AutoNetSim: Intent-Driven Wireless Network Experimentation with Self-Evolving Agents

Runnan Si*, John Song*, Haijian Sun, and Zhenlin An
University of Georgia, USA

***Abstract*—Wireless network simulation and optimization are difficult because they require solving a complex cross-layer configuration problem while interacting with complex 3D radio environments. For decades, engineers have relied on channel models, channel simulators, and extensive standards to guide this process, but using them correctly still requires substantial domain expertise and engineering effort. Recent language-model agents can automate parts of wireless reasoning and code generation, yet they are not trained for the cross-layer setting in which an agent must construct a 3D radio environment, bind it to radio and network assumptions, and run simulations inside it. We present *AutoNetSim*, a self-evolving language-agent system for intent-driven wireless network simulation. Given a natural-language request, it constructs an executable *radio environment* that combines a 3D scene with a cross-layer wireless-system configuration and optimization. The system uses a multi-agent architecture with specialized scene-building, simulation, reflection, planning, and skill-learning agents. It learns from previous tutorials, worked examples, prior simulation results, and its own failed trajectories, then updates a procedural skill and knowledge base through verifier-driven optimization. We evaluate the system on benchmark suites covering 3D radio environment generation, wireless simulation, and token efficiency across multiple indoor and outdoor scenarios. On 3D radio environment generation, *AutoNetSim* reaches 72.5% held-out pass rate, whereas both baselines fail to solve any test task. Across ten ray-tracing, physical-layer, and network/system-level simulation families, it averages 95.5% pass rate, while both baselines stay below 70.0%.**

I. Introduction

Wireless-network deployment is a high-stakes engineering task whose difficulty appears before any optimization or planning algorithm can be run. A deployment question must first be translated into a faithful cross-layer simulation problem that specifies the physical environment, the radio-system configuration operating inside it, and the link- or system-level to evaluate. This translation is hard not merely because a wireless network is tied to the physical world, or because it has a large parameter space. It is a coupled network-system problem because the behavior of a radio configuration depends on the physical environment in which it is deployed. Propagation is shaped by geometry, radio materials, and device placement, while the meaning of that propagation depends on radio access choices, physical-layer settings, link behavior, and system-level objectives [1], [2].

The wireless community has spent decades managing this complexity through standardization and softwareization. Industrial and academic engineers document common assumptions in standards, channel models, and material recommendations, including 3GPP TR 38.901 [3] and ITU-R P.2040 [4]. They also release software platforms that encode this expertise into reusable tools, ranging from packet-level network simulators and commercial planning suites to channel-sounding and radio-propagation toolchains [5]–[7]. In this paper, we use NVIDIA Sionna as the representative backend because it is an open-source wireless simulation ecosystem that spans radio environment modeling, link-level communication experiments, and system-level studies [8], [9]. Recent integrations further connect Sionna with network simulators and wireless digital-twin stacks [10], [11]. These efforts make wireless research reproducible, but they do not make it simple. As networks become denser and more environment-dependent, users must still read large bodies of documentation, choose compatible abstractions across layers, assemble simulator artifacts, and interpret multi-modal results. The expertise bottleneck therefore shifts from writing every model from scratch to keeping standardized knowledge, simulator software, physical assumptions, and network objectives mutually consistent.

The emergence of large language models has opened a new path for reducing this burden. LLMs can already generate code and operate tools in many software-engineering settings [12], [13], and recent wireless agents use LLMs for network slicing, mobile service assurance, wireless homework, radio-map generation, base-station siting, and optimization formulation [14]–[18]. These systems show that wireless tasks benefit from structured agent workflows, but they also expose a missing capability. If an LLM agent is asked only to operate a network through code or text interfaces, it can remain at the level of abstract states, logs, and optimization variables. A wireless simulation agent must instead act more like an embodied agent living inside a radio environment, where every network decision is tied to physical geometry, radio materials, device topology, protocol settings, and downstream objectives. This requirement is also different from existing language-guided 3D scene generation, which focuses on visually plausible layouts or embodied-AI environments [19]–[21]. For wireless simulation, the scene is not the endpoint; it is the substrate on which the radio system operates. The agent must understand which objects matter for propagation, how material choices affect the channel, how topology changes the wireless network, how physical-layer choices change the interpretation of the result, and how all of these choices should be translated into simulator artifacts and follow-up experiments. Existing wireless agents largely operate on abstract network states, optimization variables, or existing

*Runnan Si and John Song contributed equally to this work. Zhenlin An is the corresponding author.

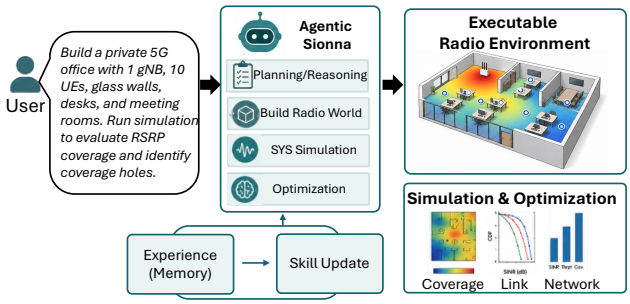


Fig. 1: ***AutoNetSim* converts natural language into executable radio environments and learns from experience.** A user describes a wireless simulation goal; the agent builds the 3D radio environment, binds it to network and experiment assumptions, executes Sionna simulations, and improves its skill through a self-evolving process driven by prior cases and current failures.

radio maps; they do not close this language-to-cross-layer-simulation understanding gap. Existing benchmarks reinforce the same limitation because they typically evaluate text-level planning, isolated optimization, existing radio maps, or visually plausible scenes, so success does not require constructing a simulator-loadable radio environment, preserving RF-material and device assumptions, or carrying those assumptions into link- and system-level metrics. Scaling training alone is also difficult because wireless simulation lacks a large, high-quality corpus that pairs tasks, simulator code, radio environment scenes, and verified outputs, while collecting such experience from engineers requires domain expertise, simulator execution, and inspection of the resulting radio map.

To bridge this gap, we present *AutoNetSim*, a self-evolving LLM agent for wireless network simulation and planning, as shown in Fig. 1. *AutoNetSim* treats wireless simulation as reasoning inside an editable *radio environment*, where it learns from prior materials and its own trials, understands 3D geometry and radio materials, creates and edits radio environments, runs cross-layer network simulations inside them, interprets multi-modal outputs, and uses the results to decide follow-up experiments. *AutoNetSim* does not introduce a new general memory mechanism or self-evolving algorithm. Instead, it applies established ideas such as reusable skills and learning from failures to wireless simulation. The resulting workflow connects simulation code generation with validation based on wireless physics. This distinction matters because the agent writes simulator code and maintains the environment, system, and experiment assumptions that make the code physically meaningful. Realizing this vision requires solving three linked problems, each paired with a system mechanism.

First, operating a 3D radio environment is difficult because existing coding agents do not have a reliable interface for understanding and manipulating coupled radio environments, where geometry, radio-material assignment, device placement, network topology, and simulator export must stay consistent. *AutoNetSim* addresses this with a Sionna-based radio environment skill that exposes these operations as procedural tools. The agent edits a structured scene representation, applies object, material, and device-topology rules, validates layout constraints, and uses rendered scenes and radio maps as feedback.

Second, reasoning through wireless simulation tasks requires more than a general ReAct-style coding loop, because ordinary software agents do not capture how wireless engineers carry assumptions from a scenario across radio access choices, physical-layer parameters, simulator code, metrics, and follow-up experiments. *AutoNetSim* adapts this loop into a hierarchical learning process that bootstraps from retrieved tutorials, repositories, and prior cases, runs and repairs simulations with execution feedback, and converts failure cases into auditable skill updates that can be reused on future tasks.

Finally, benchmarking radio environment competence is itself missing because there is no standard benchmark for measuring whether an agent can turn language into executable, physically meaningful wireless simulations. *AutoNetSim* therefore builds a Sionna-centered benchmark in the spirit of execution-grounded software benchmarks. We distill tasks from public tutorials, documentation, and worked examples, stratify them by radio environment, link-level, and system-level difficulty, and evaluate them with verifiers that check both executable artifacts and cross-layer domain constraints.

This paper makes the following three contributions:

1) **Executable radio environment generation.** *AutoNetSim* converts natural-language environment descriptions into simulation-ready radio environments.
2) **Agentic wireless simulation.** *AutoNetSim* turns open-ended user requests into executable cross-layer network simulation and optimization experiments with high pass rate.
3) **Self-improving simulation skills.** *AutoNetSim* improves the agent skill substrate through failure-driven discrete updates scored by execution-based wireless verifiers.

We instantiate *AutoNetSim* on NVIDIA Sionna with Claude Sonnet 4.6 [22] as the primary LLM backbone, and evaluate it on 3D radio environment generation, wireless simulation, and token efficiency. On 3D radio environment generation, *AutoNetSim* reaches 72.5% held-out pass rate, compared with 0.0% for both Naive LLM and Self-Written Skill. Across ten ray-tracing, physical-layer, and network/system-level simulation families under simple natural-language prompts, *AutoNetSim* averages 95.5% pass rate, while Naive LLM stays at 60.0% and Self-Written Skill at 66.5%; on system-level tasks even under heavily engineered prompts *AutoNetSim* retains a 35.5% and 29.0% lead over the two baselines, and reduces generated tokens by 32.0% on average. The codes and demo traces are available on our project page at pervasive-intelligence-lab.github.io/agentic-sionna.

II. Related Work

LLM agents for wireless networks. Recent work has explored LLM agents for wireless-network reasoning, control, and optimization. WirelessAgent [14] introduces a structured perception–memory–planning–action architecture for network slicing, while WirelessAgent++ [15] extends this line by representing agent workflows as executable programs and optimizing them through domain-adapted search. WirelessBench [23] provides a tolerance-aware benchmark for evaluating LLM agents on wireless-network intelligence. ComAgent [18] coordinates

multiple LLMs to produce solver-ready wireless optimization formulations, and other efforts target radio-map generation [16], base-station siting [17], and network-control interfaces [24]. These approaches either rely on predefined agent structures or optimize workflows in a dedicated design stage; they do not continuously convert execution failures into reusable procedural knowledge for future wireless tasks. In contrast, *AutoNetSim* uses verifier feedback and failed simulation trajectories to iteratively improve its procedural skills and memory while grounding this learning in executable 3D radio environments and cross-layer wireless simulations.

