throughout, we distinguish **hand-authored CARLA towns** from **CarlaGen4D-generated scenes**, with both held out at the map level. We compare models trained on hand-authored towns alone with models trained on mixtures that include generated scenes. In the *matched frame budget* setting, the total number of training frames is fixed, with half coming from generated scenes, allowing us to compare the data sources without increasing the training volume. We also report results when all available data are used, and train on generated scenes alone as a control.

4.4.1 Driving policy

We train a branched camera policy from scratch, without driving-pretrained weights, to predict waypoints at 1, 2, and 3 s and evaluate it using standard open-loop metrics.

Table 3highlights the benefit of CarlaGen4D as a source of complementary training coverage. Most importantly, under the matched frame budget, replacing half of the hand-authored data with CarlaGen4D improves L_2 error on held-out generated scenes from 7.79 to 3.10m, while also improving performance on hand-authored towns. Thus, the gain cannot be attributed simply to having more training data i.e. with the same amount of data, generated scenes expose the policy to different road configurations that improve generalization to both generated and hand-authored maps. Using all available data provides a further improvement to both the sets.

4.4.2 Camera-to-BEV perception

Following prior monocular BEV work (Roddick & Cipolla, 2020; Philion & Fidler, 2020; Zhou & Krähenbühl, 2022), we predict a camera-visible BEV semantic map covering 50×50m, in front of the ego vehicle at 0.5m/px. Supervision is limited to pixels visible to the camera, using masks for the horizontal and vertical field of view as well as occlusions from buildings, terrain, and other agents.

As shown in Table 3, the matched data experiment shows that replacing half of the hand-authored data with CarlaGen4D more than triples mIoU on held-out generated scenes, while also improving performance on hand-authored towns. The gains on generated scenes and hand authored scenes are consistent across object classes (Table 12), showing that the generated data adds coverage that is missing from the hand-authored training set.

4.4.3 Driving caption generation

We fine-tune ADAPT (Jin et al., 2023), a driving captioner trained on BDD-X (Kim et al., 2018), on narration/reasoning pairs derived from the simulator's own logged state, and evaluate on held-out CarlaGen4D-generated scenes.

Table 4: **Driving caption generation.** Category accuracy is measured against the finite label space the references are drawn from; the shuffled-clip metric reports CIDEr against the clip's own reference minus CIDEr against randomly reassigned references, and isolates video-conditioned captioning from learning the reference phrasing. Two seeds.

	Category acc.↑		Shuffle gap↑		CIDEr	
Model	narration	reasoning	narration	reasoning	narr.	reas.
Majority class	0.259	0.429	–	–	–	–
ADAPT, zero-shot (Jin et al., 2023)	0.134	0.036	0.002	0.035	0.142	0.155
+ fine-tuned on generated scenes	**0.371**	**0.545**	**0.428**	**0.554**	1.087	1.647

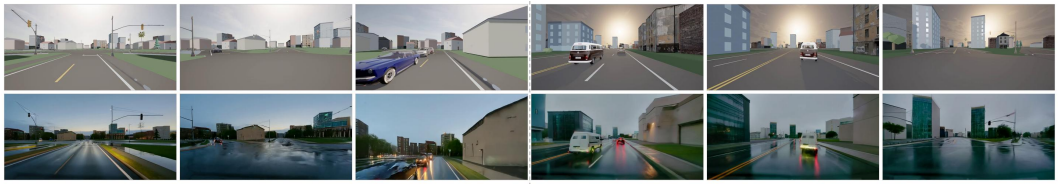


Figure 7: **Generated scenes translated to photorealistic video.** The *top row* is the simulator render, the *bottom row* the same frames after translation with Cosmos-Transfer2.5. *Left:* a left turn through a six-way junction. *Right:* a vehicle weaving ahead of the ego.

Table 4 reports the results. We focus on category accuracy, shuffled-clip control, and the n-gram metrics, as they better capture whether the generated captions reflect the underlying driving semantics. The released checkpoint shows almost no difference between the two conditions ($\Delta\text{CIDEr} = 0.002$), suggesting that its predictions are largely independent of the video. After fine-tuning, the gap increases to 0.42-0.56 across seeds, while reasoning accuracy improves beyond the majority-class baseline. This suggests that fine-tuning helps the model connect the described event to the observed driving behavior, rather than relying mainly on generic caption patterns. Figure 6 shows this difference qualitatively.

4.4.4 Transfer to photorealistic imagery

The generated scenes by CarlaGen4D are rendered in CARLA, but the visual gap between simulation and real camera data can limit its use for training real world policies. We address this by applying Cosmos-Transfer2.5 (NVIDIA et al., 2026) to the rendered videos, using the simulator's semantic segmentation as the conditioning signal. The underlying scene remains unchanged: road layout, markings, buildings, and agent trajectories are preserved, while the video appearance is resynthesized (Figure 7). This allows the same generated episode to be used either in its original CARLA representation, where simulator annotations are available, or as a more realistic video representation for tasks that depend on visual appearance.

5 Conclusion

We introduced CarlaGen4D, an end-to-end pipeline for text-driven 4D scene generation in CARLA. Our compact DSL and compiler-in-the-loop agent enable efficient generation of valid, drivable road topologies, while the resulting scenes provide a diverse testbed for evaluating driving policies across map configurations and traffic scenarios. We further show that generated scenes can augment training to improve performance on both generated layouts and on existing towns. Together, these results highlight the potential of text-driven 4D generation for autonomous driving simulation.

Limitations. As seen in Section 4.4, generated scenes complement rather than replace hand-authored ones and training on generated scenes alone performs substantially worse on hand-authored towns. Future work could explore more targeted generation strategies to improve transfer to unseen hand-authored environments. Generation is not yet interactive i.e. while the DSL is $2.1\times$ faster than direct OpenDRIVE generation, a single road network still takes $\approx$10 minutes and $\sim$52K output tokens. This limits rapid generation of large, diverse scenario suites; caching, verified sub-layout retrieval, and smaller fine-tuned generation models could reduce this cost.

Reproducibility Statement

The prompt suite, system prompts, tool interfaces and metric definitions needed to reproduce this work are given in the appendix. Appendix E and Appendix H.2 list the map prompts and the per-family breakdown. The agents are specified in full: Appendix B and Appendix A give the road topology agent's system prompt and the DSL, Appendix C and Appendix C.1 the foreground agent and its trajectory helpers, and Appendix D the background agent. Appendix F.2 defines every Tier A and Tier B metric and Appendix F.3 reports the same ranking under a judge from a second provider. Appendix F.4 states the ablation protocol, including the shared driver, the per-run budget and the definition of a clean run; Appendix G states the trajectory comparison protocol; Appendix I states the patch decomposition and the descriptors used for the nearest-neighbour analysis. Model identifiers are given where they matter: the agents run on `claude-sonnet-5` and the judge on `claude-opus-5`.

AI Use Statement

Large language models appear in this work in three distinct roles, and we separate them deliberately.

As the object of study. The system we propose *is* an LLM-agent pipeline: the road topology, foreground and background agents are prompted language models (`claude-sonnet-5`) that write DSL programs, trajectory code and placement calls. This is the contribution of the paper and it is specified in Section 3 and Appendices B-D.

As an evaluation instrument. Prompt adherence and fidelity are scored by a vision-language judge (`claude-opus-5`); the rubric is in Appendix F.2 and a cross-provider check is in Appendix F.3.

As authoring assistance. We used an LLM to draft and refine parts of the evaluation prompt suite, which was then reviewed and edited by the authors before use, and to polish prose of the paper for readability. No result, number, figure or claim in this paper was produced by an LLM acting on its own: the experiments were designed, run and interpreted by the authors, who are responsible for the entire contents of the submission.

References

Anthropic. Claude opus 5. `https://platform.claude.com/docs/en/models/opus-5/overview`, 2026.

ASAM e.V. Opendrive® format specification. `https://www.asam.net/standards/detail/opendrive/`, 2024. Accessed: 2026-05-07.

Sherwin Bahmani, Xian Liu, Wang Yifan, Ivan Skorokhodov, Victor Rong, Ziwei Liu, Xihui Liu, Jeong Joon Park, Sergey Tulyakov, Gordon Wetzstein, Andrea Tagliasacchi, and David B. Lindell. TC4D: Trajectory-conditioned text-to-4D generation. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2024.

Holger Caesar, Juraj Kabzan, Kok Seang Tan, Whye Kit Fong, Eric Wolff, Alex Lang, Luke Fletcher, Oscar Beijbom, and Sammy Omari. nuPlan: A closed-loop ML-based planning benchmark for autonomous vehicles. In *CVPR Workshop on Autonomous Driving*, 2021.

Xuan Cai, Xuesong Bai, Zhiyong Cui, Danmu Xie, Daocheng Fu, Haiyang Yu, and Yilong Ren. Text2scenario: Text-driven scenario generation for autonomous driving test, 2025. URL `https://arxiv.org/abs/2503.02911`.

Dian Chen, Brady Zhou, Vladlen Koltun, and Philipp Krähenbühl. Learning by cheating. In *Proceedings of the 3rd Conference on Robot Learning (CoRL)*, 2019.

Felipe Codevilla, Eder Santana, Antonio M. López, and Adrien Gaidon. Exploring the limitations of behavior cloning for autonomous driving. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 9328–9337, 2019.

Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. Carla: An open urban driving simulator. In *Proceedings of the Conference on Robot Learning (CoRL)*, 2017.

Karim Elmaaroufi, Devan Shanker, Ana Cismaru, Marcell Vazquez-Chanlatte, Alberto Sangiovanni-Vincentelli, Matei Zaharia, and Sanjit A. Seshia. Scenicnl: Generating probabilistic scenario programs from natural language, 2024. URL https://arxiv.org/abs/2405.03709.

Daniel J. Fremont, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L. Sangiovanni-Vincentelli, and Sanjit A. Seshia. Scenic: A language for scenario specification and scene generation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 63–78, 2019. doi: 10.1145/3314221.3314633.

Lili Gao, Yanbo Xu, William Koch, Samuele Ruffino, Luke Rowe, Behdad Chalaki, Dmitriy Rivkin, Julian Ost, Roger Girgis, Mario Bijelic, and Felix Heide. ScenarioControl: Vision-language controllable vectorized latent scenario generation. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2026.

Xinmeng Hou, Wuqi Wang, Long Yang, Hao Lin, Jinglun Feng, Haigen Min, and Xiangmo Zhao. Driveagent: Multi-agent structured reasoning with llm and multimodal sensor fusion for autonomous driving, 2025. URL https://arxiv.org/abs/2505.02123.

Senkang Hu, Zhengru Fang, Zihan Fang, Yiqin Deng, Xianhao Chen, and Yuguang Fang. AgentsCoDriver: Large language model empowered collaborative driving with lifelong learning. *arXiv preprint arXiv:2404.06345*, 2024.

Yihan Hu, Jiazhi Yang, Li Chen, Keyu Li, Chonghao Sima, Xizhou Zhu, Siqi Chai, Senyao Du, Tianwei Lin, Wenhai Wang, Lewei Lu, Xiaosong Jia, Qiang Liu, Jifeng Dai, Yu Qiao, and Hongyang Li. Planning-oriented autonomous driving. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023.

Bernhard Jaeger, Kashyap Chitta, and Andreas Geiger. Hidden biases of end-to-end driving models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 8240–8249, 2023.

Bo Jiang, Shaoyu Chen, Qing Xu, Bencheng Liao, Jiajie Chen, Helong Zhou, Qian Zhang, Wenyu Liu, Chang Huang, and Xinggang Wang. VAD: Vectorized scene representation for efficient autonomous driving. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 8340–8350, 2023.

Bu Jin, Xinyu Liu, Yupeng Zheng, Pengfei Li, Hao Zhao, Tong Zhang, Yuhang Zheng, Guyue Zhou, and Jingjing Liu. ADAPT: Action-aware driving caption transformer. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2023.

Jinkyu Kim, Anna Rohrbach, Trevor Darrell, John Canny, and Zeynep Akata. Textual explanations for self-driving vehicles. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018.

Jiangfan Liu, Yongkang Guo, Fangzhi Zhong, Tianyuan Zhang, Zonglei Jing, Siyuan Liang, Jiakai Wang, Mingchuan Zhang, Aishan Liu, and Xianglong Liu. Adversarial generation and collaborative evolution of safety-critical scenarios for autonomous vehicles, 2025. URL https://arxiv.org/abs/2508.14527.

Qiujing Lu, Xuanhan Wang, Yiwei Jiang, Guangming Zhao, Mingyue Ma, and Shuo Feng. Multimodal large language model driven scenario testing for autonomous vehicles, 2024. URL https://arxiv.org/abs/2409.06450.

Jiageng Mao, Junjie Ye, Yuxi Qian, Marco Pavone, and Yue Wang. A language agent for autonomous driving, 2024. URL https://arxiv.org/abs/2311.10813.

NVIDIA, :, Arslan Ali, Junjie Bai, Maciej Bala, Yogesh Balaji, Aaron Blakeman, Tiffany Cai, Jiaxin Cao, Tianshi Cao, Elizabeth Cha, Yu-Wei Chao, Prithvijit Chattopadhyay, Mike Chen, Yongxin Chen, Yu Chen, Shuai Cheng, Yin Cui, Jenna Diamond, Yifan Ding, Jiaojiao Fan, Linxi Fan, Liang Feng, Francesco Ferroni, Sanja Fidler, Xiao Fu, Ruiyuan Gao, Yunhao Ge, Jinwei Gu, Aryaman Gupta, Siddharth Gururani, Imad El Hanafi, Ali Hassani, Zekun Hao, Jacob Huffman, Joel Jang, Pooya Jannaty, Jan Kautz, Grace Lam, Xuan Li, Zhaoshuo Li, Maosheng Liao, Chen-Hsuan Lin, Tsung-Yi Lin, Yen-Chen Lin, Huan Ling, Ming-Yu Liu, Xian Liu, Yifan Lu, Alice Luo, Qianli

Ma, Hanzi Mao, Kaichun Mo, Seungjun Nah, Yashraj Narang, Abhijeet Panaskar, Lindsey Pavao, Trung Pham, Morteza Ramezanali, Fitsum Reda, Scott Reed, Xuanchi Ren, Haonan Shao, Yue Shen, Stella Shi, Shuran Song, Bartosz Stefaniak, Shangkun Sun, Shitao Tang, Sameena Tasmeen, Lyne Tchapmi, Wei-Cheng Tseng, Jibin Varghese, Andrew Z. Wang, Hao Wang, Haoxiang Wang, Heng Wang, Ting-Chun Wang, Fangyin Wei, Jiashu Xu, Dinghao Yang, Xiaodong Yang, Haotian Ye, Seonghyeon Ye, Xiaohui Zeng, Jing Zhang, Qinsheng Zhang, Kaiwen Zheng, Andrew Zhu, and Yuke Zhu. World simulation with video foundation models for physical ai, 2026. URL https://arxiv.org/abs/2511.00062.

OpenAI. Gpt-5.6 sol. https://developers.openai.com/api/docs/models/gpt-5.6-sol, 2026.

Jonah Philion and Sanja Fidler. Lift, splat, shoot: Encoding images from arbitrary camera rigs by implicitly unprojecting to 3d. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2020.

Qwen Team. Qwen 3.8. https://qwen.ai/blog?id=qwen3.8, 2026.

Thomas Roddick and Roberto Cipolla. Predicting semantic map representations from images using pyramid occupancy networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

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

Hao Shao, Letian Wang, Ruobing Chen, Hongsheng Li, and Yu Liu. Safety-enhanced autonomous driving using interpretable sensor fusion transformer. In *Proceedings of the 6th Conference on Robot Learning (CoRL)*, 2022.

Uriel Singer, Shelly Sheynin, Adam Polyak, Oron Ashual, Iurii Makarov, Filippos Kokkinos, Naman Goyal, Andrea Vedaldi, Devi Parikh, Justin Johnson, and Yaniv Taigman. Text-to-4D dynamic scene generation. In *International Conference on Machine Learning (ICML)*, 2023.

Pei Sun, Henrik Kretzschmar, Xerxes Dotiwalla, Aurelien Chouard, Vijaysai Patnaik, Paul Tsui, James Guo, Yin Zhou, Yuning Chai, Benjamin Caine, et al. Scalability in perception for autonomous driving: Waymo open dataset. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

Shuhan Tan, Boris Ivanovic, Xinshuo Weng, Marco Pavone, and Philipp Kraehenbuehl. Language conditioned traffic generation. In *Proceedings of the 7th Conference on Robot Learning (CoRL)*, volume 229 of *Proceedings of Machine Learning Research*, pp. 2714–2752, 2023.

Shihao Wang, Zhiding Yu, Xiaohui Jiang, Shiyi Lan, Min Shi, Nadine Chang, Jan Kautz, Ying Li, and Jose M. Alvarez. OmniDrive: A holistic vision-language dataset for autonomous driving with counterfactual reasoning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2025.

Licheng Wen, Daocheng Fu, Xin Li, Xinyu Cai, Tao Ma, Pinlong Cai, Min Dou, Botian Shi, Liang He, and Yu Qiao. DiLu: A knowledge-driven approach to autonomous driving with large language models. In *International Conference on Learning Representations (ICLR)*, 2024.

Yicheng Xiao, Yangyang Sun, and Yicheng Lin. ML-SceGen: A multi-level scenario generation framework. *arXiv preprint arXiv:2501.10782*, 2025.

Chejian Xu, Wenhao Ding, Weijie Lyu, Zuxin Liu, Shuai Wang, Yihan He, Hanjiang Hu, Ding Zhao, and Bo Li. SafeBench: A benchmarking platform for safety evaluation of autonomous vehicles. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, volume 35, 2022.

Zhenhua Xu, Yujia Zhang, Enze Xie, Zhen Zhao, Yong Guo, Kwan-Yee K. Wong, Zhenguo Li, and Hengshuang Zhao. DriveGPT4: Interpretable end-to-end autonomous driving via large language model. *IEEE Robotics and Automation Letters*, 2024.

Jiawei Zhang, Chejian Xu, and Bo Li. Chatscene: Knowledge-enabled safety-critical scenario generation for autonomous vehicles, 2024. URL https://arxiv.org/abs/2405.14062.

Zhejun Zhang, Alexander Liniger, Dengxin Dai, Fisher Yu, and Luc Van Gool. End-to-end urban driving by imitating a reinforcement learning coach. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021.

