



Degree Project in the Field Information and Communication Technology and the Main
Field Machine Learning
Second cycle, 30.0 credits

Automating Data Plane Configuration: Leveraging LLMs for P4 Code Generation

ERIK HALME

Automating Data Plane Configuration: Leveraging LLMs for P4 Code Generation

ERIK HALME

Degree Programme in Information and Communication Technology and Machine Learning

Date: November 12, 2025

Supervisors: Changjie Wang, Mandar Joshi

Examiner: Marco Chiesa

School of Electrical Engineering and Computer Science

Host company: Saab AB

Swedish title: Automatiserad konfiguration av dataplanet: Användning av LLM:er för P4-kodgenerering

Abstract

The increasing complexity of network infrastructures, driven by Software Defined Networking (SDN), has made data plane programming with languages like P4 a challenging and error-prone task. This thesis investigates the potential of Large Language Models (LLMs) to automate and assist in P4 code generation. Using an experimental methodology, this study evaluates the performance of both open-weight (Phi-4, M-32B) and closed-weight (o4-mini) LLMs on a benchmark of P4 programming exercises. The thesis compares multiple approaches, including various prompt engineering strategies and Parameter-Efficient Fine-Tuning (PEFT) on a custom P4 dataset.

The results demonstrate that off-the-shelf open-weight models fail to generate any compilable P4 code, highlighting the need for specialization. Fine-tuning proved highly effective, with the fine-tuned M-32B model achieving a 64.3% compilation success rate. The smaller fine-tuned Phi-4 model showed higher relative functional correctness, which suggests a trade-off between model size and understanding of syntax and semantics. For the state-of-the-art model, o4-mini, an iterative feedback strategy that mimics a human developer's workflow was the most successful prompting technique, increasing compilation success to 57.1%.

This study concludes that while LLMs show significant promise as a tool for P4 development, fine-tuning is important for handling niche Domain-Specific Language (DSL). The findings indicate that both specialized models and advanced prompting techniques can significantly enhance developer productivity. However, human oversight remains essential for ensuring code correctness and security.

Keywords

Artificial Intelligence, Machine Learning, Large Language Models, Code Generation, Software Defined Networking, P4

Sammanfattning

Den ökande komplexiteten i nätverksinfrastrukturer, som drivs av Mjukvarudefinierade Nätverk (Software Defined Networking, SDN), har gjort dataplanprogrammering med språk som P4 till en utmanande och felbenägen uppgift. Denna uppsats undersöker potentialen hos stora språkmodeller (Large Language Models, LLMs) för att automatisera och assistera vid generering av P4-kod. Genom en experimentell metodik utvärderar studien prestandan hos både öppet tillgängliga modeller (Phi-4, M-32B) och slutna modeller (o4-mini) på ett benchmark bestående av programmeringsuppgifter i P4.

Uppsatsen jämför flera tekniker, inklusive olika strategier för prompt engineering och parameter-effektiv finjustering (Parameter-Efficient Fine-Tuning, PEFT) på ett specialanpassat P4-dataset.

Resultaten visar att färdiga modeller med öppen viktning inte lyckas generera någon P4-kod som kan kompileras, vilket understryker behovet av specialisering. Finjustering visade sig vara mycket effektiv, där den finjusterade M-32B-modellen nådde en kompilationsframgång på 64,3%. Den mindre finjusterade Phi-4-modellen visade högre relativ funktionell korrekthet, vilket antyder en avvägning mellan modellstorlek och förståelse av syntax och semantik. För den avancerade modellen o4-mini var en iterativ återkopplingsstrategi, som efterliknar en mänsklig utvecklarens arbetsflöde, den mest framgångsrika promptningstekniken, och ökade kompilationsframgången till 57,1%.

Studien drar slutsatsen att även om LLM:er visar stor potential som verktyg för P4-utveckling, är finjustering avgörande för att hantera nischade domänspecifika språk (Domain-Specific Languages, DSLs). Resultaten visar att både specialiserade modeller och avancerade promptningstekniker avsevärt kan förbättra utvecklarens produktivitet, även om mänsklig granskning fortfarande är nödvändig för att säkerställa kodens korrekthet och säkerhet.

Nyckelord

Artificiell Intelligens, Maskininlärning, Stora Språkmodeller, Kod-generering, Programvarudefinierade Nätverk, P4

Acknowledgments

First and foremost, I would like to express my gratitude to my academic supervisor, Changjie Wang, for his invaluable guidance, consistent support, and insightful feedback throughout this thesis journey. I am also very thankful to my examiner, Marco Chiesa, for his time and careful review of my work and openness to good discussion along the way.

