TeleSWEBench: A Commit-Driven Benchmark for Evaluating LLM-Powered Software Engineering in Telecommunications

Pranshav Gajjar NCSU, USA
Ali Mamaghani UC San Diego, USA
Dinesh Bharadia UC San Diego, USA
Vijay K Shah NCSU, USA

Abstract

With the telecommunications field embracing *zero-touch* management alongside novel O-RAN and AI-RAN frameworks, contemporary telecom networks now function as immensely intricate and heavily softwarized codebases. While automated software engineering (ASE) tools and Software Engineering (SWE) *Agents* hold the potential to alleviate the critical code generation bottleneck in this domain, their ability to navigate and modify specialized, mathematically rigorous wireless stacks like `srsRAN 5G` remains unverified. General-purpose coding benchmarks fail to capture the stateful logic and strict requirements of telecommunications, leaving a critical evaluation gap. In this paper, we introduce **TeleSWEBench**, the first commit-driven benchmark specifically designed to measure an agent's performance in the telecom domain. We mine real developer commits from the `srsRAN 5G` repository and distill them into structured test cases across three difficulty tiers (Easy, Medium, and Difficult). Our benchmark consists of 734 questions that are accompanied by executable unit tests. To avoid the rigidity of test cases, we further propose a hierarchical LLM-as-a-Judge framework called *TeleJudge* that scores agent outputs at the file level and aggregates verdicts holistically. This follows an evaluation based on context and semantic similarity in parallel to a standard unit test-based evaluation. Using this benchmark, we evaluate AIDER, OpenHands, and the Claude Code frameworks, powered by state-of-the-art reasoning LLMs, including Qwen, GPT-OSS, Gemma 4, Kimi, GLM, and Qwencoder 2.5. Our two-stage evaluation reveals that models suffer from a lack of both localization accuracy and functional correctness, with the strongest ASE tools achieving up to 25% of shippable changes.

1 Introduction

The telecommunications industry is undergoing a fundamental paradigm shift toward fully autonomous, *zero-touch* network provisioning and management. Driven by the advent of Open RAN (O-RAN) (Tripathi and Shah, 2025) and the emerging Artificial Intelligence-Radio Access Network (AI-RAN) (Feng et al., 2026) architectures, traditional proprietary telecom hardware is being rapidly supplanted by highly softwarized, cloud native network functions (Tripathi and Shah, 2025). Consequently, modern wireless networks, from the physical layer up to the core, are essentially massive, highly complex codebases deployed on commercial off-the-shelf hardware (Polese et al., 2023). As the demand for dynamic resource allocation, continuous protocol upgrades, and rapid feature deployment accelerates, the primary bottleneck in realizing these next-generation networks has shifted from hardware limitations to automated software engineering and code generation.

Concurrently, the rapid advancement of Large Language Models (LLMs) has catalyzed a revolution in automated software engineering (ASE). Autonomous *SWE* agents can now navigate codebases, generate multi-file patches, and resolve standard GitHub issues with increasing autonomy (Hou et al., 2024; Zhang et al., 2026). These tools promise to alleviate the critical software bottleneck in telecom by automating the maintenance and expansion of network stacks. However, the efficacy of these general-purpose agents remains largely unverified when applied to the mathematically rigorous, highly specialized, and hardware-adjacent domain of wireless communications. Wireless software stacks, such as the widely adopted open-source `srsRAN 5G` (Paisana et al., 2026; Gajjar and Shah, 2025b), impose unique and unforgiving constraints: strict asynchronous timing requirements, intricate state machine management, layered protocol hierarchies, and rigid adherence to complex 3GPP standardization documents (Baron and Gupta, 2018; Ganiyu et al., 2025).

Despite the proliferation of these automated cod-

Proceedings of the 15th International Joint Conference on Natural Language Processing and the 5th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics, pages 4911–4925
November 6-10, 2026 ©2026 Association for Computational Linguistics

ing tools, the research community lacks a systematic methodology to evaluate their true utility in this domain. Existing software engineering benchmarks, such as SWE Bench (Jimenez et al., 2023), MBPP (Austin et al., 2021), and HumanEval (Chen et al., 2021), evaluate models primarily on generic web development tasks, standard algorithms, or popular Python libraries. They fundamentally fail to capture the domain-specific nuances and cascading cross-file dependencies inherent to a 5G network stack. Existing instruction tuning evaluations, when applied to telecom, often produce shallow or factually incorrect questions.

As a result, we are left with a critical evaluation gap: *While we have access to powerful automated software coding tools, there is currently no quantitative way to know which models actually work for softwarized network generation, nor how they fail when confronted with complex telecom logic*. To bridge this gap, we introduce **TeleSWEBench**[1], the first commit-driven benchmark specifically designed to evaluate LLM-powered ASE frameworks and SWE Agents in the telecommunications domain. Rather than relying on synthetic or instruction-tuned queries, TeleSWEBench is constructed by mining real-world developer commits from the `srsRAN 5G`[2] repository (Paisana et al., 2026), distilling authentic bug fixes, feature additions, and protocol updates into structured test cases across three difficulty tiers.

We establish a rigorous, two-stage evaluation pipeline that decouples the capability of an agent to navigate a complex repository from its ability to write factually correct domain logic. By structuring the benchmark this way, we can measure scope localization entirely independently of functional correctness. Furthermore, we propose a novel hierarchical LLM-as-a-Judge framework called `TeleJudge`, which is rigorously validated against repository native executable unit tests to accurately score multiple file patches. Through extensive evaluation of state-of-the-art models within the AIDER (paul-gauthier et al., 2026), OpenHands (Wang et al., 2025) and Claude Code (Liu et al., 2026) frameworks, we expose a significant capability gap in modern LLMs regarding domain-specific code generation. Ultimately, this benchmark provides a critical foundation for the future development of telecom native SWE agents.

[1]The benchmarking suite is available at `https://teleswebench.github.io/`.

[2]The leading open-source O-RAN 5G solution from SRS with an AGPL-3.0 license.

2 Related Work

Recent years have seen a proliferation of benchmarks designed to evaluate the coding capabilities of Large Language Models (LLMs). Early datasets, such as HumanEval (Chen et al., 2021), focused on isolated, single-function Python generation tasks, evaluating models on *164* hand-crafted programming problems. As model capabilities advanced, the focus shifted from standalone function synthesis to repository-scale software engineering. SWE-bench (Jimenez et al., 2023) introduced a paradigm shift by requiring models to resolve real-world GitHub issues by navigating multi-file codebases and generating functional patches, offering *2,294* evaluation instances. SWE-Bench has been further extended to other benchmarking suites like (Yang et al., 2024b; Zhang et al., 2025). Similarly, benchmarks such as RepoBench (Liu et al., 2023) target repository-level code auto-completion and context retrieval, with execution-based variants such as ExecRepoBench (Yang et al., 2024a) that evaluate *1,200* repository-level unit tests. The current gold standard for evaluation is SWE-Bench-Verified (Chowdhury et al., 2024), which comprises *500* samples verified by experts, and all prominent benchmarks, along with their scale, are shown in Figure 1.