Brady Zhou and Philipp Krähenbühl. Cross-view transformers for real-time map-view semantic segmentation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

SUPPLEMENTARY MATERIAL

CONTENTS

A Road-Network DSL Design and Implementation **15**

A.1 Overview and Motivation 15
A.2 Architecture 15
A.3 Coordinate System 16
A.4 Cross-Sections 16
A.5 Geometry Primitives 16
A.6 Elevation 17
A.7 Connectivity 17
A.8 Compilation and Diagnostics 18
A.9 Routing and Validation 18
A.10 Rendering 18
A.11 OpenDRIVE Export 18
A.12 Limitations and Unsupported Features 19

B Road Topology Agent Details **19**

B.1 System Prompt 19
B.2 Layout Agent Tool Calls 19

C Foreground Agent Details **20**

C.1 Tool Definitions: Foreground Agents 24

D Background Agent Details **26**

D.1 System Prompt 26
D.2 Helper Data Structures 26
D.3 Tool Call Definitions 28
D.4 Free Loop Detection via Planar Graph Face Tracing 30
D.5 Roadside Plots 31

E Evaluation Prompts **31**

E.1 Prompts for Evaluation - Map Generation 31
E.2 Prompts for Evaluation - Scenarios 33

F Map Generation Evaluation **39**

F.1 VLM as a Judge 39
F.2 Metric Definitions 40
F.3 Cross-judge check 42
F.4 Ablation: the DSL and the compile loop 42

G Trajectory Generation Evaluation **43**

G.1 VLM as a Judge 43

G.2 Other Metrics 45

G.3 Results on Other Metrics 47

H Additional Experimental Details and Results **48**

H.1 Setup 48

H.2 Q1: Map Generation Quality and Diversity 49

H.3 Q2: Generated Topologies as a Policy Benchmark 50

H.4 Q3: Generated Simulations as Training Data 51

H.5 Per-class and per-region breakdown for Q3 53

I Layout Diversity against Real and Hand-Authored Road Networks **53**

I.1 What the measurement has to get right 53

I.2 Embedding road networks in a common space 55

I.3 Results 56

A Road-Network DSL Design and Implementation

A.1 Overview and Motivation

`layout_dsl` is a Python domain-specific language (DSL) for describing, compiling, validating, and exporting road networks. It is designed to bridge three concerns simultaneously: (i) *geometric precision*, producing reference lines and elevation profiles that map onto the OpenDRIVE standard ASAM e.V. (2024) without approximation; (ii) *explicit connectivity*, requiring that every place where two roads meet is stated by the author rather than inferred, so that a map which compiles is a map whose topology was intended; and (iii) *ergonomics*, exposing an API in which a complex network is a few dozen lines of Python.

A map is written as three things: the cross-sections that occur in it, the roads that carry those cross-sections along a reference line, and the connections between road ends.

```
L = Layout("t_junction")
main = L.road("main", pose(0, 0, 0), [line(120)],
              profile=two_way(lanes=2, sidewalk=2.0))
stem = L.road("stem", radial(60, 0, 90, 14, facing="out"), [line(60)],
              profile=two_way(lanes=1))
L.junction("j", [main.start, main.end, stem.start])
net = L.compile()
```

Every program ends with `L.compile()`, which returns a `CompiledLayout` exposing `errors`, `warnings`, `describe(name)`, `route(...)`, `render_to_file(path)` and `export_xodr(path)`.

A.2 Architecture

The language is built from a small number of dataclasses, each owning one concern:

- **`LaneDef`, `Profile`** - a cross-section as an ordered list of lanes from the centre line outward, each with a type, a width, and an optional tag used to pair lanes across a profile change.

- **`Seg`, `RefLine`** - reference-line geometry as a sequence of lines, arcs, constant-curvature turns and spirals, with arc-length parameterisation and curvature queries.
- **`RoadSpec`** - one road: a start pose or port, its geometry, its profile schedule, its elevation knots, and its road class.
- **`Port`, `PortLane`** - a road end, carrying its pose, its cross-section, and the lateral interval of each of its lanes. Ports are the only objects that may be connected.
- **`Layout`** - the authored map: roads, joints and junctions.
- **`CompiledLayout`** - the resolved map: per-road lane sections, the connecting roads synthesised for each junction, the lane-level edge set, and the diagnostics.
- **`Diagnostics`** - a list of records, each carrying a severity, a *where* naming the road, joint or junction responsible, a machine-readable code, and a message stating the remedy.

A.3 Coordinate System

The DSL adopts a standard right-handed 2-D Cartesian frame: $+x$ points East and $+y$ points North. Orientations are specified in degrees measured counter-clockwise from $+x$, so 0° is East and 90° is North. Elevation is a third coordinate carried per road rather than per point, as described below. Lane identifiers follow the OpenDRIVE convention: right lanes are $-1, -2, \ldots$ numbered inward to outward, left lanes are $1, 2, \ldots$, and lane 0 is the reference line itself.

Three helpers construct poses: `pose(x, y, heading)` states one directly; `offset_pose(p, ahead, left, turn_deg)` states one relative to another pose or port; and `radial(cx, cy, bearing, distance, facing)` places a pose on a circle about a centre, with `facing="out"` for a road leaving a junction and `"in"` for one arriving. The last of these is how junction arms are laid out, since it guarantees that the arms are at the intended bearings and at a common radius from the junction centre.

A.4 Cross-Sections

A `Profile` lists lanes outward from the centre line on each side, written as a compact string, for example `"driving 3.5 x2, shoulder 2.5"`. Right-hand traffic is assumed: lanes on the right carry traffic along the road direction and lanes on the left carry traffic against it. Supported lane types are `driving`, `shoulder`, `sidewalk`, `biking`, `parking`, `median`, `border`, `restricted` and `bidirectional`. Two constructors cover the common cases:

```
two_way(lanes=2, width=3.5, shoulder=2.5, sidewalk=2.0, median=1.0)
one_way(lanes=3, shoulder_right=2.5)   # reference line is the LEFT edge
```

A road may change cross-section along its length with `at(s, profile, taper)`, where the new profile holds from s onward and is reached over `taper` metres before it; a negative s counts back from the road end. Lanes are paired across the change by type, innermost first, or explicitly by tag. A lane present on one side of the change and absent on the other is widened from, or narrowed to, zero width over the taper, which is how acceleration and deceleration lanes are expressed.

A.5 Geometry Primitives

Reference lines are sequences of four primitives, each mapping onto an OpenDRIVE geometry record without approximation:

- `line(length)` - a straight segment.
- `arc(length, radius)` - a constant-curvature arc; a positive radius bends left, a negative radius bends right.
- `turn(radius, angle_deg)` - the same arc parameterised by the angle turned through, which is usually the quantity a prompt states.
- `spiral(length, start_radius, end_radius)` - a clothoid easing between two curvatures, with `None` denoting straight, so `spiral(20, None, 60)` eases from a tangent into a 60 m curve.

Geometry may be omitted entirely. A road given an `end` pose or port instead of a geometry list has a curve fitted between its endpoints, which is the natural way to write a connecting road whose exact shape is not the point.

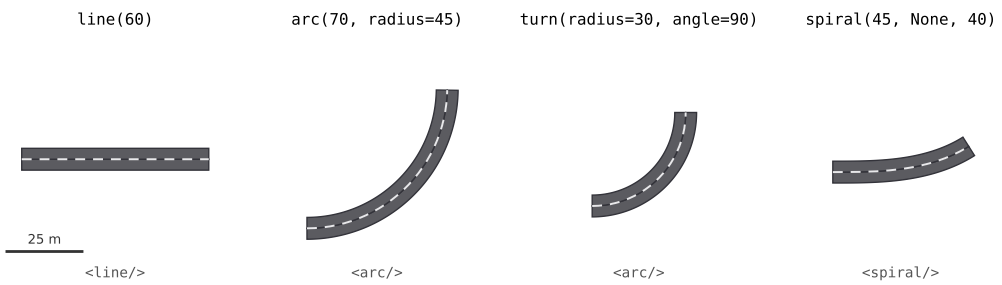


Figure 8: **The four geometry primitives.** Each panel is rendered by compiling the call printed above it, and the OpenDRIVE geometry record it becomes is printed below. `turn` is a second parameterisation of an arc, by the angle turned through rather than by arc length, so it emits `<arc/>` as well. All four panels share one scale. The language provides no compound macros: junctions, roundabouts and ramps are composed from these pieces together with the explicit port connections of Section A.7, which is what keeps the compiler's diagnostics addressed to a named road or port rather than to a black-box template.

A.6 Elevation

Elevation is specified on the road as a small set of height knots, either a single value, a pair of endpoint values, or a list `[(s, z), ...]` in which `"end"` denotes the road end and a negative s counts back from it. The profile eases between knots and is level at each of them, so

```
z=[(0, 0), (70, 6), (-70, 6), ("end", 0)]
```

describes a bridge that climbs, runs flat across its span, and descends. A road that does not state a height inherits the height of the port it starts at, so chained roads remain level with one another without repetition.

Two properties are enforced. Roads that cross without a junction between them must be vertically separated by at least 4.5 m at the crossing point, and grade is bounded: the peak grade of an eased climb is 1.5 times rise over run, which is warned above 8% and rejected above 15%. Junctions are required to be level, so a ramp climbs on its own road before reaching the junction rather than within it.

A.7 Connectivity

Every road exposes `r.start` and `r.end` as ports, and every port participates in at most one connection. Ports that participate in none are map edges. There are exactly two ways to connect them.

Joints. `L.join(port_a, port_b)` states a one-to-one continuation. It is accepted only if the two positions coincide to within 5 cm, the headings continue straight, and every lane meets a lane of the same type occupying the same lateral interval. When the lanes do not correspond, the diagnostic names the offending lane rather than reporting that the join failed. An explicit mapping `lanes={a: b}` overrides the automatic pairing.

Junctions. `L.junction(name, ports, ...)` states that two or more ports meet, and the compiler synthesises the connecting roads. By default it builds a two-way through road for each pair of mutually opposite arms carrying all of their lanes, and one connecting road for every remaining movement. Left turns are routed through the innermost lanes, right turns through the outermost, and near-straight merges and diverges onto the side they arrive from. Movements may be restricted, either by enumerating the permitted ones or by forbidding individual pairs, and U-turns are excluded unless

requested. Kerbs, shoulders and footways are continued through the junction when no movement turns by more than $35°$, which distinguishes a merge from an intersection.

The consequence is that the language has no separate constructor for a T-junction, a four-way junction or a roundabout. A junction of any arity is the same declaration with a different number of ports, and an unusual intersection is therefore no harder to write than a common one.

A.8 Compilation and Diagnostics

`L.compile()` resolves the authored map into per-road lane sections, synthesises the junction connecting roads, and builds the lane-level directed edge set over which routing is answered. It never repairs the map. Every problem is reported as a diagnostic carrying a severity, the name of the road, joint or junction responsible, a machine-readable code, and a message stating what would fix it. The codes fall into five groups:

- *Connection failures*: `unmatched_lane`, `type_change`, `lane_offset`, `port_reused`, `port_faces_away`, `ports_too_close`, `open_port`.
- *Geometry failures*: `geometry_end`, `loopy_fit`, `tight_radius`, `overlap`, `clearance`.
- *Cross-section defects*: `width_step`, `offset_step`, `lane_ends_abruptly`, `lane_starts_abruptly`, `profile_schedule`.
- *Elevation*: `grade`, `elevation_knots`, `junction_not_level`, `grade_separated`.
- *Junction interior*: `lane_no_movement`, `connector_loop`, `loopy_connector`, `tight_connector`, `dead_end`.

Errors make the map invalid. Warnings denote defects that are visible but sometimes intended, such as an abrupt lane end on a road whose prompt asks for exactly that, and are therefore reported rather than enforced.

A.9 Routing and Validation

`net.route(road_a, lane_a, road_b, lane_b)` answers whether a movement is drivable in the compiled lane graph, traversing junction connecting roads and respecting lane direction. Because a map may satisfy the compiler and still fail in the simulator, the same movements are additionally checked against CARLA's own OpenDRIVE parser, loaded offline from the exported file with no simulator process. A movement is considered verified only when both agree. Routes that must be *impossible*, such as travelling against a one-way arm, are expressed in the same form with the expected outcome negated, so that a map which is over-permissive is caught as readily as one that is disconnected.

A.10 Rendering

`net.render_to_file(path)` draws a plan view of the compiled map, with lanes filled by type, junction interiors and their connecting roads distinguished from ordinary roads, and open ports marked. Roads at greater height are drawn over lower ones with a bold outline and labelled with their height range, which is what makes a grade-separated structure legible in a top-down image.

A.11 OpenDRIVE Export

`net.export_xodr(path)` writes an OpenDRIVE 1.4 file. Reference-line primitives are emitted directly as their corresponding geometry records, lane sections follow the profile schedule, elevation is written as the cubic polynomials implied by the height knots, and each junction becomes an OpenDRIVE junction whose connections enumerate the synthesised connecting roads and their lane links. The resulting file is what is loaded into CARLA and what is evaluated in Section 4.2.

A.12 Limitations and Unsupported Features

The language models road surface, lane structure, connectivity and elevation. It does not model traffic control devices, which are added in a separate pass after export, nor lane markings beyond the defaults implied by lane type, nor variable superelevation or crossfall. Junctions are required to be level, so a structure in which the intersection itself is graded must be decomposed into level junctions joined by graded roads. Curve fitting between two poses selects a smooth interpolant and does not attempt to satisfy a specified length, so a road whose length matters should be written with explicit geometry.

B Road Topology Agent Details

B.1 System Prompt

System Prompt

```
You design drivable road maps for CARLA with layout_dsl.  A map is
primitives (roads with cross-sections and reference-line geometry)
plus explicit connections (ports that meet at joints or junctions).
Most of the work is the decomposition:  which cross-sections exist,
which roads carry them, where each road's ends are, and which ends
meet.

Workflow
    1. Call layout_reference() once.
    2. Write the decomposition (profiles, roads, connections) as a
       comment block, then the program.  The program must build a
       Layout and bind net = L.compile().
    3. run_layout(code).  Fix every error.  Warnings describe visible
       defects (kinks, width steps, overlapping arms); fix them
       unless the prompt asks for that feature.
    4. describe(name) when you need a port's exact pose or lane
       positions.
    5. check_routes() with the movements the prompt implies, and ones
       that must be impossible, for example against a one-way road.
       Both the compiled lane graph and CARLA's parser must agree.
    6. finish(summary) once the map compiles with no errors, the
       routes pass and the picture matches the prompt.

Rules.  Use print() to inspect, never raise SystemExit.  Do not
import anything else.  Keep road names short and unique (letters,
digits, _ and -).
```

B.2 Layout Agent Tool Calls

The road topology stage is a single LLM-backed agent, `LayoutAgent`, which converts a natural-language road-scene description into an executable `layout_dsl` program and repairs that program against the compiler until the map is drivable. It communicates with the LLM through a fixed set of *tool calls*, structured function invocations whose schemas are derived automatically from Python type annotations and registered with the underlying API at construction time. The agent routes to the correct backend by model name: names beginning with `claude` use the Anthropic Messages API, and all other names use the OpenAI-compatible Responses API.

There is no separate critique agent. The earlier version of this stage paired a `DSLGeneratorAgent` with a `DSLCritiqueAgent` that scored the program, checked its syntax, and critiqued it from the code and the rendered image. That arrangement asked a second language model to infer, from the program text, defects that the compiler can state exactly: which joint failed to pair, which lane found no partner of its own type, which junction arms overlap. Replacing the critic with attributed compiler diagnostics removes a model call from every iteration and replaces an opinion with a fact.

LayoutAgent tools. The agent exposes five tool calls that together implement a *decompose – compile – inspect – verify* loop.

`layout_reference()` → `str` Returns the `layout_dsl` API, the modelling rules, and a set of worked example programs covering an asymmetric T-junction, an on-ramp merge, a five-way mixed junction, an asymmetric corridor and an overpass. The agent is instructed to call this once, before writing anything, so that the language definition is in context rather than recalled.

`run_layout(code: str)` → `report + image` Executes a complete program, in which the `layout_dsl` names are pre-imported and which must bind `net = L.compile()`, and returns a JSON report together with a rendering of the map. The report contains the errors and warnings, each attributed to the joint, junction or road responsible; the ports left open; the roads with their port poses; the connecting roads the compiler synthesised for each junction; the pieces of the traffic-lane graph; and anything the program printed. This is the tool that carries the feedback signal: a failure is reported as a named object and a stated remedy rather than as an aggregate score.

`describe(name: str)` → `str` Returns, for one road, its ports with their poses and the lateral interval and flow direction of each lane, together with its geometry and lane sections; or, for one junction, its ports and the connecting roads built between them. It reads from the last successful run, and exists so that a geometric question can be answered exactly instead of estimated from the rendered picture.

`check_routes(routes_json: str)` → `str` Takes a list of movements, each written as a source and destination road, lane and position along the road, together with the expected outcome. Every movement is evaluated twice, once in the compiled lane graph and once in CARLA's own OpenDRIVE parser loaded offline from the exported file, and a movement counts as verified only when the two agree. Because the expected outcome may be negative, the tool also tests movements that must be *impossible*, such as travelling against a one-way arm, so that an over-permissive map is caught as readily as a disconnected one. The response additionally reports dead-end lanes, U-turn connectors, and the connected pieces of CARLA's driving route graph.

`finish(summary: str)` → `str` Ends the design and records a short statement of how the map was decomposed and what was verified. The call is refused unless the most recent `run_layout` compiled with no errors, so the agent cannot terminate on a map that does not compile.

Termination and budgets. A run ends in one of three ways. The agent calls `finish` on a clean compilation, which is the intended outcome; it exhausts a cap of K tool-calling rounds; or it exhausts a total output-token budget across all calls. In both budget cases the last successfully compiled map is kept and the run is recorded as not having converged, which is how the non-converged cases in Section 4.2 are counted. Automatic termination on the first clean compilation whose route checks pass was implemented and then disabled by default: runs became shorter but the maps became worse, because the agent stopped before comparing the rendered picture against the prompt, and structures implied by the prompt but not required for drivability, such as the blocks in "each arm contains a row of blocks", were never built.