My sincere thanks also go to my supervisor, Mandar Joshi, at Saab. His practical advice and expertise were essential to the success of my research. I am grateful to Saab for providing the resources and facilitating a welcoming environment to conduct my research. The opportunity to work within their offices and collaborate with their team offered a great environment to share and discuss ideas.

Contents

List of Figures	ix
List of Tables	xi
List of Listings	xiii
1 Introduction	1
1.1 Background	1
1.2 Problem	2
1.3 Purpose	2
1.3.1 Benefits, Ethics and Sustainability	2
1.4 Goals	3
1.5 Research methodology	3
1.6 Delimitations	4
1.7 Structure of the thesis	4
2 Background	7
2.1 Software-Defined Networking	7
2.1.1 The P4 Programming Language	7
2.2 Mininet	9
2.2.1 P4-Utills & P4-Learning	9
2.3 Large Language Models	10
2.3.1 Prompt Engineering	12
2.3.2 Fine-Tuning	13
2.3.3 Low-Rank Adaptation	14
2.3.4 State-of-the-art LLMs	15
2.4 Related Work	16
3 Method	19
3.1 Research Philosophy and Methodology	19
3.2 Literature review	20
3.3 Evaluation Data	20
3.4 Training Data	22
3.5 Model Selection	22

3.6	Model Fine-Tuning	23
3.7	Testing and Model Inference	24
3.7.1	Testing Methodology	24
3.7.2	Model Evaluation	25
3.8	Prompting Techniques	26
3.9	Inference Settings	29
4	Implementation	31
4.1	Testing pipeline	31
4.1.1	Compilation tests	32
4.1.2	Output tests	32
4.1.3	Model Prompting	33
4.2	Training Procedure	33
4.2.1	Leonardo	33
4.2.2	Unslloth & TRL	34
4.3	Model Inference	34
5	Result & Analysis	35
5.1	Training	35
5.2	Dataset Analysis	36
5.3	Compilation Success Rate	38
5.4	Functional Correctness	38
6	Discussion	41
6.1	Effectiveness of Fine-Tuning	41
6.2	Effectiveness of Prompting	42
6.3	Limitations	42
6.3.1	Dataset Validity and Scope	43
6.3.2	Pre-training of base models	43
6.3.3	Methodological Choices	43
6.3.4	Evaluation Metrics and Process	43
6.4	Ethical Implications	44
6.5	Sustainability Considerations	44
7	Conclusions & Future Work	45
7.1	Conclusions	45
7.2	Future Work	46
	Bibliography	49
	A List of Insights	55

List of Figures

2.1	SDN architecture. Image from [1].	8
2.2	The Transformer architecture as described in <i>Attention is all you need</i> [2]. Image from [3].	11
3.1	Character counts per exercise problem description, formatted as Markdown, used in the evaluation dataset.	28
5.1	Training loss, smoothed training loss using exponential moving average ($\alpha = 0.95$) and evaluation loss for Phi-4 (a) and M-32B (b) over 1 and 2 epochs respectively.	36
5.2	Histogram showing the distribution of sample lengths in the training dataset. The 200 largest samples are omitted from the histogram because they are way larger than the average sample character length.	37
5.3	Character counts per summarized exercise problem description, formatted as Markdown, used in the evaluation dataset.	37
5.4	Bar plot of compilation success rate for Phi-4-p4, M-32B-p4 and o4-mini and prompting strategies. The metric used is pass@5 for all prompting strategies except 5-iter, which was run only once with no independent attempts.	39
5.5	Bar plot of absolute and relative output success rate for Phi-4-p4, M-32B-p4 and o4-mini and prompting strategies. The metric used is pass@5 for all prompting strategies except 5-iter, which was run only once with no independent attempts.	40

