P4Bench: A Scalable and Adaptive Oracle-Backed Benchmark for P4 Programming with Large Language Models

ZIRONG GAN, University of Connecticut, USA
SHUYI LIAO, University of Connecticut, USA
YANNIAN NIU, University of Connecticut, USA
XINYUE ZHENG, University of Connecticut, USA
TINGTING YU, University of Connecticut, USA
MINMEI WANG*, University of Connecticut, USA

Large language models (LLMs) are increasingly evaluated on code generation across popular programming languages, but existing benchmarks provide little coverage for P4, a domain-specific and low-resource language for programmable data planes. As LLM-assisted programming becomes increasingly relevant to programmable data planes, it is essential to provide a P4 benchmark that covers diverse network-function tasks, validates packet-level behavior, and remains useful as model capabilities improve, without requiring extensive manual task authoring.

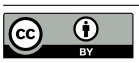
We present P4Bench, an oracle-backed benchmark for LLM-based P4 programming. P4Bench introduces a Task Pattern Language (TPL) for specifying source-grounded network-function patterns and constructing diverse network-function tasks. For each task, we introduce Python oracles that derive packet-level ground truth independently of any P4 reference implementation. Each benchmark task is then evaluated end to end: the LLM must generate both a P4 data-plane program and the corresponding control-plane entries, and correctness is automatically measured by deployed packet behavior. TPL also enables mutation-based task variants, allowing P4Bench to recalibrate task difficulty as LLMs improve, while the oracle-derived packet expectations decouple task behavior from any single P4 implementation, providing a basis for future target-specific evaluators. To control artifact quality, P4Bench introduces an admission-control process that incorporates LLM-as-a-judge review and human validation.

We build the current P4Bench corpus with 73 TPL patterns, 130 full-program tasks, and 1,914 packet tests. Using this corpus, we evaluate six state-of-the-art LLMs and further analyze model failures on both BMv2 and Tofino targets. Even the strongest model, Opus-4.8, compiles only 82.6% of the generated programs and achieves a functional score of only 0.733 out of 1.0. These results reveal substantial gaps in P4 language validity, target-specific constraint handling, packet-level semantics, and consistency between generated data-plane programs and control-plane state. Such failures arise from the deployed nature of programmable-data-plane artifacts and are largely invisible to general-purpose code-generation benchmarks, underscoring the need for domain-specific LLM-based agents and systems for network programming.

CCS Concepts: • **Networks** → **Programmable networks**; • **Software and its engineering** → *Domain specific languages*; • **Computing methodologies** → *Artificial intelligence*.

*Corresponding author.

Authors' Contact Information: Zirong Gan, zirong.gan@uconn.edu, University of Connecticut, Storrs, Connecticut, USA; Shuyi Liao, shuyi.liao@uconn.edu, University of Connecticut, Storrs, Connecticut, USA; Yannian Niu, yannian.niu@uconn.edu, University of Connecticut, Storrs, Connecticut, USA; Xinyue Zheng, xinyue.zheng@uconn.edu, University of Connecticut, Storrs, Connecticut, USA; Tingting Yu, tingting.yu@uconn.edu, University of Connecticut, Storrs, Connecticut, USA; Minmei Wang, minmei.wang@uconn.edu, University of Connecticut, Storrs, Connecticut, USA.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2476-1249/2026/12-ART75
https://doi.org/10.1145/3857378

Additional Key Words and Phrases: P4, programmable data planes, large language models, code generation, benchmark, packet testing

ACM Reference Format:
Zirong Gan, Shuyi Liao, Yannian Niu, Xinyue Zheng, Tingting Yu, and Minmei Wang. 2026. P4Bench: A Scalable and Adaptive Oracle-Backed Benchmark for P4 Programming with Large Language Models. *Proc. ACM Meas. Anal. Comput. Syst.* 10, 3, Article 75 (December 2026), 35 pages. https://doi.org/10.1145/3857378

1 Introduction

Network functions are essential building blocks of modern networked systems, providing the performance, security, reliability, and visibility required by today's infrastructure and applications. Programmable network devices enable these functions to evolve beyond fixed-function packet processing, allowing operators to customize and deploy diverse network behaviors directly in the data plane [1]. Among programmable network devices, programmable switches provide a practical platform for line-rate packet processing, with P4 serving as a widely adopted domain-specific language for specifying data-plane behavior [2]. However, as networks grow in scale, heterogeneity, programmability, and workload dynamism, designing and maintaining correct network functions becomes increasingly difficult.

Recent advances in Large Language Models have demonstrated strong capabilities in automated code generation across widely used programming languages, such as Python, substantially improving programming efficiency and helping automatically solve challenging coding tasks [3–6]. These advances make LLM-assisted programming a promising direction for automated network-function design and implementation. By translating operator intent, protocol specifications, and technical documentation into executable programs, LLMs could reduce development effort and accelerate the deployment of programmable data-plane functions. However, the ability of LLMs to generate correct and deployable network functions using P4 remains insufficiently evaluated.

Existing benchmarks, including HumanEval [7], MBPP [8], SWE-bench [9], and LiveCodeBench [10], provide standardized tasks, executable tests, and evaluation metrics for widely used programming languages, but none targets P4 programming. The closest prior work [11] explores LLM-based P4 generation through natural-language translation, fine-tuning, and a 50-task code-completion benchmark. Although its training corpus is curated from public P4 repositories, the benchmark itself consists primarily of manually constructed tasks derived from P4 tutorials, provides packet-level unit tests for only 11 tasks, and targets only the BMv2/v1model architecture. To systematically assess the P4 programming capabilities of current LLMs and LLM-based agentic systems on network-function tasks, it is therefore essential to develop a comprehensive benchmark framework that spans diverse network functions, evaluates end-to-end functional correctness, and captures the architectural and resource constraints of heterogeneous target platforms.

Developing such a benchmark involves several key design requirements. First, the task suite should cover a broad range of network-function behaviors and span multiple difficulty levels, enabling meaningful differentiation among models and agentic systems with varying capabilities. Second, because a deployable P4 implementation consists of both a data-plane program and the corresponding control-plane configuration, the benchmark must evaluate their joint correctness rather than assessing either artifact in isolation. Third, as LLMs and LLM-based systems evolve rapidly, the benchmark should support efficient task expansion and difficulty recalibration with limited additional expert effort. Fourth, P4 programs are inherently target-dependent, the benchmark should support heterogeneous execution platforms. BMv2/v1model and Intel Tofino/TNA, for example, differ in architecture, available externs, compiler constraints, and resource models. Restricting evaluation to a single target may therefore fail to expose important target-specific correctness and deployability issues. Satisfying these requirements simultaneously is challenging.

We first examine existing benchmark-construction methodologies to understand their strengths and tradeoffs, thereby informing the design of a benchmark for LLM-based P4 programming. Existing code benchmarks are typically constructed using one of three approaches. Expert-authored benchmarks, such as HumanEval and MBPP [7, 8], provide carefully designed, high-quality tasks with reliable evaluation, but require substantial domain expertise and manual effort to scale. Corpus-derived benchmarks, such as SWE-bench and LiveCodeBench [9, 10], leverage real-world programming artifacts to improve realism and coverage, but are fundamentally constrained by the availability, diversity, and quality of existing corpora. Synthetic benchmarks, such as AutoCodeBench [12], use LLMs to generate tasks at scale, but unconstrained synthesis may introduce unreliable, redundant, insufficiently grounded, or semantically inconsistent benchmark instances.

Building on these design considerations, we propose P4Bench, a scalable and adaptive benchmark for evaluating automated P4-based network-function design and implementation. P4Bench introduces an LLM-assisted construction pipeline that transforms selected source materials, including research papers, RFCs, tutorials, and technical specifications, into validated network-function benchmark tasks. Specifically, an LLM-based construction agent extracts source-grounded network-function behavior and represents it as a task pattern using our Task Pattern Language (TPL), an intermediate representation that specifies each task pattern in a structured yet target-independent form. The pipeline then instantiates each pattern into a complete task bundle containing a model-facing specification, public and hidden packet tests, and a Python oracle that independently derives packet-level ground truth. This design decouples task semantics from any target-specific P4 implementation and supports evaluation across different targets. To ensure artifact quality, each task undergoes an admission-control process that combines agent self-check, independent LLM review, and human validation. P4Bench further supports mutation-based difficulty calibration, enabling the task corpus to be expanded and recalibrated as model capabilities evolve.

This paper makes the following contributions:

(1) **Task Pattern Language (TPL).** We introduce TPL, a structured and target-independent intermediate representation for formally specifying network function tasks.
(2) **A comprehensive P4 benchmark.** We develop P4Bench, a scalable and adaptive benchmark that covers diverse network-function tasks and evaluates complete artifacts generated by the model under test, including both P4 data-plane programs and their corresponding control-plane configurations. The current corpus comprises 130 tasks spanning multiple difficulty levels and 1,914 packet-level tests. **P4Bench is publicly available at https://github.com/ANSS-Lab/P4Bench.**
(3) **Oracle-backed ground truth.** We use Python oracles that derive packet-level expected outputs independently of any reference P4 implementation, decoupling task semantics from a specific backend and supporting evaluation across heterogeneous P4 targets.
(4) **Empirical study.** We evaluate frontier LLMs on P4Bench and conduct a detailed failure analysis. Even the strongest model, Opus-4.8, compiles only 82.6% of generated programs and achieve 0.733 functional score, whereas DeepSeek-V4-Pro compiles only 1.5% of programs. Our analysis further reveals persistent failures in P4 syntax, target-specific constraint handling, packet-level semantics, and consistency between generated data-plane programs and control-plane configurations, providing insights for the design of more reliable P4 code-generation models and agentic systems.

The remainder of the paper is organized as follows. §2 reviews P4 and related benchmarks, §3 presents the design challenges, §4 introduces TPL, and §5 describes P4Bench. §6 and §7 present the experimental methodology, results, and failure analysis, while §8 discusses the limitations and future work, and §9 concludes the paper.

2 Background and Related Work

2.1 Programmable Data Plane and P4

Programmable data plane allows network operators and researchers to design and implement customized network functions and algorithms without redesigning the hardware [13, 14]. This capability accelerates network innovation, extends hardware lifetime, and enables advanced applications in cloud computing, network security, traffic engineering, and in-network computing.

To make this flexibility usable in practice, developers need a programming abstraction that can describe packet-processing behavior while remaining portable across different switch targets. Programming Protocol-Independent Packet Processors (P4) [2] has become the dominant domain-specific language (DSL) for this role. P4 enables developers to specify, implement, and deploy custom packet-forwarding logic across heterogeneous network targets. P4 abstracts target-specific constraints by formalizing data-plane behavior around the target-independent Protocol-Independent Switch Architecture (PISA). The logical workflow of a PISA pipeline is structurally organized into five sequential stages: (1) a parser, (2) an ingress match-action pipeline, (3) a traffic manager, (4) an egress match-action pipeline, and (5) a deparser. P4 targets include software switches such as BMv2, programmable ASICs such as Intel Tofino, and FPGA-based platforms.

2.2 Benchmarks for LLM-based Code Generation

Code generation has become a major application of Large Language Models (LLMs), and benchmarks are the primary mechanism for quantifying model capabilities and identifying weakness and limitations [15, 16]. HumanEval [7] and MBPP [8] established a functional-correctness paradigm for general-purpose programming: a model produces a function-level solution, the solution is executed against unit tests, and performance is measured with metrics such as pass@k. As frontier-model performance has improved, saturation on established code benchmarks has made it increasingly difficult to distinguish among leading models. [17].

Benchmarking LLMs on code generation has therefore evolved in several directions. SWE-bench moves from isolated functions to repository-level software repair and targeting real-world software engineering capability, where models must navigate real codebase and generate integration patches for GitHub issues [9]. LiveCodeBench refreshes evaluation tasks over time to reduce data contamination and measure zero-shot generalization on newly released programming problems [10]. To reduce the manual effort required for benchmark creation, AutoCodeBench uses LLMs and sandboxed execution to synthesize benchmark problems and testing inputs [12], while DyCodeEval generates logically-equivalent variations with initial seed programming problems from existing code benchmarks [18]. For domain-specific languages (DSLs), specialized benchmarks are typically constructed either manually by domain experts or by collecting design problems from publicly available datasets [19–22]. In contrast, P4 programming remains comparatively under-benchmarked: the most closely recent study on LLM-based P4 code generation [11] constructs only a small tutorial-derived dataset, covering mainly entry-level network functions.

2.3 Benchmark Creation with LLMs

Benchmark tasks are traditionally created by human experts, leveraging their domain knowledge and execution experience or transforming existing artifacts from other sources, e.g. github repositories, competitive programming platform, mathematical competitions [23]. This process is labor-intensive, time-consuming and expensive, and it is often difficult to scale to new domains or to refresh existing benchmarks. In response, researchers have explored the use of LLMs to assist in benchmark creation [12], under the condition that LLMs are pretrained on large corpora that

include code, documentation, and technical text and have shown emerging capabilities such as comprehension, reasoning, and in-context learning [24].

The cononical procedure for LLM-assisted benchmark creation is to instruct the model to extract domain knowledge from existing materials, and to generate the benchmark tasks [12]. Prior work has demonstrated the feasibility and effectiveness of extraction by LLMs from human-facing artifacts. PROSPER [25] develops Artifact Miner to extract protocol FSM state and transition information from RFC with LLMs, obtaining lower false positives and better accuracy compared to SOTA. In terms of generation, Intermediate Representations (IRs) play an important role when LLMs are applied to domain-specific generation tasks, where the representation can encode the domain concepts, constraints, and relationships that the model must preserve. E.g., in the hardware description language domain, VerilogCoder [5] will first generate Task and Circuit Relation Graph as IR to guide the generation of Verilog code and construct an Abstract Syntax Tree (AST) to assist the model in bug fixing, achieving higher correctness in Verilog code generation.

3 Motivations and Challenges

With the growing demand of LLM-assisted network-function design and implementation on programmable data planes, it becomes increasingly important to evaluate whether LLMs can produce correct and deployable artifacts across diverse network function tasks. Prior work shows that LLMs can make substantial progress in generating high-quality code when guided by advanced prompting strategies [26–28], enhanced by Retrieval-augmented generation [29–31] or equipped with tools for interacting with their environment. [32–34]. In some settings, LLM-generated code can even outperform human-written code in readability and maintainability [35]. However, the ability of LLMs to automate network-function design and implementation remains underexplored, where correctness depends on packet-level behavior, control-plane/data-plane consistency, and target-specific constraints. Inspired by recent work on LLM-assisted benchmark creation [12, 36] and evidence that LLMs can understand network protocols [25], we propose to leverage an LLM-based agent to construct diverse network-function tasks and build a benchmark framework for systematically evaluating the code-generation capabilities of LLM-based code generators.

However, developing such a benchmark framework raises three key challenges.

C1: Constructing various fresh and realistic tasks is difficult. Public codebases and repositories for network functions are much scarcer than those for general-purpose programming languages. Existing examples are often tutorials, course assignments, or long-lived reference programs, which may be limited in diversity and may have been exposed during model pretraining. As a result, directly reusing these artifacts risks evaluating memorization rather than a model's ability to synthesize behaviorally correct programs for new network-function tasks. A useful benchmark must therefore generate fresh task instances while preserving realism. This is challenging because each task must specify not only packet-processing logic, but also stateful behavior, target constraints, control-plane entries, topology assumptions, and packet-level tests. The benchmark must also span diverse network-function families so that performance reflects general P4 programming capability rather than success on a narrow set of hand-written examples.