LLM-assisted wireless and scientific simulation. RadioSim Agent [25] combines an LLM with deterministic radio simulators and vision-based interpretation, but focuses on interactive analysis of radio maps in existing scenarios. LLM–ns-3 integration instead generates packet-level simulation code [26], while VLM-guided differentiable radio simulation estimates material parameters for a fixed scene [27]. In adjacent scientific domains, FeaGPT [28], AutoMOOSE [29], MCP-SIM [30], and COSMO-Agent [31] demonstrate agentic workflows for finite-element analysis, phase-field modeling, and general physics. These systems establish the value of LLM-assisted simulation, but do not jointly construct a 3D radio environment, execute cross-layer wireless experiments within it, and diagnose the resulting spatial and numerical artifacts.

Learning reusable agent skills. *AutoNetSim* builds on the tool-augmented reasoning paradigm introduced by ReAct [32] and on work that formalizes agent skills as portable procedural capabilities [33], [34]. AFlow searches over executable agent workflows [35], whereas SKILLRL distills successful trajectories into reusable skills [36]. Unlike them, our focus is verifier-guided skill evolution for wireless simulation: a proposed update is retained only when execution-based checks show that it improves simulator-facing artifacts while preserving geometry, radio configuration, output schemas, and cross-layer consistency.

Language-guided 3D scene generation. LayoutGPT [19], FlairGPT [37], Holodeck [20], and OptiScene [21] generate visually plausible indoor environments from language. Their goal is scene layout or embodied-agent interaction, rather than wireless simulation, so they do not assign materials according to radio behavior or verify simulator loadability. In contrast, *AutoNetSim* treats the generated scene as an executable simulation input: objects remain individually addressable, carry radio-material and device metadata, and are checked for geometric validity before downstream wireless experiments.

III. Background: Wireless Simulation

Wireless network simulation ecosystem. We use NVIDIA Sionna as the representative backend because it provides a mature open-source ecosystem for wireless-system research [8], [38]. As shown in Fig. 2, Sionna connects three levels that are often handled by separate tools. At the environment level, its ray-tracing interface builds 3D radio scenes, models propagation through geometry and radio materials, and produces channels and radio maps [9]. At the communication level, its

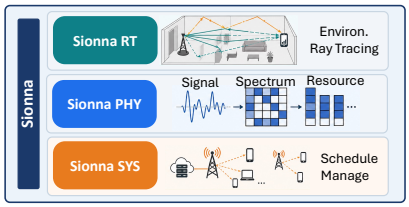


Fig. 2: **Sionna Wireless Network Simulator Architecture.**

physical-layer and link-level APIs support modulation, coding, MIMO processing, OFDM chains, and error-rate evaluation. At the network-abstraction level, its system-level APIs support studies based on physical-layer abstraction.

From radio environment generation to simulation artifacts. A central strength of the ray-tracing simulator is that it can import customized radio environments and generate site-specific channels. To build such a radio environment, we need to build detailed indoor scenes in 3D modelling software (e.g., Blender) by placing the objects that define each room, assigning every object a radio material, importing the scenarios into Sionna with transmitter and receiver locations, and then generating channels [39]. Outdoor workflows impose a similar burden at city scale [40]. The conventional workflow, however, remains engineering-heavy. An engineer must construct or edit the 3D scene, place each mesh at the right scale and pose, assign a radio-relevant material to every object, export a Mitsuba/Sionna-compatible XML scene description, and import that scene into the simulator. The workflow also requires configuring transmitters, receivers, antenna arrays, camera or viewpoint objects, and rendering or radio-map parameters. This process remains difficult because the environment is not a passive visual asset. Many studies further require dynamic scene edits to emulate motion or deployment changes, such as moving furniture, users, or access points, making the workflow even more tedious to set up and maintain.

Why the stack is hard for general agents. Sionna exposes the right abstractions, but it also exposes the main difficulty for LLM agents because correctness is distributed across the whole wireless stack. A valid simulation requires consistency across scene geometry, material models, exported files, device placement, channel assumptions, physical-layer parameters, requested metrics, system objectives, and plotting or reporting code. Many failures are semantic. A scene may load but use visually plausible materials with implausible radio behavior, a script may execute but evaluate a mismatched channel assumption, or a heatmap may look reasonable while answering the wrong deployment question. This is the documentation-to-artifact burden that *AutoNetSim* targets with radio environment skills, Sionna worked examples, and execution-based checks.

IV. System Overview

AutoNetSim is built on two observations. First, modern agents can perform multi-step reasoning, but wireless simulation makes each reasoning chain long because the agent must coordinate geometry, materials, Sionna APIs, plots, metrics, and diagnosis. We use a multi-agent workflow to shorten each worker's reasoning path and improve stability. Second, agents can improve through self-iteration when failures are tied to

executable feedback. We use training cases, verifier reports, and failed trajectories to update a reusable procedural skill.

Fig. 3 shows how these ideas are implemented as a closed-loop system. (1) The Orchestrator converts a natural-language request into a typed simulation plan that records the scenario, required radio environment artifacts, target wireless experiment, cross-layer assumptions, constraints, expected outputs, and stopping condition. It dispatches workers with scoped skills and keeps the session state consistent. (2) The Scene Builder owns the radio environment: it creates or edits a structured scene representation, grounds objects in an asset catalog, assigns radio materials, validates geometry and clearance constraints, and exports simulator-ready artifacts. (3) The Simulator owns executable wireless experiments across ray-tracing, physical-layer, and system-level simulation; it adapts worked examples to the current task, runs the generated code, and records parameters, plots, metrics, and provenance. (4) The Verifier and Reflector check the generated artifacts, interpret heatmaps and numerical metrics, and hand structured diagnoses to the Experiment Planner for follow-up actions such as access-point relocation, carrier changes, or parameter sweeps. (5) The Skill Learner uses verifier and reflection outputs from failed trajectories to update the procedural skill and memory bank. Each subagent can query this memory bank when it is invoked, so successful patterns, failure cases, and accepted skill edits become reusable context for later tasks. This architecture lets *AutoNetSim* behave as a radio environment agent. It can understand a prompt, build the environment implied by the prompt, run the requested wireless simulation inside that environment, check whether the physical, link-level, and system-level artifacts are consistent, and improve from the cases where the workflow fails.

V. System Design

AutoNetSim is built around 4 mechanisms that keep wireless reasoning tied to a coupled radio environment and network stack. First, a multi-agent network simulation and optimization architecture specializes the ReAct paradigm [32] into a wireless plan-act-validate-reflect loop over typed artifacts. Second, a shared RAG memory loads learned skills, prior cases, Sionna templates, API notes, and verifier rules without placing the full memory into every worker context. Third, a radio environment construction skill turns natural language into a structured scene coding task whose outputs can be exported to Sionna-compatible XML. Fourth, a skill-improvement loop converts failed trajectories into auditable updates to the procedural skill.

A. Multi-Agent Network Simulation Architecture

ReAct shows that language agents can interleave reasoning with tool actions [32]. Wireless network simulation and optimization need a stricter variant because the intermediate objects include radio environment states, Sionna scripts, optimization candidates, plots, numerical metrics, and verifier outcomes whose assumptions must remain consistent across propagation, physical-layer, and system-level layers. *AutoNetSim* therefore implements plan-act-validate-reflect as the control logic of a multi-agent architecture with scoped worker contexts. In

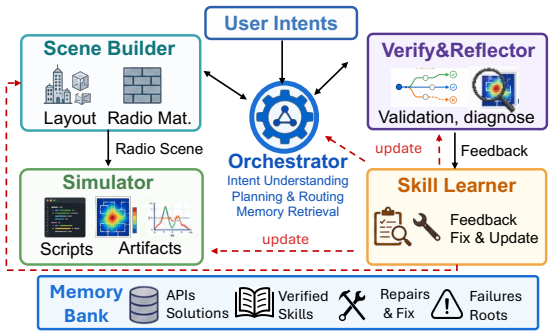


Fig. 3: **Architecture of *AutoNetSim*.** A central Orchestrator dispatches typed tasks to isolated subagents. The outputs form an executable radio environment, bind it to a cross-layer experiment, and feed a skill-learning loop that improves future simulations.

the planning step, the Orchestrator translates the user request into the next typed artifact to produce. In the acting step, a specialized agent edits a scene state, writes Sionna code, runs a simulation, evaluates optimization candidates, or proposes a follow-up experiment. In the validation, deterministic checks and simulator execution decide whether the artifact is acceptable. In the reflection step, the Reflector explains why validation failed or why a successful run suggests the next experiment or optimization move. For example, validation can flag an invalid material assignment or a missing radio map, while reflection can attribute a coverage gap to wall loss, furniture blockage, distance attenuation, or an inconsistent AP placement assumption.

Runtime workflow. At inference time, the architecture in Fig. 3 routes each user intent through five named components. (1) The **Orchestrator** performs intent understanding, planning and routing, and memory retrieval. It converts the request into a typed task plan that records the radio environment, device and RF settings, simulator layer, expected artifacts, and validation conditions. (2) The **Scene Builder** constructs the layout and radio-material assignments, producing a structured radio scene while preserving constraints such as locked objects, device locations, and room dimensions. (3) The **Simulator** writes and executes Sionna scripts and produces simulation artifacts for ray tracing, physical-layer simulation, cross-layer workflows, and optimization sweeps. (4) The **Verify & Reflector** validates prompt preservation, geometry, simulator loadability, execution, finite outputs, and cross-layer consistency, diagnoses failures or follow-up needs, and returns feedback to the Orchestrator. (5) The **Skill Learner** uses this feedback to fix and update procedural skills and records API solutions, verified skills, repairs, and failure root causes in the **Memory Bank**. Because the components exchange typed, inspectable artifacts, each radio scene, simulation artifact, diagnosis, and skill update can be validated, replayed, and reused. For parameter sweeps, the Orchestrator can dispatch multiple Simulator runs in parallel and merge their result schemas.