While these SWE benchmarks provide excellent testbeds for general-purpose programming and web development logic, they predominantly rely on Python or Java environments. They fundamentally lack the hardware-adjacent, stateful, and mathematically complex C++ paradigms required in telecommunications. Another important aspect is evaluating the multi-file patches generated by autonomous agents. While execution-based evaluation, like compiling and running unit tests, remains the gold standard (Yang et al., 2024b; Chowdhury et al., 2024), it becomes inherently difficult to evaluate domains or code repositories where **no** public unit tests are available. Consequently, the use of strong LLMs as judges (Gu et al., 2024; Li et al., 2024, 2025) has become a standard practice for evaluating generative LLM outputs for multiple domains, including software development (He et al., 2026). The core intuition here is that finding problems in anything is a *significantly* easier task than creating something new; hence, using a model that is only prompted to find issues with targeted instructions is

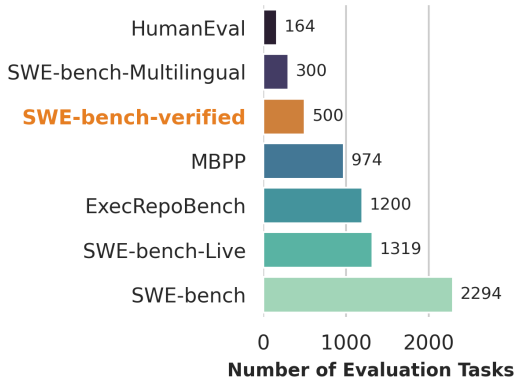


Figure 1: Comparison of the total number of evaluation tasks across prominent Software Engineering (SWE) benchmarks.

capable of reaching an expert-level performance in evaluation. Despite rapid advancements in general software engineering, the application and evaluation of LLMs within the telecommunications domain have historically been restricted to Natural Language Processing (NLP) and static information retrieval. Consequently, existing telecom evaluation frameworks are overwhelmingly confined to Multiple-Choice Question Answering (MCQA) formats. Benchmarks such as ORANBench (Gajjar and Shah, 2025a) and TeleQnA (Maatouk et al., 2025), alongside holistic evaluation suites introduced by the GSMA (GSMA, 2026), are strictly designed to test a model's ability to parse complex 3GPP standardization documents, summarize 5G architectures, or answer theoretical protocol queries.

When it comes to actual codebase evaluation in telecom, efforts remain severely limited. The most notable initiative is srsRANBench (Gajjar and Shah, 2025b); however, it strictly measures static code understanding and repository comprehension, completely lacking any assessment of end-to-end code generation. To address domain-specific telecom challenges, several fine-tuned, telecom-adapted models have recently been developed, such as TelecomGPT (Zou et al., 2025) and ORANSight (Gajjar and Shah, 2025b). Notably, ORANSight attempts to bridge the software gap by explicitly including the srsRAN codebase in its pretraining corpus. Yet, despite this domain-specific exposure, the practical application and evaluation of these systems remain restricted to code comprehension or the generation of isolated, single-function snippets. Current telecom AI frameworks do **not** assess the critical capabilities required for autonomous network management: end-to-end code generation, active multi-file codebase modification, cross-layer compilation, and the management of cascading execution logic necessary to maintain a softwarized protocol stack. TeleSWEBench directly addresses this void by moving beyond MCQA and static comprehension, providing the first rigorous testbed for functional, repository-scale code generation in telecommunications.

3 TeleSWEBench

We leverage `srsRAN 5G` as the foundation for our benchmark due to its prominence as an industry-standard open-source wireless stack. We sourced a comprehensive history of repository modifications spanning from 2022 to 2025, yielding a total of over 15k commits. Within this extensive history, the average number of file changes per commit stands at 5.57. This high degree of multi-file modification underscores the immense complexity and tightly coupled nature of telecommunication software. Unlike standard application development, where logic might be isolated to a single localized script, modifying a modern cellular protocol stack inherently requires synchronized updates across various interconnected layers, state machines, and hardware interfaces. Consequently, this statistical reality clearly demonstrates that evaluating autonomous software engineering agents in the telecommunications domain demands a benchmark capable of rigorously testing broad codebase navigation and deep repository context. By anchoring our evaluation on authentic developer activity with such high cross-file dependencies, we ensure that models are tested on the genuine architectural rigors they would face in real-world network deployment. A comprehensive overview of the TeleSWEBench pipeline, from commit mining to evaluation, is illustrated in Figure 2.

3.1 Benchmark Generation

We partition benchmark items into three difficulty levels, each corresponding to how much of the solution is spelled out in the natural language task and how much the model must infer from the repository. *Easy* tasks are written as near mechanical edit instructions. A valid easy prompt must give exact file paths and line references, specify precisely which code to remove and which code to insert, and remain fully self-contained so that, in principle, the change could be carried out with

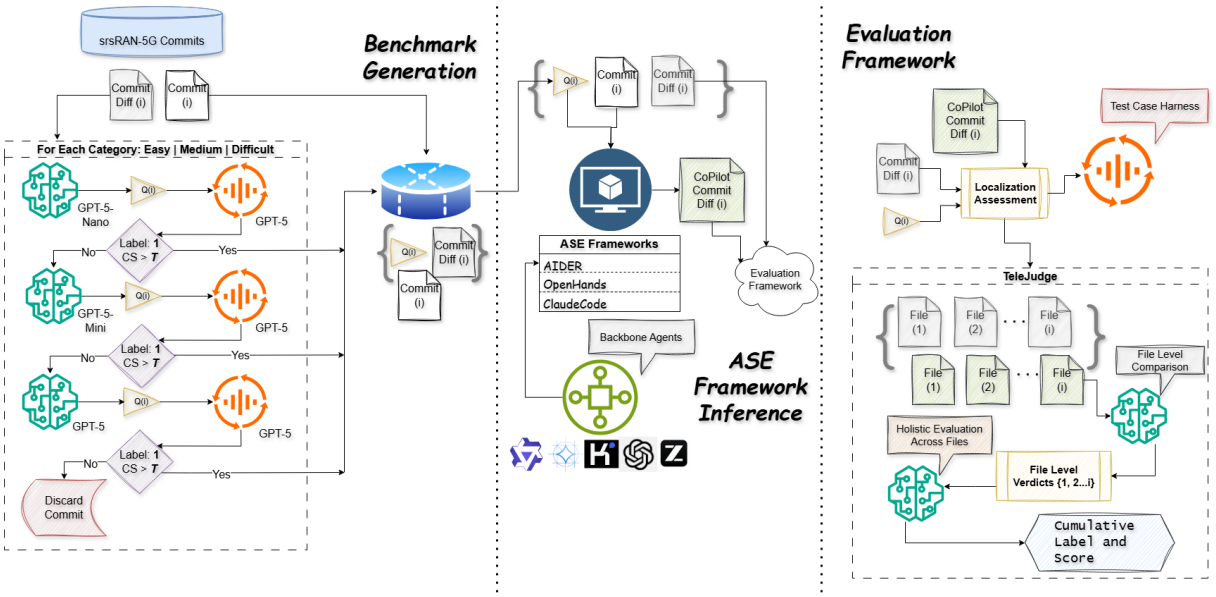


Figure 2: High-level overview of TeleSWEBench from benchmark creation to evaluation.

straightforward search and replace style operations without additional exploration. *Medium* tasks sit between specification and discovery. They explain the *what* and *why* of the change, name the affected files or subsystems, and supply salient facts such as constants, function names, or version identifiers, but deliberately omit exact line-level edit recipes. Answering them is assumed to require some structural understanding of where the change belongs. *Difficult* tasks state only high-level objectives and minimal hints such as a target value, endpoint, or version. They must not name specific file paths or functions, so the assistant must locate the right implementation context and plan edits autonomously. This tier is intended to stress repository-scale reasoning rather than literal transcription of a patch description.