C2: Keeping benchmarks adaptive as LLM-based coding models rapidly improve. Network functions can be realized through different designs, data structures and algorithms, each introducing distinct tradeoffs in memory usage, throughput, latency, and implementation complexity. Evaluating LLM-based generators in this space therefore requires a scalable and extensible benchmark-construction method that can produce valid tasks across varying difficulty levels and resource constraints. The benchmark must also evolve with model progress: task difficulty should be measured and adjusted systematically, rather than determined by author intuition, so that the benchmark continues to expose meaningful capability gaps over time.

C3: LLM-generated benchmark artifacts require validation. As we introduce LLM-based agents during benchmark construction, generated artifacts may hallucinate behavior, omit corner cases, or deviate from source specifications. A reliable benchmark therefore requires a controlled validation process for accepting generated artifacts, including task descriptions, test cases, and ground truth.

4 Task Pattern Language: A Structured Representation for P4 Network Tasks

P4Bench requires each benchmark task to support downstream artifacts generated by LLMs. Unstructured natural language task descriptions suffer from semantic ambiguity, which causes high variance in LLM-generated artifacts [37]. Also, directly prompting to generate raw P4 source code lacks intermediate modular control. We therefore introduce a Task Pattern Language (TPL), an intermediate representation for specifying network-function tasks in a structured but target independent form.

TPL functions as a domain-specific structured abstraction that bridges informal natural language intents and executable P4 programs. It decomposes networking tasks into explicit components, providing more explicit scaffolding for LLMs to generate complete task bundles (P4 code, control-plane rules, and test harnesses). Meanwhile, TPL explicitly exposes configurable complexity knobs (e.g., table sizes, header depth, stateful memory access), enabling calibrated benchmark generation that is not directly exposed by conventional P4 programming interfaces.

TPL captures the behavioral core of a task, including the relevant entities, packet processing rules, trace level invariants, parameters, test scaffolding, mutation operators, and grounding sources. This representation serves as the bridge between the construction phase and the evaluation phase described in Section 5. During construction, P4Bench uses TPL patterns to instantiate concrete tasks, generate model-facing specifications, synthesize Python oracles, and produce public and hidden packet tests. During calibration, P4Bench uses TPL mutation operators to derive task variants with adjusted difficulty. During admission control, the explicit grounding and structured semantics make each task easier to inspect and validate. In this way, TPL provides the common task representation that makes P4Bench scalable, oracle-backed, adaptive, and auditable.

4.1 Design Principles Behind TPL

Research papers, RFCs, tutorials, and vendor specifications provide a principled basis for benchmark task design by documenting realistic network functions, deployment assumptions, packet formats, failure modes, and performance constraints. By systematically examining these sources, we identify recurring abstractions and requirements that make network tasks realistic, verifiable, and extensible. These source-derived observations inform the design of the Task Pattern Language (TPL), which formalizes auditable and reusable task patterns. A task pattern captures a modular network behavior that can stand alone or be composed with other patterns to represent diverse network tasks. We distill the design of TPL into five guiding principles.

- **Entity-scoped behavioral abstraction.** A network-function task defines behavior over entities, such as packets, flows, connections, sessions, or links. Making these entities explicit is important because different network functions operate over different semantic units. For example, a stateless forwarder makes decisions on individual packets, a load balancer maintains consistent decisions across packets in the same flow, and a firewall updates state across a connection. TPL therefore requires each task pattern to explicitly specify the entities over which the intended behavior is defined.
- **Rules and invariants as the semantic core.** Data-plane functions can be viewed as reactive transition systems that perform a sequence of operations upon packet arrival. TPL needs

to capture this behavior through rules that specify how each packet is processed, and use rules to model per-packet transitions. However, some correctness requirements cannot be checked with single packet. Thus, invariants should be introduced to express trace-level obligations across packets and over time, such as preserving flow affinity, keeping forwarding decision stable, preventing forwarding loops, and cleaning up state after a session end.

- **Parameterized and mutable patterns for scale.** A task pattern should be reusable rather than dedicated to a single task instance. TPL should support reuse by exposing pattern parameters and mutation operators, allowing one source-grounded network-function pattern to generate many task instances. Mutation operators adjust predefined dimensions such as table capacity, workload size, policy strictness, protocol corner cases to create variant as model capabilities evolve.
- **Ground-truth construction.** Test packets are a key part of the benchmark because they determine whether a generated program is functionally correct. Each pattern should therefore include test scaffolding that specifies how test packets are generated for evaluating the intended behavior. Importantly, this scaffolding is used only to construct the benchmark test suite and is not exposed to the evaluated system, such as an LLM-based code generator.
- **Source-grounded and auditable.** A task pattern should be grounded in explicit source evidence rather than unsupported fabrication. Each pattern therefore needs to record the references that justify its intended behavior, assumptions, and parameter ranges, including relevant papers, RFC sections, tutorials, or vendor specifications. This can make the resulting semantics auditable by human reviewers.

4.2 Task Pattern Language Design

Following these design principles, a TPL pattern consists of several components: entities define the semantic objects over which the task is specified; rules and invariants capture the intended behavior; parameters and seeds instantiate realistic operating points; grounding records the supporting sources; and mutation operators enable adaptive corpus construction.

We thus formalize this structure by defining a TPL pattern as a parameterized object $P = (\mathcal{E}, \Theta, R, \Phi, \mathcal{T}, \mathcal{M}, G)$, where $\mathcal{E}$ is a typed set of entities, Θ is a set of named parameters with types and constraints, $R = (r_1, \ldots, r_m)$ is an ordered rule sequence, Φ is a set of trace invariants, $\mathcal{T}$ is test scaffolding, $\mathcal{M}$ is a set of mutation operators, and G records the grounding sources used to justify the behavior.

Formally, each rule r_j is represented as: $r_j = (\mathsf{when}_j, \mathsf{given}_j, \mathsf{then}_j)$. The *when* clause specifies the rule **preconditions**, expressed as predicates over the current packet, abstract state, and seed. The *given* clause specifies the entities over which the rule operates, including those it reads or updates. And the *then* clause specifies the rule **postconditions**, defined as actions performed by the rule, including forwarding, dropping, header-field rewriting, state binding, eviction, and packet emission. Rules are interpreted by deterministic first-match evaluation. For packet x_t and state σ_t, the first enabled rule produces an output decision d_t and successor state σ_{t+1}:$(x_t, \sigma_t) \xrightarrow{R} (d_t, \sigma_{t+1})$. Applying R repeatedly over an input sequence yields a trace $\tau = ((x_1, \sigma_1, d_1), \ldots, (x_n, \sigma_n, d_n))$.

Invariants in Φ are predicates over τ and express properties not captured by a single transition, such as flow persistence, priority stability, bounded latency, loop freedom, and reclamation after a state lifetime expires.

A *seed* s is a concrete binding of all parameters in Θ. Applying s to the pattern P yields a task instance $I = P[s]$ with no free parameters. For each task instance, P4Bench generates three artifacts for benchmark construction and evaluation: $\mathsf{render}(I), \mathsf{oracle}(I), \mathsf{tests}(I, \mathcal{T})$. The $\mathsf{render}(I)$ is the natural-language specification shown to the model under test; $\mathsf{oracle}(I)$ is the executable

Table 1. TPL vocabulary. Entities define the objects named by rules, preconditions define what rules can observe, postconditions define what rules can change, and invariants define trace-level obligations.

Class	Vocabulary
Entities	`packet`, `flow`, `connection`, `session`, `bucket`, `window`, `time_tick`, `link`.
Preconditions	`match`, `exists`, `value`, `aggregate`, `count_exceeds`, `rate_exceeds`, `elapsed`, `protocol_state`, `ingress`, `header_present`.
Postconditions	`forward`, `drop`, `mirror`, `set`, `rewrite`, `inc`, `dec`, `bind`, `evict`, `assign_rank`, `emit`, `transition`.
Invariants	`persistence`, `stability`, `bounded_latency`, `fairness`, `no_loop`, `no_orphan`.

ground-truth model that materializes expected packet outputs; and tests$(I, \mathcal{T})$ instantiates the test scaffolding into public and hidden packet tests. Public tests expose representative instances of the required behavior, while hidden tests evaluate additional branches and edge cases that are not revealed to the model under test during program generation.

A mutation operator is a typed operation that P4Bench can apply to an instantiated task to create a related task variant. Such an operator may change parameter bindings, workloads, resource bounds, coverage requirements, or selected semantic clauses, depending on which difficulty axis needs adjustment. Formally, each $\mu \in \mathcal{M}$ maps a pattern instance to a sibling instance, $\mu(P, s) = (P', s')$, while recording the moved difficulty axis and preserving the intended network-function identity. Semantic mutations that change R or Φ trigger renewed oracle and specification audits.

TPL vocabulary. TPL uses a predefined vocabulary so that patterns are reviewable and downstream artifact generation remains constrained. Table 1 summarizes the vocabulary used by the current corpus. The vocabulary captures the recurring semantic atoms of benchmark-relevant network functions while leaving realization choices to the model under test. For example, `bind` states that flow or connection state must be created, but it does not say whether a P4 solution should use registers, exact-match entries, hashed indices, or a different target-specific realization.

Intended User & Usability Scope. TPL is designed for benchmark creators, network researchers, and automated LLM workflows rather than general network operators. It is written in familiar networking terms (match–action rules, per-flow state, header fields), so users can read and verify a pattern without learning a new formalism. Hence, TPL serves as a practical, heuristic-driven task representation tailored for guided LLM generation and calibrated benchmarking, rather than a formally verified language with mathematical soundness guarantees.

4.3 Case Study: From Pattern to Task

Figure 1 illustrates how to generate an access control list (ACL) task from a structured TPL pattern. The ACL task evaluates each IPv4 packet against a priority-ordered set of policy entries: the highest-priority matching entry determines whether the packet is forwarded or dropped, while unmatched packets follow the default policy. During the task construction, the TPL pattern records the information needed to instantiate the task, including entities, parameters, rules, invariants, test scaffolding, mutation axes, and grounding. A seed then binds the pattern to concrete ports and ACL entries, from which P4Bench derives both the task specification provided to the LLM code generator and the packet tests used to evaluate the generated program. Appendix A presents the complete case study.

5 P4Bench Design

This section presents the design of P4Bench, which uses TPL as the intermediate task representation for scalable benchmark construction, oracle-backed validation, difficulty calibration, and end-to-end model evaluation. Guided by the challenges in Section 3, P4Bench is organized around

1. Network-function. Priority ACL over IPv4 packets. The highest-priority matching entry determines the action; denied and unmatched packets are dropped.

2. Structured TPL pattern.

$\mathcal{E}$	Packet p and ACL entry e; each entry has a match predicate, priority, and action.
Θ	Example parameters include the default action d_0 and the output port p_{out}; a seed assigns concrete values.
R	**Rule: when** $M = \{e \in A \mid e.\text{match}(p)\}$ is nonempty, $e^\star = \arg\max_{e \in M} e.\text{priority}$, and $e^\star.\text{action} = \text{permit}$. **given** p, A, p_{out}. **then** forward p unchanged to p_{out}.
Φ	If multiple ACL entries match the same packet, the packet outcome must be determined only by the highest-priority matching entry. A permitted packet must leave with its Ethernet and IPv4 headers unchanged.
$\mathcal{T}$	Generate an overlapping-entry packet that matches both a high-priority deny rule and a lower-priority permit rule; the expected result is drop.
$\mathcal{M}$	Increase overlap depth or change the default action to adjust difficulty while preserving ACL semantics.
G	RFC3198.

3. Seed and derived tests.

Seed $d_0 = \text{deny}, p_{\text{in}} = 1, p_{\text{out}} = 2, A = [e_1, e_2]$, where $e_1 = (20, \text{srcIP} = 10.0.1.10, \text{deny})$ and $e_2 = (10, \text{srcIP} \in 10.0.1.0/24, \text{permit})$.

Testing case	Packet source IP	Expected result
explicit deny / overlap priority	`10.0.1.10`	select e_1 over e_2; drop

4. Model-facing specification.

Implement an IPv4 ACL in P4-16 for v1model. Packets arrive on port 1. Apply ACL entries by priority. Forward permitted packets to port 2 without header changes; drop denied or unmatched packets.

Fig. 1. Compact ACL case-study instantiation. Each TPL component is shown with one representative example; a seed binds the pattern to a concrete packet test and the model-facing specification.

two phases, as shown in Figure 2: an offline construction phase that creates, validates, and calibrates benchmark tasks, and an evaluation runtime phase that uses the validated tasks to measure model performance.

In the offline phase, P4Bench starts from source materials such as papers, RFCs, P4 tutorials, and vendor documents. A construction agent extracts network-function semantics from these sources and encodes them as TPL patterns. P4Bench then instantiates each pattern with concrete seeds to produce task bundles, including a model-facing task specification, control-plane entries, packet tests, and a Python oracle that derives packet-level ground truth independently of any target-specific P4 reference implementation. To ensure corpus reliability, each candidate task bundle passes through an admission-control protocol before release. To keep the benchmark informative as models improve, P4Bench further applies a difficulty-calibration loop that probes tasks with baseline models and uses TPL mutation operators to generate calibrated variants when needed.

In the runtime evaluation phase, a model under test receives the task specification, target information, required output format, and public tests. The model must generate both a P4 data-plane program and the corresponding control-plane entries. P4Bench compiles and deploys the generated program, installs the entries, replays packet tests, and compares observed packet behavior against the oracle-derived expected outputs. The remaining sections describe the three main components of this design: task construction in Section 5.1, task difficulty calibration in Section 5.2, and model evaluation in Section 5.3.

5.1 Task Construction

This section presents the design of P4Bench task construction. We first introduce the task-space map that guides source selection, then describe the construction pipeline that converts selected sources into candidate task bundles. We then explain how oracle-backed ground-truth generation and admission control ensure artifact reliability.

5.1.1 Task Space and Source Selection. Constructing diverse network-function tasks from validated sources is essential for building a reliable and representative benchmark. Source selection requires explicit coverage criteria; otherwise, the corpus may overrepresent entry-level examples

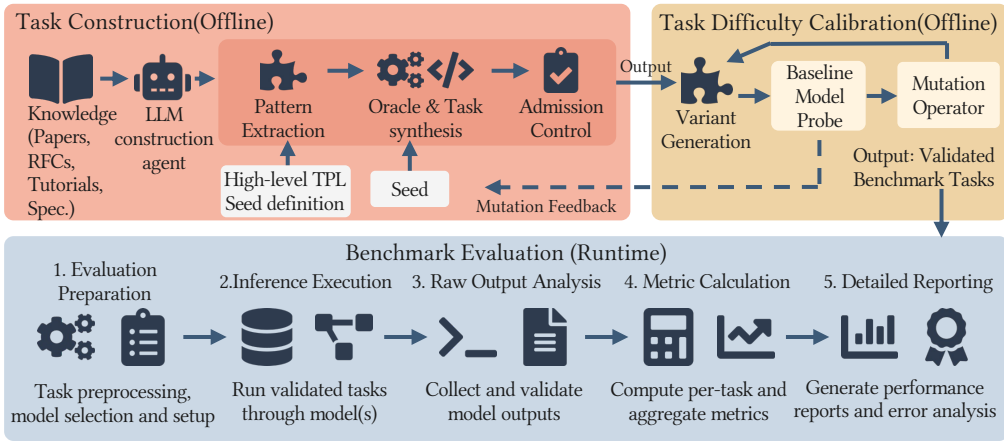


Fig. 2. Overview of the P4Bench.

or a small set of canonical network functions. P4Bench therefore uses a task-space map to guide source selection across network-function families and P4 programming capabilities.

