Trust, But Verify: An Empirical Evaluation of AI-Generated Code for SDN Controllers

Felipe Avencourt Soares, Muriel F. Franco, Eder J. Scheid, Lisandro Z. Granville

Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, Brazil
Federal University of Health Sciences of Porto Alegre (UFCSPA), Porto Alegre, Brazil
{felipe.avencourtsoares, ejscheid, granville}@inf.ufrgs.br
muriel.franco@ufcspa.edu.br

***Abstract*—Generative Artificial Intelligence (AI) tools have been used to generate human-like content across multiple domains (*e.g.*, sound, image, text, and programming). However, their reliability in terms of correctness and functionality in novel contexts such as programmable networks remains unclear. Hence, this paper presents an empirical evaluation of the source code of a POX controller generated by different AI tools, namely ChatGPT, Copilot, DeepSeek, and BlackBox.ai. To evaluate such a code, three networking tasks of increasing complexity were defined and for each task, zero-shot and few-shot prompting techniques were input to the tools. Next, the output code was tested in emulated network topologies with Mininet and analyzed according to functionality, correctness, and the need for manual fixes. Results show that all evaluated models can produce functional controllers. However, ChatGPT and DeepSeek exhibited higher consistency and code quality, while Copilot and BlackBox.ai required more adjustments.**

***Index Terms*—generative AI, programmable networks, software-defined networking, empirical analysis**

I. Introduction

Generative Artificial Intelligence (AI), that is, a type of AI capable of producing human-like content [1], has been applied in multiple domains, such as text, music, and image generation [2]. However, its applications go beyond these cases. One specific example is the automatic generation of source code in various programming languages (*e.g.*, C, Python, C++, and Java) based on textual descriptions of program objectives, which, when well-defined [3], can produce results that meet the user's specified requirements. Hence, "vibe coding", which is practice of developing software through conversations with Large Language Models (LLM) has become popular [4].

Nevertheless, programs generated by generative AI applications (*e.g.*, ChatGPT [5]) are not always correct and may pose security concerns [4], [6], as they may contain logical errors or performance issues. Therefore, developers cannot blindly trust the output produced by such systems. This problem becomes even more critical when the generated code targets network devices that must ensure security (*e.g.*, firewalls) or maintain performance to meet Quality of Service (QoS) [7] and Service Level Agreement (SLA) policies [8].

Although the popularity and use of such tools are increasing rapidly, research on this topic remains limited due to its novelty. Moreover, as computer networks evolve to become increasingly programmable and open [9], several efforts have already attempted to automate code generation for these environments [10], [11], often without properly considering the correctness or reliability of the produced code. Hence, the behavior and trustworthiness of AI-generated code for programmable networks (*e.g.*, for Software-Defined Networking - SDN) deserve further investigation.

In this context, this paper analyzes different codes generated by generative AI applications (*e.g.*, ChatGPT, CoPilot, DeepSeek, and BlockBox.ai) within the scope of programmable networks (*e.g.*, SDN controllers [12]) to verify whether they achieve the user-defined objectives, are functionally correct, or present security issues. Finally, it empirically compares and discuss the differences and variations between the codes generated by the selected AI tools.

The remainder of this paper is organized as follows. Section II reviews the literature on AI-based code generation for both SDN and general-purpose applications. Section III describes the methodology adopted in this study, including the definition of AI tools, prompts, and network topologies used for testing. Next, Section IV details the experiments conducted with the code generated by the selected tools, while Section V discusses the results across different dimensions. Finally, Section VI concludes the paper and outlines directions for future work.

II. Related Work

A first step in the research involves the analysis of the state-of-the-art to help identify challenges that require further exploration. Table I presents a concise summary relating the two main topics of this work.

The first work [20] explores the capability of a Large Language Model (LLM) to autonomously manage a network from scratch through the proposed LLM-NetCFG framework. Another relevant study [16], which had a significant influence on this research, evaluates the ability of ChatGPT-3.5 to solve computing and networking problems. The authors follow a systematic methodology by stating the problem, presenting the expected solution, describing the prompt provided to the LLM, and analyzing the model's output. However, the paper leaves open the evaluation of other natural language models and notes inaccuracies that have since been corrected due to model evolution.

TABLE I: Comparision of Related Work

Reference	Year	Context	Programmable Network	AI Tool/Technique/Model	Code Generation	Evaluation
[13]	2022	Networking	No	BERT	No	Study
[14]	2023	Computing	No	GPTs, CodeGen, StarCoder	Yes	Experiment
[15]	2023	Computing	No	BERT and GPT-3	No	Experiment
[16]	2023	Computing	SDN	GPT-3.5	Yes	Experiment
[17]	2023	Networking	SDN	NetLM	No	Study
[18]	2023	Networking	No	Bard and GPT-3/4	Yes	Study
[19]	2024	Networking	SDN	Several	No	Discussion
[20]	2024	Networking	No	GPT-4, NetCFG	No	Experiment
[21]	2025	Networking	SDN	LLM	Yes	Experiment
This Work	2025	Networking	SDN	ChatGPT, Copilot, DeepSeek, Blackbox.io	Yes	Experiment

The review in [19] provides a broad overview of AI and LLM applications in networking, referencing multiple studies across various subfields since the rise of these technologies. The work in [17] proposes the implementation of NetLM, a ChatGPT-derived model designed to enable intent-based communication and autonomous network management. The study [15] highlights the role of log analysis for fault prediction and operational support. It employs LLMs such as GPT-3 and BERT, trained on large datasets to improve accuracy and reliability in anomaly detection.