Shared RAG memory. The agents share an explicit retrieval memory that stores learned procedures, prior cases, Sionna templates, verifier rules, and compact session summaries. This memory contains (1) learned skill blocks, (2) prior successful

and failed trajectories, (3) Sionna templates and API notes, (4) verifier rules, and (5) compact session summaries. Retrieval is necessary because this memory is expected to grow. Injecting the full skill file, all templates, and all failure notes into every worker would be token-expensive and would distract the worker with procedures for unrelated tasks. The RAG layer therefore acts as the multi-agent memory manager by loading only the procedural knowledge needed by the current worker and leaving the rest of the memory out of context.

Task-conditioned hierarchical retrieval. Given a user request, the Orchestrator first normalizes it into a structured query with fields for task family, simulator layer, wireless entities, target artifact, scene context, requested metric, and likely verifier constraints. Following standard RAG and dense retrieval methods [41], [42], the Orchestrator combines the structured query with the latest message. All-MiniLM-L6-v2 [43], [44] encodes this text, averages its token representations, and normalizes the result to produce a 384-dimensional vector. Memory entries are converted into vectors in the same way and stored in ChromaDB [45] using HNSW [46]. The system filters the entries by task family and simulator layer, ranks the remaining entries by cosine similarity to the query, and returns the top four. For example, a request such as "generate a 28 GHz coverage map in a built-in urban scenario and compare two base-station positions" activates the coverage family, the selected Sionna scenario, a RadioMapSolver-style template, heatmap and path-gain outputs, and transmitter-placement checks. Retrieval then proceeds coarse-to-fine. First, a router searches compact module cards and selects one or two regions such as radio environment construction, coverage-map generation, path extraction, cross-layer channel conversion, BER/BLER simulation, MIMO-OFDM, or system-level mobility. Second, a procedure search runs only inside the selected regions to retrieve concrete skill blocks and worked templates. Third, a failure/verifier search retrieves narrow notes keyed to the selected procedure, such as invalid Sionna scene names, incompatible antenna-array dimensions, LDPC code-rate limits, resource-grid shape mismatches, or missing radio-map outputs. The Orchestrator then assembles a worker prompt with the user goal, selected skill excerpt, reference templates, expected artifact schema, and verifier checklist. As the skill and failure library grow, retrieval keeps token cost tied to the task-specific fragments loaded for each worker.

B. Radio Environment Construction

To simulate a network or optimize a deployment, *AutoNetSim* must first construct the radio environment in which candidate devices, propagation paths, and network decisions are evaluated. We therefore design a dedicated radio environment construction skill for simulator-facing scene construction. A naive alternative would be to call an existing language-driven 3D scene generator, such as LayoutGPT, FlairGPT, Holodeck, or OptiScene [19]–[21], [37]. These systems are useful for producing visually plausible layouts or embodied-AI environments, but visual plausibility is not the contract required by wireless simulation. For a radio environment, correctness depends on whether the scene can expose object geometry, poses, material behavior, and device metadata to the simulator and to later deployment-optimization steps.

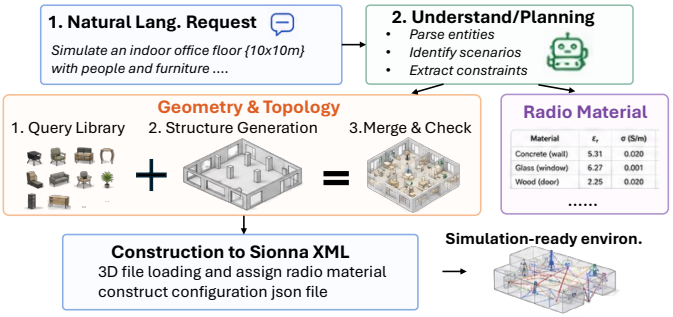


Fig. 4: **Radio environment construction pipeline.**

Network and radio environment simulators, with Sionna as one representative backend, typically require a structured scene with explicit object-level representation. Here, a structured scene means a collection of individually addressable 3D mesh objects or geometric primitives, where each object carries metric geometry, coordinate pose, semantic identity, radio-material assignment, and optional device metadata. This object-level structure is what lets the agent independently edit a transmitter, preserve locked furniture during scene revision, assign a high-loss material to one partition without changing the rest of the room, and verify that every RF-relevant object is covered by a valid material. Naive text-to-3D or image-to-3D generation is therefore insufficient because it may produce an attractive visual artifact while entangling multiple objects into one surface, hiding semantics in texture, omitting controllable radio-material mappings, and failing to expose the scene graph that a wireless simulator or verifier needs.

AutoNetSim converts environment generation into a constrained scene-programming workflow over a structured scene state. As shown in Fig. 4, the Scene Builder (1) parses the prompt into room geometry, architectural elements, furniture requirements, material hints, device topology, and locked user constraints; (2) derives a hybrid geometric representation for architectural envelopes; (3) grounds furniture in a 3D asset catalog with measured dimensions; (4) assigns object-level radio materials and device metadata; (5) optimizes layout under geometric constraints; and (6) exports and validates the resulting scene through simulator-facing middleware. This pipeline keeps the generated environment editable at the object level while still producing the strict simulator artifact needed for ray tracing and downstream network experiments.

Hybrid geometric representation. *AutoNetSim* does not ask the language model to invent every wall mesh directly. Instead, the Scene Builder emits a compact state with (1) a floor polygon or connected room polygons, (2) room height, (3) wall-segment attributes such as windows and doors, (4) radio-material assignments, (5) furniture categories and selected mesh IDs, and (6) device locations. Deterministic geometry code then derives the simulator geometry from this state. For a rectangular room, the floor polygon contains four vertices; for an L-shaped or multi-room task, it contains the additional

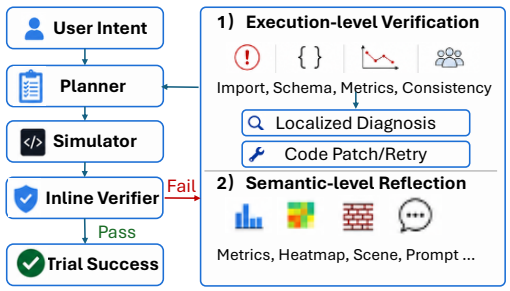


Fig. 5: **Multi-level reflection.** The framework combines execution-level verification and semantic-level reflection to iteratively diagnose, patch, and refine wireless simulation workflows.

concave vertices or connected room polygons. Each adjacent vertex pair becomes a wall segment that is extruded by the room height to form a 3D wall surface, while floors and ceilings are generated from the same polygon. Windows and doors are stored as attributes on wall segments, allowing the geometry code to cut openings and insert glass or door materials while preserving the same wall coordinate system. This representation is hybrid because architectural envelopes are programmatic and consistent, while furniture is selected from the 3D-FUTURE [47] mesh catalog and placed using measured AABBs. The language model therefore controls semantic and RF-relevant choices, such as room shape, material assignment, furniture selection, and placement constraints, while low-level wall geometry is mechanically derived from the floor plan.

Scene middleware. *AutoNetSim* maintains the radio environment as a structured scene state and exports it through multiple middleware formats. A 2D floor plan supports quick human inspection. The Mitsuba representation is the simulator-facing artifact used by the ray-tracing backend. It assigns ITU radio materials to walls, floors, windows, ground, and furniture categories, and reconciles coordinate conventions before Sionna loads the scene. This middleware is important because it gives the agent an editable representation of the environment while still producing the strict file format expected by the simulator.

Layout and material validation. The layout stage uses deterministic constraints for placement and clearance. The optimizer combines in-bounds placement, AABB collision avoidance, door and pathway clearance, wall affinity, room-shape constraints, and user-specified locks. After export, the verifier checks whether the generated scene preserves the prompt, satisfies geometry and capability constraints, covers architectural and furniture objects with valid radio materials, and loads through Sionna. This final check closes the construction pipeline by ensuring that visual plausibility, object semantics, RF material control, and simulator loadability agree in the same artifact.

C. Simulation Reasoning and Skill Improvement

Once a radio environment exists, *AutoNetSim* must generate the cross-layer wireless experiment that answers the user's question. The Simulator does not treat worked examples as fixed templates to copy. Following the multi-step reasoning used by agents, the Orchestrator decomposes a request into a task plan and retrieves relevant knowledge for its individual steps through vector search. A new task may therefore combine knowledge from coverage-map generation, path extraction, link-quality evaluation, or system-level aggregation even when no single example matches the request. The Simulator adapts and composes the retrieved skills, API notes, and worked examples into a new program for the current scene and task. The generated code is screened for valid package imports, network configurations, and consistency with the radio environment and the requested link- or system-level objective. Every successful run emits a result schema containing metrics, plots, simulation parameters, cross-layer assumptions, and provenance.

Reflection in *AutoNetSim* operates at two complementary timescales. *Online reflection* occurs inside a single trial and keeps the agent on a feasible execution trajectory, as shown in Fig. 5. *Offline reflection* operates across trials and distills persistent lessons into the procedural skill that governs all subsequent runs. The two are coupled through a shared verifier: the same checks that gate a single trial's success also produce the structured failure traces that drive offline updates.