C Foreground Agent Details

System Prompt — Asset Placer (Node-Hopping Pipeline)

```
You are an expert CARLA autonomous-driving scenario designer and
simulation engineer.  In a single pipeline you will:

PHASE A -- Explore the road graph to understand the topology.  Use
as few tool calls as possible; the necessary information is usually
found in the first few calls, especially around intersections.
PHASE B -- Submit a placement plan:  exactly where every actor
spawns.
```

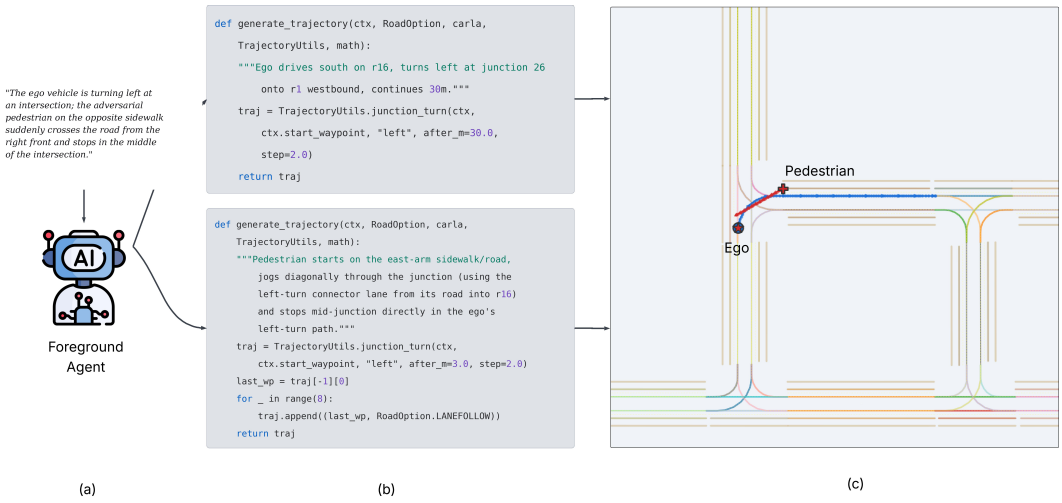


Figure 9: **Foreground asset placement and trajectory generation.** *Left:* from a natural-language scenario the agent writes an executable trajectory function per actor, built on map-aware utilities such as route planning and lane following, with a fallback to lane following on the spawn lane when no candidate trajectory is accepted. *Right:* the trajectories those functions produce, rendered on the map.

PHASE C -- For *every* actor, write and verify a Python trajectory function.

PHASE A - Road-graph exploration tools

- _tool_get_intersections -- list intersection nodes (start here)
- _tool_get_road_info -- all lanes for a given road_id
- _tool_find_nearby_nodes -- nearby graph nodes sorted by distance
- _tool_get_lane_neighbors -- left/right neighbour lanes
- _tool_get_road_successors -- successor roads from an edge exit

PHASE B -- Placement plan

Call _tool_submit_placement_plan(plan_json) where plan_json is a JSON array of objects with the following fields:

type	str	vehicle|ambulance|police|firetruck|two_wheeler|pedestrian
is_ego	bool	true for exactly *one* actor
road_id	int	
lane_id	int	
section_id	int	default 0
waypoint_index	int	0-based index along the lane
motion_description	str	plain-English motion brief

Rules:

- Exactly one is_ego=true actor (type="vehicle").
- Actors spread realistically -- different lanes, varied distances.
- All road_id/lane_id values must exist in the graph.

PHASE C -- Scene trajectory generation

Every actor is planned in one conversation. The opening message already gives each actor's spawn lane and pose, requested motion, lane neighbours, where its spawn lane leads and its distance to the ego, together with nearby traffic lights and signs and each light's schedule from the scenario start, so no per-actor lookup is needed to begin.

1. submit_trajectory(label, python_code, motion_json) -- runs the actor's code, gates it on staying on the drivable surface, and validates its motion specification, in one call. Fix and resubmit on failure.

2. check_interactions() -- simulates all accepted actors under their motion specifications and reports starts, stops, closest approaches and who reaches each crossing when, so the scenario's interaction can be made to occur rather than assumed.
3. simulate_rollout(horizon_s) -- runs the scene closed-loop in CARLA with the real executors and reports collisions, signal violations and blocked or fallen actors.
4. finish() -- refused unless every actor is accepted and a rollout of exactly this plan completed without blocking problems.

You **must** have every actor accepted and the final plan rolled out before finishing.

Trajectory function requirements
Exact signature (do not deviate):

```
def generate_trajectory(ctx, RoadOption, carla, TrajectoryUtils,
math):
"""<short docstring>"""
...
return trajectory  # List[carla.Location], min 20
```

All imports are injected -- do **not** write import statements.

ctx fields: world, wmap, grp, road_id, lane_id, section_id, waypoint_index, start_waypoint, ego_waypoint, compressed_graph.

Why List[carla.Location] (not waypoints)? Waypoints carry an orientation baked into the road graph. Returning bare Location objects lets the controller compute heading from the tangent between consecutive points, so actors always face the direction they are actually travelling -- avoiding the 180° spin that waypoint-snapping can cause.

Coding rules:
- Return List[carla.Location], minimum 20 entries.
- **No hardcoded** (x, y, z) coordinates -- derive every location from ctx.start_waypoint, TrajectoryUtils helpers, or graph look-ups.
- Wrap the body in a try/except that falls back to TrajectoryUtils.follow_lane(ctx, ctx.start_waypoint, 30).
- Output *only* the Python function -- no markdown fences, no prose.

Available TrajectoryUtils helpers (use *exact* parameter names shown):
- follow_lane(ctx, start_wp, n_waypoints, step=2.0) -- advance along the current lane from start_wp.
- route_between(ctx, origin_loc, dest_loc) -- plan a road-graph route between two carla.Location objects.
- zigzag(ctx, start_wp, n_cycles, cycle_length, lateral_offset, step=1.0) -- weave side-to-side while advancing.
- circle(ctx, center, radius, n_points, z=None) -- closed circular path.
- offset_lane(ctx, locations, lateral_m) -- shift a path laterally by ±lateral_m metres.
- loop(locations, n_repeats) -- repeat a path n times end-to-end.
- get_waypoints_on_lane(ctx, road_id, lane_id, section_id=0) -- all waypoints on a lane edge as locations.

Example 1 -- simple lane following:

```
def generate_trajectory(ctx, RoadOption, carla, TrajectoryUtils,
math):
"""Drive straight along the current lane for  60 m."""
try:
return TrajectoryUtils.follow_lane(
ctx, ctx.start_waypoint, 30, step=2.0)
except Exception:
return TrajectoryUtils.follow_lane(
ctx, ctx.start_waypoint, 30)
```

Example 2 -- zigzag / weaving:

```
def generate_trajectory(ctx, RoadOption, carla, TrajectoryUtils,
math):
"""Cyclist weaves side-to-side."""
try:
return TrajectoryUtils.zigzag(
ctx, ctx.start_waypoint, n_cycles=4,
cycle_length=6.0, lateral_offset=1.2, step=0.8)
except Exception:
return TrajectoryUtils.follow_lane(
ctx, ctx.start_waypoint, 30)
```

Example 3 -- stationary actor:

```
def generate_trajectory(ctx, RoadOption, carla, TrajectoryUtils,
math):
"""Actor remains stationary (parked / blocked)."""
loc = ctx.start_waypoint.transform.location
return [loc] * 30
```

System Prompt — Scene Trajectory Agent

You are an expert CARLA autonomous-driving simulation engineer. Write a single Python function generate_trajectory that produces a realistic trajectory for one simulated actor.

EXACT SIGNATURE (do not deviate)

```
def generate_trajectory(ctx, RoadOption, carla,
TrajectoryUtils, math):
    """<short docstring>"""
    ...
    return trajectory  # List[Tuple[carla.Waypoint,
RoadOption]]
```

All imports are injected -- do **not** write import statements.

ctx fields: world, wmap, grp, road_id, lane_id, section_id, waypoint_index, start_waypoint, ego_waypoint, compressed_graph.

[The TrajectoryUtils API listing is injected here; it is given in Appendix C.1.]

WORKFLOW -- you MUST follow this order

1. Call _tool_get_spawn_info to confirm spawn location.
2. Optionally call graph exploration tools as needed.
3. Call _tool_submit_trajectory_code with your function. The tool only syntax-checks the code; it does *not* run it.
4. Call _tool_verify_and_run_code to actually execute it. If it returns an error, read the traceback, fix the code, re-submit with _tool_submit_trajectory_code, then verify again.
5. Call _tool_visualise_trajectory to see a BEV plot. Check that the path matches the intended motion description, that it stays on the road and crosses lanes in the right place, and that the road-option annotations are correct. If the trajectory looks wrong, fix the code and repeat from step 3.
6. Only stop once _tool_visualise_trajectory shows a plausible path.

STAY ON THE ROAD (enforced -- _tool_verify_and_run_code rejects violations)

- The trajectory must start at ctx.start_waypoint (the actor spawns there).
- Every waypoint, and the straight line between consecutive waypoints, must lie on a lane the actor may use: vehicles on driving lanes; cyclists and pedestrians also on sidewalks and shoulders (crossing the road is fine). Anywhere else is empty space in a generated map: the actor falls through it.

- Consecutive waypoints at most MAX_GAP_M metres apart.
- Derive points from map waypoints (next(), get_left_lane()/get_right_lane(), junction lane waypoints); never add raw metre offsets that leave the lane.

CODING RULES
- Return List[Tuple[carla.Waypoint, RoadOption]], minimum 20 entries.
- No hardcoded (x, y, z) -- derive from ctx.start_waypoint or graph lookups.
- Circles → TrajectoryUtils.interpolate_circle.
- Lane changes → get_left_lane()/get_right_lane() with CHANGELANELEFT/CHANGELANERIGHT.
- Turns → carla.Junction.get_waypoints() filtered by cross-product.
- Wrap the body in a try/except that falls back to TrajectoryUtils.follow_lane(ctx, ctx.start_waypoint, 30) on any error.
- Output *only* the Python function -- no markdown fences, no prose.

C.1 Tool Definitions: Foreground Agents

The foreground stage runs two agents. The *Asset Placer* decides which actors exist and where they start, exploring the road graph on demand rather than receiving it in full (*node-hopping*); its tools are given in Phases A and B below. The *Scene Trajectory Agent* then writes the trajectory and motion specification of every actor in a single conversation, and verifies the resulting scene against the simulator; its tools are given in Phase C. Planning every actor in one conversation, rather than one agent per actor, is what allows an interaction between two actors to be timed; the rationale is given in Section 3.2.

Phase A — Road-Graph Exploration Tools

`_tool_get_intersections(max_results=15)`. Returns up to `max_results` intersection nodes from the road graph (segments where `is_junction=True`), sorted by their x-coordinate. Each entry contains `road_id`, `lane_id`, entry/exit world-space coordinates (metres), and the number of waypoints on that segment. This is the recommended *first call* in any scenario session: intersections are the most interesting locations for multi-actor placement.

`_tool_get_road_info(road_id)`. Returns all lane segments belonging to `road_id`, including per-lane `lane_id`, entry/exit coordinates, `num_waypoints`, and an `intersection` Boolean flag. Useful for understanding the full lateral cross-section of a road (e.g. to distinguish opposing-direction lanes or identify merge points).

`_tool_find_nearby_nodes(road_id, lane_id, max_results=12)`. Returns up to `max_results` graph nodes sorted by Euclidean distance from the *entry* point of `(road_id, lane_id)`. Each result includes `distance_m`, the `intersection` flag, and entry coordinates. Used to discover adjacent roads, merge/diverge points, and nearby intersection arms without a full graph traversal, implementing the *node-hopping* strategy.

`_tool_get_lane_neighbors(road_id, lane_id)`. Queries the CARLA map API for the immediately adjacent drivable lanes to the left and right of the entry waypoint of `(road_id, lane_id)`. Returns `road_id`, `lane_id`, `lane_type`, and world-space location for each neighbour (or `null` if none exists). Essential for multi-lane placement and lane-change scenarios.

`_tool_get_road_successors(road_id, lane_id)`. Traverses the road graph from the *exit* of `(road_id, lane_id)` and returns all reachable successor edges. Each successor includes `road_id`, `lane_id`, edge `type` (`LANEFOLLOW`, `CHANGELANELEFT`, `CHANGELANERIGHT`, etc.), and the `intersection` flag. Used to trace valid routes through intersections and to reason about topological connectivity before writing trajectory code.

PHASE B — PLACEMENT PLAN

`_tool_submit_placement_plan(plan_json).` Accepts a JSON-serialised array of actor placement objects and validates the plan. Each object must contain:

- `type` — one of `vehicle`, `ambulance`, `police`, `firetruck`, `two_wheeler`, `pedestrian`;
- `is_ego` — Boolean, exactly *one* actor set to `true`;
- `road_id`, `lane_id` — must exist in the graph;
- `section_id` — integer, default 0;
- `waypoint_index` — 0-based index along the lane edge path;
- `motion_description` — plain-English brief used by Phase C to guide trajectory generation.

On success the tool resolves `carla.Waypoint` objects for every actor, constructs `TrajectoryContext` instances (one per actor), and returns a confirmation message listing each `actor_index`. The agent must then proceed to Phase C without delay. On failure a `ValidationError` string is returned.

PHASE C — SCENE TRAJECTORY TOOLS

The Scene Trajectory Agent receives, in its opening message, every actor with its type, spawn lane and pose, requested motion, lane neighbours, where its spawn lane leads and its distance to the ego, together with the traffic lights and signs near the actors and each light's measured schedule from the scenario start. It therefore needs no per-actor exploration round trip to discover the scene, and its tools are concerned with committing, checking and verifying.

`submit_trajectory(label, python_code, motion_json).` Runs one actor's trajectory code against that actor's live context, gates the result on staying within the drivable surface, and validates the accompanying motion specification, in a single call. The code is the complete source of `generate_trajectory(ctx, RoadOption, carla, TrajectoryUtils, math)`. The motion specification is a JSON object giving the speed profile, acceleration and deceleration limits, the start delay or trigger, intermediate stops, the end behaviour, a reactivity flag and light states. On failure the traceback and the failing check are returned and the agent repairs the actor.

`check_interactions().` Simulates every accepted actor under its motion specification and reports how the scene plays out: when each actor starts, stops and finishes; for every pair, the closest approach with both actors' speeds and states; and, where paths cross, the time at which each actor reaches the crossing. This is the tool that establishes whether the scenario's interaction, such as a crossing in front of the ego, a cut-in or an overtake, actually occurs, as opposed to merely whether the paths intersect somewhere.

`simulate_rollout(horizon_s).` Runs the whole scene closed-loop in CARLA with rendering disabled, for a default horizon of 40 simulated seconds. Every actor including the ego is spawned and driven by the same executors that data collection and evaluation use, starting from the scenario-start signal state. Returns per-actor progress, collisions, red-light and stop-sign violations, and blocked or fallen actors.

`visualise_scene().` Draws every accepted trajectory on the road map in one image, with a start dot, a direction arrow and a label per actor.

`inspect_graph_node(road_id, lane_id, section_id).` Returns the topology of one lane: entry and exit coordinates, length, junction flag and successor lanes. A section id of -1 returns every section of that lane.

`find_nearby_nodes(label, max_results).` Returns the graph lanes nearest to one actor's spawn point, sorted by distance.

`finish().` Ends planning. It is refused unless every actor has an accepted trajectory and a `simulate_rollout` of exactly the current plan has completed without blocking problems, so a scenario cannot be returned that has not been run in the simulator.

D Background Agent Details

D.1 System Prompt

System Prompt

You are the Designer Agent for a city sub-block planner. Your job is to populate a sub-block (a free-space area between roads) with buildings, apartments, trees, and other assets according to the provided design brief.

Key design principles:

1. **Peripheral Density.** The boundary ring ($\leq$ 15 m from the edge) must be densely packed so the sub-block looks full from the outside. There should be *no gaps* along the edge of the peripheral ring.
2. **No Roads.** Sub-blocks contain *only* buildings, vegetation, and civic/utility structures -- never roads.
3. **Orientation.** Orient assets so their long axis runs parallel to the nearest boundary edge for a tidy street-front appearance.
4. **Variety.** Mix large anchor buildings with smaller units and vegetation.
5. **No Overlaps.** Always validate placements. Use the overlap-check tool before committing a batch.

Workflow -- when modifying the scene, follow these steps:

(a) Call get_scene_summary to understand the current state.

(b) Use get_peripheral_positions and get_interior_positions for candidate spots.

(c) Use place_asset, place_asset_batch, or place_asset_grid to add assets.

(d) Call validate_scene to check for violations.

Available asset IDs:

residential_apartment_big, residential_apartment_small, pine_tree, post_office, residential_family_house, chicago_building, city_building, coconut_tree_low_poly, industrial_building, nyc_bronx_buildings, oak_tree, textured_building_with_interior, tree_low_poly, tree

Use *tool calls exclusively* to interact with the scene.

D.2 Helper Data Structures

Before presenting the tool calls, we describe the two core helper classes that all tools operate on.

D.2.1 SceneState

`SceneState` (defined in `scene_state.py`) is the shared, mutable state object that is passed to the Designer Agent at construction time. It owns the sub-block geometry, the placed asset registry, and all validation logic.

Key attributes:

- `boundary` — A closed polygon expressed as a list of (x, y) tuples in scene-local coordinates (metres). All asset corners must lie strictly inside this polygon.

- `assets` — A `dict` mapping `instance_id` → `PlacedAsset`. The Designer Agent reads and mutates this mapping through the tool API.
- `metadata` — Free-form `dict`; carries the design brief and block type forwarded by the orchestrator.
- `forbidden_zones` — A list of convex polygons (each a list of (x, y) corners) that mark road segments running through the loop. Any asset whose bounding rectangle overlaps a forbidden zone is rejected by `validate_placement()`.
- `area`, `centroid`, `bbox` — Pre-computed scalar / tuple properties derived from the boundary polygon via the Shoelace formula and a standard bounding-box pass.

Derived metrics available to the agent:

- `coverage_ratio()` — Ratio of summed asset footprint area to total sub-block area, clamped to $[0, 1]$.
- `peripheral_density(band_width)` — Fraction of placed assets whose centre lies within `band_width` m of the boundary (default $15\,\mathrm{m}$).
- `suggest_peripheral_positions(band_width, step)` — Returns a list of (x, y) candidate positions inside the peripheral ring, sampled on a grid with resolution `step` and filtered to exclude road forbidden zones.
- `suggest_interior_positions(margin, step)` — Returns candidate (x, y) positions that lie strictly more than `margin` m from the boundary, again excluding forbidden zones.

Road-zone detection. When a `GlobalRoutePlanner` (GRP) and the world-space loop polygon are provided at construction time, `SceneState` automatically calls `extract_road_zones_from_grp()`, which:

1. Iterates over every segment in `grp.compressed_graph`.
2. Skips non-drivable lane types (sidewalks, parking, shoulders).
3. Tests each segment endpoint pair against the loop polygon using the ray-casting algorithm and a segment–polygon intersection test.
4. For every overlapping segment, constructs a rectangular *forbidden strip* of half-width w_{road} (default $10\,\mathrm{m}$) aligned with the segment direction and converts it to scene-local coordinates by subtracting $(loop_min_x,\ loop_min_y)$.

D.2.2 PLACEDASSET

`PlacedAsset` is a `@dataclass` that records one placed instance.