In [13], BERT-based learning is adapted to produce semantic numerical representations (embeddings) that enhance model generalization and network device security. The research in [21] examines network automation in 5G environments using LLMs like BERT, focusing on intent interpretation and transformer-based time-series prediction for validation. The framework described in [18] combines synthesis techniques to automatically generate graph-based code for communication networks. Finally, [14] proposes an automated benchmarking and validation framework for assessing the quality and correctness of LLM-generated code.

It can be concluded from this research that most of the reviewed studies and experiments rely heavily on LLMs, particularly ChatGPT. In general, these models are employed as management or detection tools within their respective applications, with learning derived from large datasets tailored to each context. Among the analyzed works focusing on code analysis generated by LLMs, only [21] is directly related to the programmable networks subcontext and includes code generation. However, it remains limited to the control plane and constrained to a single LLM (*i.e.*, BERT).

III. Methodology

In this section, the AI tols to be analyzed are defined, followed by the definition of the prompts. Next, the network topologies on which these AIs will operate are presented, along with the execution of the respective prompts. Finally, the functionality of the codes generated by the Large Language Models (LLMs) is analyzed.

A. Artificial Intelligence Tools

A preliminary review of AI reveals a wide range of available options. Given this diversity, specific criteria were established to narrow the scope of analysis, aiming to capture different aspects in terms of both popularity and technical approach. Based on these filters, the following AIs were selected for study: ChatGPT [5], Coplit [22], DeepSeek [23], and BlackBox.ai [24].

ChatGPT, developed by OpenAI, is the most popular among those selected and is known for its versatility in solving general-purpose problems. Similar to DeepSeek and BlackBox.ai, it provides a conversational interface in which the user inputs a text description containing the necessary information, referred to here as a *prompt*. The initial interaction stage is crucial, as it guides the solution through keywords. Therefore, a well-crafted prompt results in a more accurate and structured response.

Copilot, developed by Microsoft, also provides a conversational interface similar to the previous models. However, its GitHub Copilot version includes functionality that assists directly within programming environments. This tool analyzes existing code and suggests the subsequent lines to accelerate software development.

DeepSeek, a Chinese company, recently released the DeepSeek-R1 model, which claims to match or even surpass OpenAI's O1 model in some aspects while offering significantly lower cost. Its training approach differs from that of OpenAI models by emphasizing reasoning trajectories learned from prior model behaviors rather than relying primarily on large-scale web data. This distinction is detailed in the study published by the company itself [23].

BlackBox.ai, originally developed by ACG, also features an interactive chat interface. Among the AIs described, it was the first to introduce the functionality of attaching files to task descriptions. Its interface remains minimalist while offering several advanced features to assist developers.

B. Tasks and Prompts

To evaluate the models mentioned above (*i.e.*, ChatGPT, DeepSeek, Copilot, and BlackBox.ai) three tasks of increasing complexity were formulated, each designed to assess the learning and reasoning capabilities of the models. In addition, two prompt-engineering techniques were selected to compose the commands for each task, zero-shot and few-shot. The defined tasks are as follows:

- **Simple:** Regardless of topology, ensure full packet delivery between all hosts.

- **Intermediate:** According to the topology, block communication between hosts as specified in the prompt.
- **Complex:** Develop a layer-three firewall for an SDN.

Initially, the *zero-shot* technique was employed to design the prompts. This approach consists of providing direct and objective instructions to the model, without prior examples or contextual information. The second technique, *few-shot*, incorporates contextual elements and examples to guide the model toward a more precise solution. Hence, reducing ambiguity.

1) Zero-Shot: For the zero-shot technique, the inputs are concise and keyword-oriented to focus the model on the intended outcome. The following prompts were defined:

- **Prompt 1**
  - *"Develop, in Python, a POX controller for a software-defined network."*
- **Prompt 2**
  - *"Develop, in Python, a POX controller for a software-defined network where it blocks pings between even hosts and odd hosts."*
- **Prompt 3**
  - *"Develop, in Python, a layer-three firewall POX controller for a software-defined network."*

2) Few-shot: For the few-shot technique, which uses examples or contextual descriptions to refine the response, the following inputs were defined. These are more complex and detailed than the zero-shot prompts, providing richer contextualization of the problem domain.