After conducting literature review about P4 and it's applications in network function design [2, 13, 14, 38–40], we define the task-design map by classifying how P4 programmability is used to realize a network function. Specifically, candidate tasks fall into four types:

- **Redesign**: translating an existing packet-processing design into a P4-based implementation;
- **Scale-up**: realizing a high-throughput network function directly on a programmable switch under data-plane resource constraints;
- **Relocation**: offloading selected host-side or coordinator-side functionality to the programmable switch;
- **Composition**: integrating two or more of the above design roles in a single task.

Meanwhile, P4Bench focuses on five categories of network functions: (1) forwarding, routing, and tunneling; (2) address and flow management; (3) traffic management and telemetry; (4) security and access control; and (5) in-network computing and machine learning. Based on our literature review, these categories capture the most common classes of network functions implementable on programmable switches. They are intended as a practical guide for source selection, rather than an exhaustive taxonomy of all possible network functions.

5.1.2 Construction Pipeline. As Figure 3 shows, given selected source material, P4Bench uses an **LLM-based construction agent** to transform network-function semantics into executable benchmark artifacts. This construction pipeline is organized into five stages:

(1) **Pattern extraction.** The LLM-based construction agent reads the selected source material, extracts the relevant network-function behavior, and encodes it as a TPL pattern using the schema defined in §4. The agent must populate each field and link each pattern element to supporting source evidence, providing the provenance needed for later audit and validation.
(2) **Seed instantiation.** The extracted pattern is instantiated to produce a concrete benchmark task. Rather than manually assigning arbitrary parameter values, the agent searches the source material for canonical settings, such as workload scale, flow population, match-action table capacity, and resource budgets or limits. These settings are encoded as a seed, which is applied to the TPL pattern to generate a concrete task instance.

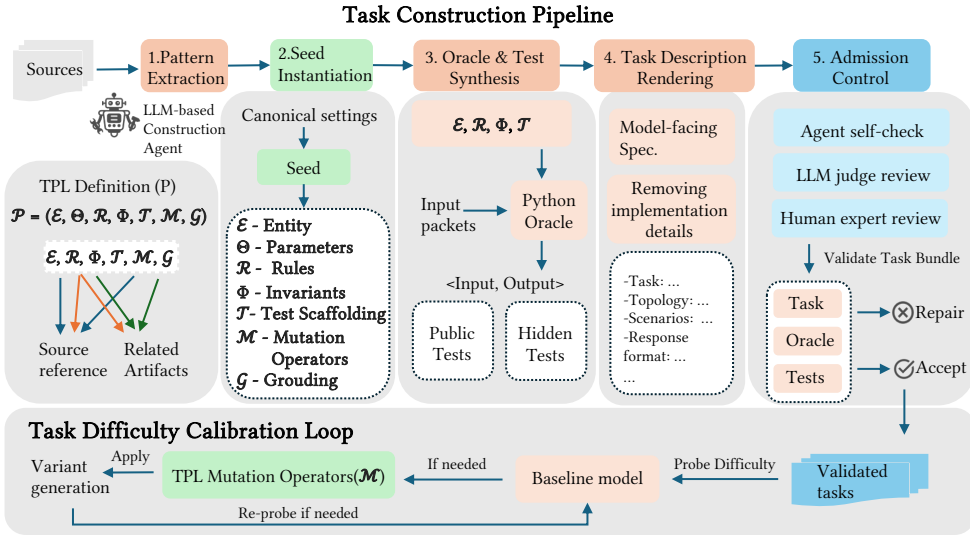


Fig. 3. Task Construction Pipeline&Task Calibration Loop.

(3) **Oracle and test synthesis.** The agent implements the behavior specified by the network-function task as a Python oracle. The entities, rules, and invariants in the TPL pattern define the intended behavior, while the test scaffolding specifies how to generate input packets. The agent then executes the oracle on these packets to derive the expected outputs. The resulting input-output pairs are added to the task bundle for evaluation. P4Bench further separates these packets into public tests, which are exposed to the model under test, and hidden tests, which are reserved for evaluation. We describe oracle-backed ground truth generation in detail in §5.1.3.

(4) **Task description rendering.** The agent converts the instantiated task into a natural-language description used in the prompt for the model under test. During this step, we instruct the agent to remove implementation details, such as specific data structure and algorithms, unless they are explicitly required by the task. This design evaluates whether the model can infer the task intent and produce a P4 implementation that is both syntactically valid for the target platform and semantically correct.

(5) **Admission control.** P4Bench finally introduces an admission-control mechanism to validate the reliability of each task bundle before it is added to the benchmark corpus. The validation process combines self-checking by the construction agent, cross-validation by an independent LLM-based agent, and human audit. Task bundles that pass this stage are admitted as validated benchmark tasks. We describe the admission-control process in §5.1.4.

5.1.3 Oracle-Backed Ground Truth. P4Bench uses Python oracles to derive the expected behavior of each instantiated task. The oracle is needed because the benchmark should evaluate whether a generated P4 program realizes the intended task semantics, rather than whether it matches a target-specific reference implementation. Directly asking an agent to enumerate expected outputs for individual packets is fragile, especially for malformed packets, boundary cases, or inputs that do not clearly exercise the main behavior. P4Bench instead encodes packet-level semantics as an executable oracle, so expected outputs are computed systematically from the task specification.

We select Python as the practical oracle language for two reasons. First, current LLMs generally perform better on Python than on many other languages [41], while low-resource languages remain challenging for code models [42]; this is important because P4 is underrepresented in

public code corpora and remains difficult for large language models to generate correctly [11]. Second, Python decouples ground truth from target-specific P4 syntax and backend constraints. Programmable-data-plane platforms such as BMv2/v1model and Intel Tofino/TNA differ in architectures, externs, and compiler constraints. A Python oracle can specify the intended packet-level behavior once and reuse the same task semantics across targets.

Test packets are generated from the instantiated pattern. For each rule and invariant , the construction agent first enumerates packet classes that should exercise the behavior: matching and non-matching packets, boundary values, malformed or unsupported inputs, and state-transition cases for stateful functions. It then instantiates these classes as concrete packet headers and payloads together with the necessary control-plane configuration or runtime state needed by the oracle. Given an input test packet and the relevant states, the oracle computes the expected outcome, such as forwarding, dropping, header rewriting, state updates, or packet emission. The resulting oracle-derived input-output pairs serve as the ground truth for evaluating the P4 program generated by the model under test.

5.1.4 Reliability of Task Artifacts: Admission Control. A reliable benchmark requires each task to be realistic, implementable, and paired with test packets that faithfully evaluate the intended packet-level behavior. Since P4Bench uses an LLM-based construction agent to generate task bundles, the resulting artifacts may contain hallucinated behavior, missing corner cases, or inconsistencies with the source material. Therefore, each benchmark task bundle in P4Bench should be validated before admitting it to the benchmark corpus.

To support this validation, we define a set of validation predicates set. Each predicate is a Boolean property that a candidate bundle must satisfy before admission. P4Bench uses three predicate types: 1) Script for mechanically decidable checks, such as TPL schema completeness; 2) Probe for empirical checks, such as whether tests reject plausible incorrect implementation; 3) and Review for semantic judgments, such as whether test cases capture the intended behavior.

P4Bench enforces these predicates through a three-layer admission control process that reviews and validates each candidate task bundle before release.

Agent self-check. Given the list of validation predicates, the LLM-based construction agent reviews both the task generation process and the generated task bundle to check for predicate violations. For example, it validates the TPL schema, checks that all required fields are present, runs the oracle on the generated test packets, and confirms that both public and hidden tests have expected outputs. Task bundles that pass this self-check are forwarded to the next admission gate together with the corresponding check report.

Independent LLM judge. To reduce bias and catch errors from the construction agent, P4Bench introduces a separate LLM-based judge to review each task bundle against the validation predicates. Notably, the judge focuses on semantic and consistency checks, including whether the task description is behaviorally complete, whether the oracle and tests remain within the specified behavior, whether public tests leak implementation strategies, and whether the evidence from the constructor self-check is sufficient to support admission. The LLM judge also produces a check report. Task bundles that pass this gate, together with the judge report, are forwarded to the final human-expert review.

Human expert review. Finally, a human expert reviewer will read the check reports from the first two gates and conducts the final validation. Notably, the review will focus on checking whether the task faithfully reflects the source material, whether the network function is realistic and implementable on the target platform, whether the tests cover important behavioral cases, and whether the task adds useful coverage to the benchmark corpus. The human reviewer makes the final admission decision.

When a task bundle is rejected at any admission gate, the corresponding gate produces feedback that is returned to the construction agent for repair. Appendix B provides the complete predicate checklist and review rubrics for the three admission gates.

5.2 Task Difficulty Calibration

Benchmark tasks should span different difficulty levels to support a comprehensive assessment of model coding capabilities. Moreover, as the models under test improve, the benchmark should adapt accordingly to preserve fair and informative evaluation of their coding capabilities. Therefore, we introduce a task difficulty calibration module that measures task difficulty and supports systematic adjustment.

5.2.1 Difficulty Quantification. A principled difficulty-measurement framework is needed to quantify task difficulty in a comparable and reproducible manner. P4Bench introduces a compile-gated functional correctness metric computed from the performance of a fixed baseline LLM. For each task, the baseline model is given multiple attempts, and task difficulty is estimated from whether the generated programs compile and how well they satisfy packet-level expectations. For attempt i, let $C_i \in \{0, 1\}$ indicate whether the generated P4 program compiles on the target. If compilation succeeds, P4Bench evaluates functional correctness separately on the public and hidden test sets. Let $T_i^{pub} \in [0, 1]$ and $T_i^{hid} \in [0, 1]$ denote the fractions of packet-level expectations satisfied on the public and hidden test packet sets, respectively, and let $\alpha \in [0, 1]$ be the weight assigned to public tests. The functional test score is $T_i = \alpha T_i^{pub} + (1 - \alpha) T_i^{hid}$. The single-attempt score is then compile-gated as $F_i = C_i T_i$.

This formulation reflects three design choices. Compilation is treated as a hard deployment gate, since a non-compiling P4 program cannot exhibit valid packet-processing behavior on the target. Correctness is measured by packet-level behavior rather than syntactic similarity, since different P4 programs and control-plane entries may implement the same network function. Finally, the public/hidden weighting separates performance on visible examples from generalization to unrevealed tests.

In practice, users may query a model multiple times when using it to automate network-function design and implementation, sampling up to k candidate solutions and selecting the best one. We therefore define the task-level difficulty score as the maximum single-attempt score over k attempts: $S_k = \max_{i=1}^{k} F_i$.

5.2.2 Task Calibration Loop. For each benchmark release, P4Bench evaluates all tasks using a chosen baseline LLM and assigns them to easy, moderate, or hard categories according to predefined thresholds on the task-level score, with lower scores indicating higher difficulty. If a difficulty level is underrepresented, P4Bench applies typed mutation operators to generate sibling task instances, re-audits the resulting semantic changes, and remeasures their difficulty until the release reaches the desired difficulty distribution. As code-generation models improve, P4Bench may update the selected baseline model. When a future release uses a stronger baseline, existing task instances are remeasured and, when necessary, further mutated to restore difficulty coverage. Tasks that no longer provide meaningful evaluation signal can be retired from the benchmark corpus.

5.3 Evaluation Framework

Once released, the benchmark is used to evaluate the performance of different code-generation models. This section introduces the evaluation framework of P4Bench.

Joint Data-Plane and Control-Plane Evaluation. A complete network-function implementation requires both a data-plane program and the corresponding control-plane configuration. Even

if the data-plane program compiles and could realize the intended behavior under correct settings, evaluation fails if the control-plane configuration refers to missing tables, uses incorrect action names, or supplies action parameters in the wrong order. This reflects real P4 development, where data-plane logic and control-plane state must be consistent. P4Bench therefore evaluates the generated program and control-plane configuration jointly.

Based on this principle, P4Bench evaluates each task through the following steps:

(1) **Prompt building.** For each task instance, the evaluator extracts the task description, topology assumptions and public test examples from that instance, then combines them with the shared system instructions and required output format to form the model prompt. The prompt template is presented in Appendix D.
(2) **Model calling.** The evaluator sends the prompt to the target model and parses the response into generated P4 code and generated control-plane configuration entries.
(3) **Compilation.** The generated P4 program is compiled for the required P4 target. Compilation is a hard gate: a non-compiling submission receives zero functional score.
(4) **Deployment and execution.** If compilation succeeds, the evaluator deploys the P4 program, installs the generated table entries, replays the input test packets, and captures the resulting outputs.
(5) **Scoring.** The evaluator computes the score by comparing the captured switch outputs with the oracle-derived expected outputs and then applying the difficulty quantification formula in §5.2. Specifically, the comparison covers packet delivery or drops, egress ports, header-field transformations, and other task-specific packet-level behaviors.

6 Experimental Design

This section summarizes the current P4Bench corpus and experimental methodology. We first describe the corpus used in our evaluation, including source selection, task composition, and target coverage. We then present the evaluated models, target platforms, and evaluation metrics. Section 7 reports the corresponding results and analysis.

6.1 P4Bench Corpus

We construct the current P4Bench corpus using the task-construction pipeline in Section 5.1. The corpus is constructed from source materials selected using the task-space map. Specifically, the tasks draw on 55 academic papers from leading networking and systems venues—including NSDI, SIGCOMM, CoNEXT, SOSR, SOSP, NDSS, OSDI, FAST, USENIX, NeurIPS, and HotNets—together with 55 IETF RFCs, 8 P4 tutorial exercises, and 3 IEEE/3GPP standards. The current corpus contains 130 tasks (`redesign` (42), `scale_up` (40), `relocation` (26), and `composition` (22)), with most tasks grounded in multiple sources, and 1,914 packet-level tests. Each task includes a natural-language task specification, control-plane configuration entries, public and hidden packet tests, and a Python oracle that provides packet-level ground truth. All admitted task bundles pass the admission-control process in Section 5.1.4.

6.2 Setup and Methodology

Models. We evaluate six state-of-the-art LLMs on P4Bench: Opus-4.8, GPT-5.5, Sonnet-4.6, Gemini-3.1-Pro, Qwen-3.7-Max, and DeepSeek-V4-Pro. Each model is evaluated with three independent, feedback-free attempts per task.

Targets. All experiments use P4-16 programs compiled and executed on either BMv2/v1model or Intel Tofino/TNA. BMv2 experiments run on an Exxact Valence full-tower workstation with Ubuntu 22.04, an AMD Ryzen Threadripper PRO 5965WX processor at 3.8 GHz with 128 MB L3

Table 2. Full-corpus performance of the state-of-the-art LLM models over 130 tasks.