Fields:

- `instance_id` (str) — Auto-generated 8-character UUID fragment; used as the key in `SceneState.assets`.
- `asset_id` (str) — Catalogue identifier (e.g. `residential_apartment_big`).
- `x`, `y` (float) — Centre position in scene-local metres.
- `orientation` (float) — Rotation in degrees, measured counter-clockwise from the $+x$ axis.
- `width`, `length` (float) — Footprint extents along the local x- and y-axes respectively, copied from the asset catalogue.
- `height` (float) — Vertical extent (metres).
- `asset_type`, `category`, `color` (str) — Copied from the catalogue; used by the visualiser.
- `label` (str) — Optional human-readable override.

Geometry method: `corners()` returns the four world-space corners of the oriented bounding rectangle:

$$\mathbf{p}_i = \begin{pmatrix} x \\ y \end{pmatrix} + R(\theta) \begin{pmatrix} \pm w/2 \\ \pm l/2 \end{pmatrix}, \qquad R(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

where w = `width`, l = `length`, and θ = `orientation` (radians). These corners are used by the overlap checker (Separating Axis Theorem) and the boundary validator (ray-casting per corner).

Asset catalogue. Catalogue data are loaded once from `assets.json` via `load_asset_catalogue()`. Each entry carries: `id`, `label`, `type`, `category`, `width`, `length`, `height`, `z_offset`, `angle_offset`, `color`, `description`, `glb_path`, and a `scale` factor used by the 3-D renderer.

D.3 Tool Call Definitions

The Designer Agent exposes the following tool calls. Each tool is implemented as a branch inside `DesignerAgent.execute_tool()`. Tools are registered in both the Anthropic (`input_schema` format) and OpenAI (`parameters` format) schema variants.

D.3.1 Observation Tools

`get_scene_summary` *Parameters: none.*
Returns a JSON object summarising the current scene state:

- `num_assets` — total count of placed instances.
- `asset_type_counts` — per-type breakdown.
- `coverage_ratio` — fraction of boundary area covered by asset footprints (range $[0, 1]$).
- `peripheral_density` — fraction of assets within $15\,\mathrm{m}$ of the boundary.
- `area_m2` — total sub-block area in m^2.
- `centroid` — $[x, y]$ of the boundary centroid.
- `num_road_zones`, `total_road_zone_area_m2` — forbidden-zone statistics appended by `get_road_zone_summary()`.

This tool is typically called first so the agent can calibrate how densely it needs to fill the remaining space.

`get_asset_catalogue` *Parameters: none.*
Returns the full JSON asset catalogue from `assets.json`. Useful when the agent needs to look up the physical dimensions of an asset before choosing a placement grid spacing.

`list_placed_assets` *Parameters: none.*
Returns a JSON array of all current `PlacedAsset` instances (via `PlacedAsset.to_dict()`), including positions, orientations, and dimensions. Used when the agent needs to inspect existing placements before making targeted modifications.

`get_boundary_info` *Parameters: none.*
Returns the boundary polygon vertices, total area, centroid coordinates, bounding box, and vertex count. Useful for computing edge directions before aligning assets.

D.3.2 Position Helper Tools

`get_peripheral_positions` *Parameters:* `band_width` *(float, default* $15\,\mathrm{m}$*),* `step` *(float, default* $6\,\mathrm{m}$*).*
Calls `SceneState.suggest_peripheral_positions()` and returns up to 200 candidate (x, y) positions sampled on a grid within `band_width` m of the boundary. Points inside any road forbidden zone are excluded automatically. The agent uses these positions to densely fill the peripheral ring in accordance with design principle 1.

get_interior_positions *Parameters:* `margin` *(float, default* 12 m*),* `step` *(float, default* 8 m*).*
Calls `SceneState.suggest_interior_positions()` and returns candidate positions that are strictly more than `margin` m from the boundary. Forbidden-zone positions are excluded.

check_position_valid *Parameters:* `asset_id` *(str),* `x` *(float),* `y` *(float),* `orientation` *(float, optional).*
Performs a trial placement via `SceneState.place_asset()` with `allow_overlap=False` and `require_inside=True`, then immediately removes the tentative instance if successful. Returns `{"valid": true}` or `{"valid": false, "message": ...}`. Useful for a lightweight pre-check before committing a whole batch.

D.3.3 PLACEMENT TOOLS

place_asset *Parameters:* `asset_id` *(str),* `x` *(float),* `y` *(float),* `orientation` *(float, optional),* `label` *(str, optional).*
Places a single asset instance. Calls `SceneState.place_asset()` which:

1. Looks up the asset dimensions from the catalogue.
2. Computes the four oriented corners.
3. Validates that all corners are inside the boundary polygon (ray-casting) and outside every road forbidden zone.
4. Runs the Separating Axis Theorem against all existing assets to rule out overlaps.

Returns `{success, instance_id, message}`.

place_asset_batch *Parameters:* `placements` *(array of objects, each with* `asset_id`*,* `x`*,* `y`*, and optional* `orientation` */* `label`*),* `skip_overlaps` *(bool, default* `true`*).*
Processes each item in `placements` sequentially via `place_asset()`. When `skip_overlaps=true`, items that fail the overlap check are silently skipped rather than causing the whole batch to abort. Returns a list of per-item results plus aggregate `placed` / `total` counts. The primary tool for efficiently populating the peripheral ring in a single call.

place_asset_grid *Parameters:* `asset_id` *(str),* `origin_x`*,* `origin_y` *(float),* `rows`*,* `cols` *(int),* `row_spacing`*,* `col_spacing` *(float),* `orientation` *(float, optional),* `label_prefix` *(str, optional).*
Places a rectangular grid of *rows* $\times$ *cols* copies of `asset_id` starting at the origin, advancing by `col_spacing` along x and `row_spacing` along y. Each cell is labelled with `label_prefix` `row-col`. Returns `{placed, failed, rows, cols}`.

place_asset_row *Parameters:* `asset_id` *(str),* `start_x`*,* `start_y` *(float),* `count` *(int),* `dx`*,* `dy` *(float),* `orientation` *(float, optional),* `label_prefix` *(str, optional).*
Places `count` copies of `asset_id` in a straight line, advancing by $(dx,\ dy)$ per copy. Useful for trees along a path or buildings along a diagonal edge.

fill_peripheral_ring *Parameters:* `asset_id` *(str),* `offset` *(float, default* $= w/2 +$ 1 m*),* `spacing` *(float, default* 2 m*).*
Iterates over every edge of the boundary polygon and packs copies of the specified asset along each edge. For edge $\mathbf{a} \to \mathbf{b}$ (length L, unit tangent $\hat{u}$, inward normal $\hat{n}$):

$$\mathbf{p}_k = \mathbf{a} + \left(\tfrac{L-(n-1)s}{2} + k\,s\right)\hat{u} + d_{\text{offset}}\,\hat{n}, \qquad k = 0, \ldots, n-1,$$

where $s = w_{\text{asset}} + \textit{spacing}$ is the step size and $n = \lfloor L/s \rfloor$ is the number of copies per edge. The asset orientation is set to the edge direction angle $\alpha = \arctan 2(\hat{u}_y, \hat{u}_x)$ so assets face the road. Invalid positions (outside boundary) are skipped.

D.3.4 MODIFICATION TOOLS

move_asset *Parameters:* `instance_id` *(str),* `x` *(float),* `y` *(float),* `orientation` *(float, optional).*

Moves an existing asset to a new (x, y) and optionally updates its rotation. The new position is validated; if it fails, the original position is restored and the error is returned.

`remove_asset` *Parameters:* `instance_id` *(str).*
Removes a single asset instance from `SceneState.assets`. Returns `{success, message}`.

`remove_assets_by_type` *Parameters:* `asset_id` *(str).*
Removes *all* placed instances whose `asset_id` matches the given catalogue ID. Returns the count of removed instances.

`clear_scene` *Parameters: none.*
Calls `SceneState.clear_all()`, deleting every asset instance. Useful when the agent decides to start the layout from scratch, for example after finding that an early large placement leaves no room for the rest of the brief.

D.3.5 Validation Tool

`validate_scene` *Parameters: none.*
Runs the full two-pass validation and returns a JSON report:

- **Boundary violations** — for each asset, calls `validate_placement()` which tests every corner against the boundary polygon and every road forbidden zone (both per-corner ray-casting and SAT rectangle overlap).
- **Overlap violations** — calls `check_all_overlaps()`, which runs SAT between every pair of placed assets in $O(N^2)$ time.
- **Summary** — includes the metrics from `get_scene_summary()`.
- **valid** (bool) — `true` only when `boundary_violations` and `overlaps` are both empty.

The agent is expected to call this tool after each major batch and correct any reported violations before finishing the design pass.

D.4 Free Loop Detection via Planar Graph Face Tracing

We detect enclosed road-free areas ("free loops") by tracing faces of the planar graph induced by the road topology segments. The procedure has four stages.

Stage 1: Planar graph construction. All segment entry and exit coordinates are snapped to a grid of resolution δ (defaulting to the map's sampling resolution) to merge near-coincident endpoints. Each unique snapped coordinate is assigned a node identifier, and each topology segment s with endpoints u, v contributes an undirected edge $\{u, v\}$ to edge set E. Self-loops ($u = v$ after snapping) are discarded.

Stage 2: Polar-angle adjacency. For each node u with 2D coordinates (x_u, y_u), its neighbor list is sorted by the polar angle

$$\theta_{uv} = \operatorname{atan2}(y_v - y_u,\ x_v - x_u), \tag{2}$$

yielding a clockwise ordering of incident edges around u. This ordering is used by the dart traversal in Stage 3.

Stage 3: Face tracing. We adopt the standard planar graph face-tracing algorithm based on *darts* (directed half-edges). For each undirected edge $\{u, v\}$ we consider both directed darts $(u \to v)$ and $(v \to u)$. Given a dart (u, v), the *next clockwise dart* is defined as

$$\operatorname{next}(u, v) = \big(v,\ \operatorname{prev}_v(u)\big), \tag{3}$$

where $\operatorname{prev}_v(u)$ is the neighbor of v immediately *before* u in the polar-angle sorted list of v's neighbors (i.e., the index $(i - 1) \bmod |\mathcal{N}(v)|$ where i is the position of u). Starting from each unvisited dart, we follow this rule iteratively until the dart is revisited, collecting the visited nodes as a face polygon. All darts are marked visited exactly once, so the algorithm terminates in $O(|E|)$ time and recovers every face of the embedding, including the single unbounded outer face.

Stage 4: Filtering and deduplication. For each raw face F (a cycle of nodes $v_1, \ldots, v_k$), we compute its signed area via the shoelace formula,

$$A(F) = \frac{1}{2} \sum_{i=1}^{k} \big(x_i y_{i+1} - x_{i+1} y_i\big), \tag{4}$$

where indices are taken modulo k. Faces with $|A(F)| < A_{\min}$ are discarded (thin inter-lane strips). Opposite-direction lane segments can yield two near-duplicate faces for the same physical loop; we group faces whose centroids lie within distance τ of one another and retain from each group the face with the smallest absolute area, which corresponds to the tightest inner boundary. The outer (unbounded) face has negative signed area under our CCW convention and is removed. Retained loops with $A(F) > 0$ are optionally shrunk by a margin m to avoid overlap with road surfaces.

Parameters. Table 5 summarises the tunable parameters and their default values.

Table 5: Parameters of the free loop detection algorithm.

Parameter	Default	Description
$A_{\min}$	1200 m^2	Minimum face area to retain
δ	sampling resolution	Endpoint snap distance
τ	30 m	Centroid proximity threshold for deduplication
m	2 m	Inward shrink margin applied to retained loops

D.5 Roadside Plots

Face tracing yields plots only where the road graph encloses a bounded region. A map whose roads enclose nothing, such as a corridor, a T-junction or a grade-separated interchange, produces no faces at all, and on a grid the land outside the outermost road is likewise unbounded and therefore never dressed. Both cases leave visible bare terrain in exactly the places a forward-facing camera looks. We therefore construct a second family of plots directly from the road surface.

Stage 1: Drivable surface. The road surface is approximated as the union of discs placed at 2 m intervals along every lane of every type, driving, sidewalk, shoulder, parking and the rest, each disc of radius equal to half the local lane width plus a clearance margin. Taking the union over all lanes rather than the driving lanes alone ensures that footways and shoulders are treated as road rather than as available ground.

Stage 2: Band extraction. The surface is dilated by a depth δ_p and the undilated surface is subtracted, leaving the band of land within δ_p metres of any road. Subtracting the surface rather than merely offsetting the road centrelines is what keeps the band clear of junction interiors, where several lanes overlap.

Stage 3: Partition and filtering. The band is split into connected components and each component is divided into plots of workable size. Plots below a minimum area are discarded, as are plots whose shape is too thin to accept any catalogue asset. The remaining plots are capped at a maximum count so that a large map does not spawn an unbounded number of agents.

Stage 4: Brief assignment. Each plot is given a textual brief derived from the prompt and from its own geometry, in the same form as the brief attached to a traced face, so that a roadside plot and a city block are indistinguishable to the Designer Agent that dresses them. Both are passed through the same placement pipeline and contribute to the same background set $\mathcal{B}$.

E Evaluation Prompts

E.1 Prompts for Evaluation - Map Generation

The map-generation evaluation uses 250 natural-language prompts in four sets. Every method is run once on every prompt, so the comparison contains no unattempted cells. The sets are constructed to

answer different questions, and the design of the suite is itself part of the evaluation: a suite of only unusual topologies would not show whether a method can still produce an ordinary street network, and a suite of only ordinary ones would not distinguish the methods at all.

Table 6: **Composition of the 250-prompt evaluation suite.** Median prompt length is given in words, and indicates how much geometric specification each set carries.

Set	Purpose	N	Median words
base	Layouts expressible in the shipped towns	87	18
novel	Topologies absent from any shipped town	100	72
ordinary	Plain networks of the kind that dominate real roads	50	64
complex	Several constraints composed in one map	13	96

E.1.1 BASE SET

The 87 base prompts are short descriptions of layouts that could plausibly be assembled from the primitives of an existing town. They span five families: *grid* (20), *arc* (20), *S-curve* (10), *junction topology* (32) and *elevation* (5). Prompts in this set are deliberately terse, with a median of 18 words, so that the method must infer the unstated geometry rather than transcribe it.

G01 An outer square loop with a 2×2 block grid inside, all blocks evenly spaced and fully connected by straight roads.

A01 A 2×2 grid of city blocks where the top edge is replaced by a smooth outward quarter-circle arc connecting the two corners.

S01 A city loop with the bottom having an outward-bulging S-shaped road smoothly connecting the two lower corners.

T02 A 6-way radial junction around a center island.

H01 A 3×3 grid of city blocks on a hillside that rises steadily by about 12 m from the south edge to the north edge, with signalized four-way junctions and two-way roads between the blocks.

E.1.2 NOVEL SET

The 100 novel prompts describe layouts that do not occur in any shipped CARLA town, and are written with enough geometric detail that adherence can be checked rather than judged loosely. They span six families: *composite* (34), which combine two structures in one map; *junction* (15), unusual intersection forms; *tessellation* (14), block patterns such as running bond that admit no four-way junction; *curve* (13); *lanes* (12), asymmetric or changing lane configurations; and *elevation* (12).

n016 A running-bond block pattern: three rows of rectangular 90 m by 45 m blocks, four blocks per row, where each row is shifted by half a block relative to the row below, so every north-south street ends at a T-junction on the next east-west street and there is no four-way junction anywhere.

n043 A 1-2-3-4 junction: a single four-way junction whose arms, clockwise from east, carry 1, 2, 3 and 4 lanes per direction, all 3.5 m wide, each arm 120 m long, joined by an outer loop on which the lane count steps between arms with 50 m tapers.

n055 A terraced grid: a 3-by-2 block layout 360 m wide with three east-west streets on level terraces at 0, 6 and 12 m elevation, joined by four north-south streets that climb the 6 m between terraces over 120 m blocks, junctions all level and grade under 8%.

n030 A wave arterial: a 560 m two-lane-per-direction arterial built as four consecutive 120 m radius arcs alternating left and right, each sweeping 40 degrees, with cross streets meeting it at the inflection points where it is momentarily straight.

E.1.3 ORDINARY SET

The 50 ordinary prompts describe the plain networks that dominate real road systems. They exist so that the comparison is not confined to unusual geometry: a method that succeeds only on exotic

prompts, or only on familiar ones, is distinguishable from one that handles both. The set covers regular grids (15), T-junction feeder patterns (15), curved arterials (10), and smaller groups of crossroads, mixed and skewed grids, one-way couplets and L-bends (10 in total). Prompts are fully specified, with a median of 64 words, so that a map which merely looks plausible can still be shown not to match.

`r001` An ordinary residential street grid: 5 parallel north-south streets spaced 40 m apart cross 5 parallel east-west streets also spaced 40 m apart, every street one lane per direction, meeting at plain four-way junctions with no turn lanes or channelization, with block corners rounded by 5 m radius fillets. The whole grid spans roughly 200×200 m and all streets end at open map edges.

E.1.4 Complex Set

The 13 complex prompts compose several constraints into a single map, typically combining grade separation with a specific movement structure. They are the longest prompts in the suite, with a median of 96 words, and are the hardest for every method evaluated.

`c001` A four-level directional stack interchange where a north-south freeway and an east-west freeway, two lanes per direction each, cross at grade-separated levels 0 m and 24 m; between them four single-lane one-way directional ramps each occupy their own level at 6, 12, 18 and 24 m, peeling directly off the origin freeway with an 80 m radius and keeping at least 5 m vertical clearance from every level they cross.

`c002` A single-point urban interchange: a four-lane-per-direction freeway passes underneath at -7 m on a 160 m sunken section while a two-lane-per-direction surface arterial crosses above through one junction footprint about 70 m across, into which both directions' left-turn movements are routed simultaneously so that the opposing left turns cross each other inside the single junction.

Availability. The prompts above are representative; the complete set of 250, together with the per-prompt outputs of every method, will be released with the code on publication so that the evaluation can be reproduced or extended.

E.2 Prompts for Evaluation - Scenarios

To assess the robustness of the ego vehicle's planning and decision-making capabilities, we design a comprehensive set of adversarial scenarios spanning diverse road topologies and agent behaviors. The scenarios are grouped into seven categories based on the driving context and the nature of the adversarial perturbation.

E.2.1 Pedestrian Intrusion on Straight Roads

These scenarios evaluate the ego vehicle's ability to respond to sudden pedestrian appearances while traveling along a straight road. Adversarial pedestrians emerge from occluded locations–such as bus stops, vending machines, driveways, and parked cars–and either stop abruptly or move unpredictably in the ego vehicle's path.