- **Prompt 1**
  - *"I created a simple software-defined network in Python using the Mininet library. In this network, I wanted to control the flow of packets between hosts. To test communication within the network, I needed to start a controller. In this context, make a POX controller that manages packet forwarding between hosts. Additionally, ensure the controller learns MAC addresses. Finally, provide a brief explanation of the implementation."*
- **Prompt 2**
  - *"I created a simple software-defined network in Python using the Mininet library. In this network, I wanted to control the flow of packets between hosts. To test communication within the network, I needed to start a controller. In this context, make a POX controller to block pings between even hosts and odd hosts. Additionally, shorten the timeout when the ping block occurs. Finally, provide a brief explanation of the implementation."*
- **Prompt 3**
  - *"I created a simple software-defined network in Python using the Mininet library. In this network, I wanted to control the flow of packets between hosts. To test communication within the network, I needed to start a controller. In this context, make a layer-three firewall POX controller to manage the software-defined network with the following functions:"*
    1) *Initialize control rules;*
    2) *Manage packet flow based on learned MAC addresses;*
    3) *Consider that the following function is implemented:*

```
def add_firewall_rules(src_ip, dst_ip,
    protocol="tcp"):
    rule=f"{src_ip}{dst_ip}{protocol}\n"
    with socket.socket(socket.AF_INET,
    socket.SOCK_STREAM) as s:
        s.connect(("127.0.0.1", 6633))
    # Assuming POX listens on this port
        s.sendall(rule.encode())
    print(f"Rule added: Block {protocol}
    from {src_ip} to {dst_ip}")
```

    4) *Finally, provide a brief explanation of the implementation.*

C. Topologies

To test the effectiveness of the controllers generated by the AIs, four different network topologies were defined, illustrated in Figure 1. These configurations aim to simulate a variety of network scenarios, ranging from simple (*e.g.*, linear), medium (*e.g.*, star), to complex (*e.g.*, , fully connected and two switches).

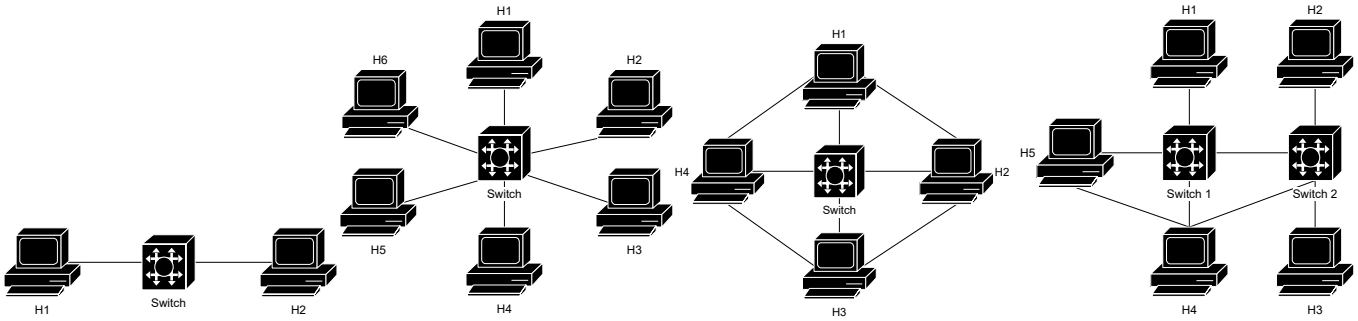
IV. Experiments

For the experiments, the following methodology was adopted. At the beginning of a new interaction with each AI tool, the input prompts defined in Section III-B were provided, corresponding to each prompt-engineering technique and specific task. The generated code was then executed in the POX controller, with network communication simulated using Mininet [25]. If the code executed successfully on the first attempt, it was marked as "*functional without fixes*"; if additional interaction or manual corrections were required, it was marked as "*functional with fixes*"; and if it failed after further attempts, it was classified as "*failure*". All scripts used to generate the network topologies and conduct the tests are publicly available at [26].

A. Task 1 - End-to-end packet delivery among hosts

1) Zero-shot: ChatGPT returned an implementation of the task with an additional feature: learning of MAC addresses, which was not explicitly requested in the prompt. MAC learning associates the source address with the switch port, improving forwarding efficiency. Beyond this optimization, the tool also provided a brief explanation of the generated code and instructions on how to start the controller.

The execution flow is triggered by the packet-in event, which controls packet distribution among network switches. It first performs a packet integrity check, discarding malformed packets and issuing a warning. After validation, the system resolves MAC addresses for proper forwarding, successfully enabling ICMP echo exchanges and meeting the task requirements.



(a) `Linear` (b) `Star` (c) `Fully Connected` (d) `Two Switches`

Fig. 1: Network topologies defined for the tests

Copilot produced a functional but simplified solution. For each packet event, forwarding is performed via broadcast (all ports), which increases network latency relative to ChatGPT's approach. Although the control flow is concise, it lacks packet validation and MAC learning; nonetheless, it solves the task.