List of Tables

3.1	List of all P4 exercises and associated P4 files from NSG’s P4-Learning repository, of which those in bold were used in the evaluation dataset.	21
3.2	Comparison of key training arguments for QLoRA fine-tuning of Phi-4 and M-32B.	24
3.3	Comparison of key inference parameters for Phi-4/-p4, M-32B/-p4 and o4-mini. Top-k for o4-mini cannot be specified in their API and is not reported in OpenAI’s documentation.	30
5.1	Compilation success for all considered models and prompting strategies. The metric used is pass@5 for all prompting strategies except 5-iter, which was run only once with no independent attempts. Cells with N/A indicate a combination that was not tested.	38
5.2	Output success rate for all considered models and prompting strategies. The metric used is pass@5 for all prompting strategies except 5-iter, which was run only once with no independent attempts. Cells with N/A indicate a combination that was not tested.	39
5.3	Relative output success rate (i.e. number of P4 files with correct output / number of successfully compiled P4 files) for all considered models and prompting strategies. The metric used is pass@5 for all prompting strategies except 5-iter, which was run only once with no independent attempts. Cells with N/A indicate a combination that was not tested.	40

List of Listings

- 3.1 System prompt for code generation. 26
- 3.2 Prompt for generating insights 27
- 3.3 System prompt used when summarizing problem descriptions. 28

- 4.1 Command for compiling a P4 file. 32

- A.1 List of insights generated by GPT-4o. 56

List of acronyms and abbreviations

AI	Artificial Intelligence. 2, 3, 10, 44
API	Application Programming Interface. 17, 29, 33, 34
ASIC	Application-Specific Integrated Circuit. 8
BMv2	Behavioral Model version 2. 9, 32
CoT	Chain-of-Thought. 12, 13, 15
CPU	Central Processing Unit. 33
DPO	Direct Preference Optimization. 13, 22
DSL	Domain-Specific Language. i, 1–3, 7, 17, 19, 20, 41, 45–47
EMA	Exponential Moving Average. 35
ETHZ	Eidgenössische Technische Hochschule Zürich. 9
FPGA	Field-Programmable Gate Array. 8
GB	Gigabyte. 33
GPU	Graphical Processing Unit. 22, 33, 34
GRPO	Group Relative Policy Optimization. 13
HLDPPL	High-Level Data Plane Programming Language. 16
LLM	Large Language Model. i, 1–4, 7, 10, 12–17, 19, 20, 26, 27, 29, 31, 33, 41, 43–47
LoRA	Low-Rank Adaptation. 14, 23, 43, 46
LSTM	Long Short-Term Memory. 10
ML	Machine Learning. 1, 10, 46
MoE	Mixture-of-Experts. 16

NF4	4-bit NormalFloat. 14
NSG	Networked Systems Group. 9, 10, 20, 24, 31
OVS	Open vSwitch. 9
P4	Programming Protocol-Independent Packet Processors. i, 1–4, 7–10, 16, 17, 19, 20, 22–27, 29, 31–33, 38, 41–43
PEFT	Parameter-Efficient Fine-Tuning. i, 14, 43, 46
PPO	Proximal Policy Optimization. 13
QLoRA	Quantized Low-Rank Adaptation. 14, 23, 35, 41, 43–46
RAM	Random-Access Memory. 33
RFC	Request For Comments. 22
RL	Reinforcement Learning. 13
RLHF	Reinforcement Learning from Human Feedback. 10, 13, 42, 46
RNN	Recurrent Neural Network. 10
RoPE	Rotary Position Embedding. 10, 34
SDN	Software Defined Networking. i, 1, 2, 7, 9, 16, 20
SEMLA	Securing Enterprises via Machine Learning-based Automation. 34
SFT	Supervised Fine-tuning. 13, 22, 46
TRL	Transformer Reinforcement Learning. 33, 34
VM	Virtual Machine. 10, 24
VRAM	Video Random-Access Memory. 33, 34

Chapter 1

Introduction

This chapter serves as an introduction to the thesis project, first presenting a short background in Section 1.1 followed by a problem description in Section 1.2 and the purpose and goals of the project in Sections 1.3 and 1.4 respectively. A brief overview of the research methodology is given in Section 1.5 and the delimitations of the project are presented in section 1.6. Finally, the structure of the thesis is given in Section 1.7.

1.1 Background

The complexity of modern network infrastructures has increased significantly with the adoption of Software Defined Networking (SDN) and programmable data planes. These technologies enable dynamic reconfiguration of networks to meet operational requirements [4]. However, manual network configuration is a tedious and time consuming process which introduces the risk of errors, inefficiencies, and security vulnerabilities, especially in environments with strict connectivity rules, firewall policies, and redundancy requirements. Programming Protocol-Independent Packet Processors (P4) is the most widely used Domain-Specific Language (DSL) for implementing data plane programs [5]. Given its structured and rule-based nature, it presents a compelling use case for automation using Machine Learning (ML). LLMs have demonstrated strong capabilities in code generation and semantic understanding for popular programming languages, making them a potential tool for generating, optimizing, and validating P4 programs.