Online reasoning with an inline verifier. A runnable script can still answer the wrong wireless question, and an executable script can still terminate with a covertly malformed result. We therefore embed a ReAct-style plan–act–observe loop [32] inside every trial: the Simulator emits a script and an expected result schema, the harness executes it, and the inline verifier inspects both the standard error stream and the structured outputs against the radio environment and user requirements. If execution fails because of an unresolved import, a type or schema mismatch, an infinite or all-zero metric, a dropped user dimension, or a fairness index inconsistent with the reported per-user rates, the verifier returns a localized diagnosis. The agent then edits the code and reruns it within the same turn budget so that low-level bugs are addressed before they propagate to the user. On a semantic-level failure, a multi-modal Reflector consumes numerical metrics, rendered heatmaps, scene context, and the user prompt to diagnose coverage completeness, signal-strength uniformity, and blind-spot location, attributing each blind spot to a closed taxonomy of wall occlusion, furniture blockage, distance attenuation, interference shadow, or material penetration loss. The Experiment Planner closes the loop by running a simulate–reflect–adjust cycle that terminates when the user's target is met, the diagnostic confidence falls below threshold, or the iteration budget is exhausted.

Offline failure-driven skill update. Online reflection corrects per-trial trajectories but cannot prevent the same class of mistake from recurring across sessions. After each training batch with ground-truth oracles, *AutoNetSim* updates the procedural skill, a human-readable Markdown file kept under version control. For each failed trajectory, the verifier's checks, the captured stdout/stderr, and the oracle comparison are fused into a five-tuple (OBSERVATION, FAILURE, CORRECTION, PRINCIPLE, UPDATETARGET). A classifier then routes the trajectory to one of seven classes, namely *skill not consulted*, *consulted but ignored*, *content wrong*, *coverage gap*, *model-capability ceiling*, *environment error*, and *reward hacking*. Only the first three drive skill edits; the rest are routed to model or

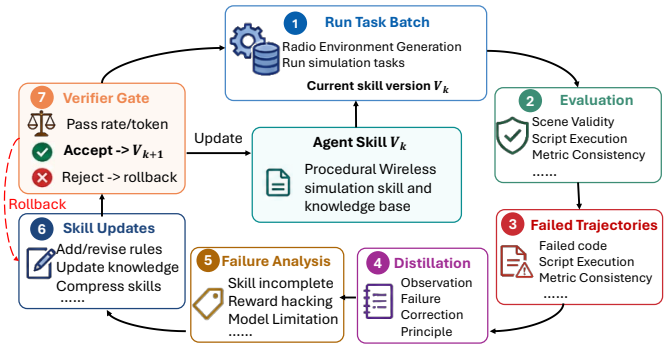


Fig. 6: **Skill improvement as discrete optimization.** *AutoNetSim* learns from failures by editing its procedural skill.

harness upgrades, or to a tighter verifier. The orchestrator picks the dominant editable class, proposes one edit to one active section, reruns the verifier on the train split, and accepts the edit only if pass rate improves by at least two percentage points with no regression. Adopted principles are also embedded into a per-skill memory index for retrieval at inference time, avoiding prompt inflation. Fig. 6 shows this loop.

This loop differs from neural training in three ways. The update unit is a named section, not a gradient step. The rationale is visible in the iteration log. Rollback is a normal version-control operation. These properties are useful in wireless simulation, where users must trust the executed script and inspect the assumptions behind the run.

VI. Implementation

A. Software Prototype

We implement *AutoNetSim* as a Python prototype with a multi-agent plan-act-validate-reflect loop adapted from ReAct [32]. The Orchestrator and each worker communicate through typed JSON artifacts. Scene workers return a structured scene state, simulator workers return runnable scripts and result schemas, reflectors return typed diagnoses, and the Skill Learner returns a proposed Markdown edit plus a verifier report. The initial task templates and procedural skill are constructed from tutorials spanning scene-aware propagation, physical-layer simulation, and system-level network simulation. The scene-generation stack uses 3D-FUTURE assets [47], Trimesh for mesh loading and axis-aligned bounding box (AABB) measurement [48], Shapely and SciPy/SLSQP for geometric constraints and layout optimization [49]–[51], and Mitsuba scene descriptions used by the ray-tracing backend [9], [52], [53]. An AABB is the smallest coordinate-aligned box enclosing a mesh; we use its width, depth, and height to test furniture placement, clearance, and collisions without relying on visual inspection. The SciPy/SLSQP constraints encode continuous layout rules such as keeping objects inside the room polygon, maintaining separation between AABBs, leaving door/pathway clearance, and honoring user-specified object locks. The wireless stack wraps high-level Sionna templates for ray tracing, physical-layer simulation, and system-level simulation, then adapts them to the scenario, devices, metrics, and output schema requested by the user. Each run records the prompt, retrieved references, active skill version, generated artifacts, validation outcomes, and simulator logs so that a failed task can be replayed during skill iteration. The prototype also includes a dashboard that displays the generated 3D radio environment together with simulation results such as coverage maps, traces, and summary metrics.

B. Model and Hardware Backends

For closed-source model experiments, we call hosted APIs. The primary backbone in the reported prototype is Claude Sonnet 4.6, and cross-model checks use OpenAI GPT and Anthropic Claude API endpoints [22], [54]. For open-source model experiments, we serve Llama-3.3-70B-AWQ [55] and Qwen3-Coder-30B-A3B-Instruct [56] on NVIDIA A100 GPUs and run the same benchmark harness, validators, and Sionna jobs against the local endpoint. This split keeps the agent code and verifier code fixed while changing only the model backend. Ray-tracing, physical-layer, and system-level experiments execute on GPU-equipped Python environments with TensorFlow and Sionna 2.0 installed [8], [38], [57]; CPU-only checks are used for structural validation, JSON/schema validation, geometry constraints, and artifact existence.

VII. Evaluation

A. Experimental Benchmark Setup

Benchmark protocol. Each benchmark instance is specified by a natural-language prompt. The prompt describes the simulation or generation goal, the relevant scenario assumptions, and the expected output form. The agent then produces the required artifact for that stage, which may be a structured radio environment, an executable Sionna script and reported metrics, or a structured network/system-level trace. For wireless simulation and optimization tasks, we evaluate each agent under two prompt settings. The *simple prompt* setting gives a concise natural-language description of the scene, devices, and desired metric, but leaves most implementation details implicit. The *enhanced prompt* setting gives an engineer-written structured input with explicit Sionna APIs, physical parameters, output schema, forbidden shortcuts, and verifier-facing requirements. All conditions receive the same task prompt, model backbone, tool access, and verifier harness unless a task explicitly evaluates enhanced prompting.

Baselines. We compare three agent settings: (1) *Naive LLM* is a plain coding agent that receives the task prompt and directly writes the requested scene artifacts or Sionna scripts. (2) *Self-Written Skill* first reads the available Sionna/domain documentation and writes its own procedural skill, then uses that self-authored skill to solve the task. (3) *AutoNetSim* uses our verifier-improved procedural workflow, initialized from tutorials and worked examples and refined through at most five accepted updates before frozen evaluation.

Verification. The benchmark uses execution-based verification with task-specific checks. The primary score is pass rate, defined as the fraction of trials that satisfy all required checks. A trial passes only if it produces the required files, the generated code, the returned schema has the required fields and dimensions, all reported values are finite and accurate, and

TABLE I: Pass rate and error-category rates.

Condition	Pass rate ↑	Geometric error ↓	Structural error ↓	Semantic placement ↓	Executability error ↓
Naive LLM	0.0%	80.0%	27.5%	15.0%	27.5%
Self-Written Skill	0.0%	87.5%	35.0%	12.5%	27.5%
AutoNetSim	**72.5%**	**15.0%**	**17.5%**	**7.5%**	**10.0%**

the task-specific metrics satisfy the verifier. Each task has an independent reference or oracle. Radio environment tasks check structured scene state and Sionna loadability, deterministic simulation tasks are compared with a trusted reference run, and optimization tasks are compared with a brute-force reference. Boundary-condition checks enforce domain sanity across geometry validity, power and rate ranges, monotonic trends, scheduler dimensionality, and fairness arithmetic. An audited tolerance window defines the acceptable difference between a generated result and its reference output for each task family. The same frozen thresholds are applied to *AutoNetSim* and all baselines.

B. Radio Environment Generation Evaluation

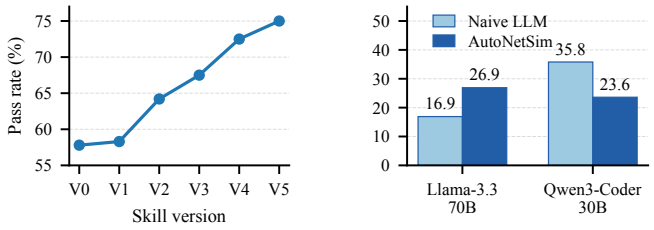
Setup. This stage evaluates whether the agent can turn natural language into a simulator-facing 3D radio environment. The suite contains 100 tasks over 20 seed scene templates, split into 60 train tasks for skill improvement and 40 for test. The tasks abstract common scene-generation operations into seven families, including basic single-room generation, RF-material variation, partitioned rooms, L-shaped rooms, multi-room apartments, building-code-style window constraints, and locked scene edits.

Verification. The verifier checks the structured scene state and exported simulator artifacts. It requires valid artifacts, prompt preservation, legal room polygons, in-bounds and collision-free furniture AABBs, valid radio-material assignments, task-specific geometry constraints, and Sionna loadability. For example, L-shaped, partition, multi-room, and scene-edit tasks are checked against their requested footprint, material, connectivity, and locked-object constraints.

Results. To explain the pass-rate difference, we group failed verifier checks into four non-exclusive categories: (1) *Geometric* errors include object overlap, out-of-bounds placement, and clearance violations; (2)*Structural* errors include incorrect room footprints, partitions, or multi-room connectivity; (3) *Semantic-placement* errors place requested objects in locations inconsistent with the prompt; (4)*executability* errors prevent the exported scene from loading in Sionna.

Table I shows that *AutoNetSim* reduces geometric errors to 15.0%, compared with 80.0% for Naive LLM and 87.5% for Self-Written Skill, through structured scene representation, deterministic placement constraints, and verifier feedback. It also reduces the other error types and achieves a 72.5% pass rate, while neither baseline produces a fully valid environment.