DeepSeek implemented an SDN controller based on MAC learning similar to ChatGPT's, but with stronger modularization, separating broadcast, packet sending, and flow-rule installation into dedicated structures.

Like the others, its controller validates packets, learns MAC addresses, and then forwards accordingly. The modular organization improves readability and maintainability while correctly solving the task.

BlackBox.ai also implemented an SDN controller with MAC learning. Its code structure resembles ChatGPT's solution, while its documentation approach is closer to DeepSeek's.

The packet-handling mechanism operates as follows: upon a packet-out event, the system checks whether the destination host has been previously learned; if so, it forwards directly on the learned port; otherwise, it broadcasts on all ports. The implementation is functional for the proposed task.

In summary (Table II), all selected tools successfully produced a POX controller for SDN. ChatGPT stood out by adding management optimizations; Copilot delivered a simpler solution relying on broadcast; DeepSeek excelled in modular design and readability; and BlackBox.ai-after an update released during this study-balanced strengths seen in ChatGPT and DeepSeek. Even under the zero-shot approach, the AIs demonstrated knowledge of software-based programmable networking.

TABLE II: Task 1 - Results with the Zero-Shot Technique

Topology	ChatGPT	Copilot	DeepSeek	BlackBox
Linear	✓	✓	✓	✓
Star	✓	✓	✓	✓
Fully Connected	✓	✓	✓	✓
Two Switches	✓	✓	✓	✓

✓ Functional without fixes; ✓✗ Functional with fixes; ✗ Failure

2) Few-shot: ChatGPT subtly improved the code generated in the previous technique. The forwarding routines were refined for clarity and cleanliness. Given the simplicity of the task, basic software-engineering refactoring sufficed to consolidate SDN problem-solving.

Among the tools, Copilot was the only one that had not implemented MAC learning in the zero-shot setting, leaving more room for improvement. Its new response incorporated MAC learning and clarified the algorithm with comments.

As a second-generation LLM with reasoning traces from other systems, DeepSeek's code is similar to ChatGPT's. It validates incoming packets before forwarding based on learned MACs. A notable difference is that packet sending was encapsulated in a dedicated function, increasing modularity and readability.

BlackBox.ai attempted to improve modularity relative to its previous version, but the resulting implementation reduced algorithmic clarity. The handler begins with a multicast check and, if not multicast, consults the MAC table for forwarding and learning. Mixing encapsulation logic with packet handling hinders comprehension. A cleaner approach would isolate components (e.g., multicast checks, encapsulation, MAC-table lookups) to improve organization and readability.

Overall, the LLMs demonstrated conceptual mastery of programmable networks and their devices. The additional context in the few-shot approach yielded solutions more aligned with sound software-engineering practices. This qualitative improvement is reflected in the quantitative results in Table III.

TABLE III: Task 1 - Results with the Few-Shot Technique

Topology	ChatGPT	Copilot	DeepSeek	BlackBox
Linear	✓	✓	✓	✓
Star	✓	✓	✓	✓
Fully Connected	✓	✓	✓	✓
Two Switches	✓	✓	✓	✓

✓ Functional without fixes; ✓✗ Functional with fixes; ✗ Failure

B. Task 2 - Blocking pings between hosts

Two alternative strategies were considered. One would generate a generic controller (as in Task 1) and then request progressive adaptations to meet Task 2 requirements. However, that path would bias the evaluation by introducing preconditions and artificial context. Therefore, we adopted a protocol

that starts a fresh dialogue context for this task, presenting the problem independently and without prior understanding.

1) Zero-shot: ChatGPT implemented a helper function to check IP parity. On each packet event, it compares the parity of source and destination IP addresses; if an even-host attempts to communicate with an odd-host (or vice versa), the controller enforces an immediate block. The result meets the requirements with coherent and functional logic.

Copilot created a simple, objective handler that validates IP and ICMP packets and then tests whether to block or forward. The flow is clear and aligns with the task.

DeepSeek provided a direct, functional solution, though with room to improve modularity relative to ChatGPT.

BlackBox.ai followed a similar line to Task 1 but without MAC-learning optimization. It added host-type checks and IPv4/ICMP validation. Finally, it validated parity (even/odd) and enforced the source–destination policy.

In the first prompt, a minor wording error ("between even hosts and odd hosts" instead of "between even hosts with odd hosts") led to ambiguous interpretation and incorrect solutions from GPT, Copilot, and BlackBox. Only DeepSeek interpreted it correctly. Such issues are common when translating prompts from Portuguese to English. We recommend careful prompt review to minimize ambiguity. After correction, results improved as summarized in Table IV.