Generation is not fixed to a single frontier model. For each commit and each difficulty, we use a fallback chain over model depth, attempting generation once per model from smaller to larger before escalating. If a cheaper model produces a question that passes validation, we accept it and stop. Only on failure do we invoke the next, more capable and typically more expensive model. This *cascade*, inspired by the paper (Jung et al., 2025), keeps average generation cost low while still allowing difficult commits or strict validator outcomes to trigger stronger models when needed. For Validation, a language model is given the same commit context, the generated question, and the declared difficulty. The validator checks whether the text is unambiguous, carries an information density appropriate to that difficulty, is plausible as a task for an IDE coding assistant, is faithful to the actual commit diff, and is free of internal contradictions. In the confidence scoring variant, the validator additionally returns a calibrated confidence score. We only retain items that are marked valid and meet a fixed *confidence threshold* of T, which is kept as 0.9, which further reduces false positives in the benchmark. We apply a second heuristic on top of the model cascade. We generate and validate difficulties in the order *difficult*, then medium, then easy, with early stopping. We only attempt medium if hard has already succeeded for that commit, and we only attempt easy if medium has succeeded. If *difficult* fails, we abandon that commit for the benchmark. If hard succeeds but medium fails, we do not proceed to easy.

This ordering reflects how constraining each tier is for both the generator and the validator. *Difficult* questions allow vague, goal-oriented wording and forbid leaking file and function names, so it is intuitive to believe that they are relatively easy to phrase in a way that remains consistent with the commit. *Easy* questions, by contrast, must satisfy the strongest structural requirements such as exact locations, exact removals and insertions, and full self-containment, which are harder to generate correctly and harder to validate as faithful to the patch. *Medium* sits between the two. By front-loading hard, we quickly filter commits that cannot support a coherent, difficult task. Furthermore, detailed

prompts used for the LLM inference are available in Appendix C

3.2 ASE Framework Inference

We specifically leverage Command Line Interface (CLI) based ASE frameworks, namely AIDER (paul-gauthier et al., 2026), OpenHands (Wang et al., 2025), and Claude Code (Liu et al., 2026), with the former two being open source. These tools have rapidly established themselves as the standard multi-agent frameworks across numerous Automated Software Engineering (ASE) studies (Shen et al., 2025; Zhang et al., 2024; Wadhwa et al., 2024), providing the essential scaffolding required for iterative repository navigation, contextual file reading, and autonomous patching. To ensure a fair and computationally feasible evaluation pipeline, we enforce a strict time-bound execution limit, defining a computational time cutoff $Time_m$ for all inference runs; if an agent fails to generate a final patch within this $Time_m$ threshold, the task is forcefully terminated. Powered by these frameworks, we benchmark a diverse suite of open-weight backbone LLMs with SOTA performance in code generation and reasoning, across different parameter scales, including Qwen3.5, Gemma4, Kimi, GLM, GPT-OSS, and Qwencoder2.5, further explained in Section 5. We evaluate backbone models from 1.5B to 1T parameters and varied context windowsto systematically identify how parameter scaling and extended context utilization impact an agent's success rate.

3.3 Evaluation Framework

We propose a two-stage evaluation pipeline, as shown in Figure 2, designed to explicitly decouple an agent's architectural navigation skills, identifying *where* to make changes, from its ability to synthesize functionally correct domain logic. The first stage evaluates task localization, measuring whether the agent correctly understood the structural scope of the required modifications. Let T represent the set of ground truth target files modified in the original developer commit, and P represent the set of files modified by the ASE framework. We classify localization performance into one of five mutually exclusive states. An **Exact Match** occurs when the agent successfully isolates the exact required files ($P = T$). A **Partial Match** ($P \cap T \neq \emptyset$ and $T \not\subseteq P$) indicates that the agent grasped elements of the problem and modified some correct files, possibly alongside unrelated ones, but failed to trace the required logic across the broader codebase. Conversely, an **Over Addressed** state ($T \subset P$) emerges when the agent correctly alters the target files but also inappropriately modifies unrelated files. Finally, we categorize complete localization failures as either a **No Match** ($P \cap T = \emptyset$, given $P \neq \emptyset$) when completely disjoint files are altered, or **No Changes** ($P = \emptyset$) when the agent merely generates text without any code changes.

The second stage evaluates functional correctness and is strictly reserved for outputs that achieve an Exact Match, ensuring that our assessment of code generation capabilities is not confounded by poor file localization. For these candidates, we first evaluate using a **Test Case Harness** across our benchmark, which consists of 734 questions that are accompanied by executable unit tests. We achieve this by natively compiling the agent's generated patch and executing it against the rigorous `srsRAN 5G` unit and integration test suites. However, rigid unit tests can often penalize the creativity of an SWE Agent, as they may fail functionally valid solutions that deviate structurally from the developer's original approach. To avoid penalizing this creativity, we further propose an LLM-as-a-Judge framework called **TeleJudge** that initially evaluates the diffs at the granularity of individual files. It then aggregates these micro-assessments into a holistic patch verdict (Pass or Fail) by reasoning about cross-file compatibility and structural coherence. We also evaluate other LLM-as-a-judge methods and show how these approaches often succumb to context overload and fail to produce reasonable output. Preliminary results regarding TeleJudge are available in Appendix B.

4 Benchmark Statistics

From the entire commit history, we can fetch all available commits that have repository-native unit tests, which comprise a small fraction of the entire commit database[3]. As the question generation process follows a heuristic, we can successfully generate Difficult questions for 313 commits, Medium questions for 279 commits, and Easy questions for 142 commits. The average number of files per question varies across these difficulty levels. Specifically, the *Easy* split has a mean of 29.82 files and a median of 7. The Medium split features a mean of 19.57 files with a median of 4, while the Diffi-

[3]More information regarding the test cases is available in Appendix F.

cult split averages 18.42 files with a median of 5. Across all generated questions, the scope ranges from a minimum of 1 file to a maximum of 300 files per question. Sample questions along with the correct code changes are available in Appendix E

5 Experimental Setup

All experiments were conducted on a dedicated workstation equipped with an Intel(R) Core(TM) i9-14900KF CPU, 62 GiB of RAM, and an NVIDIA GeForce RTX 4090 GPU with 24 GB of GDDR6X memory. We utilized the backbone models detailed in Table 1, employing the specific quantization formats available through the NAUTILUS framework[4] (Weitzel et al., 2025) to optimize for our local hardware limits. The only exception is the QwenCoder2.5, which is evaluated locally using Ollama (Yang et al., 2026). Furthermore, to adhere to the constraints of our available compute resources, we enforced a strict maximum execution limit $Time_m$ of 3600 seconds for the system to report a final answer. To first understand why the LLM Agents fail, we primarily leverage the **AIDER** framework because it is fully open-source and the most computationally efficient when compared to other open-source options, including OpenHands (Ni et al., 2026). Upon identifying the best-performing backbone model, we conducted an extended ablation study by replacing AIDER with **OpenHands** and **Claude Code**. This secondary evaluation phase allowed us to determine the extent to which the ASE framework and backbone architecture dictate telecom-specific success.

Model	Parameters	CL	Quant
Qwen 3.5 (Team, 2026)	397B	1M	FP8
Qwen 3.5 Small (Team, 2026)	35B	1M	FP8
GPT-OSS (OpenAI et al., 2025)	120B	131K	MXFP4
Gemma 4 (Farabet, 2026)	31B	262K	FP16
Kimi K2.5 (Team et al., 2026)	1T	262K	MXFP4
GLM 4.7 (Team et al., 2025)	358B	202K	FP8
QwenCoder 2.5 (Hui et al., 2024)	1.5B	32K	Int4

Table 1: Parameters, maximum context lengths (CL), and quantization formats (Quant) for the evaluated backbone LLMs.