The skill-evolution process on the 60-task training split reveals what the agent learns. As shown in Fig. 7a, the accepted skill iterations steadily improve the pass rate from 57.8% at V0 to 75.0% at V5. Early accepted edits add naming and prompt-preservation rules so verifier-visible objects match the user's request, later iterations compress the capability



(a) **Skill evolution.** (b) **Backbone impact.**

Fig. 7: **Effects of skill evolution and model backbone.** (a) Pass-rate improvement across accepted skill versions. (b) Pass rates of Naive LLM and *AutoNetSim* with two open-source backbones.

glossary so relevant rules remain active without consuming the context budget, and the final edit adds a fast path for locked scene edits. The largest gains occur in window/clearance rules, multi-room topology, and partitioned spaces, where the skill translates natural language into executable radio environment requirements that are not explicit in the original prompt.

C. Ray-Tracing and Physical-Layer Simulation

Setup. The second evaluation stage measures whether *AutoNetSim* can generate executable Sionna experiments for ray tracing and physical-layer simulation. Ray-tracing tasks exercise scene-aware propagation, coverage maps, path extraction, and received-power/SINR metrics [9]; physical-layer tasks exercise link-level computations such as BER/BLER, OFDM throughput, and spectral efficiency [2], [8]. We use four Sionna built-in scenarios, including two indoor testbeds and two outdoor city scenes, which isolates simulation-coding failures from scene-construction failures. The ray-tracing and physical-layer suite contains 28 base prompts; Table III summarizes the simulation and optimization families and their prompt counts. The simulation families cover common radio-map, ray-tracing, and OFDM/NR evaluation practice, while the optimization families instantiate standard beamforming and power-control objectives [2], [9], [58]–[60]. We run each prompt five times under each of the three agent settings.

Verification is tied to physical behavior. For each task, a human-written reference script produces the ground-truth output, and the verifier accepts an agent run only when its reported metrics match that reference within audited tolerance windows and satisfy basic boundary checks. For P1/P2, the reference additionally sweeps the allowed control variables, including antenna orientation, beam angle, and TX power, and compares the agent's selected configuration with the brute-force optimum or Pareto frontier. To complement the binary pass rate, we also measure continuous result error against the reference outputs. We report path-gain MAE and RSS-grid MAE for channel quality, SINR error and BER log-domain error for link quality, and throughput relative error for end-to-end performance. Each metric is pooled over the benchmark cells for which the corresponding output is defined. These metrics reveal how close a result is to the reference even when one verifier condition causes the complete trial to fail.

Results. Table IV reports pass rates by task family, prompt type, and agent condition. Under simple prompts, *AutoNetSim* averages 82.5% across the simulation families N1 to N4,

TABLE II: Continuous wireless-result error (mean ± standard deviation); lower is better.

Condition	Path-gain MAE (dB)	RSS-grid MAE (dB)	SINR error (dB)	BER log-error	Throughput RE (%)†
Naive LLM	1.48 ± 3.11	7.86 ± 10.70	1.27 ± 1.78	0.25 ± 0.71	10.88 ± 16.20
Self-Written Skill	0.97 ± 1.68	7.21 ± 10.05	0.77 ± 1.54	0.02 ± 0.01	8.91 ± 17.91
AutoNetSim	$\mathbf{0.75 \pm 1.93}$	$\mathbf{4.03 \pm 8.53}$	$\mathbf{0.43 \pm 0.95}$	$\mathbf{0.01 \pm 0.00}$	$\mathbf{3.30 \pm 8.17}$

†For throughput, mean ± standard deviation is computed over trials with RE $\leq 100\%$. Larger errors predominantly reflect unit-conversion failures; their rates are 6.9% (Naive LLM), 11.4% (Self-Written Skill), and 5.8% (*AutoNetSim*).

TABLE III: Ray-tracing and physical-layer simulation and optimization task families.

ID	Type	#	Task
N1	Sim.	8	Single-AP coverage map or point-level link probe
N2	Sim.	4	AP configuration edit and before/after rerun
N3	Sim.	4	Multi-AP coverage, interference, or serving-AP assignment
N4	Sim.	4	BER/BLER, throughput, SINR, or spectral efficiency
P1	Opt.	4	AP azimuth/downtilt → maximize one-UE OFDM throughput
P2	Opt.	4	Beam angle + TX power → rate/power Pareto trade-off

compared with 0.0% for Naive LLM and 1.25% for Self-Written Skill. The gain appears on both AP-centric tasks (80.0% on N1 and 90.0% on N2) and harder cross-layer tasks (85.0% on N3 and 75.0% on N4). On the simple optimization families P1/P2, *AutoNetSim* reaches 75.0% average pass rate, and both baselines remain at 0.0%. Enhanced prompts improve the completed N1/N2 simulation rows: Naive LLM averages 35.0%, Self-Written Skill averages 47.5%, and *AutoNetSim* averages 97.5%. The enhanced N3/N4 simulation rows average 22.5%/32.5%/90.0% for Naive LLM, Self-Written Skill, and *AutoNetSim*, respectively, and the enhanced P1/P2 optimization rows average 92.5%/75%/100%. These results support the same conclusion as the system-level study: structured prompts help, but verifier-trained procedural memory is needed for robust executable simulation. The baseline failures are typically analytical shortcuts, such as replacing ray tracing with closed-form path loss or replacing coded-link throughput with a BER curve or theory channel estimate. The verifier rejects these shortcuts because they ignore multipath, wall attenuation, finite-blocklength coding gain, or the required sweep over the specified physical controls. Table II shows that *AutoNetSim* achieves the lowest mean error on all five metrics. The largest differences appear in RSS-grid MAE, which falls to 4.03 dB from 7.86 dB for Naive LLM and 7.21 dB for Self-Written Skill, and in throughput relative error, whose filtered mean falls to 3.30% from 10.88% and 8.91%, respectively.

D. Network/System-Level Verification

Setup. The third stage evaluates whether the agent can preserve assumptions at the network and system level. The network suite has the same two task types, summarized in Table V. Simulation tasks ask the agent to run the network as specified. One reports overall system throughput and per-user rates for a fixed multi-AP/multi-user deployment, and the other reports scheduler behavior, SINR, and fairness metrics under a fixed resource-allocation rule. Optimization tasks ask the agent to change the network configuration. The single-objective task tunes multi-cell beamforming vectors to maximize sum capacity under co-channel interference, while the multi-objective task allocates OFDM resource blocks and power across users to balance total throughput against edge-user fairness. These tasks are different from the ray-tracing and physical-layer stage because the desired answer is often a structured system trace or aggregate network metric.

TABLE IV: Ray-tracing and physical-layer task pass rate.

ID	Naive LLM	Self-Written Skill	*AutoNetSim*
Simulation tasks: simple prompts			
N1	0.0%	5.0%	**80.0%**
N2	0.0%	0.0%	**90.0%**
N3	0.0%	0.0%	**85.0%**
N4	0.0%	0.0%	**75.0%**
Simulation tasks: enhanced prompts			
N1	35.0%	40.0%	**95.0%**
N2	35.0%	55.0%	**100.0%**
N3	25.0%	30.0%	**90.0%**
N4	20.0%	35.0%	**90.0%**
Optimization tasks: simple prompts			
P1	0.0%	0.0%	**75.0%**
P2	0.0%	0.0%	**75.0%**
Optimization tasks: enhanced prompts			
P1	85%	50%	**100%**
P2	100%	100%	**100%**

TABLE V: Network simulation and optimization task families.

ID	Type	#	Task
S1	Sim.	4	Fixed deployment → system throughput and per-user rates
S2	Sim.	4	Fixed scheduler → SINR, RB usage, and fairness trace
S3	Opt.	4	Multi-cell beamforming → maximize sum capacity
S4	Opt.	4	RB scheduling → throughput/fairness trade-off

Verification. Each S trial must pass three verifier layers. First, the expected artifacts must be present and follow the required result schema. Second, the generated artifacts must execute successfully in the benchmark environment and produce finite structured outputs. Third, the verifier compares those outputs against a task-specific oracle. For S1, it checks per-user rates, sum throughput, and Jain's fairness, including fairness self-consistency from the reported rates. For S2, it checks the scheduler trace, per-user throughput, sum throughput, and fairness. For S3, it checks the beamforming sweep table and verifies that the reported best point is supported by the agent's own sweep. For S4, it checks the resource-allocation sweep and verifies that the reported Pareto set contains valid, non-dominated entries. For S1–S3, we additionally use relative throughput error to quantify the distance from the reference beyond the binary tolerance decision; fairness remains checked both against the reference and against the value recomputed from the submitted per-user rates.

Results. Table VI summarizes the system-level stage, reporting pass rates over 20 trials per task family and separating simple-prompt from enhanced-prompt variants. Under simple prompts, *AutoNetSim* attains a perfect 100.0% pass rate on both simulation tasks (S1, S2) and reaches 45.0% and 50.0% on the substantially harder optimization tasks (S3, S4), whereas both baselines collapse to 0.0% across all four families. This contrast indicates that, absent explicit scaffolding, neither a Naive LLM nor a Self-Written Skill can recover the structured outputs required by network/system-level workloads. Enhanced prompts render S1 and S2 tractable for every agent, with all methods clustered between 95% and 100%. The discriminative regime, therefore, lies in the optimization tasks: on S3, *AutoNetSim* achieves 90% versus 35% (Naive LLM) and 55% (Self-

TABLE VI: Network/system-level task pass rate.

Family	Naive LLM	Self-Written Skill	*AutoNetSim*
Simulation tasks: simple prompts			
S1	0.0%	0.0%	**100.0%**
S2	0.0%	0.0%	**100.0%**
Simulation tasks: enhanced prompts			
S1	95%	100%	**95%**
S2	100%	100%	**100%**
Optimization tasks: simple prompts			
S3	0.0%	0.0%	**45%**
S4	0.0%	0.0%	**50%**
Optimization tasks: enhanced prompts			
S3	35%	55%	**90%**
S4	70%	90%	**95%**

Written Skill); on S4, *AutoNetSim* attains 95% versus 70% and 90%, respectively. Averaged over the enhanced-prompt S1–S4 suite, *AutoNetSim* yields a 95.0% pass rate, compared with 75.0% for the Naive LLM and 86.25% for the Self-Written Skill, corresponding to absolute gains of 20.0% and 8.75%. These results corroborate the hypothesis that system-level tasks demand structural consistency of outputs in addition to syntactically executable code. Concretely, an agent may produce a plausible aggregate throughput while silently dropping users, report a fairness index inconsistent with its own per-user rates, or emit malformed scheduling traces.