1. The ego vehicle is driving on a straight road when a pedestrian suddenly crosses from the right front and suddenly stops as the ego vehicle approaches.
2. The ego vehicle is driving on a straight road; the adversarial pedestrian stands behind a bus stop on the right front, then suddenly sprints out onto the road in front of the ego vehicle and stops.
3. The ego vehicle is driving on a straight road; the adversarial pedestrian is hidden behind a vending machine on the right front, and abruptly dashes out onto the road, and stops directly in the path of the ego.
4. The ego vehicle is driving on a straight road; the adversarial pedestrian appears from a driveway on the left and suddenly stops and walks diagonally.
5. The ego vehicle is driving on a straight road; the adversarial pedestrian suddenly appears from behind a parked car on the right front and suddenly stops.

E.2.2 INTERSECTION TURNS WITH ADVERSARIAL AGENTS

These scenarios challenge the ego vehicle during left and right turns at intersections. Adversarial pedestrians, cyclists, and motorcyclists appear unexpectedly or exhibit deceptive behaviors–such as feinting a crossing, stopping in the middle of the intersection, or dismounting–designed to disrupt the ego's turning maneuvers.

1. The ego vehicle is turning left at an intersection; the adversarial motorcyclist on the right front pretends to cross the road but brakes abruptly at the edge of the road, causing confusion.
2. The ego vehicle is turning left at an intersection; the adversarial pedestrian on the opposite sidewalk suddenly crosses the road from the right front and stops in the middle of the intersection.
3. The ego vehicle is turning right at an intersection; the adversarial motorcyclist on the opposite sidewalk abruptly crosses the road from the right front and comes to a halt in the center of the intersection.
4. The ego vehicle is turning right at an intersection; the adversarial pedestrian on the left front suddenly crosses the road and stops in the middle of the intersection, blocking the ego vehicle's path.
5. The ego vehicle is turning left at an intersection; the adversarial cyclist on the left front suddenly stops in the middle of the intersection and dismounts, obstructing the ego vehicle's path.

E.2.3 LANE CHANGE INTERFERENCE

This category evaluates the ego vehicle's ability to execute safe lane changes when adversarial vehicles actively interfere. Behaviors include sudden merges back into the original lane, speed-matching to close gaps, abrupt braking, and unpredictable weaving that makes the target lane difficult to assess.

1. The ego vehicle is attempting to change lanes to avoid a slow-moving leading vehicle; the adversarial car in the target lane suddenly merges into the ego vehicle's original lane, blocking the ego vehicle from returning to its initial position.
2. The ego vehicle is changing to the right lane; the adversarial car is driving parallel to the ego and blocking its path.
3. The ego vehicle is attempting to change lanes to avoid a slow-moving leading vehicle; the adversarial car in the target lane suddenly slows down, matching the speed of the leading vehicle, and effectively blocking the ego vehicle from completing the lane change.
4. The ego vehicle is performing a lane change to evade a slow-moving vehicle; the adversarial car in the target lane on the right front suddenly brakes, causing the ego vehicle to react quickly to avoid a collision.
5. The ego vehicle is preparing to change lanes to evade a slow-moving leading vehicle; the adversarial car in the target lane starts weaving between lanes, making it difficult for the ego vehicle to predict its position and safely execute the lane change.

E.2.4 OPPOSITE-LANE BYPASS WITH ONCOMING HAZARDS

These scenarios place the ego vehicle in a situation where it must temporarily use the oncoming lane to bypass a parked, lane-blocking vehicle. While executing this maneuver, the ego encounters unexpected hazards–jaywalking pedestrians, swerving motorcyclists, pedestrians stopping abruptly, unsignaled turning vehicles, and suddenly accelerating oncoming cars–each requiring rapid reactive behavior.

1. The ego approaches a parked car that is blocking its lane and must use the opposite lane to bypass the vehicle, cautiously monitoring oncoming traffic, and suddenly encounters a jaywalking pedestrian, requiring the ego to quickly assess the situation and respond appropriately to avoid a collision.
2. The ego encounters a parked car blocking its lane and must use the opposite lane to bypass the vehicle, carefully assessing the situation and yielding to oncoming traffic, when an

oncoming motorcyclist swerves into the lane unexpectedly, necessitating the ego to brake or maneuver to avoid a potential accident.

3. The ego encounters a parked car blocking its lane and must use the opposite lane to bypass the vehicle when an oncoming pedestrian enters the lane without warning and suddenly stops, necessitating the ego to brake sharply or steer to avoid hitting the pedestrian.
4. The ego approaches a parked car obstructing its lane and must use the opposite lane to go around when an oncoming car suddenly turns into the ego's path without signaling, requiring the ego to react quickly and take evasive action to prevent a collision.
5. The ego encounters a parked car blocking its lane and must use the opposite lane to bypass the vehicle when an oncoming car suddenly accelerates, closing the gap for the ego to safely return to its lane, necessitating the ego to quickly decide whether to accelerate or brake to avoid a collision.

E.2.5 Intersection Right-of-Way Violations

These scenarios test the ego vehicle when traversing intersections under various right-of-way conditions. Adversarial vehicles run red lights, cut off the ego mid-crossing, execute abrupt turns, or brake suddenly. Both straight-through and turning movements by the ego are included, along with unprotected left-turn yielding scenarios.

1. The ego is driving straight through an intersection when a crossing vehicle runs the red light and unexpectedly accelerates, forcing the ego to quickly reassess the situation and perform a collision avoidance maneuver.
2. The ego vehicle is moving straight through the intersection; the adversarial agent, initially on the left front, runs the red light and makes an abrupt right turn, forcing the ego vehicle to perform a collision avoidance maneuver.
3. The ego vehicle is going straight through the intersection; the adversarial vehicle approaches from the left front and cuts off the ego vehicle.
4. The ego vehicle is moving straight through the intersection; the adversarial agent, initially on the left front, runs the red light and makes an abrupt left turn, forcing the ego vehicle to perform a collision avoidance maneuver.
5. The ego moves straight at an intersection when a crossing vehicle runs the red light from the right and brakes abruptly, causing the ego to rapidly adapt its trajectory and perform a collision avoidance maneuver.
6. The ego starts an unprotected left turn at an intersection while yielding to an oncoming car when the oncoming car's throttle malfunctions, leading to an unexpected acceleration and forcing the ego to quickly modify its turning path to avoid a collision.
7. The ego attempts an unprotected left turn at an intersection while yielding to an oncoming car when the oncoming car suddenly brakes, necessitating the ego to rapidly reassess the situation and adjust its turn.
8. The ego commences an unprotected left turn at an intersection while yielding to an oncoming car when the adversarial car, coming from the right, blocks multiple lanes by driving extremely slowly, forcing the ego vehicle to change lanes.
9. The ego vehicle is making an unprotected left turn; the adversarial vehicle approaches the intersection at a normal speed but then suddenly attempts to make a last-second right turn.
10. The ego attempts an unprotected left turn at an intersection while yielding to an oncoming car when the oncoming car veers erratically.
11. The ego is performing a right turn at an intersection when the crossing car suddenly speeds up, entering the intersection and causing the ego to brake abruptly to avoid a collision.
12. The ego vehicle is turning right; the adversarial car (positioned ahead on the right) blocks the lane by braking suddenly.
13. The ego vehicle is turning right; the adversarial vehicle enters the intersection from the left side, swerving to the right suddenly.

14. The ego vehicle is turning right; the adversarial car (positioned ahead on the right) reverses abruptly.
15. The ego vehicle is turning right; the adversarial car (positioned behind on the right) suddenly accelerates and then decelerates.
16. The ego vehicle is approaching the intersection; the adversarial car (on the left) suddenly accelerates and enters the intersection first and suddenly stops.
17. The ego vehicle is approaching the intersection; the adversarial car (on the right) suddenly accelerates and enters the intersection first and suddenly stops.
18. The ego vehicle is maintaining a constant speed; the adversarial car, coming from the right, blocks multiple lanes by driving extremely slowly, forcing the ego vehicle to change lanes.
19. The ego vehicle is entering the intersection; the adversarial vehicle comes from the opposite direction and turns left and stops, causing a near collision with the ego vehicle.
20. The ego vehicle is entering the intersection; the adversarial vehicle comes from the right and turns left and stops, causing a near collision with the ego vehicle.

E.2.6 Erratic and Non-Monotonic Behavior on Standard Roads

These scenarios introduce agents exhibiting anomalous, non-standard motion patterns on conventional road segments–including zig-zagging vehicles, looping cyclists, oscillating lane changes, indecisive pedestrians, rhythmic braking patterns, and repetitive U-turns. These test the ego's ability to handle behaviorally atypical road users in familiar topologies.

1. Ego vehicle drives on a city street; a vehicle ahead performs wide zig-zag motion across lane boundaries, occasionally entering the adjacent lane.
2. Ego vehicle approaches an intersection; another car keeps circling continuously without turning, forcing ego to stop.
3. Ego vehicle drives on a residential road; a pedestrian suddenly changes direction multiple times, appearing indecisive while crossing.
4. Ego vehicle is on a highway; a car ahead suddenly slows down, accelerates, and then performs a partial U-turn-like maneuver toward the shoulder.
5. Ego vehicle is on a straight road; a pedestrian walks in a small circular loop near parked cars.
6. Ego vehicle drives on a multi-lane urban road; one vehicle performs aggressive zig-zag lane changes, while another vehicle ahead brakes intermittently, creating compounding uncertainty.
7. Ego vehicle approaches a 4-way intersection; a cyclist enters and begins circling unpredictably in the intersection, while cross traffic proceeds normally.
8. Ego vehicle drives in a dense city environment; a pedestrian moves in a non-linear path (zig-zag) while a nearby vehicle performs a sudden U-turn into the ego's lane.
9. Ego vehicle drives on an arterial road; a leading vehicle repeatedly oscillates between lane center and lane divider, never committing to a lane.
10. Ego vehicle approaches a roundabout; a car inside continuously loops multiple times without exiting, blocking entry.
11. Ego vehicle is on a suburban street; a pedestrian repeatedly starts and stops crossing mid-lane, reversing direction unpredictably.
12. Ego vehicle drives through a tunnel; a vehicle ahead slows down abruptly at regular intervals, creating a rhythmic brake–accelerate pattern.
13. Ego vehicle is on a highway merge lane; a merging car oscillates between accelerating to merge and dropping back into the ramp repeatedly.
14. Ego vehicle drives in a parking lot; a vehicle ahead performs tight figure-eight loops between parking rows.
15. Ego vehicle approaches a T-intersection; cross traffic vehicle repeatedly turns left and immediately re-enters the intersection in a loop.

16. Ego vehicle is on a curved urban road; a cyclist weaves in widening serpentine paths across the bike lane and adjacent car lane.
17. Ego vehicle drives on a straight boulevard; a delivery van repeatedly pulls toward the curb and then re-enters traffic without stopping.
18. Ego vehicle approaches a crosswalk; a pedestrian walks diagonally, pauses, then resumes in a zig-zag path across the crossing zone.
19. Ego vehicle drives on a highway; a car ahead performs repeated partial lane changes, never fully crossing lane boundaries.
20. Ego vehicle is on a residential cul-de-sac; a vehicle continuously circles the cul-de-sac while intermittently reversing direction.
21. Ego vehicle drives through a downtown grid; a vehicle ahead repeatedly blocks intersections by entering, stopping, and circling within junction space.
22. Ego vehicle is on a multi-lane road; two adjacent vehicles exhibit conflicting behaviors—one drifting laterally in oscillations, the other performing sudden braking pulses.
23. Ego vehicle approaches an on-ramp; a vehicle ahead performs S-shaped weaving while trying to merge but repeatedly aborts.
24. Ego vehicle is on a narrow rural road; an oncoming vehicle swerves into the shoulder, returns, then repeats this pattern in cycles.
25. Ego vehicle drives near a school zone; multiple pedestrians cross in staggered, non-monotonic paths, reversing mid-crossing unpredictably.
26. Ego vehicle is at a signalized intersection; a vehicle enters on green, then repeatedly loops within the intersection instead of clearing it.
27. Ego vehicle drives on a coastal road; a cyclist performs wide oscillatory arcs from shoulder to centerline in a sinusoidal pattern.
28. Ego vehicle is in a shopping district; a pedestrian repeatedly circles a parked car while occasionally stepping into traffic lanes.

E.2.7 Complex and Non-Standard Road Topologies

These scenarios challenge the ego vehicle in topologically complex environments that go beyond conventional road designs. Adversarial agents exhibit looping, oscillating, or conflicting behaviors within multi-way junctions, spiral roadways, cloverleaf interchanges, figure-eight structures, stacked roundabouts, and other intricate layouts. These scenarios assess the generalization of the ego's planning stack to rare and structurally demanding environments.

1. Ego vehicle approaches a 5-way star intersection; a vehicle continuously rotates through consecutive exits without leaving the junction while incoming traffic attempts to merge.
2. Ego vehicle drives through a 6-way radial junction; two vehicles enter from opposite spokes and repeatedly swap lanes while circling the center island.
3. Ego vehicle navigates a spiral roadway descending into an underground plaza; a vehicle ahead continuously spirals inward and outward without committing to an exit.
4. Ego vehicle enters a figure-eight interchange; a cyclist loops repeatedly across the crossover point, intersecting multiple traffic flows unpredictably.
5. Ego vehicle drives in an L-shaped city layout where each block forms a 3×3 grid; a bus repeatedly shortcuts diagonally through intersections against expected flow.
6. Ego vehicle traverses a 3×3 downtown grid with the center column merged into a pedestrian boulevard; delivery robots intermittently enter and block vehicle crossings.
7. Ego vehicle approaches a stacked double-roundabout; a car exits the lower loop, climbs to the upper loop, then re-enters the first roundabout repeatedly.
8. Ego vehicle drives on a cloverleaf highway interchange; a vehicle continuously loops through the clover ramps without merging onto any highway segment.
9. Ego vehicle navigates a triangular city block network; a pedestrian repeatedly cuts across triangle medians and reverses direction at corners.

10. Ego vehicle enters a hexagonal plaza surrounded by six connecting roads; a vehicle circles the hexagon while drifting unpredictably across lane markings.
11. Ego vehicle drives through a maze-like medieval street topology; a scooter repeatedly emerges from narrow alleys and re-enters hidden side passages.
12. Ego vehicle approaches a split-level intersection with overlapping bridges; a truck repeatedly changes between upper and lower connectors in rapid succession.
13. Ego vehicle travels through a spiral parking structure connected directly to city streets; a vehicle ahead alternates between ascending and descending loops erratically.
14. Ego vehicle drives along a serpentine mountain road; an oncoming vehicle repeatedly cuts hairpin corners into the ego's lane before correcting back.
15. Ego vehicle approaches a turbine interchange; a merging car continuously switches between connector ramps, never fully entering the target highway.
16. Ego vehicle navigates a circular city layout with concentric ring roads; a motorcycle repeatedly jumps between inner and outer rings through radial connectors.
17. Ego vehicle drives through a bowtie-shaped junction; a vehicle repeatedly crosses the central crossover zone, blocking both halves of the intersection.
18. Ego vehicle approaches a Y-shaped highway split; a confused driver repeatedly commits to one branch before sharply returning to the split point.
19. Ego vehicle traverses a checkerboard urban topology with alternating one-way streets; a vehicle repeatedly enters incorrect directions and performs abrupt corrective maneuvers.
20. Ego vehicle enters a kidney-shaped roundabout; a cyclist continuously loops around the asymmetric curves while drifting between inner and outer lanes.
21. Ego vehicle drives on an elevated spiral bridge crossing over itself; a vehicle ahead repeatedly slows at the self-overlap point, creating cascading congestion.
22. Ego vehicle approaches a 7-way plaza intersection; pedestrians begin crossing simultaneously from all spokes in intersecting diagonal patterns.
23. Ego vehicle navigates a city with alternating wide and narrow blocks; a vehicle repeatedly attempts U-turns inside undersized junctions and becomes partially obstructive.
24. Ego vehicle drives through a road network shaped like a figure-eight with a central signalized crossing; a bus repeatedly misses turns and loops both halves continuously.
25. Ego vehicle enters a flower-petal road topology with looping cul-de-sac petals around a center hub; a car repeatedly exits one petal and immediately enters another.
26. Ego vehicle approaches a multilayer roundabout interchange; a vehicle oscillates between outer bypass lanes and inner circulation lanes without exiting.
27. Ego vehicle drives through a compressed grid where every second intersection is merged into a super-junction; vehicles from multiple approaches simultaneously block the shared center.
28. Ego vehicle navigates a star-shaped residential layout; a pedestrian repeatedly crosses between radial spokes through landscaped medians unpredictably.
29. Ego vehicle travels on a corkscrew tunnel ramp; a vehicle ahead repeatedly brakes near each rotational apex, causing rhythmic stop-and-go waves.
30. Ego vehicle approaches a mirrored double-figure-eight roadway; two cyclists loop in opposite directions and intersect near crossover regions repeatedly.
31. Ego vehicle enters a figure-eight interchange; a cyclist traverses the crossover point in a looping pattern.
32. Ego vehicle drives through a downtown grid with a central pedestrian plaza; a delivery robot repeatedly enters and blocks vehicle crossings while navigating the plaza.

F MAP GENERATION EVALUATION

F.1 VLM AS A JUDGE

The judge sees two views of the same map, because neither alone is sufficient. The rendered image carries topology that a reader takes in at a glance, but it is a lossy picture whose scale and drawing style differ between methods, and being top-down it cannot show height at all. The structured facts parsed from the produced map file are exact and method-independent, and are what a simulator consumes, but they do not convey layout. Scoring the image alone would reward attractive renders; scoring the file alone would miss the arrangement. The judge is therefore also asked whether the two *agree*, which catches a render that flatters a file it does not match.

Structured facts. For a method emitting OpenDRIVE, the file is parsed with a plain XML reader rather than through the simulator's loader, so that the facts describe what the file says rather than what a tolerant loader salvages from it. The judge receives the number of roads and junctions, total road length, the minimum and maximum road length and lane count, the number of distinct incoming arms at each junction, the elevation range and span, the maximum gradient, a flatness flag, and a census of the geometry primitives used. Elevation is evaluated from the cubic polynomial at both ends of every elevation record: reading only the constant term reports the height at the start of a record, so a road that climbs within a single record through its linear and higher-order terms is recorded as perfectly flat, which in one case reported a real -6 to $+6$ m saddle as level ground. For a method emitting a vectorised scene rather than OpenDRIVE, the equivalent quantities are summarised from that format, so it is judged on the same axes rather than penalised for a different file type.

Protocol. Each method–prompt pair is judged independently: no judgement sees another method's output for the same prompt, so there is no ordering effect to control for. Images are downscaled to a longest edge of 1100 pixels, which bounds the input cost per judgement. Judgements are made with `claude-opus-5`.

System Prompt

Role. You are an expert evaluator of automatically generated road networks for driving simulation.

Inputs. You will be given: (a) a natural-language prompt describing a road layout, (b) structured facts parsed directly from the map file the method produced, and (c) a rendered image of that map.