TABLE IV: Task 2 - Results with the Zero-Shot Technique

Topology	ChatGPT	Copilot	DeepSeek	BlackBox
Linear	✓	✓	✓	✓
Star	✓	✓	✓	✓
Fully Connected	✓	✓	✓	✓
Two Switches	✓	✓	✓	✓

✓ Functional without fixes; ✓✗ Functional with fixes; ✗ Failure

2) Few-shot: In this setting, ChatGPT's initial code mismapped IPs in the parity function; a second interaction fixed and improved it. Unlike other tools, it implemented the check as a dedicated `is_even` function, reflecting software-engineering learning. The overall logic mirrored prior approaches: ICMP/IPv4 checks followed by host classification (even/odd) to decide to block or to forward.

Copilot's first answer performed a simple source/destination IP test and blocked when required, but used an incorrect library, causing packet-distribution errors during tests. A second iteration corrected the issue and completed the task.

DeepSeek initially produced a partial solution, requiring a second interaction to fix forwarding. After the fix, the solution was more direct but less structured than before. With the exception of a timeout accelerator initialized early, flow installation and host checks were handled in the packet-event function-undesirable for maintainability.

BlackBox.ai produced a solution similar to its zero-shot version. Improvements included the host parity test and a configurable timeout set at the beginning of the code (akin to constants in other languages).

Although all AIs solved the problem (*cf.* Table V), the few-shot setting revealed greater difficulty in producing well-structured, modular code relative to zero-shot, often requiring more interactions.

TABLE V: Task 2 - Results with the Few-Shot Technique

Topology	ChatGPT	Copilot	DeepSeek	BlackBox
Linear	✓✗	✓✗	✓✗	✓
Star	✓✗	✓✗	✓✗	✓
Fully Connected	✓✗	✓✗	✓✗	✓
Two Switches	✓✗	✓✗	✓✗	✓

✓ Functional without fixes; ✓✗ Functional with fixes; ✗ Failure

C. Task 3 - Layer-three firewall

1) Zero-shot: This stage exhibited wide variability due to the unconstrained zero-shot prompt regarding firewall rules. Each tool explored a different solution strategy.

ChatGPT produced a functional IP-based firewall. For each packet event, it checked source/destination IPs against the rule table to block or forward. Forwarding was initially incomplete, but a single-line fix in the send path resolved the issue and the task was completed.

Copilot produced a very similar L3 firewall, but left an explicit *TODO* regarding ARP handling, unlike ChatGPT. ARP maps IP addresses to MAC addresses and must be considered in practical deployments.

After a second interaction to fix a parameter error, DeepSeek delivered a functional firewall with clear separation of initialization, default rules, rule insertion, and packet handling. Forwarding logic followed the familiar IP-based approach.

BlackBox.ai returned code very similar to previous tasks, indicating limited learning about L3 firewalls. The flow consisted of brief initialization with connections and allowed IPs, followed by per-packet IP checks for forwarding. Even after two interactions, the implementation did not complete successfully.

Overall (*cf.* Table VI), the LLMs produced responses consistent with the requested functionality under the zero-shot approach. Although more interactions were needed to fix syntax or parameter issues, L3 firewalls were structurally implemented.

TABLE VI: Task 3 - Results with the Zero-Shot Technique

Topology	ChatGPT	Copilot	DeepSeek	BlackBox
Linear	✓✗	✓✗	✓✗	✗
Star	✓✗	✓✗	✓✗	✗
Fully Connected	✓✗	✓✗	✓✗	✗
Two Switches	✓✗	✓✗	✓✗	✗

✓ Functional without fixes; ✓✗ Functional with fixes; ✗ Failure

2) Few-shot: While the previous approach assessed flexibility, the few-shot methodology here enables deeper exploration via code fragments. With a specific prompt, we evaluated and integrated previously developed code, as detailed in Section III-B.

ChatGPT correctly integrated prior code into a working firewall, extending the controller to support local rule insertion, block checking, and packet management with MAC learning.

For rule insertion, it checks whether a rule is already active before installing it. For the block checker, it first validates packet structure, discards malformed packets, identifies the protocol, and then consults the active rule table.

Copilot implemented a firewall with more specialized functionality than in zero-shot while meeting the prompt's requirements. Notably, it removed per-packet ARP checks in favor of a generalized rule-list mechanism, yielding cleaner code. Like ChatGPT, it supported dynamic local rule insertion.

DeepSeek proposed a distinct approach: an L3 firewall backed by a TCP server. Benefits include *(i)* real-time rule updates, *(ii)* easier integration with external systems, and *(iii)* higher modularity and scalability. Downsides include higher development complexity and potential security implications. Initialization differs from conventional controllers; other functions (rule definition, insertion, and packet handling) align with methods formalized earlier.

BlackBox.ai implemented an L3 firewall similar to DeepSeek's but added MAC-learning. Unlike the others, it did not support dynamic local rule insertion. After TCP-server initialization, the algorithm focuses on packet processing and forwarding: *(i)* integrity validation for MAC learning and *(ii)* IP-specific filtering according to predefined rules. This effectively controls traffic but limits policy update flexibility.