E. Representative Failure Analysis

Across the 440 trials reported in Tables IV and VI, 59 trials (13.4%) fail. The observed failures include bypassing the requested Sionna solver with an analytical model such as FSPL or TR 38.901, producing incomplete or misaligned structured outputs, terminating before producing a valid result file, and returning values outside the audited tolerance window. These cases show that the agent does not always strictly follow the skill during execution. System-level optimization tasks S3 and S4 account for 24 of the 59 failures (40.7%), reflecting the difficulty of combining ray tracing with multi-user sweeps and Pareto-frontier search. The main remaining weaknesses are therefore enforcing the requested simulator backend and ensuring complete sweeps and structured outputs.

F. Impact of Model

Setup. The main results above use Claude Sonnet 4.6 as the default backbone. To separate the effect of the procedural skill from the effect of the underlying model, we also run cross-model checks while keeping the prompts, tools, verifier, and benchmark harness fixed. Closed-source checks use hosted Claude and OpenAI GPT endpoints, while open-source checks serve the model locally on A100 GPUs.

Results. Fig. 7(b) summarizes the open-source model checks. Llama-3.3-70B improves with *AutoNetSim*, while the Qwen3-Coder-30B partial run regresses under the same harness. The procedural skill exceeded the context window of our local Qwen deployment, so part of the skill was truncated.

G. Efficiency and Usability

We evaluate the efficiency of *AutoNetSim* through token and tool-use accounting. Token efficiency is also part of the skill-optimization objective. When proposing skill updates, *AutoNetSim* tracks token usage and prefers edits that preserve verifier pass rate while reducing generated tokens. The resulting skill can add cached context at input time, but it makes the correct API sequence and verifier-facing invariants available up front, shortening the agent's reasoning and repair trajectory. Table VII reports per-trial means over the ray-tracing and physical-layer Sionna script generation benchmark (N1 to N4 and P1 to P2, 420 trials), using Claude Sonnet 4.6 as the default backbone.

TABLE VII: Token economy and tool calling.

Per-trial mean	Naive LLM	Self-Written	*AutoNetSim*
Cache-read tokens	1.13 M	1.27 M	1.56 M
Live input tokens	53	56	49
Output tokens	12.5 K	12.1 K	**8.5 K**
Tool-use turns	30.1	29.6	**29.1**
Time per turn	6.8 s	7.1 s	7.0 s
Wall-clock time	205 s	210 s	**204 s**

The main trade-off is that *AutoNetSim* reads more cached context but generates fewer new tokens. Compared with Naive LLM, cache-read tokens increase from 1.13 M to 1.56 M per trial (38.1%) because of the multi-turn repair loop and cached procedural skill. In contrast, generated tokens decrease from 12.5 K to 8.5 K, a reduction of 32.0% relative to Naive LLM and 29.8% relative to Self-Written Skill. Tool-use turns also decrease slightly from 30.1 to 29.1 per trial. The three methods require similar wall-clock time: 205, 210, and 204 seconds per trial, respectively. Because human-expert completion time was not measured, we do not claim a speedup over experts.

VIII. Discussion and Conclusion

Discussion: *(a) Backend compatibility: AutoNetSim* is a methodology for constructing and verifying wireless simulation workflows rather than a system that depends on Sionna alone. Its agent coordination, skill management, and verifier-guided refinement can also be used with other wireless simulators. *(b) Real world Alignment:* The current system begins with a natural-language description or a simulator-compatible structured scene and does not process raw geometry captured from the real world. Future work can extend the scene-input stage to support captured environments and evaluate the resulting simulations against measurements. *(c) Scalability:* The present training set and skill bank are relatively small. As the system supports more scenarios and cross-layer tasks, hierarchical retrieval and agent-memory designs [61], [62], together with skill deduplication, merging, pruning, and removal of outdated entries, can help manage a larger skill bank and support more complex workflows.

Conclusion: We presented *AutoNetSim*, a self-evolving agent architecture for intent-driven wireless network simulation. *AutoNetSim* translates natural-language requests into executable wireless simulations and iteratively improves its procedural skills through execution feedback and failed trajectories. More broadly, this work points toward a new direction in which self-evolving agents can serve as intelligent interfaces to complex simulators, potentially benefiting network simulation, optimization, and management beyond the wireless setting.

IX. Acknowledgement

We appreciate the insightful comments and feedback from the anonymous reviewers and shepherd. This work was supported in part by the NSF under Grants 2521925 and 2523752.

References

[1] T. S. Rappaport, *Wireless Communications: Principles and Practice*, 3rd ed. Pearson, 2024.

[2] E. Dahlman, S. Parkvall, and J. Sköld, *5G NR: The Next Generation Wireless Access Technology*, 2nd ed. Academic Press, 2020.

[3] 3GPP, "Study on channel model for frequencies from 0.5 to 100 GHz," 3GPP TR 38.901, Release 17, 2022.

[4] ITU-R, "Effects of building materials and structures on radiowave propagation above about 100 MHz," Recommendation ITU-R P.2040-3, 2023.

[5] T. R. Henderson, M. Lacage, G. F. Riley, C. Dowell, and J. Kopena, "Network simulations with the ns-3 simulator," in *Proceedings of the ACM SIGCOMM Conference Demo Session*, 2008.

[6] Keysight Technologies, "Channel sounding solutions for 5G NR and beyond," https://www.keysight.com, 2024.

[7] Ranplan Wireless, "Ranplan: Indoor wireless network design and optimization," https://www.ranplanwireless.com, 2024.

[8] J. Hoydis, S. Cammerer, F. Aït Aoudia, A. Vem, N. Binder, G. Marcus, and S. ten Brink, "Sionna: An open-source library for next-generation physical layer research," *IEEE Communications Magazine*, vol. 61, no. 3, pp. 52–58, 2023.

[9] J. Hoydis, F. Aït Aoudia, S. Cammerer, M. Nimier-David, N. Binder, G. Marcus, and A. Keller, "Sionna RT: Differentiable ray tracing for radio propagation modeling," *arXiv preprint arXiv:2303.11103*, 2023.

[10] A. Zubow, Y. Pilz, S. Rösler, and F. Dressler, "Ns3 meets Sionna: Using realistic channels in network simulation," *arXiv preprint arXiv:2412.20524*, 2024.

[11] T. Iye, M. Sakamoto, S. Takaya, E. Sato, Y. Susukida, Y. Nagaoka, K. Maruta, and J. Nakazato, "Open wireless digital twin: End-to-end 5G mobility emulation in O-RAN framework," *arXiv preprint arXiv:2503.12177*, 2025.

[12] J. Yang, C. E. Jimenez, A. Wettig, K. Liber, K. Narasimhan, and O. Press, "SWE-agent: Agent-computer interfaces enable automated software engineering," *arXiv preprint arXiv:2405.15793*, 2024.

[13] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh, H. H. Tran, F. Li, R. Ma, M. Zheng, B. Qian, Y. Shao, N. Muennighoff, Y. Zhang, B. Hui, J. Lin, R. Brennan, H. Peng, H. Ji, and G. Neubig, "OpenHands: An open platform for AI software developers as generalist agents," *arXiv preprint arXiv:2407.16741*, 2024, formerly OpenDevin.

[14] J. Tong, W. Guo, J. Shao, Q. Wu, Z. Li, Z. Lin, and J. Zhang, "WirelessAgent: Large language model agents for intelligent wireless networks," *arXiv preprint arXiv:2505.01074*, 2025.

[15] J. Tong, Z. Li, F. Liu, W. Guo, and J. Zhang, "WirelessAgent++: Automated agentic workflow design and benchmarking for wireless networks," *arXiv preprint arXiv:2603.00501*, 2026.

[16] H. Quan, W. Ni, T. Zhang, X. Ye, Z. Xie, S. Wang, Y. Liu, and H. Song, "Large language model agents for radio map generation and wireless network planning," *arXiv preprint arXiv:2501.11283*, 2025.

[17] Y. Wang, Y. Zhang, and W. Chen, "BSS-LLM: Large language model as catalyst for base station siting," *arXiv preprint arXiv:2408.03631*, 2024.

[18] H. Li, M. Xiao, K. Wang, R. Schober, D. I. Kim, and Y. L. Guan, "ComAgent: Multi-LLM based agentic AI empowered intelligent wireless networks," *arXiv preprint arXiv:2601.19607*, 2026.

[19] W. Feng, W. Zhu, T.-J. Fu, V. Jha, A. Calber, M. Chang, and W. Y. Wang, "LayoutGPT: Compositional visual planning and generation with large language models," in *Advances in Neural Information Processing Systems*, vol. 36, 2023.

[20] Y. Yang, F.-Y. Sun, L. Weihs, E. VanderBilt, A. Herrasti, W. Han, J. Wu, N. Haber, R. Krishna, L. Liu *et al.*, "Holodeck: Language guided generation of 3D embodied AI environments," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024.

[21] Y. Yang, Z. Luo, T. Ding, J. Lu, M. Gao, J. Yang, V. Sanchez, and F. Zheng, "LLM-driven indoor scene layout generation via scaled human-aligned data synthesis and multi-stage preference optimization," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

[22] Anthropic, "The Claude model family," *Technical Report*, 2024, https://www.anthropic.com.

[23] J. Tong, F. Liu, L. Xv, S. Lu, K. Li, Y. Zhang, Y. Song, Z. Xue, and J. Zhang, "Wirelessbench: A tolerance-aware llm agent benchmark for wireless network intelligence," *arXiv preprint arXiv:2603.21251*, 2026.

[24] S. Lin, J. Zhou, and M. Yu, "An LLM-based agentic framework for accessible network control," in *AI Crossroads Workshop at ACM SIGMETRICS*, 2025.