Task. Score on the following axes, each an integer 0–5:

- **prompt_adherence** -- does the map realise what the prompt asks for (counts of arms, lanes and streets, angles, spacing, stated distances)? Quantitative claims in the prompt are checkable against the structured facts; use them.
- **topology_correctness** -- are junctions, connectivity and road continuity sane and consistent with the described structure, as opposed to disconnected fragments or accidental extra junctions?
- **geometric_plausibility** -- is this drivable road geometry (sane lane widths, curvature, no degenerate or overlapping segments)?
- **image_file_consistency** -- does the render agree with the structured facts? Score 5 when they tell the same story, low when the picture looks richer or poorer than the file supports.
- **elevation_correctness** -- ONLY when the prompt makes a vertical claim (a climb, descent, grade, bridge, overpass, underpass, ramp, multi-level or sunken structure, or an explicit height in metres). Judge it from the STRUCTURED FACTS, never from the image: every render here is top-down 2D, so a flat map and a correct 3D one look identical. Use elevation_range_m, elevation_span_m, max_gradient_pct and is_flat, and check them against the height the prompt asks for. A map with is_flat=true for a prompt that demands

```
   a grade separation scores 0.  If the prompt makes NO vertical
   claim, return null for this field; do not guess.

Be strict and comparative-free:  judge only this one map against this
one prompt.  A map that is a valid road network but does NOT match
the prompt should score low on prompt_adherence even if it scores
high elsewhere.  If the structured facts show an empty or trivial
network, say so.

Output format.  Reply with ONLY a JSON object, no prose outside it:

       {
       "prompt_adherence":  ⟨0-5⟩,
       "topology_correctness":  ⟨0-5⟩,
       "geometric_plausibility":  ⟨0-5⟩,
       "image_file_consistency":  ⟨0-5⟩,
       "elevation_correctness":  ⟨0-5 or null⟩,
       "prompt_has_vertical_claim":  ⟨bool⟩,
       "verdict":  "⟨≤25 words⟩",
       "evidence":  "⟨≤40 words citing a specific number from
       the facts or a specific image feature⟩"
       }
```

Cost. Judging all 250 prompts for all five methods is 1250 judgements. At the image resolution and prompt above this consumed 2.94M input and 799K output tokens, of which 69% of the output was reasoning, at a mean of 8.9 s per judgement.

F.2 Metric Definitions

Table 7 gives every metric for every method, including the connectivity and layout-diversity columns that Table 1 omits for space. The definitions follow.

Table 7: **Text-to-map generation over 250 prompts, all columns** (87 base, 100 novel, 50 ordinary, 13 complex), pooled. Tier A is computed on the lane-graph abstraction all methods share; Tier B only exists for methods that emit OpenDRIVE. *Div.* is the effective distinct-layout ratio; *Ext.* is generated spatial extent over requested extent (pooled over the 163 prompts that state a size); *SPG* is the shuffled-perturbation control, where 0 means the output is indistinguishable from one produced for a different prompt; its six attribute axes are standardised before the cosine, and prompts stating no mechanically checkable claim are excluded, leaving 177–183 of 250 per method; *VLM* is the mean of four 0–5 axes from a vision-language judge that sees both the render and facts parsed from the map file; *Elev.* is the fraction of elevation-prompt outputs carrying non-zero grade. *Conn.* is the fraction of lane nodes in the largest connected component. This is the full version of Table 1, which omits the connectivity and diversity columns for space.

	Tier A (all methods)						**Tier B** (OpenDRIVE)	
Method	Success	Conn.↑	Div.↑	Ext.→ 1	SPG↑	VLM↑	Load	Elev.↑
ScenarioControl (Gao et al., 2026)	**100.0%**	0.62	**0.99**	0.28	0.041	2.36	n/a	n/a
LCTGen (Tan et al., 2023)	99.6%	0.42	0.14	0.43	0.009	1.75	n/a	n/a
ML-SceGen (Xiao et al., 2025)	95.6%	0.22	0.56	0.56	0.106	2.93	100%	0.0%
OmniTester (Lu et al., 2024)	97.2%	**0.82**	0.98	1.20	0.160	4.02	100%	28.0%
Ours	98.4%	0.81	0.93	**1.28**	**0.196**	**4.54**	100%	**69.4%**

Alongside the learned judge we compute a set of mechanical metrics, so that the headline ordering rests on two instruments with different failure modes rather than on a model's opinion. All of them operate on a common lane-graph abstraction extracted from whatever each method emits, so that a method is not rewarded or penalised for its file format. Metrics that can only exist for a simulator-ready map are reported separately and marked *n/a* for methods that cannot express them, rather than scored as zero.

Generation success. A sample succeeds when the method produced a parseable primary artifact for that prompt. The adjudication is taken from the run manifest rather than re-derived, because the record files written by different batches follow different conventions and a stale record can sit beside a freshly successful output; the manifest resolves this by trusting a record only when it is at least as recent as the artifact it describes.

Structural validity. Computed on the lane graph: the fraction of lane nodes lying in the largest connected component, the rate of orphan nodes, and the rate of self-intersecting road geometry.

Prompt fidelity: CCS and SPG. We adopt the control-adherence protocol of Gao et al. (2026). Each prompt is reduced to an attribute vector u and each generated map to a vector v over the same attributes: junction arity claimed and realised, the presence of a loop, the number of enclosed blocks, the requested and achieved extent, whether elevation is mentioned and present, and the lanes per direction claimed and measured. The control correlation score is

$$\mathrm{CCS} \;=\; \frac{1}{N}\sum_{i=1}^{N} \cos\big(u_i,\, v_i\big), \tag{5}$$

and the shuffled-prompt gap subtracts the same quantity computed under random reassignment of prompts to maps, averaged over $T = 200$ permutations:

$$\mathrm{SPG} \;=\; \mathrm{CCS} \;-\; \frac{1}{T}\sum_{t=1}^{T}\frac{1}{N}\sum_{i=1}^{N} \cos\big(u_{\sigma_t(i)},\, v_i\big). \tag{6}$$

A method that emits plausible maps while ignoring the text scores near zero on SPG however good those maps look, which is the property the metric exists to test.

Two departures from a naive implementation are necessary and we state them. First, the axes are standardised before the cosine is taken. On their native units the length-versus-extent axis carries 99.8% of the centred dot product, measured over our corpora, because it is expressed in metres while every other axis is a small count; without standardisation the control reduces to a test of whether the map came out the right size, and junction arity, loops, blocks and elevation contribute nothing. Second, prompts that state no mechanically checkable claim yield an all-zero u and are excluded rather than contributing a degenerate cosine; N is reported as the number of prompts actually used, which is 177–183 of 250 per method. The parser accepts numerals written as words as well as digits, without which 114 of the 250 prompts, including every prompt in the ordinary set, have their stated junction arity silently read as absent.

Diversity. Each map is reduced to a normalised occupancy fingerprint and pairwise cosine similarities are computed within a prompt set. Two maps are near-duplicates above a similarity of 0.90, and the reported quantity is the effective number of distinct layouts, obtained by counting connected components under that threshold, divided by the number of samples.

Spatial extent. For the subset of prompts that state a size, we report the ratio of the extent realised to the extent requested. The base set is excluded from the pooled figure because only five of its prompts state a size and their ratios are extreme enough to dominate the mean; the reported value pools the 163 prompts of the other three sets.

Simulator loadability. For methods emitting OpenDRIVE, the file is loaded with a client-side `carla.Map` parse and waypoints are generated at 1 m spacing, with no simulator process and no GPU. Loadability alone does not separate the methods, since every file that was produced loads; the informative quantities are what the loaded map contains, namely the number of topology edges and the number of generated waypoints, which vary with the prompt for some methods and are near-constant for others.

Elevation. Restricted to prompts whose family requires a height change, we report the fraction of produced maps carrying non-zero grade. This is reported only for methods that emit OpenDRIVE, since the others have no representation for height and would otherwise be scored zero for a format limitation rather than a failure.

F.3 CROSS-JUDGE CHECK

The map scores in the main paper come from a single VLM judge, and the agents that generate the maps are backed by `claude-sonnet-5` while the judge is `claude-opus-5`. A judge and a generator from the same vendor invite the question of whether the ranking is an artefact of one model family scoring its own outputs. We therefore re-ran the judge with `gpt-5.6-sol`, a model from a different provider, on a stratified subset of 50 of the 250 prompts (17 base, 20 novel, 10 ordinary, 3 complex), judging all five methods on each, for 250 judgements. The system prompt, the structured facts and the rendered image are identical; only the model differs.

Table 8: The same 250 judgements scored by two judges from different providers. Scores are means on a 0–5 scale over the 50-prompt subset; the Claude column is restricted to the same cells so the two are directly comparable.

	Adherence		Topology		Geometry		Img/file		Elevation	
Method	Cl.	GPT	Cl.	GPT	Cl.	GPT	Cl.	GPT	Cl.	GPT
CARLAGEN4D (ours)	**4.26**	**4.12**	**4.64**	**4.66**	**4.60**	**4.56**	**4.90**	**4.90**	**4.30**	**4.40**
OmniTester	3.62	3.74	4.08	4.42	3.96	4.00	4.34	4.66	1.90	1.90
ML-SceGen	0.56	0.58	2.74	2.72	3.56	3.38	4.46	4.70	0.00	0.00
ScenarioControl	0.70	0.46	2.24	1.68	2.86	2.06	3.80	4.80	0.00	0.00
LCTGen	0.40	0.00	2.06	0.00	2.94	0.10	1.48	0.00	0.00	0.00

What transfers and what does not. Our method ranks first on all five axes under both judges, and the two judges place our absolute scores within 0.14 of each other on every axis. The margin over the strongest baseline is preserved: on adherence 4.26 vs 3.62 under Claude and 4.12 vs 3.74 under GPT. Per-judgement agreement is closest on adherence, the axis the main claim rests on (63% exact, 96% within one point, mean absolute difference 0.41), and looser on the axes that require reading the rendered image (topology 43% exact, mean difference 0.89; geometry 44%, 0.99).

F.4 ABLATION: THE DSL AND THE COMPILE LOOP

The DSL is motivated by the cost of asking a language model for OpenDRIVE directly, and the repair loop by the claim that compiler diagnostics are worth another model call. We test both by removing one at a time. All three arms share one driver, one model, one 1200 s wall-clock budget per run and one definition of success, on a 50-prompt subset that weights the four prompt sets equally (13 base, 12 ordinary, 13 novel, 12 complex). Our own arm is re-run here rather than lifted from the main results, so the comparison is like-for-like.

Table 9: Removing the DSL or the compile loop, 50 prompts per arm. *Clean* is a declared, error-free completion; *produced* counts runs that returned any map; *stopped* counts graceful budget exhaustion, which keeps the last map rather than discarding the run. Tokens and seconds are means per run.

				Mean cost		Clean by prompt set			
Arm	Clean	Produced	Stopped	Tokens	Sec.	base	ord.	novel	cplx.
Ours (DSL + loop)	**39/50**	**50**	9	51 671	578	11/13	12/12	**9/13**	**7/12**
No loop (DSL, single shot)	33/50	48	**1**	**33 480**	**350**	**12/13**	11/12	6/13	4/12
No DSL (raw OpenDRIVE, loop)	12/50	43	38	131 784	1224	8/13	1/12	1/13	2/12

The DSL. Writing OpenDRIVE directly costs 2.6× the tokens and 2.1× the wall-clock time of writing DSL, and returns a clean map on 12 of 50 prompts against 39. It also exhausts the budget on 38 of 50 runs, so most of that cost buys an unfinished map. The gap widens with prompt difficulty: on the ordinary and novel sets the raw-OpenDRIVE arm produces 1 clean map out of 12 and 13 respectively. This is the measurement behind the claim that direct map generation is prohibitively expensive.

The compile loop. Removing the iteration is cheaper, not more expensive: a single shot costs 35% fewer tokens and finishes in 60% of the time. What it loses is the harder prompts. On the base set the single-shot arm is nominally ahead (12/13 vs 11/13), and on ordinary prompts the two are within

one map; the separation is on novel ($9/13$ vs $6/13$) and complex ($7/12$ vs $4/12$) prompts, where a first attempt is unlikely to compile cleanly.

G Trajectory Generation Evaluation

G.1 VLM as a Judge

System Prompt

You are an expert autonomous driving evaluator.

You will receive:

1. A Bird's-Eye View (BEV) image of a multi-agent trajectory scenario.
   - Trajectories for different agents are shown as solid lines.
   - Agents are labelled and colour-coded.
2. A list of agents, each with:
   - agent_id
   - type (vehicle / ambulance / police / firetruck / two_wheeler / pedestrian)
   - motion_description -- the natural-language description the planner assigned to this actor.
3. Supporting context: collision results, heading-discontinuity statistics.

Your Task

For *each* agent, assign a **Motion Fidelity Score** from 1 to 5:

Score	Meaning
5	Trajectory *perfectly* matches the described motion. *E.g.* a "weaving cyclist" clearly zigzags; a "stationary vehicle" stays still.
4	Mostly matches; minor deviations that do not contradict the description.
3	Partial match; some aspects are correct but overall behaviour differs.
2	Weak match; general direction is right but specific motion is wrong.
1	Does not match at all; behaviour *completely* contradicts the description.

Focus especially on:

- Zigzag / weaving patterns -- does the actor *actually* zigzag?
- U-turns -- does the actor actually reverse direction?
- Stationary actors -- does the path stay in place?
- Speed profiles -- fast, slow, or stopping.
- Lane changes -- does the actor actually change lane?
- Following / leading relationships between actors.

Instructions

1. Call analyse_trajectory_image with a detailed description of *each* agent's path as you observe it in the BEV.
2. Call score_motion_fidelity with a JSON list of per-agent scores.
3. Call generate_fidelity_verdict with the final structured report.

Respond **only** using the provided tools.

G.2 Other Metrics

Notation

Let $\mathcal{A} = \{a_1, \ldots, a_N\}$ be the set of N simulated actors. For actor a_i let $\tau_i = \left[(x_i^t, y_i^t, z_i^t)\right]_{t=1}^{T}$ denote its executed trajectory of T waypoints sampled at uniform time steps. The ego actor is denoted a_0. Each actor has a declared type $\rho_i \in \{\texttt{vehicle}, \texttt{ambulance}, \texttt{police}, \texttt{firetruck}, \texttt{two_wheeler}, \texttt{pedestrian}\}$.

Metric 1 — Collision Rate

Each actor a_i is modelled as an axis-aligned bounding box (AABB) with type-specific half-extents $(\ell_i^{\mathrm{hw}}, \ell_i^{\mathrm{hl}})$ in the x- and y-directions respectively (see Table 10).

Two actors a_i, a_j *collide at timestep* t iff

$$\left|x_i^t - x_j^t\right| < \ell_i^{\mathrm{hw}} + \ell_j^{\mathrm{hw}} \quad \text{and} \quad \left|y_i^t - y_j^t\right| < \ell_i^{\mathrm{hl}} + \ell_j^{\mathrm{hl}}.$$

A pair (i, j) is marked as *colliding* if the above condition holds for *any* $t \in \{1, \ldots, T\}$. Let $P = \binom{N}{2}$ be the total number of unique actor pairs.

Overall collision rate

$$\mathrm{CR}_{\mathrm{overall}} = \frac{\#\{\text{colliding pairs}\}}{P} \in [0, 1].$$

Ego–actor collision rate

$$\mathrm{CR}_{\mathrm{ego}} = \frac{\#\{j \neq 0 \mid (a_0,\, a_j) \text{ collides}\}}{N - 1} \in [0, 1].$$

Actor–actor collision rate (excluding the ego)

$$\mathrm{CR}_{\mathrm{aa}} = \frac{\#\{(i, j) : i \neq 0,\, j \neq 0,\, (a_i, a_j) \text{ collides}\}}{\binom{N-1}{2}} \in [0, 1].$$

This quantity is reported as `N/A` when fewer than two non-ego actors are present, because the denominator is zero.

Table 10: AABB half-extents per actor type (metres).

Type	ℓ^{hw}	ℓ^{hl}
`vehicle`	2.2	1.0
`ambulance`	2.8	1.1
`police`	2.2	1.0
`firetruck`	4.0	1.4
`two_wheeler`	1.0	0.4
`pedestrian`	0.3	0.3

Metric 2 — Lane Adherence Score

For each trajectory point (x_i^t, y_i^t, z_i^t), query the CARLA map API for the nearest drivable-lane waypoint w_i^t. The *lateral offset* is

$$d_\perp(i,t) = \sqrt{\left(x_i^t - w_{i,x}^t\right)^2 + \left(y_i^t - w_{i,y}^t\right)^2} \ \text{[metres]}.$$

Per actor, compute:

$$\bar{d}_\perp^i = \frac{1}{T}\sum_{t=1}^{T} d_\perp(i,t), \qquad d_{\perp,90}^i = \mathrm{P}_{90}\big[d_\perp(i,1), \ldots, d_\perp(i,T)\big].$$

Aggregate over all actors:

$$\bar{d}_\perp^{\text{agg}} = \frac{1}{NT}\sum_{i=1}^{N}\sum_{t=1}^{T} d_\perp(i,t), \qquad d_{\perp,90}^{\text{agg}} = \mathrm{P}_{90}\big[\{d_\perp(i,t)\}_{i,t}\big].$$

Actors of type `two_wheeler` are flagged as *intentional-deviation* agents and reported separately, since their motion descriptions may legitimately include zigzag or weaving patterns.

If the CARLA map is unavailable (offline evaluation), all offsets are set to 0 and a note is appended to the report.

Metric 3 — Trajectory Completion Rate

Actor a_i is deemed to have *completed* its trajectory iff:

1. a_i was **not** involved in any collision (i.e. $a_i \notin \{\text{colliding actors}\}$), **and**
2. The Euclidean distance from its final executed position $\mathbf{p}_i^T$ to the last planned waypoint $\hat{\mathbf{p}}_i^{|\hat{\tau}_i|}$ satisfies

$$\left\| \mathbf{p}_i^T - \hat{\mathbf{p}}_i^{|\hat{\tau}_i|} \right\|_2 \ \le \ \delta_{\text{goal}} \quad (\delta_{\text{goal}} = 3 \text{ m by default}).$$

The **trajectory completion rate** is

$$\text{TCR} = \frac{\#\{i \mid a_i \text{ completed}\}}{N} \ \in [0,1].$$

Metric 4 — Heading Discontinuity Rate

The instantaneous heading of actor a_i at step $t \geq 2$ is derived from consecutive positions:

$$\theta_i^t = \operatorname{atan2}\big(y_i^t - y_i^{t-1},\ x_i^t - x_i^{t-1}\big) \ \in (-\pi,\, \pi].$$

A *heading discontinuity* occurs at step t iff the wrapped angular change

$$\Delta\theta_i^t = \min\big(|\theta_i^t - \theta_i^{t-1}|,\ 2\pi - |\theta_i^t - \theta_i^{t-1}|\big)$$

exceeds a threshold ϑ (default $\vartheta = 45°$). This directly detects the *180° spin artefact* that arises when waypoints are used instead of bare `carla.Location` objects. Because heading is undefined for a stationary displacement, a transition is eligible to be counted only when both adjacent displacement magnitudes are at least $\epsilon_{\text{move}} = 10^{-4}$ m. This tolerance removes floating-point jitter without suppressing physically meaningful motion.