Model	Compilation rate	S_1	S_3	pass@1	pass@3
Opus-4.8	**82.6%**	**0.733**	**0.834**	**60.8%**	**72.3%**
Sonnet-4.6	71.8%	0.617	0.805	51.8%	69.2%
GPT-5.5	54.4%	0.479	0.782	41.5%	66.2%
Gemini-3.1-Pro	49.5%	0.431	0.695	37.7%	61.5%
Qwen-3.7-Max	21.3%	0.174	0.349	16.2%	32.3%
DeepSeek-V4-Pro	1.5%	0.008	0.023	0.8%	2.3%

cache, and two NVIDIA RTX A6000 GPUs. Tofino experiments run on an S9180-32X programmable switch with SDE version 9.6.0.

Generation and evaluation settings. Each model is queried through its provider's own interface . Every draw is a separate, stateless request consisting of a single user message built from the template in Appendix D, which contains the task description, the topology, and the public tests; hidden tests and the oracle are never shown. BMv2 submissions are compiled with p4c 1.2.4.2 (`--target bmv2 --arch v1model`), and each test runs in a fresh BMv2 `simple_switch` 1.15.0 process; Tofino submissions are compiled with bf-p4c (SDE 9.6.0). Grading is deterministic, because each test's expected output is derived from the audited Python oracle at construction time and stored with the task. Appendix E lists the per-model decoding parameters, the sampling and prompting protocol, the full evaluation environment, and the models used for task construction and failure judging.

6.3 Metrics

We use the scoring metric defined in Section 5.2.1. Specifically, for each task instance $I \in \mathcal{I}$, the model is given k independent attempts, and its task-level score is the best score across these attempts: $S_k = \max_{1 \le i \le k} F_i$. We set $\alpha = 0.2$, assigning 20% of the functional score to public tests and 80% to hidden tests. We use a retry budget $k = 3$ in all model evaluations, and report S_1 for single-attempt performance and S_3 for the best of three attempts. Since each task has three independent draws, we estimate the expected single-attempt score S_1 by the mean of F_i over the three draws, rather than using any single draw. All scores are then averaged over the tasks in the corpus. Meanwhile, we use *compilation rate* to measure the fraction of generated programs across all attempts that are accepted by the target compiler. Higher compilation rate means better understanding of P4 syntax for model under test.

Finally, we use *pass*@1 and *pass*@3 as stricter success metrics. A draw counts for *pass*@1 only if it achieves a functional score of 1.0. A task counts for *pass*@3 if at least one of its three draws achieves a functional score of 1.0. Thus *pass*@1 and *pass*@3 measure exact task completion, while S_k also gives partial credit for partially correct packet behavior.

7 Results and Analysis

7.1 Model Performance on P4Bench

This section evaluates whether the models under test can generate functionally correct P4 programs and the corresponding control-plane entries required to realize the intended network-function behavior.

As shown in Table 2, Opus-4.8 achieves the best overall performance on the full corpus across all main metrics, including compilation rate, S_1, S_3, and strict pass rates. Sonnet-4.6 is the strongest

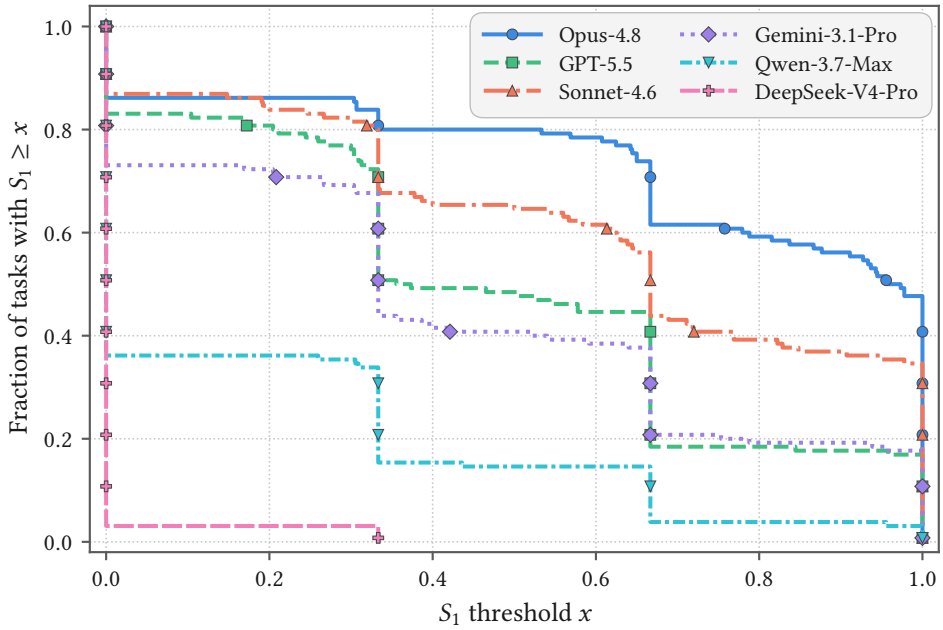


Fig. 4. CCDF of task-level S_1 over the full corpus. For each threshold x, the curve shows the fraction of tasks with $S_1 \geq x$. Higher curves indicate stronger single-attempt performance across more tasks.

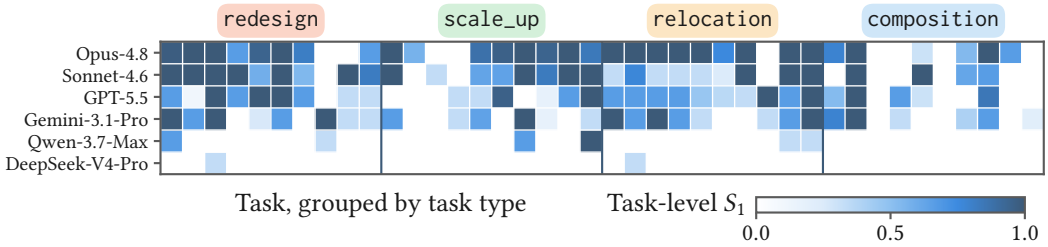


Fig. 5. Task-level S_1 heatmap by model and task type. Each cell is the mean functional score for one model-task pair. To keep the grid readable, the figure uses a balanced random sample of 40 tasks, with 10 tasks from each task type.

non-Opus model, outperforming GPT-5.5 and Gemini-3.1-Pro in both compilation rate and functional correctness. GPT-5.5 remains competitive under multiple-attempt evaluation, improving from $S_1 = 0.479$ to $S_3 = 0.782$, while Gemini-3.1-Pro improves from 0.431 to 0.695. In contrast, Qwen-3.7-Max and DeepSeek-V4-Pro are primarily limited by compilation failures. Since compilation is a hard gate in our scoring, non-compiling draws receive zero functional credit. This strongly suppresses their final scores: Qwen-3.7-Max compiles only about one fifth of its draws, while DeepSeek-V4-Pro almost never compiles.

Figure 4 further shows the Complementary Cumulative Distribution Function(CCDF) of task-level S_1 for each model under single-attempt evaluation. The curves show clear separation among the evaluated models. Opus-4.8 dominates across most score thresholds, indicating that it achieves high packet-level correctness on the largest fraction of tasks. Sonnet-4.6 is the strongest non-Opus model and consistently outperforms GPT-5.5 and Gemini-3.1-Pro. Although GPT-5.5 and Gemini-3.1-Pro remain competitive at low and moderate thresholds, their curves decline more rapidly

at high thresholds, suggesting that they complete fewer tasks with near-perfect packet-level correctness in a single attempt. Qwen-3.7-Max and DeepSeek-V4-Pro remain substantially below the frontier models, which is consistent with their low compilation rates.

This distribution is also important for validating the benchmark construction process. P4Bench uses Gemini-3.1-Pro as the baseline model for task-difficulty calibration when selecting tasks into the corpus. The fact that Opus-4.8 and Sonnet-4.6 shift the CCDF upward relative to Gemini-3.1-Pro shows that stronger models can make measurable progress beyond the calibration baseline. At the same time, the much lower curves of Qwen-3.7-Max and DeepSeek-V4-Pro show that the same corpus remains difficult for less capable models. These trends indicate that the calibrated corpus provides meaningful difficulty separation: it includes tasks that are accessible to current frontier models, tasks that distinguish stronger models from the calibration baseline, and tasks that remain challenging even for the best-performing model.

Figure 5 further breaks down task-level S_1 by model and task types on a balanced 40-task sample. The heatmap shows substantial variation across both dimensions: stronger models such as Opus-4.8 and Sonnet-4.6 achieve more consistent scores, while weaker models obtain nonzero scores only on a small number of tasks. Across task types, `composition` is the most challenging category, with a mean task-level S_1 of 0.24, below `scale_up` (0.37), `relocation` (0.45), and `redesign` (0.46). This gap indicates that compositional network functions remain harder than single-pattern tasks, suggesting that current models struggle most when they must combine multiple P4 design roles, data-plane behaviors, and control-plane requirements in a single implementation.

7.2 Failure Analysis

Understanding why existing LLMs fail to produce compilable and functionally correct implementations for benchmark tasks is essential for identifying the remaining bottlenecks in LLM-based P4 programming. This section therefore presents a failure analysis based on the evaluation results of the models studied above.

We classify model failures into two broad categories: compilation failures and functional failures. Compilation failures occur when a generated program is rejected by the target compiler. We further divide these failures into three types: `grammar_error`, where the program contains invalid syntax or violates P4 language or type rules; `target_constraint_violation`, where the program is structurally plausible but violates compiler, architecture, or pipeline constraints; and `hallucination`, where the program refers to unsupported or nonexistent constructs, metadata, externs, actions, or declarations. Samples for which the model returns no P4 program are also counted as compilation failures and reported separately as `api_timeout`, since no compiler invocation takes place. Functional failures occur when a generated artifact compiles successfully but does not realize the intended packet-level behavior. We classify these failures by the earliest stage at which task semantics are violated: `comprehension`, where the model misinterprets the required behavior; `design`, where it selects an unsuitable algorithm or data structure; `implementation`, where the intended design is encoded incorrectly in the data plane; and `configuration`, where the generated P4 program is inconsistent with the corresponding control-plane or runtime state.

We collect compilation and functional failures from all six models evaluated on the full 130-task corpus, using three samples per task, for a total of 390 samples per model. Model calls are limited to 30 minutes. The functional-failure analysis covers 2,010 failed packet tests from 274 compile-passing but functionally incorrect submissions across 74 tasks.

Figure 6 shows the distribution of compilation failures across models and target platforms, revealing several key findings.

BMv2/v1model: Opus-4.8 has fewer hallucination errors, but target constraints remain a bottleneck. Figure 6(a) shows that Opus-4.8 achieves the highest compilation success rate on

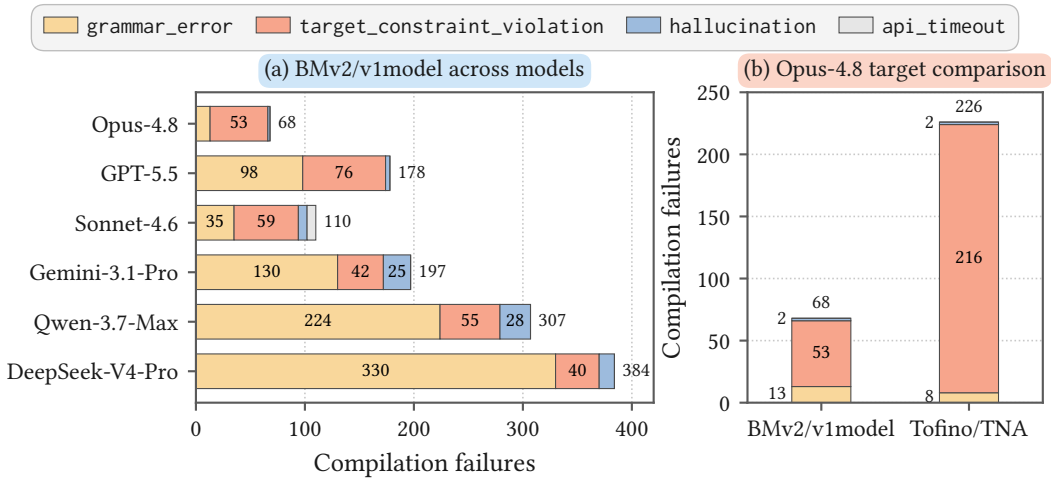


Fig. 6. Compile-failure distribution by model and target. (a) BMv2/v1model failures on full corpus across models. (b) Opus-4.8 failures on full corpus under BMv2/v1model and Tofino/TNA.

BMv2/v1model, compiling 322 of 390 samples (82.6%). Among its 68 compilation failures, only 2/68 (2.9%) is due to hallucinated target constructs, while 53/68 (78%) are `target_constraint_violation` errors. Sonnet-4.6 shows a similar failure profile: 59 of its 110 compilation failures (54%), which include 8 `api_timeout` samples, are target-constraint violations. In contrast, GPT-5.5, Gemini-3.1-Pro, Qwen-3.7-Max, and DeepSeek-V4-Pro fail more often due to `grammar_error`. These results suggest that stronger models are less likely to invent unavailable constructs or declarations, but still struggle to generate P4 programs that satisfy the BMv2/v1model compiler, architecture, and pipeline constraints.

Tofino/TNA: hardware constraints substantially reduce compilation success. Figure 6(b) compares Opus-4.8 on the same 130 tasks across BMv2/v1model and Tofino/TNA. The model compiles 322 of 390 samples (82.6%) on BMv2, but only 164 of 390 (42.1%) on Tofino. Of the 226 Tofino compilation failures, 216 are classified as `target_constraint_violation`. These failures are predominantly hardware-specific, including the use of BMv2/v1model program structure, violations of stateful-ALU and gateway-condition limits, unsupported action sources, invalid checksum placement, actions that span multiple pipeline stages, and backend target failures. These results show that syntactically plausible P4 is insufficient for hardware deployment unless the generated program also respects the target's physical pipeline and resource constraints. Therefore, P4 code generation must explicitly account for target specific hardware and architectural constraints (e.g., pipeline stages, parser restrictions). A key future challenge lies in encoding this domain knowledge directly into LLM prompts or retrieval-augmented pipelines. Notably, feeding compiler diagnostics and test harnesses output back to LLMs helps them iteratively correct both syntactic and semantic errors. We leave the exploration of such feedback loops to future work.

Figure 7 further summarizes the distribution of functional failures for three representative models, Opus-4.8, GPT-5.5, and Gemini-3.1-Pro, with *Overall* denoting the aggregate over all six evaluated models.