[25] S. Hussain and C. Brennan, "RadioSim Agent: Combining large language models and deterministic EM simulators for interactive radio map analysis," *arXiv preprint arXiv:2511.05912*, 2025.

[26] F. Rezazadeh, A. Ashtari Gargari, S. Lagen, H. Song, D. Niyato, and L. Liu, "Toward generative 6G simulation: An experimental multi-agent LLM and ns-3 integration," *arXiv preprint arXiv:2503.13402*, 2025.

[27] Z. Kang, Y. Lim, Z. Gu, S.-W. Ko, T. Q. S. Quek, and J. Park, "Vision-language-model-guided differentiable ray tracing for fast and accurate multi-material RF parameter estimation," *arXiv preprint arXiv:2601.18242*, 2026.

[28] Y. Qi *et al.*, "FeaGPT: An end-to-end agentic-AI for finite element analysis," *arXiv preprint arXiv:2510.21993*, 2025.

[29] S. Manna, H. Chan, and S. K. R. S. Sankaranarayanan, "AutoMOOSE: An agentic AI for autonomous phase-field simulation," *arXiv preprint arXiv:2603.20986*, 2026.

[30] D. Park, H. Moon, and S. Ryu, "MCP-SIM: A self-correcting multi-agent LLM framework for language-based physics simulation and explanation," *npj Artificial Intelligence*, vol. 2, no. 10, 2026.

[31] L. Deng, S. Deng, Y. Chen, Y. Dai, Z. Zhong, L. Li, X. Sun, Y. Shi, and H. Huang, "COSMO-Agent: Tool-augmented agent for closed-loop optimization, simulation, and modeling orchestration," *arXiv preprint arXiv:2604.05547*, 2026.

[32] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "ReAct: Synergizing reasoning and acting in language models," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2023.

[33] R. Xu and Y. Yan, "Agent skills for large language models: Architecture, acquisition, security, and the path forward," *arXiv preprint arXiv:2602.12430*, 2026.

[34] Y. Jiang, D. Li, H. Deng, B. Ma, X. Wang, Q. Wang, and G. Yu, "SoK: Agentic skills – beyond tool use in LLM agents," *arXiv preprint arXiv:2602.20867*, 2026.

[35] X. Zhang *et al.*, "AFlow: Automating agentic workflow generation," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2025.

[36] P. Xia, J. Chen, H. Wang, J. Liu *et al.*, "SKILLRL: Evolving agents via recursive skill-augmented reinforcement learning," *arXiv preprint arXiv:2602.08234*, 2026.

[37] X. Zhou, S. Qian, and W. Feng, "FlairGPT: Repurposing LLMs for interior designs," *arXiv preprint arXiv:2404.01067*, 2024.

[38] NVIDIA, "Sionna documentation," https://nvlabs.github.io/sionna/, 2026, accessed 2026-05-19.

[39] L. Zhang, H. Sun, J. Sun, and R. Q. Hu, "WiSegRT: Dataset for site-specific indoor radio propagation modeling with 3D segmentation and differentiable ray-tracing," *arXiv preprint arXiv:2312.11245*, 2023.

[40] P. Testolina, M. Polese, P. Johari, and T. Melodia, "BostonTwin: The Boston digital twin for ray-tracing in 6G networks," in *Proceedings of the ACM Multimedia Systems Conference 2024*, 2024.

[41] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Kuttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 9459–9474.

[42] V. Karpukhin, B. Oğuz, S. Min, P. Lewis, L. Wu, S. Edunov, D. Chen, and W.-t. Yih, "Dense passage retrieval for open-domain question answering," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2020, pp. 6769–6781.

[43] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence embeddings using siamese BERT-networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 2019, pp. 3982–3992.

[44] W. Wang, F. Wei, L. Dong, H. Bao, N. Yang, and M. Zhou, "MiniLM: Deep self-attention distillation for task-agnostic compression of pre-trained transformers," in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 5776–5788.

[45] Chroma Core, "Chroma: The open-source embedding database," https://www.trychroma.com/, 2023, accessed: 2026-08-16.

[46] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs,"

IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 42, no. 4, pp. 824–836, 2020.
[47] H. Fu, R. Jia, L. Gao, M. Gong, B. Zhao, S. Maybank, and D. Tao, "3D-FUTURE: 3D furniture shape with texture," *International Journal of Computer Vision*, vol. 129, pp. 3313–3337, 2021.
[48] Dawson-Haggerty *et al.*, "Trimesh: Python library for loading and using triangular meshes," https://trimesh.org/, 2024.
[49] Shapely Contributors, "Shapely: Manipulation and analysis of geometric objects in the Cartesian plane," https://shapely.readthedocs.io, 2024.
[50] P. Virtanen, R. Gommers, T. E. Oliphant *et al.*, "SciPy 1.0: Fundamental algorithms for scientific computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.
[51] D. Kraft, "A software package for sequential quadratic programming," *DFVLR Forschungsbericht*, vol. 88, no. 28, 1988.
[52] M. Nimier-David, D. Vicini, T. Zeltner, and W. Jakob, "Mitsuba 2: A retargetable forward and inverse renderer," *ACM Transactions on Graphics*, vol. 38, no. 6, pp. 1–17, 2019.
[53] W. Jakob, S. Speierer, N. Roussel, M. Nimier-David, D. Vicini, T. Zeltner, B. Nicolet, K. Camsari *et al.*, "Mitsuba 3 renderer," *ACM Transactions on Graphics*, 2022.
[54] OpenAI, "GPT-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.
[55] A. Dubey, A. Jauhri, A. Pandey *et al.*, "The Llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.
[56] A. Yang, A. Li, B. Yang *et al.*, "Qwen3 technical report," *arXiv preprint arXiv:2505.09388*, 2025.
[57] M. Abadi, P. Barham, J. Chen *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," https://www.tensorflow.org/, 2015.
[58] Z. Yun and M. F. Iskander, "Ray tracing for radio propagation modeling: Principles and applications," *IEEE Access*, vol. 3, pp. 1089–1100, 2015.
[59] M. Chiang, C. W. Tan, D. P. Palomar, D. O'Neill, and D. Julian, "Power control by geometric programming," *IEEE Transactions on Wireless Communications*, vol. 6, no. 7, pp. 2640–2650, 2007.
[60] Q. Shi, M. Razaviyayn, Z.-Q. Luo, and C. He, "An iteratively weighted MMSE approach to distributed sum-utility maximization for a MIMO interfering broadcast channel," *IEEE Transactions on Signal Processing*, vol. 59, no. 9, pp. 4331–4340, 2011.
[61] P. Sarthi, S. Abdullah, A. Tuli, S. Khanna, A. Goldie, and C. D. Manning, "RAPTOR: Recursive abstractive processing for tree-organized retrieval," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024.
[62] C. Packer, V. Fang, S. G. Patil, K. Lin, S. Wooders, and J. E. Gonzalez, "MemGPT: Towards LLMs as operating systems," *arXiv preprint arXiv:2310.08560*, 2023.

Supplementary Materials

AutoNetSim: Intent-Driven Wireless Network Experimentation with Self-Evolving Agents

These supplementary materials provide additional reproducibility details for the paper "AutoNetSim: Intent-Driven Wireless Network Experimentation with Self-Evolving Agents." Section I presents the prototype interface and representative prompts across the benchmark task families, and Section II reports the complete frozen acceptance rules used by the benchmark verifier.

I. System Interface and Prompt Specifications

This section shows how requests are presented to the agent and how a user inspects an AutoNetSim run. The request and prompt examples make the evaluation inputs concrete, while the dashboard illustrates the user-facing interface.

A. Representative Dashboard Request

Table S1 gives the request used to generate the interactive example. It specifies the scene, controls, overlays, and reported metrics shown in the dashboard.

TABLE S1: Representative interactive simulation prompt.

Prompt
Create an indoor Sionna dashboard for a furnished room. Build the 3D radio environment with walls, furniture, and a TX marker; expose web controls for transmit power, array size, TX position, coverage-grid height/depth/cell size, z-slice, and layer visibility; run the ray-tracing computation on demand; overlay the coverage heatmap and optional ray paths in the 3D viewport; and report BER-vs-SNR, coverage distribution, PDP/OFDM traces, peak signal, mean loss, coverage, frequency, and array statistics so the user can adjust the scenario and recompute.

B. Representative Benchmark Prompts

The following prompts show how the benchmark expresses simulation and optimization requests. The two prompt conditions use the same task and verifier: an enhanced prompt provides implementation details, output fields, and prohibited shortcuts, whereas a natural-language prompt states the scenario and requested outputs more concisely.