Metrics Building upon the evaluation framework established in Section 3.3, we formalize our assessment into distinct metrics for both stages of the pipeline. For Stage 1, we primarily measure task localization using the *Exact Match* (EM) rate, which represents the proportion of questions where the correct files are modified ($P = T$). Given N completed questions and N_{EM} instances of exact matches, EM is defined as $\text{EM} = N_{\text{EM}}/N$. Similarly, we calculate the rates for Partial Match (PM), Over Addressed (OA), No Match (NM), and No Changes (NC) by dividing their respective occurrence counts by N. For Stage 2, we evaluate the code generation capabilities strictly on the subset of N_{EM} candidates. To measure functional correctness, we define the *Unit Test Acceptance Rate* (UAR). If N_{UT} represents the number of generated patches that successfully compile and pass all associated unit and integration tests, UAR is calculated as $\text{UAR} = N_{\text{UT}}/N_{\text{EM}}$. Finally, to quantify the outputs processed by our TeleJudge framework, we define the *TeleJudge Acceptance Rate* (TAR). This metric identifies the proportion of structurally creative and accurate solutions that pass the holistic verdict. Letting N_{pass} denote the number of patches accepted by the judge for file level and finally the holistic verdict, TAR is given by $\text{TAR} = N_{\text{pass}}/N_{\text{EM}}$. To capture end-to-end deployability, we introduce the *Ship-Ready Percentage* (SRP), which jointly requires semantic acceptance and executable correctness on the same exact-match subset. Let N_{SR} denote the number of Stage 2 candidates (from N_{EM}) whose patches are *accepted* by TeleJudge and also *pass* all required build and test checks. Then $\text{SRP} = \frac{N_{\text{SR}}}{N_{\text{EM}}}$. Formally, let $i \in \{1, \ldots, N_{\text{EM}}\}$ index the exact-match instances, which gives us $N_{\text{SR}} = \sum_{i=1}^{N_{\text{EM}}} \mathbf{1}[\text{JudgeAccept}_i \wedge \text{TestPass}_i]$ so SRP measures the proportion of candidates that are simultaneously holistically correct and functionally verified. This makes SRP a stricter end-to-end metric than either UAR or TAR alone.

6 Results and Analysis

6.1 Stage 1 Localization Performance

Table 2 presents the comprehensive results for the initial localization stage across all difficulty tiers. A clear trend emerges regarding the inverse relationship between task complexity and localization success. Across all models evaluated, the Exact Match (EM) rates degrade significantly as the scope of the tasks shifts from Easy to Difficult. Concurrently, the No Match (NM) and No Changes (NC) metrics exhibit a sharp increase in the more complex tiers. For example, QwenCoder 2.5 achieves a

[4]NAUTILUS serves Qwen3.5 and Qwen3.5 Small under the model identifiers `qwen3` and `qwen3-small`; we retain these identifiers in Table 2 and Figures 4-5.

Table 2: Stage 1 Localization Results across Difficulty Tiers. Runs that ended in `timeout` are excluded.

Model	Easy					Medium					Difficult					Cumulative				
	EM	PM	OA	NM	NC	EM	PM	OA	NM	NC	EM	PM	OA	NM	NC	EM	PM	OA	NM	NC
`Qwen3`	24.0	19.2	0.0	0.0	56.7	25.0	27.0	3.2	4.8	39.9	1.3	6.7	0.7	16.2	75.1	14.0	16.5	1.5	9.2	58.7
`Qwen3-small`	1.6	11.9	0.8	0.8	84.9	7.4	14.8	0.0	2.6	75.3	1.0	2.3	0.0	8.7	88.0	3.5	8.8	0.1	5.0	82.6
`Kimi-K2.5`	5.3	22.8	0.0	0.0	71.9	13.8	19.5	0.8	4.1	61.8	0.9	6.4	0.0	6.0	86.8	5.3	12.6	0.2	4.6	77.3
`Gemma4`	1.5	9.2	0.0	0.0	89.2	7.6	16.5	0.0	0.6	75.3	0.3	1.4	0.0	5.4	92.9	2.8	7.2	0.0	3.2	86.8
`GPT-OSS`	19.7	13.1	0.8	1.6	64.8	19.9	24.0	2.6	4.5	49.1	0.0	5.4	0.0	14.4	80.2	11.0	13.8	1.1	8.4	65.7
`QwenCoder-2.5`	37.8	21.9	2.1	20.3	17.9	20.4	6.4	1.9	32.6	38.7	5.3	4.0	1.7	43.5	45.5	17.4	8.1	1.8	34.8	37.8
`GLM-4.7`	3.2	29.0	0.0	0.0	67.7	6.0	4.0	0.0	0.0	90.0	0.0	2.6	0.0	0.5	96.8	1.5	5.9	0.0	0.4	92.2

37.8% EM rate on Easy tasks, but this performance drops to a mere 5.3% on Difficult tasks, accompanied by a corresponding surge in its NM rate to 43.5%. This demonstrates that as the necessary context spans deeper across the repository hierarchy without explicit file hints, models increasingly fail to isolate the target logic. Interestingly, our evaluation highlights a counterintuitive dynamic regarding model scale and localization efficacy. The smallest model in our evaluation suite actually localizes remarkably well and acts more decisively to modify the codebase. In contrast, the larger, structurally more capable models demonstrate a highly reserved performance profile. This is heavily reflected in their disproportionate NC rates; for instance, GLM-4.7 and Gemma4 output cumulative NC rates of 92.2% and 86.8%, respectively. We hypothesize that this phenomenon is due to the larger foundation models being overly aligned for safety and conversational timidity. When faced with the dense, interconnected C++ architecture of the telecom repository, these models become overly cautious and refuse to execute concrete modifications. We provide qualitative examples of these interactions in Appendix A.

6.1.1 Failure Points

Beyond the quantitative performance drops observed in the localization metrics, we identified three distinct behavioral failure modes that prevent models from successfully resolving tasks. As illustrated in Figure 3, the first prevalent point of failure across the *modern* LLMs (except `QwenCoder-2.5`) occurs when the model successfully deduces which files and lines require modification, but strictly outputs plain-text suggestions or pseudocode rather than generating the actionable code edits, resulting in a *timid* behavior. The second failure mode involves models generating excessively verbose outputs that lack actionable code suggestions, frequently causing the model to halt execution to ask clarifying counter-questions instead of proceeding autonomously. Finally, the third failure point is context exhaustion. Given the massive scale of the srsRAN 5G repository, the iterative file exploration required for Medium and Difficult tasks overwhelms the context windows of the agents, leading to truncated reasoning loops and aborted runs before any edits can be localized.

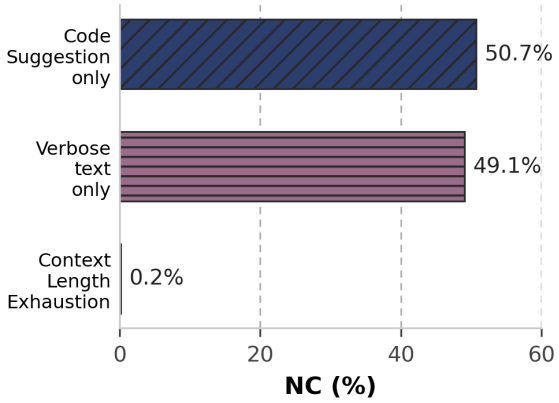


Figure 3: Distribution for the three failure points across all *NC* outputs.

6.2 Stage 2 Functional Correctness

By restricting our Stage 2 evaluation strictly to instances where models achieved an Exact Match in localization, we assess true functional correctness through three complementary metrics: UAR, TAR, and SRP. Relying solely on unit tests or qualitative judgments fails to capture true deployment readiness, making this joint evaluation critical for assessing telecom code quality. Figure 4 illustrates a pronounced gap between executable and semantic correctness across the evaluated models.