Configuration is the dominant source of functional failure on BMv2. Among compile-passing BMv2 submissions, configuration errors account for 67% of failed packet tests, substantially exceeding implementation errors at 24% ; comprehension and design together contribute

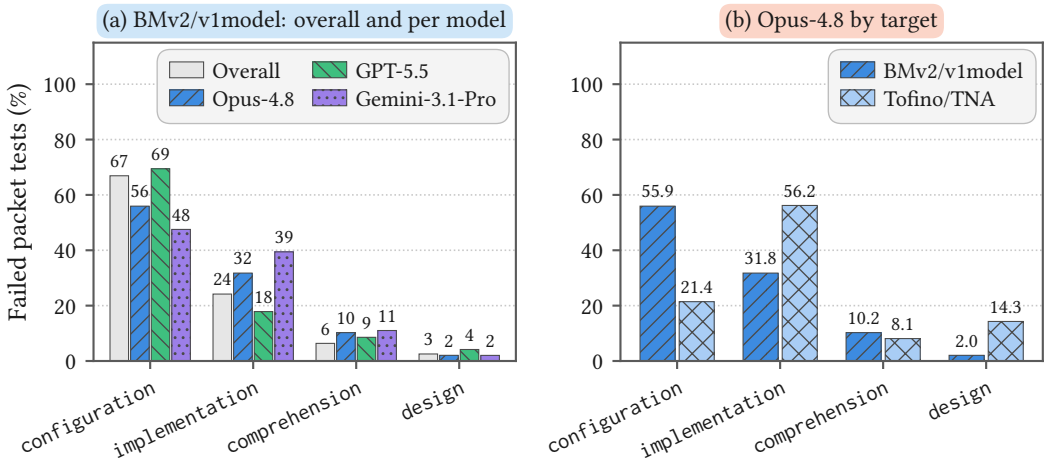


Fig. 7. Functional-failure category distributions. (a) shows BMv2/v1model failures aggregated over all six models (Overall) and for Opus-4.8, GPT-5.5, and Gemini-3.1-Pro. (b) compares Opus-4.8 functional failures under BMv2/v1model and Tofino/TNA.

only 9%. This result indicates that generating a syntactically valid P4 program is insufficient: models frequently fail to produce control-plane or runtime state that is consistent with the generated data plane. This observation is consistent with Chimera's finding that many P4 infrastructure bugs are triggered by control-plane inputs [43]. The per-model breakdown shows that configuration remains a major failure source across all three models, although Gemini-3.1-Pro exhibits a comparatively larger share of implementation errors. Ensuring functional correctness requires semantic alignment between table entries/control-plane configurations and data-plane logic. Our results suggest that automated validation layers may be important when deploying LLMs for end-to-end network programming

Implementation is the second major bottleneck. Implementation failures are considerably more frequent than comprehension or design failures across the evaluated models. Thus, models often identify a plausible network-function behavior and solution structure, but fail to encode that design faithfully in P4 under the constraints of the target architecture. The main difficulty is therefore not only selecting what the program should do, but realizing that behavior as a correct and deployable data-plane implementation.

Hardware targets shift failures toward design and implementation. Figure 7(b) compares Opus-4.8 across BMv2/v1model and Tofino/TNA. On Tofino, the share of implementation failures increases from 31.8% to 56.2%, while design failures increase from 2.0% to 14.3%; configuration failures decrease from 55.9% to 21.4%. This shift reflects constraints that are weak or absent on the software-based BMv2 target but mandatory on Tofino's physical pipeline, including stage placement, stateful-resource limits, and hardware-supported action forms. These results show that target constraints affect both high-level solution design and low-level P4 realization. Detailed failure are reported in Appendix G.

7.3 Difficulty Calibration Validity

We evaluate the mutation effectiveness using 42 base–variant task pairs. Figure 8 compares the base and variant scores under single-attempt S_1 for Opus-4.8, GPT-5.5, and Gemini-3.1-Pro. Overall, in 41 of the 42 pairs, at least one model's score changes, indicating that mutation generally produces a measurable shift in task difficulty.

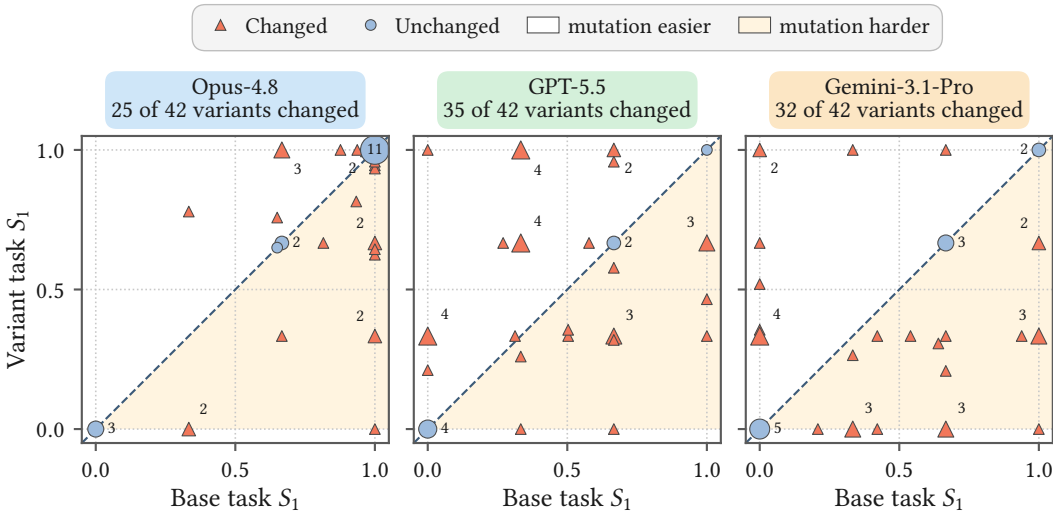


Fig. 8. Base–variant score changes on S_1. Each point compares a base task with its variant; points on the dashed line are unchanged, while points below or above it have lower or higher variant scores. Marker size and labels indicate overlapping points.

Mutation effects are model-dependent. Among the 42 variants, 19 affect all three models, 13 affect two models, 9 affect one model, and 1 affects none. Opus-4.8 changes on 25 variants, compared with 35 for GPT-5.5 and 32 for Gemini-3.1-Pro. The latter two models are more sensitive because their scores more often lie in the intermediate range, where a mutation can increase or decrease measured difficulty. In contrast, Opus-4.8 more frequently already solves or fails the base task, limiting the observable effect of some mutations.

Structural mutations are more reliable. Variants that modify resource capacity or introduce additional data-plane structure, such as tables, registers, or stateful features, change 77% of individual model scores and affect at least one model in 23 cases. By comparison, variants that primarily modify the task specification change 65% of individual model scores, and the only variant with no observable effect belongs to this category. These results suggest that mutations are most effective when they alter the concrete implementation burden, whereas specification-level changes have less predictable effects because they may also influence how readily a model recognizes the intended solution.

7.4 Artifact Feasibility by Human Experts Validation

Finally, we verify task feasibility by implementing a reference P4 program and corresponding control-plane configuration for every task. All 130 reference implementations compile and pass all packet tests, achieving a functional score of 1.0. This confirms that each benchmark task is solvable and that observed model failures are not caused by infeasible specifications or limitations of the evaluation harness.

8 Discussion

Standardized prompting for controlled capability evaluation. During evaluation, P4Bench uses a standardized prompt containing the task specification, target architecture, topology, public test cases, and required output format, without incorporating advanced prompting strategies. This design enables a controlled assessment of the models' underlying P4 programming capabilities

for automated network-function implementation. The resulting failure analysis further provides insights into how LLM-assisted network programming can be improved.

Mutation strategies. Our experiments show that well-designed mutations can systematically adjust task difficulty, reducing the need and effort to construct new tasks from scratch. Through iterative experimentation, we have already identified several effective mutation strategies. As future work, we plan to develop a more principled taxonomy and empirically characterize how different mutations affect difficulty across models, tasks, and target platforms. This analysis will provide practical guidance for efficiently recalibrating benchmark difficulty as model capabilities evolve.

Benchmark scalability. The current P4Bench corpus contains 130 tasks spanning diverse network-function categories and supports evaluation on both BMv2 and Intel Tofino targets. Its construction pipeline enables the corpus to be expanded with additional tasks as new network functions and source materials become available. Moreover, the task-difficulty calibration mechanism allows P4Bench to remain informative by recalibrating or generating task variants as the capabilities of evaluated models and systems evolve. Extending to other target requires adapting the testing scripts and integrating corresponding dedicated toolchain including compilers and control plane interfaces.

Corpus limitation. We focus on localizable network functions evaluated on a singleswitch topology to directly benchmark data-plane generation quality without multi-hop confounding factors. Support for multi-switch topologies is a natural extension, but it requires additional considerations for topology representation, control-plane configuration, and packet-level evaluation.

9 Conclusion

We presented P4Bench, a scalable and adaptive benchmark for evaluating LLM-based P4 network-function design and implementation. P4Bench combines TPL-based task representation, an LLM-assisted construction pipeline, oracle-backed packet-level evaluation, and mutation-based difficulty calibration. Its current corpus contains 73 TPL patterns, 130 full-program tasks, and 1,914 packet tests. P4Bench evaluates complete P4 data-plane programs together with their corresponding control-plane configurations across BMv2/v1model and Intel Tofino/TNA.

Our evaluation of six frontier LLMs shows that P4 programming remains challenging: even Opus-4.8 compiles only 82.6% of generated BMv2 programs, achieves a single-attempt functional score of 0.733, and compiles only 42.1% on Tofino. The failure analysis reveals persistent weaknesses in target-specific validity, implementation correctness, packet-level semantics, and data-plane/control-plane consistency. By exposing these systems-level failures while supporting scalable task construction and difficulty recalibration, P4Bench provides a foundation for measuring progress and guiding the development of more reliable, target-aware P4 code generators and agentic systems for automated network-function design and implementation.

Acknowledgments

We would like to thank our shepherd and anonymous reviewers for their valuable feedback. The work of Shuyi Liao, Yannian Niu, Xinyue Zheng, and Minmei Wang was partially supported by the National Science Foundation under Grant CNS-2426030. The work of Tingting Yu was partially supported by the National Science Foundation under Grant CCF-2403747.

References

[1] Oliver Michel, Roberto Bifulco, Gabor Retvari, and Stefan Schmid. 2021. The Programmable Data Plane: Abstractions, Architectures, Algorithms, and Applications. *Comput. Surveys* 54, 4 (2021), 1–36.

[2] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-Independent Packet Processors.

ACM SIGCOMM Computer Communication Review 44, 3 (July 2014), 87–95. doi:10.1145/2656877.2656890

[3] Jianxun Wang and Yixiang Chen. 2023. A Review on Code Generation with LLMs: Application and Evaluation. In *Proceedings of the 2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*. IEEE, Piscataway, NJ, USA, 284–289.

[4] Yihong Dong, Xue Jiang, Jiaru Qian, Tian Wang, Kechi Zhang, Zhi Jin, and Ge Li. 2025. A Survey on Code Generation with LLM-Based Agents. arXiv:2508.00083 https://arxiv.org/abs/2508.00083

[5] Chia-Tung Ho, Haoxing Ren, and Brucek Khailany. 2025. VerilogCoder: Autonomous Verilog Coding Agents with Graph-Based Planning and Abstract Syntax Tree (AST)-Based Waveform Tracing Tool. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 39. AAAI Press, Washington, DC, USA, 300–307.

[6] Sahil Bhatia, Jie Qiu, Niranjan Hasabnis, Sanjit A. Seshia, and Alvin Cheung. 2024. Verified Code Transpilation with LLMs. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 37. Curran Associates, Inc., Red Hook, NY, USA, 41394–41424.

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] https://arxiv.org/abs/2107.03374

[8] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. arXiv:2108.07732 [cs.PL] https://arxiv.org/abs/2108.07732

[9] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues?. In *The Twelfth International Conference on Learning Representations (ICLR)*. OpenReview.net.

[10] Naman Jain, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. In *The Thirteenth International Conference on Learning Representations (ICLR)*. OpenReview.net.

[11] Mihai-Valentin Dumitru, Vlad-Andrei Bădoiu, Alexandru M. Gherghescu, and Costin Raiciu. 2025. Generating P4 Data Planes Using LLMs. *Computer Networks* 272 (2025), 111709. doi:10.1016/j.comnet.2025.111709

[12] Changzhi Zhou, Ao Liu, Yuchi Deng, Zhiying Zeng, Tao Zhang, Haotian Zhu, Jianwei Cai, Yue Mao, Chenchen Zhang, Lingyun Tan, Ziyan Xu, Bohui Zhai, Hengyi Liu, Speed Zhu, Wiggin Zhou, and Fengzong Lian. 2026. AutoCodeBench: Large Language Models Are Automatic Code Benchmark Generators. In *The Fourteenth International Conference on Learning Representations (ICLR)*. OpenReview.net.

[13] Frederik Hauser, Marco Häberle, Daniel Merling, Steffen Lindner, Vladimir Gurevich, Florian Zeiger, Reinhard Frank, and Michael Menth. 2023. A Survey on Data Plane Programming with P4: Fundamentals, Advances, and Applied Research. *Journal of Network and Computer Applications* 212 (March 2023), 103561. doi:10.1016/j.jnca.2022.103561

[14] Elie F. Kfoury, Jorge Crichigno, and Elias Bou-Harb. 2021. An Exhaustive Survey on P4 Programmable Data Plane Switches: Taxonomy, Applications, Challenges, and Future Trends. *IEEE Access* 9 (2021), 87094–87155. doi:10.1109/ACCESS.2021.3086704

[15] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. In *Proceedings of the 2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, Piscataway, NJ, USA, 31–53.

[16] Shiwen Ni, Guhong Chen, Shuaimin Li, Xuanang Chen, Siyi Li, Bingli Wang, Qiyao Wang, Xingjian Wang, Yifan Zhang, Liyang Fan, et al. 2025. A Survey on Large Language Model Benchmarks. arXiv:2508.15361 https://arxiv.org/abs/2508.15361

[17] Mubashara Akhtar, Anka Reuel, Prajna Soni, Sanchit Ahuja, Pawan Sasanka Ammanamanchi, Ruchit Rawal, Vilém Zouhar, Srishti Yadav, Chenxi Whitehouse, Dayeon Ki, Jennifer Mickel, Leshem Choshen, Marek Suppa, Jan Batzner, Jenny Chim, Jeba Sania, Yanan Long, Hossein A. Rahmani, Christina Q. Knight, Yiyang Nan, Jyoutir Raj, Yu Fan, Shubham Singh, Subramanyam Sahoo, Eliya Habba, Usman Gohar, Siddhesh Milind Pawar, Robert Scholz, Arjun Subramonian, Jingwei Ni, Mykel Kochenderfer, Sanmi Koyejo, Mrinmaya Sachan, Stella Biderman, Zeerak Talat, Avijit Ghosh, and Irene Solaiman. 2026. When AI Benchmarks Plateau: A Systematic Study of Benchmark Saturation. In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*. PMLR. https://openreview.net/forum?id=YC1Otscjbs

[18] Simin Chen, Pranav Pusarla, and Baishakhi Ray. 2025. Dynamic Benchmarking of Reasoning Capabilities in Code Large Language Models Under Data Contamination. arXiv:2503.04149 https://arxiv.org/abs/2503.04149

[19] Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. 2023. VerilogEval: Evaluating Large Language Models for Verilog Code Generation. In *Proceedings of the 2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, Piscataway, NJ, USA, 1–8.

[20] Wamiq Para, Shariq Bhat, Paul Guerrero, Tom Kelly, Niloy Mitra, Leonidas J. Guibas, and Peter Wonka. 2021. SketchGen: Generating Constrained CAD Sketches. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 34. Curran Associates, Inc., Red Hook, NY, USA, 5077–5088.