As expected, the higher complexity of this task led to greater variability among the LLMs. ChatGPT and Copilot adopted a consolidated, local controller architecture favoring implementation simplicity and robustness, whereas DeepSeek and BlackBox prioritized dynamic updates, system integration, modularity, and scalability. All solutions satisfied the specified requirements, as summarized in Table VII.

TABLE VII: Task 3 - Results with Few-Shot Technique

Topology	ChatGPT	Copilot	DeepSeek	BlackBox
Linear	✓	✓	✓	✓
Star	✓	✓	✓	✓
Fully Connected	✓	✓	✓	✓
Two Switches	✓	✓	✓	✓

✓ Functional without fixes; ✓✗ Functional with fixes; ✗ Failure

V. Discussion

In this section, we critically analyze the results of the study. In Section V-A, we revisit prompt construction to compare the methodological differences between the adopted approaches. Next, in Section V-B, we provide a comprehensive comparative analysis of the evaluated tools, highlighting similarities and differences with implementation examples. Finally, in Section V-C, we discuss the current technological landscape, characterized by accelerated advances and continuous evolution in SDN and applied AI.

A. Comparison of Approaches: Zero-shot vs Few-shot

When crafting a first prompt for the task execution, a zero-shot approach can be adopted. As shown in this study, it offers distinct advantages and limitations. This technique, which provides no prior context or training examples, allows greater flexibility and creativity in the responses. However, effective prompt design requires a precise understanding of the problem so that it can be expressed unambiguously and at an adequate level of detail, as in the few-shot approach, thereby aligning the responses with the intended objectives. This perspective is consistent with the reflections of Kevlin Henney, a well-known software engineer and author who has written extensively on programming practice and design clarity. Henney emphasizes that clarity and specificity are central to effective software development [27].

The importance of precise problem formulation for AI is illustrated by Listings 1 and 2, which show zero-shot and few-shot code, respectively, and exemplify concrete improvements driven by software-engineering principles.

```
def _handle_PacketIn(self, event):
    packet = event.parsed
    log.info("Packet in %s", packet)

    msg = of.ofp_flow_mod()
    msg.match = of.ofp_match.from_packet(packet)
    msg.idle_timeout = 10
    msg.hard_timeout = 30
    msg.actions.append(of.ofp_action_output(port=of.
    OFPP_FLOOD))
    msg.data = event.ofp
    self.connection.send(msg)
```

Listing 1: Copilot-generated code for Task 1 , Zero-shot

```
def _handle_PacketIn(self, event):
    packet = event.parsed
    in_port = event.port

    # Learn the source MAC address
    self.mac_to_port[packet.src] = in_port

    # Check if we know the destination MAC address
    if packet.dst in self.mac_to_port:
        out_port = self.mac_to_port[packet.dst]
        log.debug("Sending packet to %s out port %s"
    , packet.dst, out_port)
        msg = of.ofp_packet_out()
        msg.data = event.ofp
        action = of.ofp_action_output(port=out_port)
        msg.actions.append(action)
        self.connection.send(msg)
    else:
        # Flood the packet if the destination MAC is
     unknown
        log.debug("Flooding packet to %s", packet.
    dst)
        msg = of.ofp_packet_out()
        msg.data = event.ofp
        action = of.ofp_action_output(port=of.
    OFPP_FLOOD)
        msg.actions.append(action)
        self.connection.send(msg)
```

Listing 2: Copilot-generated code for Task 1 - Few-shot

Although both approaches yield functional solutions, the few-shot technique stands out by producing implementations

with greater conceptual refinement, optimized performance, and improved readability. Performance benefits arise from MAC-based forwarding (line 6 of Listing 2), where the switch learns and stores the output port per host.

Regarding readability, and following Fowler's refactoring principles [28], code clarity outweighs mere functionality. As observed empirically, algorithms developed with sound software-engineering practices offer significant advantages over ad-hoc implementations, particularly in maintainability and scalability.

B. Comparison Across AI Tools

ChatGPT emerged as the most mature and widely used tool among those evaluated. Its responses exhibited consistent technical knowledge of software-controlled programmable networks. In Task 1 under the zero-shot approach, the model proposed an optimization not explicitly requested, indicating contextual inference and generalization. Under few-shot, we observed enhanced responses, incorporating software-engineering best practices and progressive optimizations across consecutive tasks.

Copilot, in turn, required more detailed and contextual prompts to reach the level of learning seen in ChatGPT,especially in software-engineering aspects such as modularity, algorithmic efficiency, and low coupling,relative to ChatGPT and DeepSeek. This limitation reinforces its primary role as a coding assistant within IDEs. Its mechanism relies on real-time static analysis and incremental suggestions for refactoring, bug fixing, and local optimization rather than comprehensive architectural solutions.

DeepSeek-R1, a state-of-the-art language model, outperformed initial expectations, showing consistency and robustness across tasks. Similar to ChatGPT-4, it was able to apply implicit prompt cues to automatically optimize implementations. In few-shot settings, it was effective at assimilating advanced software-engineering concepts, particularly modularization, component cohesion, and algorithmic structure optimization, demonstrating adaptability under limited supervised guidance.