Per actor:

$$\text{HDR}_i = \frac{100}{T-1} \sum_{t=2}^{T} \mathbf{1}\big[\Delta\theta_i^t > \vartheta\big] \ \ \text{[discontinuities per 100 steps]}.$$

Aggregate over all actors:

$$\text{HDR}_{\text{agg}} = \frac{100}{\sum_i (T-1)} \sum_i \sum_{t=2}^{T} \mathbf{1}\big[\Delta\theta_i^t > \vartheta\big].$$

Also reported: $\max_{i,t}\ \Delta\theta_i^t$ in degrees (worst single jump).

Metric 5 — Motion Description Fidelity (VLM Judge)

Each actor a_i is assigned a natural-language *motion description* m_i by the planner (e.g. "weaving cyclist", "stationary parked car", "ambulance overtaking at high speed").
A **VLM-as-a-Judge** (`VLMMotionFidelityJudge`) is given:

- A Bird's-Eye View (BEV) image of all executed trajectories.
- The motion description m_i for each actor.
- Collision context and heading-discontinuity statistics.

The judge assigns a **Motion Fidelity Score** $s_i \in \{1, 2, 3, 4, 5\}$ per actor according to:

s_i	Interpretation
5	Trajectory perfectly matches m_i.
4	Mostly matches; minor deviations only.
3	Partial match; overall behaviour differs.
2	Weak match; general direction right, motion wrong.
1	No match; behaviour completely contradicts m_i.

The **average motion fidelity score** over all actors is

$$\bar{s} = \frac{1}{N} \sum_{i=1}^{N} s_i \ \in [1,\, 5].$$

If VLM evaluation is disabled (e.g. `-no-vlm`), all scores default to 0 (not evaluated).

G.3 Results on Other Metrics

We compare trajectory generation against TTSG Ruan et al. (2024) on the same 39 prompts and the same CARLA build. The two methods do not run on the same road network: TTSG places its actors

on the shipped CARLA towns, while our pipeline synthesises a map for each prompt. This difference matters for the success comparison and we return to it below. A run counts as a success when the scenario executes and produces a trajectory bundle. Quality metrics are computed over successful executions only, with 95% bootstrap confidence intervals.

Prompt distribution. The 39 prompts are drawn from seven scenario categories: erratic and non-monotonic behaviour (11), intersection right-of-way violations (10), complex trajectory geometry (8), pedestrian intrusion (3), lane-change interference (3), intersection turns with adversarial agents (2), and opposite-lane bypass with oncoming hazards (2).

Table 11: Trajectory generation on 39 prompts. Arrows give the preferred direction. Quality metrics are averaged over successful executions, so TTSG's are computed over 12 scenes and ours over 39; brackets are 95% bootstrap confidence intervals.

	CarlaGen4D (ours)	**TTSG**
Execution success	$\mathbf{39/39}$ (1.00)	$12/39$ (0.31)
Overall collision ↓	$\mathbf{0.057}$ [0.00, 0.13]	0.071 [0.00, 0.25]
Ego–actor collision ↓	$\mathbf{0.065}$ [0.00, 0.15]	0.077 [0.00, 0.25]
Actor–actor collision ↓	0.000	0.000
Heading discontinuity ↓	$\mathbf{0.221}$ [0.08, 0.38]	1.266 [0.00, 3.94]
Motion fidelity ↑	$\mathbf{3.06}$ [2.79, 3.34]	2.99 [2.61, 3.36]

Why TTSG fails so often. Twenty-six of TTSG's 27 failures occur before the simulation starts; the remaining one is the simulator timeout. It could not find a road matching the scenario it was asked for in 9 cases (`no_valid_road`), and could not place the ego and the adversarial agent together in 11 (`failed_to_spawn_ego_plus_actor`). A further 6 crashed with an `AttributeError` while querying a waypoint that its own site selection had not found. TTSG selects a spawn site by matching the scenario against road attributes of the loaded map, so a prompt whose requirements no road satisfies never reaches simulation. Our method writes a trajectory function against the lane graph of a map built for the prompt, so no search for a matching site is needed and these two failure modes cannot arise. Much of the gap in execution success therefore reflects generating the road network, not trajectory generation alone.

Comparison. Among the scenes that do execute, the two methods are close on per-scene quality. Two differences are worth noting, with the caveat that TTSG's scores come from the 12 prompts it was able to place, which are not a random subset. The heading discontinuity point estimate is lower for us, 0.221 against 1.266, though with only 12 TTSG scenes the interval is too wide to separate the methods. And our generation executes more often, $39/39$ against $12/39$, for the reason given above.

H Additional Experimental Details and Results

H.1 Setup

Prompt suite and maps. The 250 prompts are distributed over four sets. The *base* set contains 87 prompts spanning five layout families (grid, arc, S-curve, junction topology, elevation). The *novel* set contains 100 prompts describing layouts absent from any shipped CARLA town, including brick-bond tessellations with no four-way junctions, turbine interchanges, corkscrew ramps and multi-level elevation. The *ordinary* set adds 50 plain grid, collector and junction prompts of the kind that dominate real road networks, included so that the comparison is not confined to unusual geometry. The *complex* set adds 13 compositional prompts that combine several constraints at once, such as simultaneous grade levels or nested ring structures. Every prompt is generated once per method. Representative prompts from each set are given in Appendix E.

Splits and seeds. All downstream experiments share a single held-out split defined at the level of maps rather than routes: two routes on the same map share its geometry, so a route-level split would leak layout into validation. For the training studies of Section 4.4, eight of the maps are used for testing, eight of the maps are held out for validation and 39 are used for training. The policy

benchmark of Section 4.3 is a separate, larger evaluation set of fifteen held-out maps (Appendix H.3) that contains these eight; the agents benchmarked there are released checkpoints trained on hand-authored towns, so none of them has seen any generated map, and the model trained in Section 4.4 is never scored on a map it trained on. Hand-authored data is collected from the CARLA towns under a disjoint town split, `Town01/03/04` for training and `Town02/05` for validation. Unless stated otherwise we report mean $\pm$ std over three seeds, with the architecture, initialisation and seed protocol held fixed across conditions so that only the data mix varies.

H.2 Q1: Map Generation Quality and Diversity

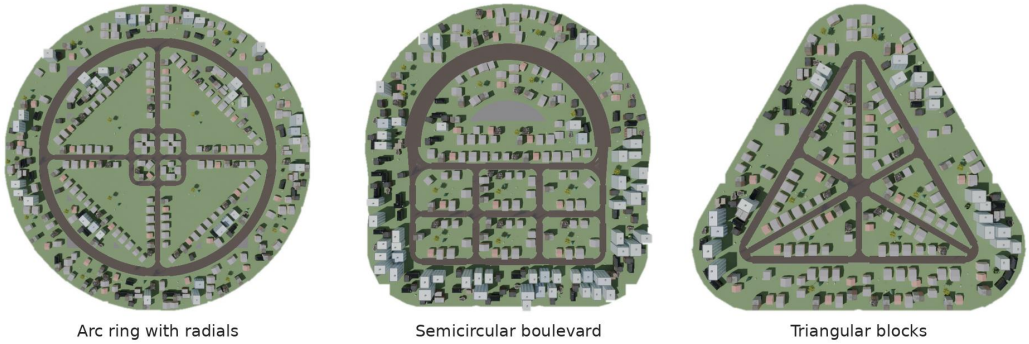


Figure 10: **Additional generated scenes**

How the baselines were run. Each baseline is run from its public release, with its own defaults, on the same 250 prompts we use for our own method. We changed nothing inside the methods; the only work was getting our prompts in and their maps out. ScenarioControl and LCTGen take the prompt directly and return a lane graph, which we convert to OpenDRIVE so that the Tier B checks can be attempted at all. ML-SceGen and OmniTester emit a road network of their own: ML-SceGen writes OpenDRIVE, and OmniTester writes a SUMO network, which we convert with SUMO's own `netconvert`. OmniTester keeps its vision-language repair loop, so a prompt it is not satisfied with is retried up to the limit its release sets; where it never converges we take its last attempt rather than discarding the prompt. Every method sees each prompt once, and no prompt is re-worded per method.

Extent and layout diversity. Extent and diversity are reported in full in Table 7, alongside the connectivity column. ScenarioControl realises 0.28 of the spatial extent its prompts request, a direct consequence of generating a single ego-centred tile rather than a network; converting its lane graph to OpenDRIVE so that it can be loaded at all yields a median of 9 topology edges over a 64 m window, against 82 edges and 4,077 waypoints for our maps. LCTGen realises 0.14 distinct layouts per prompt: retrieval from a fixed template library cannot follow a prompt off the manifold of the corpus it was built from, and the real-world map corpus is itself the binding constraint.

Elevation and per-family breakdown. The vision-language judge is instructed to read grade from the map file rather than from the image, since elevation is invisible to a top-down rendering; on the elevation axis it scores 4.62 for our maps, 1.82 for OmniTester and 0.00 for ML-SceGen. The shuffled-perturbation control is likewise not uniform across families: ML-SceGen attains its best score on the ordinary prompts (0.258) and its worst on the complex ones (-0.072).

Additional qualitative results. Figure 11 covers the families the main figure leaves out: an ordinary grid, a curve, an arc ring, a tessellation with no four-way junction, and two prompts asking for relief. The ordinary grid is worth noting, since it is the case least favourable to us: OmniTester builds it as well as we do, which is what we would expect of a method that tracks familiar layouts. The differences open up on the last three rows, where the prompt asks for a pattern or a grade the other methods have no way to express.

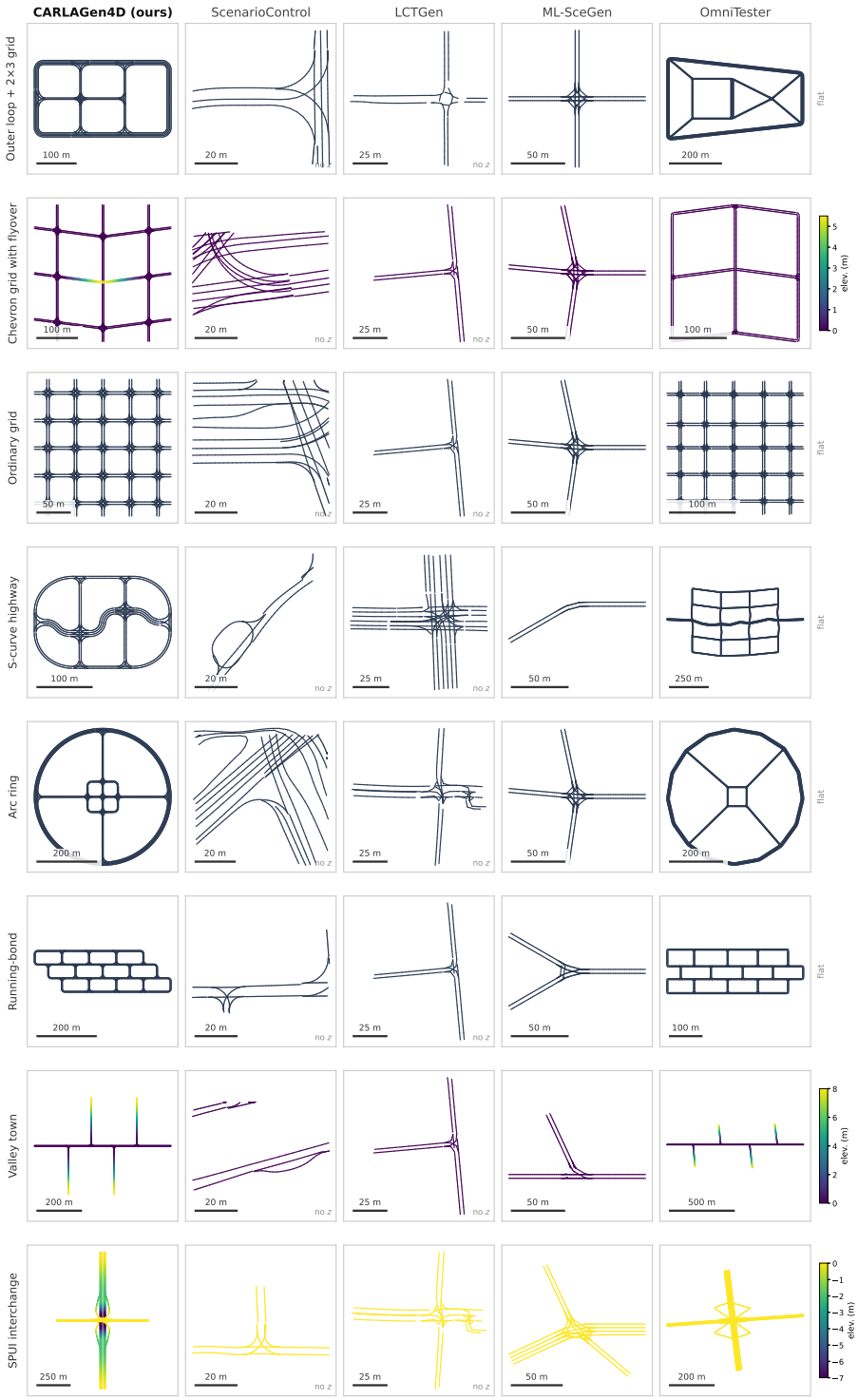


Figure 11: **Map generation methods**Road colour encodes elevation; rows whose prompts describe no relief are drawn in one ink and marked *flat*, and the two methods whose output format carries no height are marked *no z*.

H.3 Q2: Generated Topologies as a Policy Benchmark

How the policies were run. Every agent runs through one evaluation script, from its own public checkpoint, with no retraining or tuning. The script asks each agent which sensors it wants and spawns exactly those, so an agent that expects a particular camera rig or LiDAR gets it.

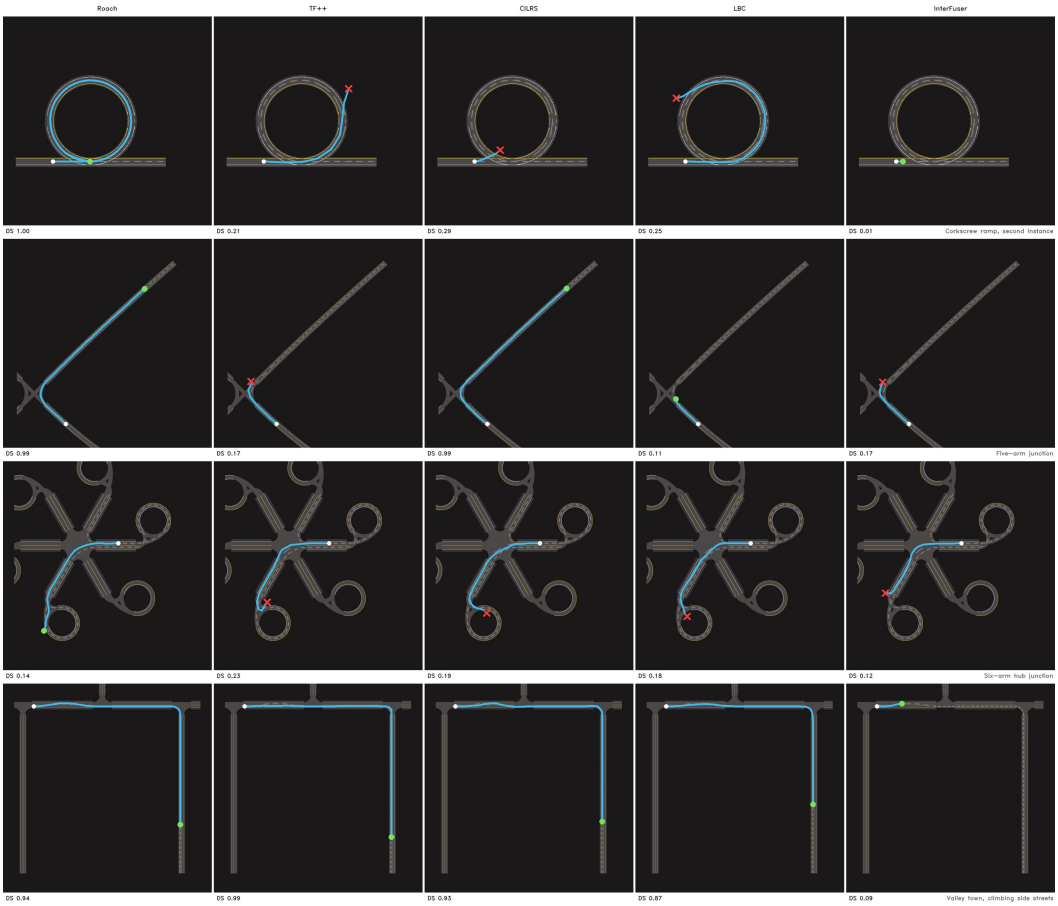


Figure 12: **Policy Evaluation** The planned route is dashed and the executed trajectory drawn over it; a white dot marks the start, a green dot where the agent came to rest, and a red cross where it left the neighbourhood of the route. The window is set by the route and is the same for every agent in a row, so a trajectory that leaves is visible as a departure rather than as a change of scale.

Where the policies fail. Figure 12 shows four episodes where the agents disagree most.

H.4 Q3: Generated Simulations as Training Data

H.4.1 Driving policy

The policy takes a single front camera image at 256×512, the current speed and the next target point, and predicts waypoints at 1, 2 and 3 s in the ego frame. Images pass through an ImageNet-pretrained ResNet-34; speed and target point each pass through a two-layer MLP, and the three features are concatenated. The head is command-conditioned in the usual way: one branch per navigation command, each a three-layer MLP with dropout 0.1, of which only the branch matching the current command is evaluated. No driving-pretrained weights are used, so the comparison between data mixtures is not confounded by what a pretrained driving model already knows.

Training uses AdamW at learning rate 10^{-4} with weight decay 10^{-4}, batch size 32, for at most 40 epochs, stopping early after 6 epochs without an improvement of at least 0.01 m. Every condition shares the architecture, the initialisation and the seed, so only the data mix differs.