[21] Prashanth Vijayaraghavan, Luyao Shi, Stefano Ambrogio, Charles Mackin, Apoorva Nitsure, David Beymer, and Ehsan Degan. 2024. VHDL-Eval: A Framework for Evaluating Large Language Models in VHDL Code Generation. In *Proceedings of the 2024 IEEE LLM Aided Design Workshop (LAD)*. IEEE, Piscataway, NJ, USA, 1–6.

[22] Theo Olausson, Alex Gu, Ben Lipkin, Cedegao Zhang, Armando Solar-Lezama, Joshua Tenenbaum, and Roger Levy. 2023. LINC: A Neurosymbolic Approach for Logical Reasoning by Combining Language Models with First-Order Logic Provers. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Stroudsburg, PA, USA, 5153–5176.

[23] Jiale Guo, Suizhi Huang, Mei Li, Dong Huang, Xingsheng Chen, Regina Zhang, Zhijiang Guo, Han Yu, Siu-Ming Yiu, Pietro Lio, and Kwok-Yan Lam. 2025. A Comprehensive Survey on Benchmarks and Solutions in Software Engineering of LLM-Empowered Agentic System. arXiv:2510.09721 [cs.SE] https://arxiv.org/abs/2510.09721

[24] Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. 2024. Large Language Models: A Survey. arXiv:2402.06196 https://arxiv.org/abs/2402.06196

[25] Prakhar Sharma and Vinod Yegneswaran. 2023. PROSPER: Extracting Protocol Specifications Using Large Language Models. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks (HotNets)*. ACM, New York, NY, USA, 41–47.

[26] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V. Le, Denny Zhou, et al. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 35. Curran Associates, Inc., Red Hook, NY, USA, 24824–24837.

[27] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. In *The Eleventh International Conference on Learning Representations (ICLR)*. OpenReview.net. https://openreview.net/forum?id=1PL1NIMMrw

[28] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language Agents with Verbal Reinforcement Learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 36. Curran Associates, Inc., Red Hook, NY, USA, 8634–8652.

[29] Shangqing Liu, Yu Chen, Xiaofei Xie, Jing Kai Siow, and Yang Liu. 2021. Retrieval-Augmented Generation for Code Summarization via Hybrid GNN. In *The Ninth International Conference on Learning Representations (ICLR)*. OpenReview.net. https://openreview.net/forum?id=zv-typ1gPxA

[30] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. 2023. Retrieval-Augmented Generation for Large Language Models: A Survey. arXiv:2312.10997 https://arxiv.org/abs/2312.10997

[31] Md Rizwan Parvez, Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval Augmented Code Generation and Summarization. In *Findings of the Association for Computational Linguistics: EMNLP 2021*. Association for Computational Linguistics, Stroudsburg, PA, USA, 2719–2734.

[32] Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. 2023. AgentCoder: Multi-Agent-Based Code Generation with Iterative Testing and Optimisation. arXiv:2312.13010 https://arxiv.org/abs/2312.13010

[33] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable Code Actions Elicit Better LLM Agents. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*. PMLR.

[34] Yoichi Ishibashi and Yoshimasa Nishimura. 2024. Self-Organized Agents: A LLM Multi-Agent Framework Toward Ultra Large-Scale Code Generation and Optimization. arXiv:2404.02183 https://arxiv.org/abs/2404.02183

[35] Mohammad Talal Jamil, Shamsa Abid, and Shafay Shamail. 2025. Can LLMs Generate Higher Quality Code than Humans? An Empirical Study. In *Proceedings of the 22nd IEEE/ACM International Conference on Mining Software Repositories (MSR)*. IEEE, Piscataway, NJ, USA, 478–489.

[36] Ruidi Qiu, Grace Li Zhang, Rolf Drechsler, Ulf Schlichtmann, and Bing Li. 2024. AutoBench: Automatic Testbench Generation and Evaluation Using LLMs for HDL Design. In *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD (MLCAD)*. ACM, New York, NY, USA, 1–10.

[37] Jia He, Mukund Rungta, David Koleczek, Arshdeep Sekhon, Franklin X. Wang, and Sadid Hasan. 2024. Does Prompt Formatting Have Any Impact on LLM Performance? arXiv:2411.10541 [cs.CL] https://arxiv.org/abs/2411.10541

[38] Athanasios Liatifis, Panagiotis Sarigiannidis, Vasileios Argyriou, and Thomas Lagkas. 2023. Advancing SDN from OpenFlow to P4: A Survey. *Comput. Surveys* 55, 9, Article 186 (Jan. 2023), 37 pages. doi:10.1145/3556973

[39] Mihai Budiu and Chris Dodd. 2017. The $P4_{16}$ Programming Language. *ACM SIGOPS Operating Systems Review* 51, 1 (Sept. 2017), 5–14. doi:10.1145/3139645.3139648

[40] Jorge Andrés Brito, José Ignacio Moreno, Luis Miguel Contreras, Manuel Alvarez-Campana, and Marta Blanco Caamaño. 2023. Programmable Data Plane Applications in 5G and Beyond Architectures: A Systematic Review. *Sensors* 23, 15 (Aug. 2023), 6955. doi:10.3390/s23156955

[41] Sathvik Joel, Jie Wu, and Fatemeh Fard. 2025. A Survey on LLM-Based Code Generation for Low-Resource and Domain-Specific Programming Languages. *ACM Transactions on Software Engineering and Methodology* (Oct. 2025). Just Accepted. doi:10.1145/3770084

[42] Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Anders Freeman, Carolyn Jane Anderson, Molly Q. Feldman, Michael Greenberg, Abhinav Jangda, and Arjun Guha. 2024. Knowledge Transfer from High-Resource to Low-Resource Programming Languages for Code LLMs. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2, Article 295 (Oct. 2024), 32 pages. doi:10.1145/3689735

[43] Jiwon Kim, Dave (Jing) Tian, and Benjamin E. Ujcich. 2025. Chimera: Fuzzing P4 Network Infrastructure for Multi-Plane Bug Detection and Vulnerability Discovery. In *Proceedings of the 46th IEEE Symposium on Security and Privacy (S&P)*. IEEE, Piscataway, NJ, USA.

A Task Pattern Language Example

This appendix gives two complete TPL examples. Figure 9 gives the complete ACL case study that is summarized in Figure 1. The task does not expose the TPL representation directly to the model under test; instead, it receives a natural-language specification rendered from the structured pattern. The oracle executes the same transition rules to derive packet-level ground truth, the tests are instantiated from the stated obligations, and calibrated variants edit typed mutation axes while preserving the underlying ACL semantics.

Figure 10 shows a second case study that requires composition rather than a single standalone pattern. The task chains a zone-based stateful firewall with IPv4 NAPT into one edge gateway, matching the `composition` task type in §5.1.1. In contrast to the stateless ACL in Figure 9, where each packet can be classified independently, this composed pattern combines stateless screening (R_1–R_2), per-connection state maintenance (R_5–R_6, R_{10}), and TCP-ordering behavior at the higher faithfulness level (ϕ=D_2). As a result, several correctness requirements are trace-level properties and cannot be validated by a single packet in isolation.

The pattern centers on a shared connection table C. Each connection record stores both the firewall-side state, including the verdict, timestamp, and last-seen metadata (c.est, c.ts, c.ls), and the NAT-side mapping (c.pip, c.xp). On the outbound path, an internal packet whose source belongs to subnet N either refreshes an existing connection and applies the stored SNAT mapping (R_5), or, on a miss, creates the outbound connection together with its inbound mirror C^{-1}, allocates an external port from Π, rewrites the source address and port, and forwards the packet externally (R_6). On the inbound path, a packet addressed to the public address ρ is admitted only when its reverse key resolves to an established connection. In that case, the packet is DNAT-ed back to the internal endpoint and forwarded inward (R_{10}). Otherwise, the packet follows the default-deny rule (R_{12}). The seed fixes a concrete instance of this behavior, and the derived tests exercise both forwarding directions as well as negative cases such as unsolicited inbound traffic and packets addressed to the wrong public destination.

This example illustrates why composition introduces obligations that are not present in either component alone. Because the firewall and NAT share one connection table, the pattern must keep the outbound and inbound table entries synchronized, prevent inbound packets from creating state, and preserve the inverse relationship between SNAT and DNAT. Its tests are therefore sequence-based: they must first establish outbound state and then verify that only the matching return traffic

1. Network-function. An ACL switch checks each IPv4 packet against a priority-ordered policy-entry set. The highest-priority matching ACL entry determines the action. A permitted packet is forwarded to port 2 without header changes; a denied packet is dropped. If no ACL entry matches, the default action is drop.

2. Structured TPL pattern.
This reusable pattern declares all components of $P = (\mathcal{E}, \Theta, R, \Phi, \mathcal{T}, \mathcal{M}, G)$.

$\mathcal{E}$	Packet p; ACL entry e; policy-entry set A; ingress port; egress port. Each entry has a priority, match predicate, and action.

Θ: parameter declarations

Parameter	Type	Constraint	Default
d_0	Action	$d_0 \in \{\text{deny}, \text{permit}\}$	deny
p_{in}	Port	valid ingress port	1
p_{out}	Port	valid egress port; $p_{\text{out}} \neq p_{\text{in}}$	2
A	List[ACLEntry]	nonempty; unique priorities; actions in {deny, permit}	

R	**Rule1: when** IPv4 packet p has matching entries $M = \{e \in A \mid e.\text{match}(p)\}$, $M \neq \emptyset$, $e^\star = \arg\max_{e \in M} e.\text{priority}$, and $e^\star.\text{action} = \text{permit}$. **given** p, A, p_{out}. **then** forward p unchanged to p_{out}. **Rule2: when** IPv4 packet p has matching entries $M = \{e \in A \mid e.\text{match}(p)\}$, $M \neq \emptyset$, $e^\star = \arg\max_{e \in M} e.\text{priority}$, and $e^\star.\text{action} = \text{deny}$. **given** p, A. **then** drop p. **Rule3: when** no entry in A matches p. **given** p, d_0, p_{out}. **then** apply d_0; a default deny drops p, and a default permit forwards p unchanged to p_{out}.
Φ	Priority correctness, default-policy correctness, and header preservation for permitted packets.
$\mathcal{T}$	Test scaffolding requires cases for explicit deny, explicit permit, overlapping-match priority, and no-match default behavior. Each test fixes an input packet and checks the resulting egress decision and header preservation when forwarded. Priority cases use overlapping entries so the expected packet result depends on selecting the highest-priority match.
$\mathcal{M}$	Mutation axes include entry count, overlap depth, default action, predicate specificity, and port assignment.
G	RFC3198.

3. Seed and derived test cases.

Seed	$d_0 = \text{deny}, p_{\text{in}} = 1, p_{\text{out}} = 2, A = [e_1, e_2]$, where $e_1 = (20, \text{srcIP} = 10.0.1.10, \text{deny})$ and $e_2 = (10, \text{srcIP} \in 10.0.1.0/24, \text{permit})$.

Testing case	Concrete packet	Expected result
explicit deny / overlap priority	`10.0.1.10`	select e_1 over e_2; drop
explicit permit	`10.0.1.20`	select e_2; forward to port 2 unchanged
default deny	`10.0.2.20`	no matching entry; drop

4. Model-facing specification.
Implement an IPv4 ACL in P4-16 for the v1model architecture. The switch receives packets on port 1. It should apply the installed ACL entries in priority order. If a packet matches multiple entries, the highest-priority matching entry must determine the action. A permitted packet must be forwarded to port 2 without modifying its headers. A denied packet must be dropped. If no ACL entry matches, the packet must be dropped. Provide both the P4 program and the control-plane entries for the ACL entries.

Fig. 9. Detailed ACL case-study instantiation showing the network-function idea, structured TPL pattern, seed-derived tests, and model-facing specification.

is admitted. Calibrated variants stress the same composed semantics by increasing the faithfulness level from D_0 to D_2, reducing κ or Π, or composing an additional rate limiter.

B Detailed Task Admission Control

P4Bench uses LLM-assisted construction to produce candidate tasks, but syntactic well-formedness alone is insufficient for admission. Admission is applied to the complete *task bundle*, covering the TPL pattern, instantiated configuration, model-facing specification, Python oracle, public and hidden tests. For tasks selected for feasibility sampling, P4Bench checks whether the intended behavior can be compiled, deployed, and exercised under the declared target assumptions, but this sample is not used to define ground truth.

Admission checks whether the task bundle defines a reliable benchmark item as a whole. Since P4 correctness is measured by packet behavior after compilation and entry installation, the artifacts in the task bundle must be mutually consistent rather than correct only in isolation. To make this consistency check reliable and auditable, P4Bench validates each bundle through a staged admission process. Deterministic scripts cover mechanically decidable checks, probes cover checks

1. Network-function. An edge gateway composes a zone-based stateful firewall with IPv4 NAPT in a single pipeline pass. The internal subnet N is masqueraded behind one public address ρ (port 1 internal, port 2 external). Outbound (internal→external): a new connection is admitted by default, an external port is allocated, and the source is SNAT-ed; the firewall verdict and the NAT mapping are written together into one shared connection table C. Inbound (external→internal): a packet is DNAT-ed and forwarded *only if* its reverse key matches an established outbound connection; all other inbound traffic falls to default-deny. Non-IPv4 and non-TCP/UDP packets are dropped before any lookup.

2. Structured TPL pattern.

This composed pattern $P = P_{\mathsf{FW}} \circ P_{\mathsf{NAT}}$ declares all components of $P = (\mathcal{E}, \Theta, R, \Phi, \mathcal{T}, \mathcal{M}, G)$.

$\mathcal{E}$ Packet p with fields p.sa, p.da, p.sp, p.dp, p.pr, p.ttl; 5-tuple key $k(p)$ and reverse key $\bar{k}(p)$; shared connection table C (entry c; attributes c.est (verdict), c.ts (tcp state), c.ls (last seen), c.pip, c.xp); inbound mirror C^{-1} : $(\rho, \mathsf{dp}, \mathsf{pr}) \mapsto c$; port free-list Π; ingress zone $z(p) \in \{\mathsf{in}, \mathsf{ex}\}$.