Crucially, it must be noted that the overall absolute performance across all models remains exceedingly low; our analysis here primarily explores relative behavioral differences rather than identifying a genuinely deployment-ready system. `GLM-4.7` emerges as the strongest relative performer, yet it only achieves an end-to-end SRP of 25.0%. How-

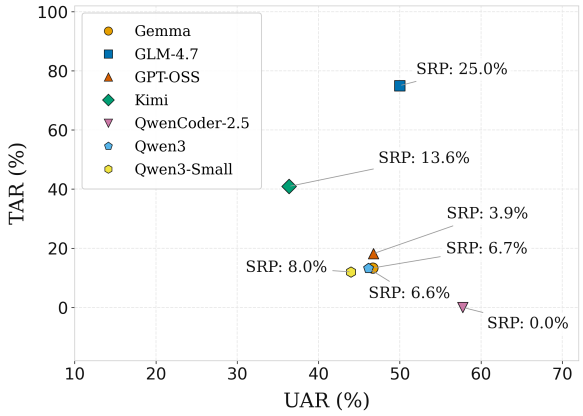


Figure 4: Each marker represents a model plotted by *UAR* on the x axis and *TAR* on the y axis and the annotations report the *SRP*.

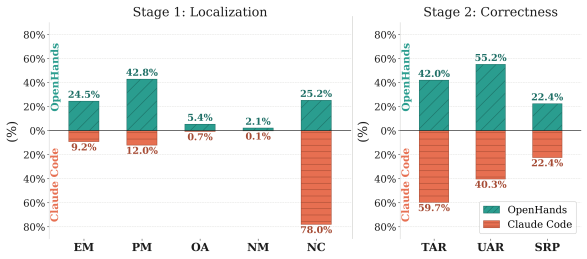


Figure 5: Performance comparison between OpenHands and Claude Code ASE frameworks using the `qwen3` backbone.

ever, this result directly contextualizes the timidity barrier identified in our Stage 1 analysis. Although `GLM-4.7` exhibited a massive NC rate exceeding 90%, this extreme caution translates into higher relative precision.

On the rare occasions when the model actually commits to an edit and correctly localizes the scope, its solutions are comparatively reliable and holistically sound. In contrast, `QwenCoder-2.5` embodies the exact opposite behavioral profile. While Stage 1 showed it to be highly decisive and willing to modify code, its Stage 2 performance reveals severe operational brittleness. The model achieves a relatively strong UAR (57.73%), successfully compiling code that passes the native test suites, but it completely fails the TeleJudge semantic evaluation with a TAR of 0.0%, resulting in an SRP of 0.0%. This discrepancy indicates that the model frequently generates narrow workarounds that blindly satisfy isolated unit tests but catastrophically violate broader cross-file compatibility, stateful logic, and strict C++ telecom conventions. The remaining models fall between these two extremes, none of which perform exceptionally well. `Kimi-K2.5` demonstrates moderate alignment across all metrics (UAR $= 36.36\%$, TAR $= 40.91\%$, SRP $= 13.64\%$). Meanwhile, models like `Qwen3-small` and `Gemma` exhibit comparable test execution behavior with UAR $\approx 44-47\%$ but suffer from drastically lower judge acceptance, yielding single-digit SRP scores. Ultimately, the persistent divergence between TAR and UAR underscores that passing unit tests alone is vastly insufficient for automated software engineering in the telecom domain.

6.3 OpenHands and Claude Code

To further investigate the impact of an ASE framework on code generation within the telecommunications domain, we conducted an extended case study utilizing the OpenHands and Claude Code frameworks backed by our top-performing foundation model from Table 2, `qwen3`. We do observe that `QwenCoder-2.5` provides a better EM rate, but due to a 0% SRP, we choose `qwen3`. As shown in Figure 5, the performance varies significantly across agent architectures. Utilizing AIDER resulted in a TAR of 13.2, a UAR of 46.2, and an overall SRP of 6.6. The alternatives delivered superior results, confirming that the ASE framework heavily influences efficacy. Additionally, AIDER recorded an EM rate of 14.0 and an NC rate of 58.7. These eclipse Claude Code, a result we attribute to *timidity*, where the tool frequently avoids making changes but maintains high correctness when it does modify the codebase.

7 Conclusion

TeleSWEBench bridges a critical gap in automated software engineering evaluation by introducing the first quantitative commit-driven benchmark tailored specifically for the telecommunications domain. By mining real-world developer commits, we constructed 734 structured test cases categorized into three distinct difficulty tiers to rigorously evaluate agent capabilities. Our comprehensive two-stage evaluation assessing both task localization and functional correctness reveals that current agents severely lack the domain-specific reasoning required to reliably patch complex wireless stacks like `srsRAN 5G`. Even the most advanced ASE frameworks struggled significantly, with the best-performing tools achieving up to a 25% success rate for producing shippable code. We observed that while models can sometimes handle

basic tasks, their performance degrades drastically on difficult tiers where they frequently fail due to context exhaustion, timidity, or a tendency to output plain text explanations instead of executable code edits.

For future work, we wish to address the Limitations mentioned below by expanding the benchmark to a wider array of repositories and broadening our baselines to encompass commercial LLM models and ASE frameworks.

Limitations

Single Repository. Currently, our benchmark is built exclusively around the srsRAN 5G repository. While srsRAN serves as a highly robust, representative, and widely used open-source telecom stack, restricting our dataset to a single codebase may narrow the scope of our findings. The unique architectural choices, coding conventions, and documentation styles specific to srsRAN might not fully capture the diverse complexities present in other open-source telecom projects or proprietary, industry-grade software, potentially limiting the broader generalizability of the evaluated agents.

Evaluated Frameworks. Furthermore, our baseline evaluations are restricted to only three ASE frameworks, omitting other closed-source or commercial alternatives. This decision was strictly necessitated by the prohibitive costs associated with executing large-scale, iterative software engineering tasks, which often require continuous API calls, environment interactions, and extensive token generation on paid, proprietary models. Although including other state-of-the-art closed-source frameworks might yield higher performance baselines, utilizing open-source frameworks ensures that our current evaluation pipeline remains fully reproducible and financially accessible to the broader research community.

Reliance on Existing Test Cases. The unit tests utilized in our benchmark are extracted directly from the `srsRAN-5G` repository rather than being custom-authored specifically for each evaluation task. Because we rely on these pre-existing developer tests, they may not be perfectly tailored to catch every edge case or unintended side effect introduced by an AI agent's specific modifications. As a result, the functional correctness evaluation might occasionally pass flawed code or fail structurally unconventional but functional code.

Infrastructure Constraints and Timeouts. Finally, as detailed in Section 5, our experiments relied on the NAUTILUS platform for model API access. Because NAUTILUS does not provide strict Service Level Agreements (SLAs), API calls occasionally drop or exit unexpectedly, leading to premature ASE tool timeouts. While hosting these large models locally could have mitigated these network-related failures and potentially allowed agents to fully complete their coding tasks, our limited compute resources precluded large-scale local deployment. Consequently, our Stage 2 performance evaluation is only conducted on the Exact Match (EM) samples where generation was completed, and the appropriate files were changed.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Justus Baron and Kirti Gupta. 2018. Unpacking 3gpp standards. *Journal of Economics & Management Strategy*, 27(3):433–461.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. 2024. Introducing SWE-bench verified.

Clement Farabet. 2026. Gemma 4: Byte for byte, the most capable open models.

Chenyuan Feng, Howard H. Yang, Kun Guo, Wenchao Xia, Chenxi Liu, and Tony Q.S. Quek. 2026. Ai-ran: The pathway to future wireless networks. *Journal of Information and Intelligence*, 4(1):5–22.