1) S1 Fixed Deployment: Enhanced Prompt:

```
TASK: Compute system-level downlink metrics for a 2-AP /
4-UE deployment
on the Sionna built-in scene "simple_street_canyon" using
sionna.rt.PathSolver.

PARAMETERS:
- Scene: sionna.rt.scene.simple_street_canyon
  (outdoor street canyon ~186x121 m)
- AP positions: AP0=(-15.0, 0.0, 15.0), AP1=(+15.0, 0.0,
15.0) m
- UE positions: (-20.0, 0.0, 1.5), (-5.0, 0.0, 1.5),
  (+5.0, 0.0, 1.5), (+20.0, 0.0, 1.5) m
- Frequency: 5 GHz; TX power per AP: 0 dBm; noise floor:
-85 dBm
- Antenna: PlanarArray(1, 1, "iso", "V"); max_depth=3;
  samples_per_src=500000

COMPUTATION:
1. PathSolver per (AP, UE) -> path_gain_db =
   10*log10(sum |a|^2 over valid paths). paths.a = (a_re,
   a_im).
2. serving_ap[j] = argmax over APs by received_dbm =
   tx_power_dbm + path_gain_db.
3. SINR with the other AP as interference + thermal noise.
   Shannon rate = log2(1+SINR_linear).
4. sum/mean throughput; Jain's fairness.

OUTPUT FILES: simulation.py, simulation_result.json with
schema
{task_family="S1_fixed_deployment",
scene_name="simple_street_canyon",
ap_positions, ue_positions, tx_power_dbm=0.0,
noise_dbm=-85.0,
frequency_hz=5e9, path_gain_db (2x4), serving_ap (len-4
ints),
sinr_db (len-4), per_user_rate_bps_hz (len-4),
sum_throughput_bps_hz, mean_throughput_bps_hz,
fairness_index,
method="sionna_rt"}.

CONSTRAINTS:
- MUST use Sionna RT. Do NOT fall back to FSPL or 3GPP TR
38.901
  closed-form.
- \$RF_SIONNA_PY simulation.py to run.
- paths.a is (a_re, a_im), not polarization.
```

2) S1 Fixed Deployment: Natural-Language Prompt:

```
Two access points are deployed in the simple_street_canyon
scene at
(-15, 0, 15) m and (+15, 0, 15) m, both with
omnidirectional isotropic
antennas at +0 dBm, with a thermal noise floor of -85 dBm.
Four users
sit on the ground at (-20, 0, 1.5), (-5, 0, 1.5), (+5, 0,
1.5),
(+20, 0, 1.5) m. Each user associates with the highest-RSS
access
point. Compute per-user SINR, per-user Shannon rate, total
system
throughput, and Jain's fairness index. Use Sionna's ray
tracer
(sionna.rt.PathSolver) for every (AP, UE) path gain in this
simulation;
the verifier compares against a Sionna reference and will
reject
closed-form FSPL or 3GPP TR 38.901 path-loss output.
```

3) N2 Frequency Edit: Enhanced Prompt:

```
TASK: Compute a baseline single-AP coverage map at 5 GHz on
the Sionna
built-in scene "box_two_screens", then re-run the same
setup at 2.4 GHz
and emit a delta map showing the frequency-induced coverage
change.

SETUP (identical at both frequencies):
- Scene: sionna.rt.scene.box_two_screens
- AP: (0.0, 0.0, 4.5) m; TX power: 20 dBm
- Antenna: PlanarArray(1, 1, "iso", "V")
- RadioMapSolver: max_depth=3, samples_per_tx=100000,
  cell_size=(0.25, 0.25) m

SOLVER CALL:
- Use the default grid derived from the scene bounding box.
Do not pass
  center or size; the verifier uses the default settings.

STEP 1: Set scene.frequency=5e9, run RadioMapSolver,
convert the map to
dBm, and save coverage_5ghz.npy and coverage_5ghz.png.
STEP 2: Set scene.frequency=2.4e9 without reloading the
scene, rerun the
same solver, and save coverage_2ghz.npy and
coverage_2ghz.png.
STEP 3: Compute coverage_delta=coverage_2ghz-coverage_5ghz
and save a
zero-centered RdBu_r visualization as coverage_delta.png.

OUTPUT: simulation.py and simulation_result.json containing
scene_name,
edit_action="frequency 5GHz -> 2.4GHz", freq_before_hz,
freq_after_hz,
before/after RSS min/max/mean, delta_dbm_mean, and
method="sionna_rt".

CONSTRAINTS:
- MUST use Sionna RT for both frequencies.
- Set scene.frequency before each solver call.
- Do not reload the scene between the two runs; retune it
in place.
```

4) N2 Frequency Edit: Natural-Language Prompt:

TABLE S2: Task-specific audited tolerance windows used by the benchmark verifiers.

Task family	Compared quantity	Acceptance window or boundary
General physical bounds	Reported wireless metrics	BER/BLER and Jain's fairness lie in $[0, 1]$; coverage lies in $[0, 100]$%; RSS lies in $[-120, 30]$ dBm and does not exceed transmit power by more than 5 dB; throughput lies in $[0, 30]$ bps/Hz; and SINR lies in $[-20, 60]$ dB.
N1–N3: coverage/RSS	Radio-map cells and summary statistics	MAE over jointly valid cells ≤ 3 dB; at least 80% of jointly valid cells within ± 5 dB; mean difference ≤ 3 dB; standard-deviation difference ≤ 2 dB; valid-cell fraction difference ≤ 25 pp.
N1–N3: physical trends	Frequency and material comparisons	Low-frequency coverage may be at most 5 pp below high-frequency coverage; drywall coverage may be at most 3 pp below concrete coverage.
N1/N3: path statistics	Path gain, path count, mean delay, and delay spread	Path-gain error ≤ 2 dB; path-count difference ≤ 2 or recovered count ≥ 70% of reference; mean-delay relative error ≤ 15%; delay-spread relative error ≤ 20%.
N2: configuration edit	Before/after coverage or link change	Reported change differs from the change recomputed from the submitted before/after results by at most 2 pp (or 2 dB for power-domain metrics).
N4: BER/BLER curve	SNR grid and per-point curve values	Maximum SNR/EbN0 grid mismatch ≤ 0.1 dB; per-point error ≤ 0.5 decades in the log domain or relative error ≤ 15%; at least four curve points must pass.
P1: single-objective optimization	Best throughput and submitted sweep	Best throughput is between 95% and 105% of the reference optimum; path-gain MAE ≤ 3 dB on at least 12 of 15 matched sweep configurations; the reported optimum must appear in the submitted sweep within 0.51° and 0.02 bps/Hz.
P2: rate/power Pareto optimization	Pareto frontier and submitted sweep	Hypervolume ≥ 90% of the reference; returned Pareto-set size ≥ 50% of the reference set; path-gain MAE ≤ 3 dB on at least 12 of 15 matched sweep configurations.
S1: fixed deployment	Sum throughput and fairness	Sum-throughput relative error ≤ 30%; fairness absolute error ≤ 0.15; reported fairness differs from fairness recomputed from per-user rates by at most 0.02.
S2: fixed scheduler	Sum throughput and fairness	Sum-throughput relative error ≤ 40%; fairness absolute error ≤ 0.20; reported fairness differs from fairness recomputed from per-user throughput by at most 0.03.
S3: beamforming optimization	Best sum rate and submitted sweep	Best sum rate is between 85% and 115% of the reference; the reported optimum must appear in the submitted sweep within 0.51° per beam angle and 0.05 bps/Hz.
S4: network Pareto optimization	Submitted Pareto set	All indices must be valid and all returned points must be non-dominated; returned Pareto-set size must be at least 50% of the reference set.

```
Using Sionna's built-in box-with-two-screens indoor
testbed, place a
base-station transmitter at (0, 0, 4.5) m with 100 mW
transmit power and
isotropic antenna arrays. Generate the 5 GHz coverage map,
then retune
the transmitter to 2.4 GHz and generate that coverage map.
Return both
maps and the per-cell delta.
```

5) Additional Natural-Language Prompts: N1: single-AP radio map.

```
Using Sionna's built-in box-with-two-screens indoor
testbed, place a
base-station transmitter at (0, 0, 4.5) m with 100 mW
transmit power and
isotropic antenna arrays. Generate the 5 GHz coverage map
for this base
station and return the map and visualization.
```

N4: physical-layer BER curve.

```
Run a Sionna PHY simulation with uncoded QPSK modulation
over an AWGN
channel. Compute the bit error rate at Eb/N0 = [0, 2, 4, 6,
8] dB using
approximately one million bits per SNR point. Return the
BER curve.
```

P1: single-objective orientation optimization.

```
An indoor low-power access point sits at (-2, 0, 4.5) m and
a user is at
(2, 0, 1.5) m inside the box_one_screen scene. The AP can
rotate its
antenna over a 5x3 grid of azimuth and downtilt angles.
Find the
orientation that maximizes the user's downlink throughput
and report the
full sweep. Use Sionna RT for each path gain and Sionna PHY
(LDPC5G,
Mapper, and AWGN) for throughput; do not substitute FSPL or
Shannon rate.
```

P2: rate/power Pareto optimization.

```
An indoor access point sits at (-2, 0, 4.5) m and a user is
at
(2, 0, 1.5) m inside the box_one_screen scene. The AP can
choose its
antenna azimuth and transmit power. Find the Pareto-optimal
trade-off
between downlink throughput and transmit power, and report
the full sweep
and all non-dominated configurations. Use Sionna RT for
path gain and
Sionna PHY (LDPC5G, Mapper, and AWGN) for throughput; do
not substitute
FSPL or Shannon rate.
```

C. Interactive Dashboard Example

Fig. S1 shows how the prototype presents the generated 3D radio environment, coverage results, traces, and summary metrics in one interface. A user can inspect these outputs, adjust the scenario, and rerun the simulation; the dashboard itself is not used to determine benchmark pass rates.

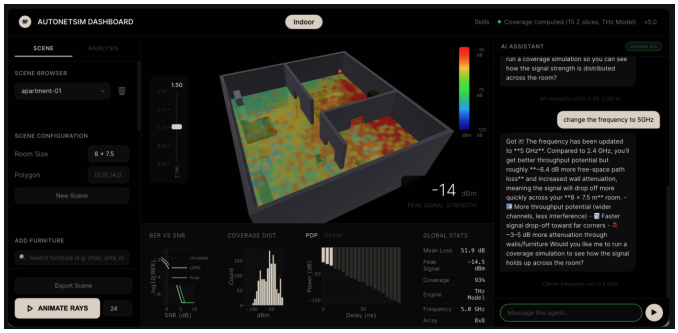


Fig. S1: **Interactive radio environment dashboard.** The prototype dashboard displays the generated 3D radio environment together with simulation results.

II. Complete Verifier Thresholds

This section records the complete frozen acceptance rules used to reproduce the benchmark decisions. An audited tolerance window is the task-specific acceptable deviation between an agent-generated result and its independently produced reference output; exact equality is inappropriate for stochastic sampling and numerical computation. Table S2 lists the compared quantity and numerical or structural boundary for every task family. The general physical bounds apply to every trial, and each task-family row adds the checks specific to that task. A trial passes only when every applicable condition is satisfied. The verifier applies these rules identically to *AutoNetSim* and all baselines; "pp" denotes percentage points.