Θ: parameter declarations

Param	Type	Constraint	Default
N	Addr	ipv4_cidr (internal subnet);required	
ρ	Addr	ipv4 (single public IP);required	
m	Enum	{stat, dyn}	stat
$[\pi_\ell, \pi_h]$	Port^2	$1024 \le \pi_\ell < \pi_h \le 65535$	[10000, 65535]
κ	Int | ∞	$[64,\ 2^{20}]$ (conntrack cap.)	∞
ε	Enum	{none, FIFO, LRU, to}	none
ϕ	Enum	$D_0 < D_1 < D_2$ (faithfulness)	D_0
ι	Bool	endpoint-indep. map (RFC 4787)	$\top$

R **R1: when** $\neg$ip4(p). **then** drop p.
R2: when ip4(p) $\wedge$ p.pr $\notin$ {tcp, udp}. **then** drop p.
R5 (outbound hit): when $z(p)$=in $\wedge$ p.sa $\in N \wedge$ p.pr $\in$ {tcp, udp} $\wedge\ \exists\, c^\star \in C:\ c^\star$.key=$k(p)$. **given** p, C. **then** $c^\star$.ls $\leftarrow$ now; p.sa $\leftarrow c^\star$.pip, p.sp $\leftarrow c^\star$.xp; p.ttl $-\!\!=1$; fwd(p, p_{ex}).
R6 (outbound miss, m=dyn): when $z(p)$=in $\wedge$ p.sa $\in N \wedge$ p.pr $\in$ {tcp, udp} $\wedge\ \nexists c \in C:$ c.key=$k(p) \wedge m$=dyn $\wedge\ \Pi \neq \emptyset$. **given** p, C, C^{-1}, Π. **then** bind$\big(C, k(p), \{\mathsf{est}{=}\top, \mathsf{ts}{=}\mathsf{NEW}, \mathsf{pip}{=}\rho, \mathsf{xp}{=}\mathsf{alloc}(\Pi)\}\big) \wedge$ bind$\big(C^{-1}, (\rho, c.\mathsf{xp}, p.\mathsf{pr}), k(p)\big)$ (same firing); p.sa $\leftarrow \rho$, p.sp $\leftarrow c$.xp; p.ttl $-\!\!=1$; fwd(p, p_{ex}).
R10 (inbound return): when $z(p)$=ex $\wedge$ p.da=$\rho \wedge c^\star{=}C^{-1}(\rho, p.\mathsf{dp}, p.\mathsf{pr})$ exists $\wedge$ $c^\star$.est=$\top \wedge (\phi \neq D_2 \vee$ legal($c^\star$.ts, p)). **given** p, C^{-1}, C. **then** $c^\star$.ls $\leftarrow$ now, $c^\star$.ts $\leftarrow$ next($c^\star$.ts, p); p.da $\leftarrow c^\star$.key.sa, p.dp $\leftarrow c^\star$.key.sp; p.ttl $-\!\!=1$; fwd(p, p_{in}).
R12 (default deny): when $z(p)$=ex (otherwise). **then** drop p.
First-match evaluation; the full pattern has 12 rules (uRPF / fragment-drop rules R_3, R_4 fire only at $\phi{=}D_2$).

Φ pair_sym: $\forall c \in C\ \exists!\, c' \in C^{-1}$ bound and evicted in the same firing; state_from_in: $z(p)$=ex $\Rightarrow$ no bind$(C, \cdot)$ (SYN-flood resilience); bidir: $\mathrm{SNAT}_{R5/R6} \circ \mathrm{DNAT}_{R10} = \mathrm{id}$; deny: $z(p)$=ex $\wedge\ \nexists c^\star \Rightarrow$ drop; eim (when ι): c.xp = $f(p.\mathsf{sa}, p.\mathsf{sp}, p.\mathsf{pr})$ only.

$\mathcal{T}$ Scaffolding requires outbound-allocate-and-SNAT, inbound-return-DNAT, reflexive-admit-only-after-outbound (z=ex with no prior z=in must drop), capacity/eviction, SYN-flood resilience, and IP+L4 checksum recomputation after every rewrite. Sequences are packet-level: the verdict depends on prior state.

$\mathcal{M}$ Axes: faithfulness ladder $D_0 \to D_1 \to D_2$, κ, $[\pi_\ell, \pi_h]$, ε, ι, and algebra breadth (compose with a rate limiter); each harden operator is paired with an ease operator.

G RFC 3022 (NAPT), RFC 4787 (UDP NAT), RFC 5382 (TCP state), RFC 2979/6092 (firewall default-deny, return traffic), RFC 3704 (uRPF); TEA / Sailfish / Sirius / Mantis.

3. Seed and derived test cases.

Seed N=`10.0.1.0/24`, ρ=`203.0.113.1`, m=stat, $\phi{=}D_0$; preinstalled mappings UDP `10.0.1.5:5000` ↔ xp `10000` and TCP `10.0.1.5:6000` ↔ xp `10001`.

Scaffolded case	Concrete packet		Expected result
outbound SNAT (UDP)	z=in, `10.0.1.5:5000→8.8.8.8:53`	UDP	admit; p.sa→ρ, p.sp→`10000`, out p_{ex}
proto-discriminated bind	z=in, `10.0.1.5:6000→8.8.8.8:443`	TCP	admit; p.sp→`10001` (pr keys the bind)
endpoint-indep. map (ι)	same (sa, sp, pr) to dst `1.1.1.1`		admit; same xp (RFC 4787)
inbound DNAT (return)	z=ex, TCP →`203.0.113.1:10001`		admit; p.da→`10.0.1.5`, p.dp→`6000`, out p_{in}
unsolicited default-deny	z=ex, TCP →`203.0.113.1:9999`		$\nexists c^\star$; drop (deny)
wrong-dst drop	z=ex, p.da=`198.51.100.7` $\neq \rho$		drop before lookup

4. Model-facing specification.

Implement a stateful firewall combined with IPv4 NAPT in P4-16 for the v1model architecture. Port 1 is the trusted internal side and port 2 the external side; the internal subnet `10.0.1.0/24` *is masqueraded behind the public address* `203.0.113.1`*. Maintain a connection table keyed by the TCP/UDP 5-tuple that records both the firewall verdict and the NAT mapping. On the internal→external path, admit a flow and SNAT its source address/port; on the external→internal path, forward a packet only if its reverse 5-tuple matches an established outbound flow, DNAT-ing it back to the internal endpoint — otherwise drop it. Drop non-IPv4 and non-TCP/UDP traffic. Recompute IPv4 and TCP/UDP checksums after every rewrite. Provide both the P4 program and the control-plane entries.*

Fig. 10. Detailed stateful-firewall ∘ NAPT case-study instantiation: a composed multi-pattern task.

Table 3. Three admission gates for validating candidate task bundles.

Gate	Role in admission	Blocks when
Gate 1: construction-agent self-check	establishes the initial evidence base using deterministic scripts and required probes	a required mechanical or executable predicate fails
Gate 2: independent LLM review	reviews the candidate bundle and Gate 1 report for semantic and cross-artifact consistency	an admission-relevant defect remains or the evidence is insufficient
Gate 3: human expert review	reviews the bundle, prior reports, grounding evidence, and corpus fit before the final decision	a severe defect, unresolved disagreement, or feasibility concern remains

Table 4. Checklist for pattern and seed admission.

ID	Self-check question	Evidence	Severity
GS1	Does each rule/invariant citation locator exist, and is the quoted passage present?	script	gate
GS2	Does each cited passage support the attached rule/invariant?	review	gate
GS3	How completely are the pattern's rules and invariants grounded?	review	flag
DS1	Does at least one rule or invariant justify each declared tag?	review	gate
DS2	Is important rule behaviour missing from the declared tags?	review	flag
DS3	How well do the tags describe the pattern's behaviour?	review	flag
VP1	Are all rule `when`/`then` clauses expressed in TPL v1.0 vocabulary?	script	gate
VP2	Does every invariant declare `enforced_when`?	script	gate
VP3	Does every parameter reference resolve to a declared parameter?	script	gate
VP4	Does the pattern declare at least one parameter used by a seed binding or mutation operator?	script	gate
VP5	Is the parameter surface appropriate for realistic seeds and difficulty mutations?	review	flag
VP6	Is this one reusable pattern rather than several hidden patterns?	review	flag
RV1	Is each seed binding inside the declared grounded range?	script	flag
RV2	Is each seed binding realistic relative to the grounding sources?	review	flag
RV3	How faithful are the bindings to the source literature?	review	flag

that require execution or measurement, and independent review covers semantic checks that require interpretation. The construction agent can generate the bundle and supporting evidence, but the final admission decision is made outside the construction process. This design prevents self-approval and reserves human review for issues that cannot be resolved automatically. We instantiate this admission process through a staged procedure and a common bundle-consistency rubric. The procedure defines how validation evidence flows across gates, while the rubric specifies the artifact-level checks applied at each gate.

B.1 Staged Admission Procedure

Table 3 summarizes the staged admission process. The gates are organized to accumulate validation evidence while increasing reviewer independence. Gate 1 establishes the initial evidence base by running the required scripts and probes over the generated bundle. Passing bundles are forwarded with the constructor's self-check report. Gate 2 adds independent LLM review. The judge reviews the candidate bundle together with the Gate 1 report, checks semantic and cross-artifact consistency, and forwards passing bundles with a judge report. Gate 3 makes the final admission decision. The human reviewer considers the bundle, prior reports, and grounding evidence, focusing on source faithfulness, target feasibility, behavioral coverage, and corpus fit. Bundles rejected at any gate are returned to the construction agent for repair.

Table 5. Checklist for oracle admission.

ID	Self-check question	Evidence	Severity
OR1	Does the oracle expose the required `step` interface?	script	gate
OR2	Does the oracle match every canonical example by value?	script	gate
OR3	Do canonical examples exercise each config-read mutation parameter at feasibility extremes?	script	gate
OR4	Does oracle output change across min/max values for config-read parameters that should affect behaviour?	script	gate
OR5	Does the oracle read seed-bound parameters from runtime config rather than baking them as constants?	script	gate
OR6	For safety or field-relation invariants, does a witness satisfy the invariant and a targeted mutant violate it?	script	gate
OR7	For fairness or distribution invariants, is the K-flow distribution within tolerance?	probe	flag
OR8	For liveness, timing, or semantic invariants, does reviewer judgement support the invariant?	review	gate
OR9	Where canonical examples do not fix the answer, does the oracle follow source semantics?	review	gate
OR10	How faithful is the oracle to the pattern and sources?	review	flag
OR11	For each rule, when its `when` condition holds, is a corresponding `step` branch reached?	script	gate
OR12	Does the oracle reproduce each published test case's `expected` block?	script	gate

Table 6. Checklist for test-suite, public-specification, and package admission.

ID	Self-check question	Evidence	Severity
TC1	Is each rule exercised by at least one test input?	script	gate
TC2	Is a mutant corrupting each rule's `then` behaviour killed by at least one hidden test?	script	gate
TC3	Does at least one hidden test assert each invariant?	script	gate
TC4	Does the hidden suite kill 100% of the injected mutant catalogue?	script	gate
TC5	Does the mutant catalogue cover plausible wrong implementations for this pattern?	review	flag
TC6	How well do the tests distinguish correct submissions from plausible incorrect ones?	probe	flag
SP1	Does any removed token reappear in the task description, test descriptions, or exposed fields?	script	gate
SP2	On inputs where independent solver models converge, what fraction converge on the oracle answer?	probe	gate
SP3	How large is the solver-accuracy drop from the full internal description to the sanitised description?	probe	gate
SP4	For each non-converged input, is divergence caused by under-specification rather than task difficulty?	review	gate
SP5	Is every oracle behaviour or branch licensed by the SUT-facing specification?	review	gate
SP6	Is there a second defensible reading of the public specification that changes a verdict?	review	gate
SP7	Overall, does the public description determine the correct behaviour?	probe	flag
PB1	Does the mock-reference submission score 1.0?	script	gate
PB2	Is the provenance record complete and consistent with the calibration record?	script	gate

B.2 Checklist and Rubric

Table 4,5,6 shows the checklist used in the self-check by construction agent. Table 7 is the rubric used by the independent review agent and human expert.

Rubric evaluation produces one of three outcomes for each candidate bundle: ADMIT, REPAIR, or REJECT. ADMIT releases a bundle whose blocking criteria all pass and whose severe defects, if any, have been resolved. REPAIR returns a near-admissible bundle to the construction agent when the remaining issue can be corrected without changing the task identity, such as incomplete provenance, insufficient hidden-test coverage, or missing admission evidence. REJECT removes a bundle from consideration when a severe defect can change the benchmark verdict or when repair would require changing the intended task semantics.

Severe defects are defects that can make the benchmark verdict unreliable. They include ambiguous model-facing specifications, oracle–specification mismatches, expected outputs that are inconsistent with the current oracle, tests that exercise behavior outside the stated specification,

Table 7. Review rubric used by the LLM judge at Gate 2 and the human reviewer at Gate 3. Each item receives PASS, PARTIAL, or FAIL; final task verdicts are ADMIT, REPAIR, or REJECT.

ID	Artefact	Review question	Self-check evidence	Severity
B-PAT1	Pattern	Does the cited source support every rule and invariant?	GS1, GS2, GS3	gate
B-PAT2	Pattern	Do the design-space tags accurately describe the pattern?	DS1, DS2, DS3	flag
B-PAT3	Pattern	Is the pattern valid TPL, with no syntax or meaning errors?	VP1–VP6	gate
B-SEED1	Seed	Are the chosen parameter values realistic for the network scenario?	RV1, RV2	flag
B-SEED2	Seed	Do the chosen values match source ranges or examples?	RV3	flag
B-ORC1	Oracle	Does the oracle implement every rule and invariant?	OR11, OR6–OR9, OR2	gate
B-ORC2	Oracle	Is the oracle free of wrong, missing, or inconsistent behaviour?	OR1–OR5, OR9, OR10, OR12, PB1	gate
B-TST1	Tests	Does the suite exercise every rule and invariant at least once?	TC1, TC3	gate
B-TST2	Tests	Does the suite cover important corner cases and likely wrong solutions?	TC2, TC4–TC6	gate
B-SPC1	Specification	Is the public specification unambiguous for all verdict-relevant behaviour?	SP4–SP6	gate
B-SPC2	Specification	Does the public specification determine the correct output for each tested input?	SP2, SP3, SP7	gate
B-SPC3	Specification	Is the public text free of leaked implementation hints?	SP1	gate
B-FEA1	Whole task	Can the required behaviour be implemented on P4/BMv2/v1model under the task constraints?	PB1, VP1	gate

infeasible target requirements, missing or invalid control-plane entries, public tests that reveal hidden implementation strategy, and vacuous tests that do not activate the intended behavior. Bundles requiring repair are returned to the construction agent with the failing criteria and supporting evidence. The provenance record retains non-blocking flags, repair history, and the final admission verdict.

C Difficulty Calibration Loop

Calibration begins by selecting a baseline LLM for a benchmark release. The baseline is not used as the final evaluation target, it rather provides a stable reference point for estimating task difficulty. For each newly constructed task, P4Bench prompts the baseline model, evaluates its generated P4 and control-plane entries, and records the functional score S_k is computed. This initial measurement establishes the task's difficulty. If S_k lies above the region the corpus needs, the task is under-discriminating and will be hardened; if S_k lies below the desired region, the task is hard or impossible to finish and will be eased

Starting point. A pattern first produces a canonical, difficulty-agnostic instance. P4Bench measures S_k by running the baseline n times, where $n \geq k$. Only then do we decide whether the release needs a sibling near that value or should seek harder or easier coverage.

Direction. A sibling above the desired interval triggers a hardening mutation; one below it triggers an easing mutation. Each mutation forks a new sibling rather than editing the parent, preserving the network-function identity and moving along one declared direction: capacity, statefulness, faithfulness, workload skew, input form, or feature breadth.

Iteration. Mutations escalate through three levels of ascending cost. L1 rebinds parameters within the pattern. L2 grows the hidden test set against the existing audited oracle, sharpening discrimination without changing semantics. L3 extends the mutation-operator library, triggering oracle regeneration and human re-audit when an operator changes rules or invariants. The baseline is rerun and S_k recomputed after each mutation.

Exit point. The loop stops when a sibling adds useful coverage: an under-covered interval, enough distance from its parent, or mutation operator is exhausted. For the later condition, we

try to extract a new task pattern by adding more behavioral requirements or combine different patterns to form a new one and restart the calibration loop.