Pranshav Gajjar and Vijay K Shah. 2025a. Oran-bench-13k: An open source benchmark for assessing llms in open radio access networks. In *2025 IEEE 22nd Consumer Communications & Networking Conference (CCNC)*, pages 1–4. IEEE.

Pranshav Gajjar and Vijay K Shah. 2025b. Oransight-2.0: Foundational llms for o-ran. *IEEE Transactions on Machine Learning in Communications and Networking*.

Abiodun Ganiyu, Pranshav Gajjar, and Vijay K Shah. 2025. Ai5gtest: Ai-driven specification-aware automated testing and validation of 5g o-ran components. In *18th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 53–64.

GSMA. 2026. Release v1.1.0 · gsma-labs/evals.

Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, and 1 others. 2024. A survey on llm-as-a-judge. *The Innovation*.

Junda He, Jieke Shi, Terry Yue Zhuo, Christoph Treude, Jiamou Sun, Zhenchang Xing, Xiaoning Du, and David Lo. 2026. Llm-as-a-judge for software engineering: Literature review, vision, and the road ahead. *ACM Transactions on Software Engineering and Methodology*.

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8):1–79.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, and 5 others. 2024. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

Jaehun Jung, Faeze Brahman, and Yejin Choi. 2025. Trust or escalate: LLM judges with provable guarantees for human agreement. In *The Thirteenth International Conference on Learning Representations*.

Dawei Li, Bohan Jiang, Liangjie Huang, Alimohammad Beigi, Chengshuai Zhao, Zhen Tan, Amrita Bhattacharjee, Yuxuan Jiang, Canyu Chen, Tianhao Wu, and 1 others. 2025. From generation to judgment: Opportunities and challenges of llm-as-a-judge. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 2757–2791.

Haitao Li, Qian Dong, Junjie Chen, Huixue Su, Yujia Zhou, Qingyao Ai, Ziyi Ye, and Yiqun Liu. 2024. Llms-as-judges: a comprehensive survey on llm-based evaluation methods. *arXiv preprint arXiv:2412.05579*.

Jiacheng Liu, Xiaohan Zhao, Xinyi Shang, and Zhiqiang Shen. 2026. Dive into claude code: The design space of today's and future ai agent systems. *arXiv preprint arXiv:2604.14228*.

Tianyang Liu, Canwen Xu, and Julian McAuley. 2023. Repobench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091*.

Ali Maatouk, Fadhel Ayed, Nicola Piovesan, Antonio De Domenico, Merouane Debbah, and Zhi-Quan Luo. 2025. Teleqna: A benchmark dataset to assess large language models telecommunications knowledge. *IEEE Network*.

Ziyi Ni, Huacan Wang, Shuo Zhang, Shuo Lu, Ziyang He, Zhenheng Tang, Sen Hu, Bo Li, Chen Hu, Binxing Jiao, and 1 others. 2026. Gittaskbench: A benchmark for code agents solving real-world tasks through code repository leveraging. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pages 32564–32572.

OpenAI, :, Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K. Arora, Yu Bai, Bowen Baker, Haiming Bao, Boaz Barak, Ally Bennett, Tyler Bertao, Nivedita Brett, Eugene Brevdo, Greg Brockman, Sebastien Bubeck, and 108 others. 2025. gpt-oss-120b & gpt-oss-20b model card. *Preprint*, arXiv:2508.10925.

Francisco Paisana, Fabian Eckermann, Pedro Alvarez, Robert Falkenberg, Xavier Arteaga, Alfredo Sáez, Supreeth Herle, carlo-gal, AlaiaL, Andre Puschmann, dvdgrgrtt, faluco, jonathan-srs, Joaquim Broquetas, Justin Tallon, Piotr, Pavel, Oriol Font-Bach, Nils Fürste, and 10 others. 2026. srsran/srsRAN_Project. https://github.com/srsran/srsRAN_Project.

paul-gauthier, Andrew Grigorev, Joshua Vial, fry69, Krazer, shladnik, IT Lackey, Titusz, Claudia Pellegrino, Daniel Vainsencher, xqyz, Peter Schilling, Jay Alammar, Antti Kaihola, Matteo Landi, Farhan Ahmad, MDW, Kenny Dizi, Ivan Fioravanti, and 11 others. 2026. Aider-AI/aider. https://github.com/Aider-AI/aider.

Michele Polese, Leonardo Bonati, Salvatore D'oro, Stefano Basagni, and Tommaso Melodia. 2023. Understanding o-ran: Architecture, interfaces, algorithms, security, and research challenges. *IEEE Communications Surveys & Tutorials*, 25(2):1376–1411.

Chihao Shen, Connor Dilgren, Purva Chiniya, Luke Griffith, Yu Ding, and Yizheng Chen. 2025. Secrepobench: Benchmarking code agents for secure code completion in real-world repositories. *arXiv preprint arXiv:2504.21205*.

GLM Team, Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, Kedong Wang, Lucen Zhong, Mingdao Liu, Rui Lu, Shulin Cao, Xiaohan Zhang, Xuancheng Huang, Yao Wei, and 152 others. 2025. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models. *Preprint*, arXiv:2508.06471.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, S. H. Cai, Yuan Cao, Y. Charles, H. S. Che, Cheng

Chen, Guanduo Chen, Huarong Chen, Jia Chen, Jiahao Chen, Jianlong Chen, Jun Chen, Kefan Chen, Liang Chen, Ruijue Chen, Xinhao Chen, and 307 others. 2026. Kimi k2.5: Visual agentic intelligence. *Preprint*, arXiv:2602.02276.

Qwen Team. 2026. Qwen3.5: Accelerating productivity with native multimodal agents.

Nishith D Tripathi and Vijay K Shah. 2025. *Fundamentals of O-RAN*. John Wiley & Sons.

Nalin Wadhwa, Atharv Sonwane, Daman Arora, Abhav Mehrotra, Saiteja Utpala, Ramakrishna B Bairi, Aditya Kanade, and Nagarajan Natarajan. 2024. Masai: Modular architecture for software-engineering ai agents. In *NeurIPS 2024 Workshop on Open-World Agents*.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, and 5 others. 2025. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*.

Derek Weitzel, Ashton Graves, Sam Albin, Huijun Zhu, Frank Wuerthwein, Mahidhar Tatineni, Dmitry Mishin, Elham Khoda, Mohammad Sada, Larry Smarr, Thomas DeFanti, and John Graham. 2025. The national research platform: Stretched, multi-tenant, scientific kubernetes cluster. In *Practice and Experience in Advanced Research Computing 2025: The Power of Collaboration*, PEARC '25, New York, NY, USA. Association for Computing Machinery.

Jian Yang, Jiajun Zhang, Jiaxi Yang, Ke Jin, Lei Zhang, Qiyao Peng, Ken Deng, Yibo Miao, Tianyu Liu, Zeyu Cui, and 1 others. 2024a. Execrepobench: Multi-level executable code completion evaluation. *arXiv preprint arXiv:2412.11990*.

John Yang, Carlos E Jimenez, Alex L Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R Narasimhan, and 1 others. 2024b. Swe-bench multimodal: Do ai systems generalize to visual software domains? *arXiv preprint arXiv:2410.03859*.

Michael Yang, Jeffrey Morgan, Daniel Hiltgen, Bruce MacDonald, Jesse Gross, Patrick Devine, Parth Sareen, Matt Williams, Devon Rifkin, Blake Mizerany, Michael, Eva H, Josh, royjhan, frob, Jeremy, Grace, nicole pardal, Dane Madsen, and 11 others. 2026. ollama/ollama. https://github.com/ollama/ollama.