H.4.2 Camera-to-BEV perception

The model reads the same 256×512 front image and predicts an 11-class semantic map over the 50×50 m in front of the ego at 0.5 m per pixel, giving a 100×100 grid. The classes are road, sidewalk, solid and broken lane markings, stop sign, the three traffic-light states, vehicle and walker,

plus an unlabelled class. An ImageNet-pretrained ResNet-34 encodes the image, a column-wise MLP maps each image column to the corresponding BEV column through eight height bins, and a small convolutional decoder produces the grid. Pixels the camera cannot see are excluded from the loss rather than labelled empty, so the model is never trained to guess at occluded ground.

Training uses AdamW at learning rate 2×10^{-4} with weight decay 10^{-4} and a cosine schedule, batch size 32, for at most 40 epochs with a minimum of 8, stopping early after 6 epochs without improvement.

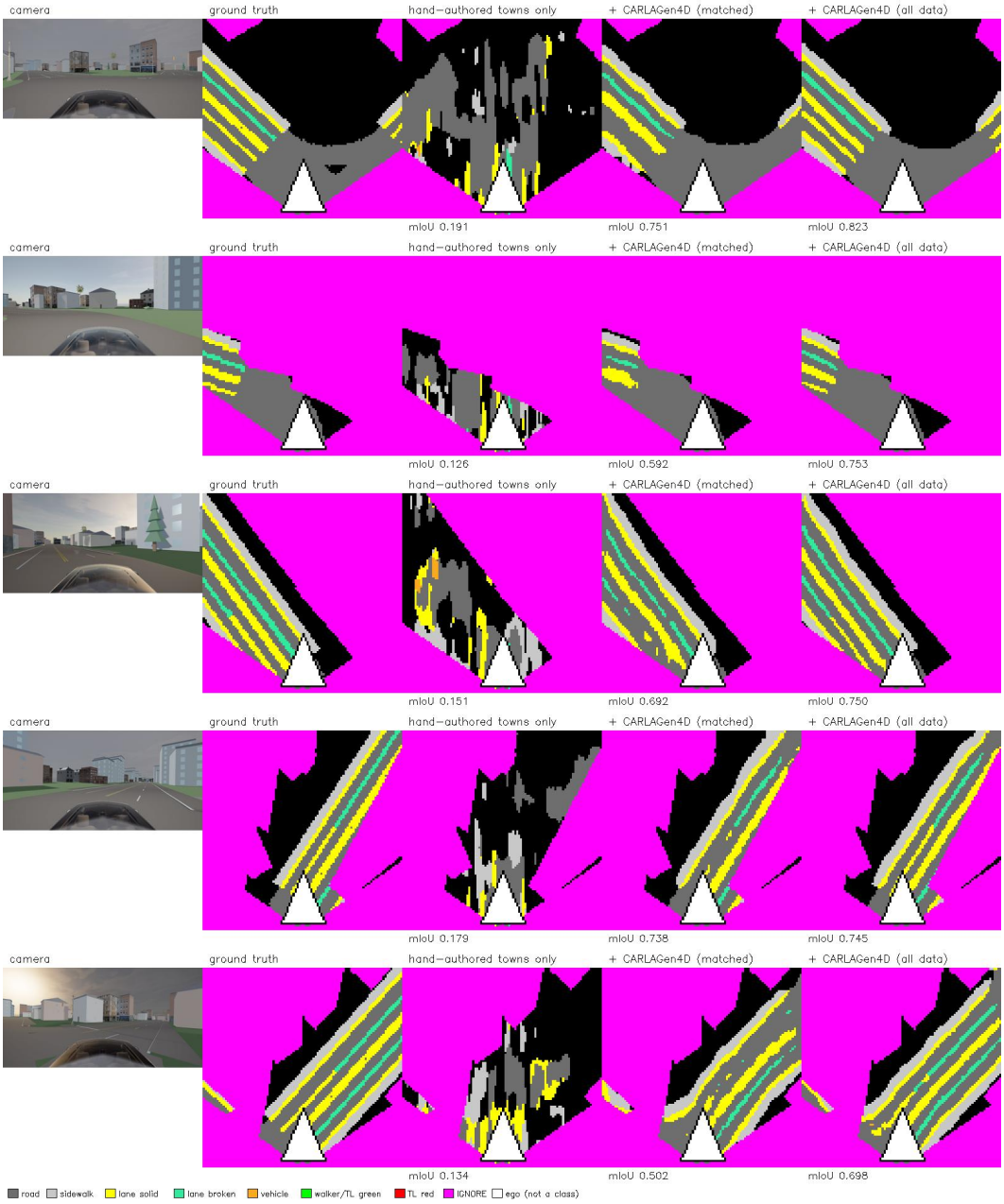


Figure 13: **Camera-to-BEV predictions** Each row is one frame: the camera image, the ground-truth map, and the three conditions. Magenta marks pixels excluded from the loss and the metric because the camera cannot see them.

Figure 13 shows five frames. The baseline trained only on hand-authored towns recovers a road surface of roughly the right shape but loses the structure on it, whereas both mixtures recover the lane markings and the extent of the junction.



Figure 14: **Driving captions** The ego pulls away on a green light. As in Figure 6, the released checkpoint narrates the manoeuvre but attributes it to the wrong cause; the fine-tune recovers the event that explains the behaviour.

Per-class IoU for every condition, on both evaluation sources, is given in Table 12.

H.4.3 Driving caption generation

We fine-tune the released ADAPT checkpoint, a video Swin encoder feeding a BERT decoder that emits a narration and a justification for a short clip. The released configuration is kept, including its separate-caption setting, so that the zero-shot and fine-tuned numbers refer to the same model in the same mode. References come from the simulator's own logged state rather than from human annotation, which is why n-gram scores are reported only for completeness: they reward reproducing our template phrasing.

Fine-tuning uses AdamW at learning rate 2×10^{-5} with a cosine schedule and weight decay 0.01 on all but the bias and normalisation parameters, batch size 4, for 6 epochs, repeated over two seeds.

Figures 15 and 16 show the failure the aggregate numbers describe. The released checkpoint narrates the manoeuvre plausibly and then supplies a cause drawn from the priors of its training distribution rather than from the clip, which is why it is indistinguishable from chance under the shuffled-clip control.

H.4.4 Transfer to photorealistic imagery

Figure 17 adds some more qualitative images. The pedestrian remains the same actor standing in the same place, and the right turn remains the same manoeuvre through the same junction, while the rendering becomes a wet street at dusk.

H.5 Per-class and per-region breakdown for Q3

The main paper reports one aggregate per task in Table 3. The two breakdowns behind those aggregates are given here: per-class IoU for camera-to-BEV perception, and the junction/non-junction split of the policy's open-loop L_2 (Table 12 and Table 13 respectively).

I Layout Diversity against Real and Hand-Authored Road Networks

I.1 What the measurement has to get right

Comparing generated maps with real ones requires a representation of road networks that is comparable across corpora. Three considerations are important for making this comparison meaningful.

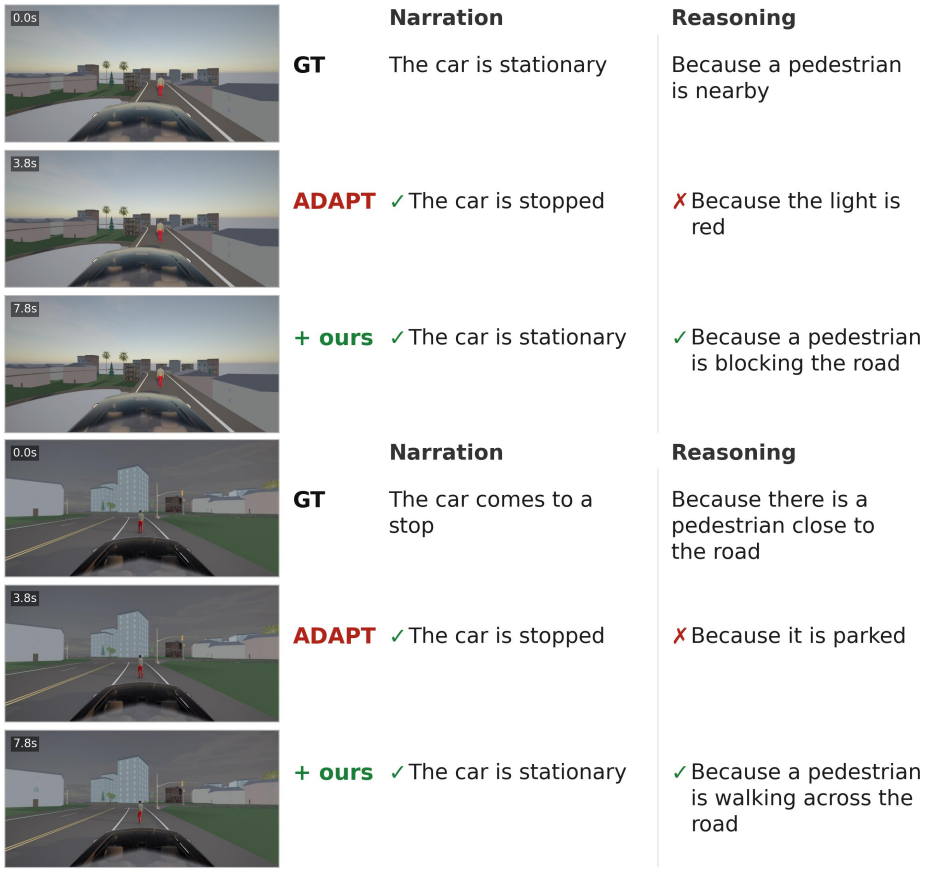


Figure 15: **Driving captions on held-out generated scenes: stopping for a pedestrian.** Each clip shows three frames, the reference caption, the released ADAPT checkpoint and our fine-tune; ticks and crosses mark agreement with the reference under the category parse. In both clips the released checkpoint reports the manoeuvre correctly and attributes it to a cause the frames contradict, a red light in the first and parking in the second, while the pedestrian responsible is visible in the road ahead.

Table 12: **Camera-to-BEV per-class IoU**, on both evaluation sources, for the classes that separate the conditions ("lane" is the solid lane marking). Bold marks the best of the three trained conditions; the control is excluded.

	hand-authored IoU↑			CarlaGen4D IoU↑		
Training data	road	lane	vehicle	road	lane	vehicle
CarlaGen4D only (control)	0.307	0.116	0.132	0.750	0.483	0.279
Hand-authored towns only	0.390	0.198	0.369	0.364	0.140	0.073
+ CarlaGen4D (matched)	0.428	0.225	0.423	0.743	0.476	0.249
+ CarlaGen4D (all data)	**0.439**	**0.237**	**0.451**	**0.748**	**0.492**	**0.289**

Lane clutter is not topological complexity. Junction arity should reflect the number of road arms rather than the number of lane directions. A four-way crossing with turn pockets, slip lanes, and driveways can otherwise appear as a six-arm junction, making it indistinguishable from a genuine six-way radial. In nuPlan, arm count correlates $+0.70$ with lane count, indicating that lane-level counting primarily captures urban lane clutter rather than topology. Parallel lanes are therefore merged into carriageways before counting: two polylines are merged when their headings agree within 20° and their lateral offset is below 6m.

Window size must be matched, not assumed. All corpora should be compared on 200×200,m patches, matching the native extent of a nuPlan sample. Roads should be clipped at patch boundaries



	Narration	Reasoning
GT	The car is picking up speed and is driving straight ahead	Now that the light is green



ADAPT	✗ The car stops	✗ Because the light is red



+ ours	✓ The car is accelerating and is continuing forward along the road	✓ Because the traffic light has turned green



	Narration	Reasoning
GT	The car is braking and is driving straight ahead	Because it is passing through an intersection



ADAPT	✗ The car slows down	✗ Because it is parking



+ ours	✓ The car is slowing down, going straight	✓ Because it is passing through an intersection

Figure 16: **Driving captions on held-out generated scenes: signals and junctions.** *Top:* the ego pulls away as the signal turns green, and the released checkpoint reports the opposite manoeuvre for the opposite reason. *Bottom:* the ego brakes through an intersection, which the released checkpoint attributes to parking. The fine-tune recovers the event that explains the behaviour in both.

Table 13: **Policy L_2 split by region**. Junction frames are those whose ego pose falls inside a junction

	CarlaGen4D $L_2 \downarrow$	
Training data	junction	non-junction
CarlaGen4D only (control)	2.13	–
Hand-authored towns only	8.39	7.57
+ CarlaGen4D (matched)	2.06	3.52
+ CarlaGen4D (all data)	**2.00**	**3.48**

rather than assigned to cells by their centroids, which can otherwise produce patches extending well beyond the reference extent. Since extent is itself a descriptor dimension, all corpora are capped at 200m and extent is excluded from the feature space.

I.2 Embedding road networks in a common space

No single compression of a road network into a vector is obviously correct, so we compute three and check whether they agree; agreement across descriptions with different blind spots is the only reason to trust any of them. All three operate on the same clipped, road-level patches.

1. **Shape**: the 40×40 occupancy raster of the patch, rotation-normalised onto its own principal axis, embedded with t-SNE. This involves no hand-designed feature and so cannot inherit the arity problem above. Rotation normalisation is essential: generated maps are axis-aligned

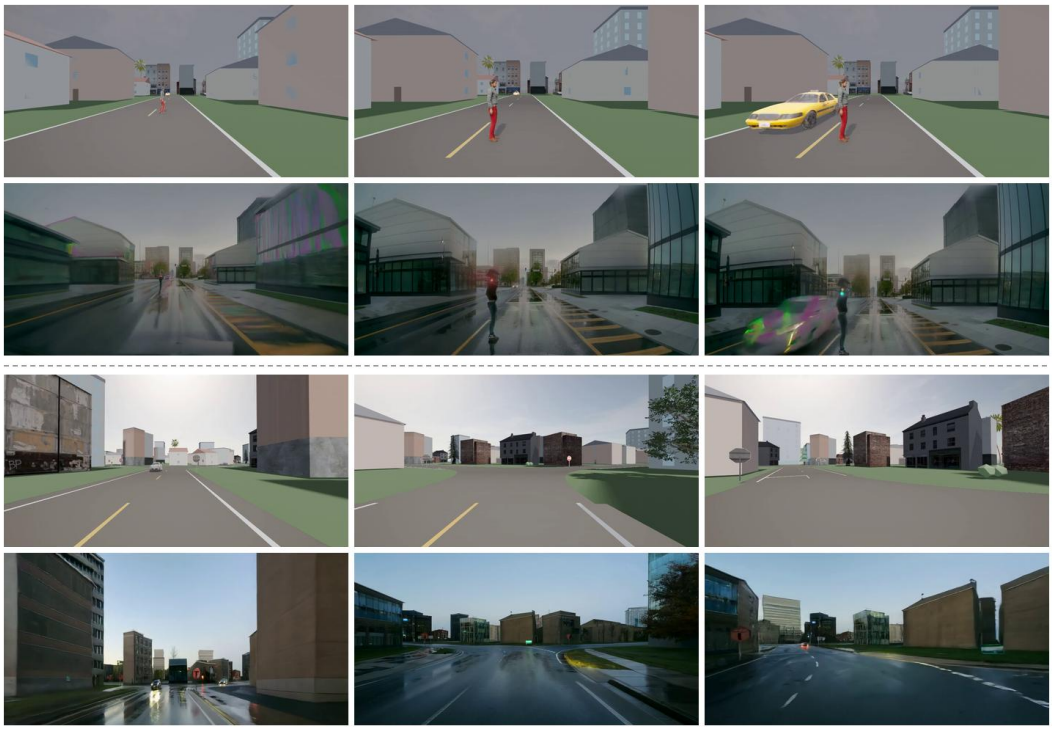


Figure 17: **Two further episodes translated to photorealistic video.** Each episode occupies two rows, separated by the dashed rule: the simulator render above and the same frames after segmentation-conditioned translation below, so a column is one frame before and after. *Top:* a pedestrian stands in the ego's path and a vehicle crosses in front of it. *Bottom:* the ego turns right at an intersection, shown from the approach through the turn. Translation is conditioned on the simulator's own segmentation and a single appearance prompt, so the actors and the road geometry are carried over and only appearance is resynthesised. The pedestrian and the junction survive the translation; the crossing vehicle, which is moving quickly across the frame, is reduced to a coloured smear, and we show it rather than omit it.

by construction and logged ones are not, so without it the embedding separates the corpora by heading and an artifact reads as a result.

2. **Structure**: road-level descriptors (road count, lanes per road, road density, curvature mean and $p90$), standardised per dimension, projected with PCA. Interpretable, so the axes can be named.
3. **Topology**: a Weisfeiler–Lehman label histogram over the junction graph, where nodes are junction clusters and edges are the roads joining them. This uses no coordinates and no arity threshold; two patches are close when their junction neighbourhoods look alike.

I.3 Results

Table 14 reports, for each corpus, the fraction of patches with *no* nuPlan patch among their ten nearest neighbours, computed in the full feature space rather than in the two-dimensional projection, which does not preserve distances. Roughly a third of our patches lie in neighbourhoods that logged data never visits, against a 1–4% self-baseline for nuPlan against itself.

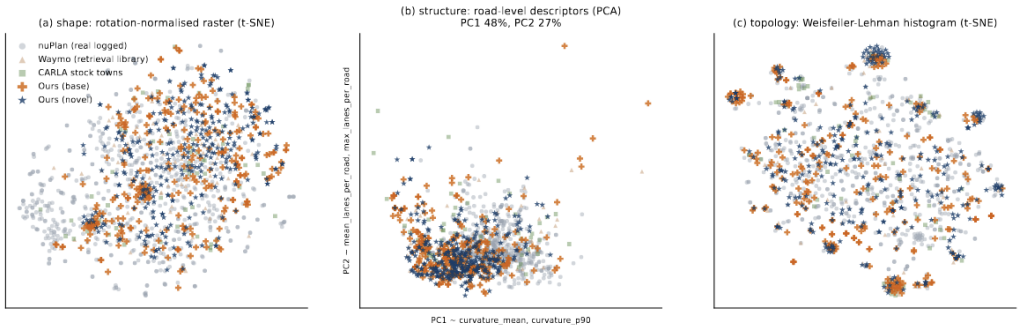


Figure 18: **Generated and real road networks in a common space.** Each point is one 200patch. Three independent compressions of the same patches; Our patches overlap the real-world distribution substantially and also occupy regions of the topology embedding that logged data does not reach.

Table 14: **Fraction of patches with no nuPlan neighbour** among their k=10 nearest neighbours, in the full space. The nuPlan row is the self-baseline: the rate at which a real patch has no other real patch nearby.

Corpus	WL topology	Road-level structure
nuPlan (self-baseline)	4.3%	1.4%
CARLA stock towns	23.6%	18.2%
Waymo (retrieval library)	28.1%	32.8%
Ours (novel prompts)	27.8%	34.8%
Ours (base prompts)	33.4%	44.8%