D Evaluation Prompt Template

P4Bench uses the following template for model evaluation. The placeholders are filled with the task description, topology, and public test scenarios before the prompt is sent to the model.

```
You are a P4-16 network engineer. Write a complete P4 program and
its control-plane configuration for the BMv2 v1model architecture.

## Task
{description}

## Network Topology
{topology}

## Test Scenarios (your solution must handle these)
{test_scenarios}

## Instructions
- Output a complete P4-16 program targeting BMv2 v1model.
- Output a control-plane configuration as JSON with table entries per switch.
- Use human-readable names for tables and actions.
- For ordinary P4 match-action tables, the legacy entry format per switch is:
  [{"table": "...", "match": {...},
    "action_name": "...", "action_params": {...}}]
- If your design needs BMv2 runtime resources that are not ordinary P4 table
  entries, use the extended per-switch object:
  {
    "entries": [...],
    "multicast_groups": [{"mgid": 1, "rid": 0, "ports": [2, 3]}],
    "mirror_sessions": [{"id": 100, "egress_port": 3}],
    "action_profiles": [...]
  }
- Use `multicast_groups` when the P4 sets `standard_metadata.mcast_grp`.
  Do not invent pseudo-tables such as `$pre.mcast_group`.
- Use `mirror_sessions` when the P4 calls `clone(...)` with a session id.
  Do not represent mirror sessions as normal `table_add` entries.

## Response Format

### program.p4
```p4
<your complete P4 program>
```

entries.json
```json
<your control-plane entries per switch:
 {"s1": [...]} or
 {"s1": {"entries": [...],
         "multicast_groups": [...],
```
```

```
        "mirror_sessions": [...]}}>
```

E Reproducibility Settings

This appendix lists the settings needed to reproduce the model evaluation in Section 6.

Table 8. Model interfaces and decoding settings. All parameters not listed, including reasoning or thinking settings and top-p, are left at the provider defaults.

Model	Identifier and interface	Temperature	Max tokens
Opus-4.8	`opus-4.8`; Claude Code CLI (`claude -p`)	CLI default	65,536
Sonnet-4.6	`claude-sonnet-4-6`; Claude Code CLI (`claude -p`)	CLI default	65,536
GPT-5.5	`gpt-5.5`; OpenAI Chat Completions API	default (fixed)	65,536
Gemini-3.1-Pro	`gemini-3.1-pro-preview`; Google GenAI API	0.8	65,536
Qwen-3.7-Max	`qwen3.7-max`; snapshot `qwen3.7-max-2026-06-08`; DashScope OpenAI-compatible endpoint	0.8	65,536
DeepSeek-V4-Pro	`deepseek-v4-pro`; DeepSeek API	0.8	65,536

Models and decoding. Table 8 summarizes the model settings. Opus-4.8 and Sonnet-4.6 are called through the Claude Code CLI in one-shot mode (`claude -p`) with all tools disabled and the effort level set to medium (a few regenerated Sonnet-4.6 draws ran at low effort). Each call is launched from an empty working directory so that no project files enter the model's context. The CLI exposes no temperature control, so both models sample at the CLI defaults. GPT-5.5 runs at its default temperature because the model does not accept other values.

Sampling protocol. Each model produces three independent draws per task. Every draw is a separate, stateless request with no execution feedback, because generation and grading run as separate phases. No random seed is passed to any API; individual outputs are therefore not reproducible, although the sampling configuration is. Requests that fail with a transport or rate-limit error are retried with exponential backoff. Draws that returned no program for a non-content reason, namely Gemini recitation blocks and interrupted runs, were regenerated. The 8 Sonnet-4.6 draws that still produced no program after regeneration are scored zero and reported as `api_timeout` in Section 7.2. Generation ran from 18 June to 10 July 2026; the July draws regenerate Sonnet-4.6 cells interrupted by a provider spend limit. All BMv2 draws were graded on 7–10 July 2026, with each draw evaluated in a fresh process. The draws of the 7 tasks whose evaluator or stored expected outputs were corrected on 10 July 2026 were re-graded on 4 October 2026 from the stored programs and entries, without regeneration.

Prompts. Each task is presented as a single user message with shared system prompt, using the template in Appendix D. The prompt contains the task description, the topology, and the public tests: 447 of the 1,914 tests, each rendered as its natural-language description and expected outcome. Hidden tests, the oracle, the implementation-level description, and the seed bindings are never shown. Notably, the `entries.json` instructions exist in two versions. The template in Appendix D is the revised version, which adds the `multicast_groups`, `mirror_sessions`, and

`action_profiles` fields for BMv2 runtime resources that are not ordinary table entries. Draws generated before 21 June 2026 used an otherwise identical template that described only the per-switch table-entry list. The 13 tasks that need multicast groups or mirror sessions were regenerated for all six models with the revised template, so every draw on those tasks saw the extended format. We extract the P4 program from the fenced `p4` blocks of the response and the control-plane entries from the `entries.json` block. If the entries cannot be parsed, the program is graded with an empty control plane. The Tofino condition uses the same template, retargeted to TNA and bf-p4c with SDE 9.6.0, and restricted to unicast forwarding and drop.

Evaluation environment. BMv2 submissions are compiled with p4c 1.2.4.2 (`--target bmv2 --arch v1model`). Each test then runs in a fresh BMv2 `simple_switch` 1.15.0 process in file-I/O mode, with entries installed through the Thrift CLI. The host runs Ubuntu 22.04 with Python 3.10 and Scapy 2.4.4. Evaluation is fully deterministic: each test's expected output is derived from the audited Python oracle at construction time and stored with the task, and the task-level `seed` field is a parameter binding, not a random seed. Scores weight public tests 0.2 and hidden tests 0.8, and a draw that does not compile scores zero. A draw or task counts as solved for pass@1 or pass@3 when its score is 1.0. For Tofino, programs are compiled with bf-p4c (SDE 9.6.0), and tests run through PTF on the S9180-32X switch with entries installed over BfRuntime. Tests requiring header or multicast support that the Tofino harness lacks are excluded.

Construction and judging. Patterns, seeds, oracles, tests, sanitised descriptions, and mutations were drafted with Claude Code (Opus-4.7, then Opus-4.8) between April and May 2026, at default settings. Every drafted artifact passed deterministic checks and the three admission gates in Appendix B.1: a creator self-check (Gate 1); an LLM judge running in a separate session without the creator's context (Gate 2), using `Opus-4.8`; and human review by the authors (Gate 3). Calibration probes used `gemini-3.1-pro-preview` at temperature 0 with a 65,536-token limit, and supplementary difficulty measurements used five draws of `gpt-5.5` at temperature 1. Compile failures are categorized by a deterministic classifier over compiler diagnostics. Functional failures are categorized by a Sonnet-4.6 judge that gives one verdict per failed test and sees the specification, the submission, and the expected versus observed packets. The mutation analysis reuses the sweep draws and generates no new ones.

F Failure Case Studies

To make the failure taxonomy in §7.2 concrete, we present five representative case studies drawn from `composition` tasks. The cases follow the same competence pipeline used in the main analysis: a submission must first pass the compilation gate, and a compile-passing submission can then fail at the earliest broken functional stage of *comprehension*, *design*, *implementation*, or *configuration*. These examples are not intended to estimate category frequency; the aggregate distributions are reported in §7.2. Instead, they show how each category appears in deployed packet-level evaluation.

The snippets, control-plane entries, scores, and error strings are taken verbatim from saved per-draw artifacts, with only irrelevant surrounding context omitted. All five failures are systematic: each reproduces across draws, and most also recur across models or related task families.

(1) Compilation: `upf_router_disc` *(GTP-U tunnel ∘ IPv4 router).* This case illustrates a target-validity failure in a composed pipeline. The generated program captures the high-level intent—decapsulate GTP-U traffic and route both inner and native IPv4 packets—but the resulting P4 violates two BMv2/v1model constraints.

First, the model applies the same FIB table on multiple control-flow branches:

```
if (!transport_ipv4_lpm.apply().hit) { ... }         // decapsulated path
   ... if (!transport_ipv4_lpm.apply().hit) { ... }  // native path
if (transport_ipv4_lpm.apply().hit) { ... }          // third apply
error: Program can not be implemented on this target since it contains
       a path from table MyIngress.transport_ipv4_lpm back to itself
```

Second, it defines a non-byte-aligned GTP-U header:

```
error: header gtpu_t: Found header with fields totaling 63 bits.
       BMv2 only supports headers totaling a multiple of 8 bits.
```

The root cause is therefore a `target_constraint_violation`. BMv2 accepts neither repeated table application along one logical pass nor headers whose total width is not a multiple of eight bits. The composition matters because the task combines two independently natural operations—tunnel handling and IP routing—inside one target pipeline. A standalone router routes once, and a standalone GTP-U parser can use a byte-aligned header; the composed task gives the model more opportunities to violate target contracts even when the intended network function is understood.

(2) Comprehension: `edge_upf_anchor` *(UPF ∘ router).* This case illustrates a boundary-condition misreading. The program compiles and has the expected UPF-plus-routing structure, but it consistently encodes the wrong TTL-expiry rule:

```
if (hdr.ipv4.ttl <= 1) { drop(); return; }   // generated.p4:272
```

The corresponding packet test expects the boundary packet to be forwarded, but the generated program drops it:

```
UNEXPECTED_DROP :: "Expected forward, no packet received"   // uplink_ttl_one_forwards_zero
```

The root cause is `comprehension`. In this task, the specification requires a packet arriving with TTL = 1 to be forwarded after the field is decremented to 0. The model instead reads the condition as "drop when TTL $\leq$ 1", so it discards exactly the boundary packet. This is not a data-structure choice or a local coding slip: the same `ttl <= 1` condition appears across Opus, GPT-5.5, and Gemini, and across `edge_upf_anchor`, `mobile_core_policer_anchor`, and `secure_router_anchor`. We therefore classify the failure as a misreading of the required behavior.

(3) Design: `secure_router_anchor` *(stateful firewall/NAT ∘ router).* This case illustrates a wrong state abstraction. The task requires reflexive admission: inbound packets should be forwarded only if they return on a flow that was previously established from the internal side. Gemini instead realizes the NAT component as a static, stateless exact-match table:

```
table dnat_table {
  key = { hdr.ipv4.dstAddr:exact; hdr.tcp.dstPort:exact;
          hdr.ipv4.srcAddr:exact; hdr.tcp.srcPort:exact; }
  actions = { dnat; ... }
}
```

The generated program therefore admits an unsolicited inbound packet whenever the static DNAT key matches:

```
UNEXPECTED_FORWARD :: "Expected drop, got packet on port [1]"  // unsolicited_inbound_drop
```

The root cause is `design`. The model chooses a stateless table structure for a stateful requirement. As a result, it can implement address translation, but it cannot enforce the composed firewall/NAT obligation that state is installed only by internal outbound traffic. This is precisely the `default_deny_external` and `state_only_installed_from_internal` property captured by Figure 10. The failure is therefore not that the model wrote the intended design incorrectly; it selected a design that cannot satisfy the intended trace-level semantics.

(4) Implementation: `bfd_frr_anchor` *(BFD liveness ∘ fast reroute).* This case illustrates a broken cross-stage dependency. The generated program has the right overall design: the BFD stage maintains liveness state, and the FRR stage should use that state to select the primary or backup next hop. The BFD component writes the liveness result into a register:

```
register<bit<1>>(512) port_down_reg;   // 0 = up, 1 = down; written by BFD
```

However, the forwarding path never reads this register when choosing the egress port. The failure appears on the reroute test:

```
WRONG_EGRESS_PORT :: "Expected port in [3], got port 2"  // detect_then_reroute_to_backup
```

The root cause is `implementation`. After BFD declares the primary next hop down, the packet should be forwarded through the backup port 3. The generated program still forwards on the primary port 2 because the FRR logic is not connected to `port_down_reg`. The intended design is appropriate and the program compiles, but the data plane does not faithfully realize the cross-component coupling introduced by composition: state written by the liveness component must be read by the forwarding component.

(5) Configuration: `router_acl_anchor` *(ACL ∘ IPv4 router).* This case illustrates a control-plane/data-plane mismatch. The generated data plane is structurally plausible: it applies a five-key ternary ACL before an LPM FIB:

```
table policy {
  key = { hdr.ipv4.srcAddr:ternary; hdr.ipv4.dstAddr:ternary;
          hdr.ipv4.protocol:ternary; meta.l4_srcPort:ternary;
          meta.l4_dstPort:ternary; }
  actions = { policy_permit; policy_deny; }
}
```

However, the generated control-plane entries specify only the fields constrained by each rule, in a P4Runtime-style partial-key format:

```
{"table":"policy","match":{"hdr.ipv4.srcAddr":["10.0.1.5","255.255.255.255"]},
 "action_name":"policy_permit"}                    // 1 of 5 keys
ENTRY_INSTALL_ERROR :: "Table policy needs 5 key fields"
```

No ACL entry is installed, and all 14 packet tests fail.

The root cause is `configuration`. The P4 program and the intended ACL behavior are consistent, but the generated runtime entries do not match the installation contract of `simple_switch_CLI`: every ternary key must be present, with don't-care fields written explicitly as `0&&&0`. By contrast, the model emits entries as if omitted keys were implicit wildcards. Composition sharpens this failure mode because the two tables have different entry contracts: the ACL requires full ternary key arity and priority, whereas the FIB requires LPM prefixes. The failure therefore occurs not in the data-plane logic alone, but in making the generated program and its control-plane state agree under the target runtime.

G Detailed Root Cause of Functional Failures

Table 9 gives the detailed root cause behind the functional-failure discussion, over all six models on BMv2/v1model (130 tasks × 3 draws per model). Counts are failed packet tests on compile-passing draws, each attributed to its primary (earliest-broken) root cause. In total, the table covers 2,010 failed packet tests.

Received 10 July 2026; revised 5 October 2026; accepted 9 October 2026

Table 9. Detailed root cause of the 2,010 collected functional failures.

Failure	Counts	Root Cause
Comprehension — 128		
Behavioral-rule misread	103	wrong condition/precedence; coherent wrong behaviour
Protocol-field semantics	17	wrong field meaning, unit, or flag
Unrecognized requirement	8	requirement never registered
Design — 51		
Infeasible on target	23	needs loops/float/unsupported extern
Stateless for stateful	20	no persistent state chosen
Wrong selection/hash	7	wrong hash or selection scheme
No mechanism for subproblem	1	sub-problem unaddressed
Implementation — 486		
Baked constant	177	CP parameter frozen at compile time
Architecture-idiom misuse	121	clone/checksum-extern/mark_to_drop misuse
Boundary-guard logic	59	off-by-one/threshold/range error
Parser/header layout	43	widths/order mismatch wire format
Over-constrained table key	25	key too narrow; packets miss
Stateful reply construction	24	synthesized reply built wrong
Transformation omitted	18	checksum/TTL edit never emitted
Control-flow ordering	10	apply block in wrong order
State access	5	bad register index/write-back
Deparser serialization	4	wrong emit order or omission
Configuration — 1,345		
Match encoding	603	LPM/ternary/range/byte-order wrong
Missing entry	414	required entries not installed
Name contract	142	names mismatch P4Info
Ternary/range priority	99	missing or wrong priority
Parameter value	49	wrong action argument value
Action arity	30	wrong parameter count
Missing runtime state	7	multicast/mirror/meter never created
Spurious catch-all entry	1	extra entry installed
Total	**2,010**	