Kexun Zhang, Weiran Yao, Zuxin Liu, Yihao Feng, Zhiwei Liu, Rithesh Murthy, Tian Lan, Lei Li, Renze Lou, Jiacheng Xu, and 1 others. 2024. Diversity empowers intelligence: Integrating expertise of software engineering agents. *arXiv preprint arXiv:2408.07060*.

Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, and 1 others. 2025. Swe-bench goes live! *arXiv preprint arXiv:2505.23419*.

Quanjun Zhang, Chunrong Fang, Yang Xie, Yaxin Zhang, Shengcheng Yu, Weisong Sun, Yun Yang, and Zhenyu Chen. 2026. A survey on large language models for software engineering. *Science China Information Sciences*, 69(4):141102.

Hang Zou, Qiyang Zhao, Yu Tian, Lina Bariah, Faouzi Bader, Thierry Lestable, and Merouane Debbah. 2025. Telecomgpt: A framework to build telecom-specific large language models. *IEEE Transactions on Machine Learning in Communications and Networking*.

A Timidness

In our evaluation, a recurring behavioral failure mode is *timidness*, as mentioned in Section 6.1. Despite successfully deducing the correct files, grasping the required logic, and understanding the necessary modifications, overly aligned models often halt their autonomous execution. Instead of acting decisively to patch the repository, they generate verbose outputs and wait for explicit user permission or manual file uploads despite running in a headless mode. The following log from the `Kimi` model illustrates this phenomenon perfectly. The model accurately analyzes the commit description and identifies all five required files, yet concludes its reasoning with a complete refusal to proceed autonomously.

```
The user wants me to identify which files need to be changed based on the commit description they provided. They want me to stop and wait for them to add the files to the chat before editing.
Let me analyze the commit description:
1. lib/asn1/asn1_utils.cpp ... mentioned as touched
2. include/srsgnb/asn1/asn1_utils.h ... mentioned as touched (contains unbounded_octstring definition)
3. unittests/asn1/asn1_rrc_nr_test.cpp ... mentioned as touched
4. unittests/asn1/asn1_utils_test.cpp ... mentioned as touched
5. apps/examples/du/du_example.cpp ... mentioned as touched
The commit describes:
... Redefining unbounded_octstring as an extension of byte_buffer
... Introducing octet_string_helper namespace with functions (to_uint, to_octet_string, to_hex_string, hex_string_to_octets, append_hex_string)
... Migrating fixed/bounded/unbounded octet string code to use helpers
... unbounded_octstring new APIs: to_string, from_string, to_number, from_number
... pack/unpack logic changes
... Updates to tests and usage (copy() calls, bit references)
So the files most likely to need changes are:
1. include/srsgnb/asn1/asn1_utils.h (contains unbounded_octstring class definition)
2. lib/asn1/asn1_utils.cpp (implementation of helpers and methods)
3. unittests/asn1/asn1_utils_test.cpp (tests for the changes)
4. unittests/asn1/asn1_rrc_nr_test.cpp (tests using octet strings)
5. apps/examples/du/du_example.cpp (usage example)
I should list these and stop....
```

B Standard LLM-as-a-Judge

We initially experimented with a single-shot setting, which is the standard way to implement LLM judges in the literature (Gu et al., 2024). Here, the judge received the diffs and the questions simultaneously to produce an output. However, this approach failed because the models either experienced context overload and generated unparsable outputs, or the total input size was so large that it exhausted the manageable context length of our compute setup. After testing several models that we could run locally, the Gemma4 31B model was ultimately selected. We then transitioned to a two-tier approach where the TeleJudge framework could maintain context across different models and deliver accurate verdicts 64.05% of the time, achieving up to 96.9% for specific models. To further enhance this framework, we explored a third tier aimed at evaluating semantics across files to reduce the overall context length. This adjustment, unfortunately, led to inaccurate results and a loss of context because the native code was abstracted away, prompting us to exclusively adopt the two-tier system for TeleJudge.

C Prompts

The four main prompt templates utilized for TeleSWEBench are provided in this section. Specifically, the *Generator* and *Validator* prompt templates pertaining to benchmark generation are available at P1 and P2, respectively. Furthermore, the TeleJudge prompt templates, as explained in Section 3.3 that are used to evaluate the *File-Level* verdicts and the *Holistic* verdicts are provided in P3 and P4. Please note that the specific `difficulty_requirements` referenced in the Generator template are detailed in Section 3.1.

P1: Generator

You are an expert software-engineer assistant. Your task is to write a {difficulty_description} instruction for another AI assistant to reproduce a code change.

Analyze the provided commit information and generate the output according to the selected difficulty.

Commit Information:
{commit_json}

Requirements ({difficulty}):
{difficulty_requirements}

Output format:
Generate {output_length}.
Use {output_style}.
Do not output JSON or code blocks unless explicitly requested.

P2: Validator

You are validating a question for GitHub Copilot evaluation. **Commit:** {commit_info}
Question: {generated_question}
Difficulty: {difficulty}

DIFFICULTY LEVEL DEFINITIONS:

EASY Difficulty:
Must provide EXACT file paths and line numbers
Must specify EXACT code to remove and EXACT code to add
Should be self-contained with ALL necessary information
MEDIUM Difficulty:
Should explain the "what" and "why" of the change
Should mention affected files/areas but NOT exact line numbers
Should provide key information (values, function names, etc.)
HARD Difficulty:
Should describe high-level objectives and goals
Should NOT mention specific file paths or function names
Should provide minimal hints (version numbers, API endpoints, etc.)

VALIDATION CRITERIA:
Is the question clear and unambiguous?
Does the question contain appropriate information level for the difficulty?
Is the question realistic for GitHub Copilot to answer?
Does the question match the actual commit changes?
Is the question free from errors or inconsistencies?

Respond with JSON only in this exact format:

```
{
    "is_valid": true/false,
    "confidence": 0.0-1.0,
    "reasoning": "Brief explanation"
}
```

Where:
`is_valid`: true if the question is valid, false otherwise
`confidence`: Your confidence level (0.0 to 1.0) that this question is valid
`reasoning`: Brief explanation of your decision

P3: File-Level Judge

You are an expert judge evaluating whether an AI coding assistant correctly addressed a software engineering task on the **srsRAN Project** 5G codebase (C++, telecom / 3GPP protocols).
You will receive:

1. The original task description (question).
2. The ground truth fix (the accepted human commit diff) for ONE file.
3. The copilot's proposed changes for the SAME file.

Accept if the copilot's changes are functionally equivalent to the ground truth for this file — they address the same root cause or implement the same feature, even if variable names, formatting, ordering, or minor stylistic details differ.
Reject if the copilot's changes miss the core issue, modify the wrong logic, introduce regressions, or are substantively incomplete.
Return ONLY a JSON object with exactly these fields (no other text):

```
{
    "verdict": "accept" or "reject",
    "confidence": 0-100,
    "reasons": ["...", "..."]
}
```

confidence: 0 = pure guess, 100 = absolutely certain.
Keep each reason to at most 25 words.

Task description:
{question}
Ground truth change for this file:
{gt_diff}
Copilot change for this file:
{copilot_diff} **Return ONLY the JSON verdict.**

P4: Holistic Meta-Judge Prompt

You are a senior reviewer making a final decision on whether an AI copilot's commit correctly addresses a task on the srsRAN 5G codebase. **Below you will receive:**

1. The original task description.
2. Per-file verdicts from a first-pass review, each with reasons.

Read the reasons carefully, think about the commit holistically, and make your own judgment. A single rejected file does NOT automatically mean the whole commit fails — consider whether the rejected file is critical to the task or a minor ancillary change. **Return ONLY a JSON object with exactly these fields (no other text):**

```
{
    "verdict": "accept" or "reject",
    "confidence": 0-100,
    "reasons": ["...", "..."]
}
```

confidence: 0 = pure guess, 100 = absolutely certain.
Keep each reason to at most 25 words.