BlackBox.ai, the last tool in this study, initially showed a significant knowledge gap regarding software-based programmable networks, failing to solve many tasks under both approaches. By the end of the study, an important update to the model,motivated by the SDN/AI interplay,improved its outputs for the same prompts, yielding better task completion.

C. Practical Aspects and Additional Remarks

LLM-based tools for programmable networks, such as ChatGPT and DeepSeek, bring significant opportunities and challenges to software engineering. These systems evolve rapidly, with frequent updates that can change performance and functionality, requiring continuous adaptation by users. Differences between free and paid tiers, processing capacity, access to advanced models, and longer context windows, can directly impact practical projects. While these tools excel at rapid prototyping and generating modular code, limitations such as response inconsistency, lack of persistent context, and occasional errors reinforce the need for human oversight.

Although few-shot helps reduce inconsistencies through richer contextual descriptions, ambiguity and prompt-induced errors persist. Minor changes in conceptual structure,or simple wording inaccuracies,can lead to interpretive drift, as observed in Task 2. Continuous verification is therefore essential, from prompt design to maintenance of generated code. Even with a well-specified problem, creativity remains key, an idea often echoed in discussions of software craftsmanship [29]. Human intervention is thus indispensable for validating and refining AI-produced code, especially in fast-evolving domains such as programmable networks.

Our results show that a network controller can range from basic packet-forwarding rules to more complex systems with advanced features, such as perimeter security management illustrated by the firewall task. Even when correctness is achieved, multiple implementation paths exist, as evidenced by ChatGPT and DeepSeek.

Even when LLMs fully or partially grasp the task, human involvement weighs heavily in the final outcome. Accurate, persistent interpretation from problem conception onward is crucial. Solid initial project scoping,fundamental to effective solutions,helps surface challenges early and enables more optimized, robust final implementations. As seen in Task 1, AIs tend to propose simpler, more direct solutions when not contextualized; small adjustments (*e.g.*, enabling MAC learning) can deliver substantial performance improvements.

VI. Summary, Conclusion, and Future Work

This study analyzed source code generated by different Artificial Intelligence (AI) models in the context of programmable networks. To this end, three prompts were designed, each representing a different level of complexity (simple, intermediate, and complex), and two prompting strategies were employed: *(i)* zero-shot and *(ii)* few-shot. The objective was to assess the effectiveness and problem-solving capabilities of each language model. For each task, a new chat interaction was initiated following the protocol of the selected prompting technique. From the experimental results, it was observed that the few-shot approach produced more consistent and robust answers, making it a more suitable choice for large-scale projects.

Overall, two main reasoning patterns emerged among the models: one more consolidated and robust, represented by ChatGPT, and another more innovative and energy-efficient, represented by DeepSeek. Since this research addresses two rapidly evolving areas, programmable networks and AI, it opens multiple avenues for future work. Potential directions include extending the study to P4-based code generation, performing similar analyses with other AI systems (*e.g.*, Gemini and custom LLM models), and replicating the experiments in the near future to capture model evolution. Given the fast-paced development of AI technologies, repeating this study even a few months after publication would likely yield distinct and updated results.

REFERENCES

[1] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "IntelliCode Compose: Code Generation Using Transformer," in *28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020, Virtual Event, USA, 2020, pp. 1433—-1443.

[2] S. S. Sengar, A. B. Hasan, S. Kumar, and F. Carroll, "Generative Artificial Intelligence: a Systematic Review and Applications," *Multimedia Tools and Applications*, vol. 84, no. 21, pp. 23 661–23 700, 2025.

[3] C. Liu, X. Bao, H. Zhang, N. Zhang, H. Hu, X. Zhang, and M. Yan, "Improving ChatGPT Prompt for Code Generation," 2023.

[4] J. Mitchell and Y. Shaaban, "Position: Vibe Coding Needs Vibe Reasoning: Improving Vibe Coding with Formal Verification," in *1st ACM SIGPLAN International Workshop on Language Models and Programming Languages (LMPL 2025)*, Singapore, Singapore, 2025, pp. 84–90.

[5] OpenAI, "ChatGPT," 2024, https://chat.openai.com/.

[6] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation," 2023.

[7] S. Khan, F. K. Hussain, and O. K. Hussain, "Guaranteeing End-to-End QoS Provisioning in SOA based SDN Architecture: A Survey and Open Issues," *Future Generation Computer Systems*, vol. 119, pp. 176–187, 2021.

[8] E. J. Scheid, B. Rodrigues, L. Z. Granville, and B. Stiller, "Enabling Dynamic SLA Compensation Using Blockchain-based Smart Contracts," in *IFIP/IEEE Symposium on Integrated Network and Service Management (IM 2019)*, Washington D.C., United States of America, 4 2019, pp. 53–61.