Task description:
{question}
Per-file verdicts:
{formatted_file_verdicts}
Think holistically and return ONLY the JSON verdict.

D Sample Hierarchical Judge Outputs

Commit: `7fabbe7` Verdict: Accept

Assessment Metadata:

- **Commit SHA:** `7fabbe7b87a800932a0c74f539cb6d888fed56f1`
- **Question ID:** `7fabbe7_medium`
- **Difficulty:** Medium
- **Copilot Model:** `kimi`
- **Judge Model:** `gemma4:31b`
- **Confidence:** 100%
- **Timestamp:** 2026-04-16 03:33:48

Overall Reasons:
Implementation of configurable search iterations is correct.
Unit tests were expanded as requested with appropriate ranges and assertions.
Both files were reviewed and accepted as functionally correct.

File Verdicts:

`lib/scheduler/support/...`

Verdict: Accept (100% confidence)

Reasons:
The copilot correctly implemented the configurable `max_prb_inc_iterations` parameter with the correct default value.
The copilot added the requested clarifying comment regarding the TBS derivation approximation, although in a slightly different location than the ground truth.

`unittests/scheduler/...`

Verdict: Accept (100% confidence)

Reasons:
The copilot's changes are functionally identical to the ground truth, including the new TBS ranges, steps, and assertion logic.
The only difference is the renaming of 'long' to 'large' for the TBS variable, which is a stylistic choice.

E Sample Questions across Difficulty Tiers

A sample from TeleSWEBench with associated questions and ground truth is shown in Figure 6.

F Extracting Test Cases

To evaluate the generated code, we leverage the framework's native unit-testing hierarchy, which follows a locality principle: test files are typically placed under dedicated test directories whose paths closely mirror the locations of their corresponding implementation files. In addition, test files often share the same base name as the implementation file, commonly with a _test suffix appended. This structural symmetry between implementation and testing components enables the efficient extraction and isolated evaluation of targeted network modules within our benchmarking framework. To identify implementation–test pairs, we employ a name-matching procedure that compares implementation filenames with candidate test files while accounting for common naming patterns such as suffix additions (e.g., _test) and path similarities. We further apply a threshold-based matching criterion to retain only sufficiently similar pairs, thereby reducing spurious associations. This process allows us to systematically construct implementation–test mappings across the repository while preserving high confidence in the extracted relationships.

However, test coverage is not available across all repository snapshots. A key challenge is that the repository tracks its test vectors through an internal Git LFS store, so an arbitrary commit does not necessarily carry the files required for successful compilation, and its test suite cannot be executed standalone. Consequently, our benchmark construction primarily relies on commits where extracted test suites remain compilable and executable in isolation. After establishing implementation–test mappings, the next step is to associate benchmark questions with their corresponding tests. To achieve this, we utilize the ground-truth file modifications available in the benchmark dataset. Specifically, if at least one file modified in the ground-truth patch matches the implementation file associated with a test case, we assign that benchmark question to the corresponding test suite. This strategy ensures that the functionality affected by the target implementation file is validated by its native testing infrastructure.

Commit *70c8b9d*: Ground Truth

Commit intent: remove DCI dependency on `vrb_to_prb::mapping_type` and use a boolean `interleaved_vrb_prb_mapping` across packing/building/tests.

Files changed (6):
`include/srsran/ran/pdcch/dci_packing.h`,
`lib/ran/pdcch/dci_packing.cpp`,
`lib/scheduler/support/dci_builder.cpp`,
`lib/scheduler/support/sch_pdu_builder.cpp`,
`tests/unittests/fapi_adaptor/mac/messages/helpers.cpp`,
`tests/unittests/ran/pdcch/dci_packing_test.cpp`.

---

include/srsran/ran/pdcch/dci_packing.h

```
- #include "srsran/ran/resource_allocation/vrb_to_prb.h"
- vrb_to_prb::mapping_type vrb_to_prb_mapping;
+ bool interleaved_vrb_prb_mapping;
```

. . . (same replacement across c_rnti/p_rnti/si_rnti/ra_rnti/tc_rnti structs)

lib/ran/pdcch/dci_packing.cpp

```
- payload.push_back(config.vrb_to_prb_mapping != vrb_to_prb::mapping_type::non_interleaved, 1);
+ payload.push_back(config.interleaved_vrb_prb_mapping, 1);
```

. . . (5 packing paths updated)

lib/scheduler/support/dci_builder.cpp

```
- si_dci.vrb_to_prb_mapping = vrb_to_prb::mapping_type::non_interleaved;
+ si_dci.interleaved_vrb_prb_mapping = false;
```

. . . (p_rnti, ra_rnti, tc_rnti, c_rnti builders aligned)

lib/scheduler/support/sch_pdu_builder.cpp

```
- pdsch.vrb_prb_mapping = dci_cfg.vrb_to_prb_mapping;
+ pdsch.vrb_prb_mapping = vrb_to_prb::mapping_type::non_interleaved;
```

. . . (5 call sites)

tests/.../helpers.cpp

```
- return {1,2,3, vrb_to_prb::mapping_type::non_interleaved, 0,2,1};
+ return {1,2,3, false, 0,2,1};
```

tests/.../dci_packing_test.cpp

```
- config.vrb_to_prb_mapping = ... ? interleaved_n2 : non_interleaved;
+ config.interleaved_vrb_prb_mapping = vrb_to_prb_mapping_dist(rgen) > 0;
- expected.push_back(config.vrb_to_prb_mapping != ...non_interleaved);
+ expected.push_back(config.interleaved_vrb_prb_mapping);
```

Easy

Question: Modify the DCI path to remove enum-based VRB-to-PRB mapping usage and replace it with a boolean flag. In `include/srsran/ran/pdcch/dci_packing.h`, replace each `vrb_to_prb::mapping_type vrb_to_prb_mapping;` with `bool interleaved_vrb_prb_mapping;` (across all relevant DCI config structs), and remove the obsolete include of `vrb_to_prb.h`. In `lib/ran/pdcch/dci_packing.cpp`, replace each mapping-bit emission that . . . Then update scheduler builders and unit tests/helpers so values are assigned and validated as booleans (typically `false` for default non-interleaved) . . .

Medium

Question: Refactor the DCI fallback/control flow so VRB-to-PRB interleaving is represented as a lightweight boolean instead of the previous enum dependency. Apply the migration consistently across (1) DCI configuration structs, . . . (3) scheduler-side DCI construction, and (4) tests/helpers that synthesize and check payload bits. Preserve functional semantics: previous "non-interleaved" behavior should remain . . . The expected end state is that enum-specific plumbing disappears from this path and all affected tests validate the boolean representation end-to-end . . .

Hard

Question: Consolidate and simplify the DCI packing surface by removing the direct dependency on `vrb_to_prb::mapping_type` while keeping externally observable behavior unchanged. The migration must be coherent across headers, packers, scheduler builders, and verification utilities, and should avoid partial conversions that leave mixed representations . . . When done, the mapping indicator should be encoded from a single boolean field and construction paths should keep a clear non-interleaved default unless explicitly overridden. **Hint:** propagate a boolean (e.g., `interleaved_vrb_prb_mapping`) through all encode/validate paths, including helper-generated fixtures . . .

Figure 6: One `srsRAN_Project` ground-truth diff (commit `70c8b9d`) shown with representative per-file chunks and corresponding questions across easy, medium and hard.