[9] N. Anerousis, P. Chemouil, A. A. Lazar, N. Mihai, and S. B. Weinstein, "The Origin and Evolution of Open Programmable Networks and SDN," *IEEE Communications Surveys & Tutorials*, vol. 23, no. 3, pp. 1956–1971, 2021.

[10] C. Wu, X. Wang, Y. Hu, S. Han, W. Meng, and D. Niyato, "Towards Intelligent SAGIN: Leveraging Big AI Models and SDN for End-to-End Automation," *IEEE Network*, pp. 1–1, 2025.

[11] H. Kukkalli, A. Dandekar, and T. Bauschert, "Prompt Engineering Based Generative AI as a Service (GAIaaS) for Intent-Based Networking," in *2025 IEEE Network Operations and Management Symposium (NOMS 2025)*, Honolulu, HI, USA, May 2025, pp. 1–7.

[12] F. A. Lopes, M. Santos, R. Fidalgo, and S. Fernandes, "A Software Engineering Perspective on SDN Programmability," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 2, pp. 1255–1272, 2016.

[13] F. Le, D. Wertheimer, S. Calo, and E. Nahum, "NorBERT: NetwOrk Representations Through BERT for Network Analysis & Management," in *International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2022)*, Nice, France, 10 2022, pp. 25–32.

[14] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation," in *International Conference on Neural Information Processing Systems (NIPS 2023)*, New Orleans, USA, 12 2023, pp. 21 558 – 21 572.

[15] P. Gupta, H. Kumar, D. Kar, K. Bhukar, P. Aggarwal, and P. Mohapatra, "Learning Representations on Logs for AIOps," in *IEEE International Conference on Cloud Computing (CLOUD 2023)*, Chicago, USA, 07 2023, pp. 155–166.

[16] J. A. Hernandez, J. Conde, P. Reviriego, and G. M. R. de Arcaute, "Is ChatGPT Capable of Solving Classical Communications and Networking Problems?" *TechRvix*, pp. 1–13, 07 2023.

[17] J. Wang, L. Zhang, Y. Yang, Z. Zhuang, Q. Qi, H. Sun, L. Lu, J. Feng, and J. Liao, "Network Meets ChatGPT: Intent Autonomous Management, Control and Operation," *Journal of Communications and Information Networks*, vol. 8, no. 3, pp. 239–255, 09 2023.

[18] S. K. Mani, Y. Zhou, K. Hsieh, S. Segarra, T. Eberl, E. Azulai, I. Frizler, R. Chandra, and S. Kandula, "Enhancing Network Management Using Code Generated by Large Language Models," in *ACM Workshop on Hot Topics in Networks (HotNets 2023)*, Cambridge, USA, 11 2023, pp. 196–204.

[19] C.-N. Hang, P.-D. Yu, R. Morabito, and C.-W. Tan, "Large Language Models Meet Next-Generation Networking Technologies: A Review," *Future Internet*, vol. 16, no. 10, 07 2024.

[20] O. G. Lira, O. M. Caicedo, and N. L. S. d. Fonseca, "Large Language Models for Zero Touch Network Configuration Management," *IEEE Communications Magazine*, pp. 1–8, 2024.

[21] M. A. Habib, P. E. I. Rivera, Y. Ozcan, M. Elsayed, M. Bavand, R. Gaigalas, and M. Erol-Kantarci, "LLM-Based Intent Processing and Network Optimization Using Attention-Based Hierarchical Reinforcement Learning," in *IEEE Wireless Communications and Networking Conference (WCNC 2025)*, Milan, Italy, 03 2025.

[22] T. Dohmke, "GitHub Copilot is Generally Available to All Developers," 2022, available at https://github.blog/news-insights/product-news/github-copilot-is-generally-available-to-all-developers/.

[23] DeepSeek, "DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning," in *arXiv*, Jan. 2025, pp. 1–22.

[24] Blackbox.ai, "Blacbox.ai," 2025, available at https://www.blackbox.ai/.

[25] R. L. S. de Oliveira, C. M. Schweitzer, A. A. Shinoda, and L. R. Prete, "Using Mininet for Emulation and Prototyping Software-Defined Networks," in *IEEE Colombian Conference on Communications and Computing (COLCOM 2014)*, Bogota, Colombia, 2014, pp. 1–6.

[26] F. A. Soares, M. F. Franco, E. J. Scheid, and L. Z. Granville, "AI-based SDN Code Generation," 2025, available at https://github.com/ederjohn/sdn-code-generation-ai.

[27] K. Henney, *97 things every programmer should know: collective wisdom from the experts*. O'Reilly Media, Inc., 2010.

[28] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2018.

[29] W. Groeneveld, L. Luyten, J. Vennekens, and K. Aerts, "Exploring the Role of Creativity in Software Engineering," in *IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS 2021)*, Madrid, Spain, May 2021, pp. 1–9.

All links were visited in October 2025