This project explores how LLMs can automate network configuration generation on a data plane level in P4 while ensuring compliance with complex rules and requirements. Techniques for improving baseline performance is tested, such as fine-tuning as well as prompt engineering by problem reformulation, error iteration and insight extraction. The project builds upon advancements in ML, code generation, natural language processing and network automation, and has practical implications for network administrators, researchers, and companies like Saab, managing complex and potentially large-scale networks.

1.2 Problem

While SDN has significantly improved the efficiency of network configuration, programming network devices remains a complex, time-consuming, and error-prone task, especially when strict security requirements must be met [6]. Writing correct and optimized P4 code requires deep expertise in networking protocols, hardware constraints, and security policies, making the development process challenging even for experienced engineers.

Recent advancements in machine learning, particularly in using LLMs for code generation, presents a promising opportunity to streamline and potentially automate aspects of P4 development. However, it remains unclear how well LLMs can handle the domain-specific challenges of P4 programming, such as protocol parsing, table matching, and packet processing. This research aims to evaluate if LLMs can effectively assist in writing P4 code, reducing development effort while maintaining correctness.

1.3 Purpose

In a broad sense this thesis aims to provide insights in how LLMs can be used to generate code in niche DSLs, of which the models has seen relatively few examples. Specifically, the thesis evaluates the ability of LLMs to generate syntactically correct and semantically meaningful P4 programs, improve developer productivity, and reduce the complexity of writing and debugging P4 code. This is done in order to assess their potential for automating or partially automating the P4 development process.

This thesis contributes to the broader field of Artificial Intelligence (AI)-assisted software development, providing insights into the applicability of LLMs in network programming and their potential to bridge the knowledge gap for developers who may not be deeply familiar with P4. The findings could have implications for both industry and academia, influencing the adoption of AI-driven tools for network programming and the future development of AI-assisted programming environments.

1.3.1 Benefits, Ethics and Sustainability

The integration of LLMs into P4 development presents several benefits, ethical considerations, and sustainability implications. By leveraging LLMs for P4 code generation, developers can significantly improve productivity, reduce onboarding time for new programmers, and enhance accessibility to network programming

which could prove to be beneficial for Saab. As P4 is a DSL requiring specialized expertise, LLM-assisted development can lower the barrier to entry, enabling a broader range of developers to contribute to network programming tasks. Furthermore, by automating repetitive or complex coding tasks, LLMs can help reduce human error and enhance code quality, ultimately leading to more efficient and reliable network implementations.

However, the adoption of AI-generated code in critical network infrastructure raises ethical concerns. Issues related to code correctness and security vulnerabilities must be addressed. An over-reliance on LLMs without proper validation mechanisms may lead to security risks, as automatically generated code could contain undetected flaws or inefficiencies. This is an especially important consideration for Saab given the sensitive nature of their work. Transparency and human oversight are important in ensuring that AI-assisted development aligns with best practices and industry standards. From a sustainability perspective, AI-driven code generation at a larger scale brings with it significant energy consumption in training and inference [7] and must be considered. Balancing the benefits of AI-assisted coding with its environmental impact requires exploring strategies such as fine-tuning smaller models, improving inference efficiency, and utilizing energy-efficient hardware.

1.4 Goals

The goal of this thesis is to provide an overview of possibilities and challenges of using LLMs to generate functional and correct P4 code with limited human interception. More specifically the aim is to compare different models and approaches, such as fine-tuning and prompt engineering, in order to provide insights into the best approach for automatically generating P4 code. To this end, a few sub-goals have been identified:

1. Produce different prompting strategies.
2. Fine-tune LLMs on a P4 code dataset.
3. Implement a pipeline for comparing the performance of the fine-tuned LLMs and prompting strategies against a baseline based on a small evaluation dataset.

1.5 Research methodology

The formulation of the research methodology in this thesis is inspired by the portal of research methods and methodologies by Håkansson [8]. The methodology used

in this thesis is one of quantitative and positivist nature. Although there is an element of interpretivism in coding, this is mainly tied to the readability of code rather than the performance of it. As this thesis only aims to study the syntax from the compiler's point of view of correctness, readability of the code will not be evaluated. The method used is experimental and applied research, well suited for this thesis because it involves comparing different approaches of LLM learning as well as developing a model or framework that produces more functional code than existing approaches can.

1.6 Delimitations

This project will strictly focus on dataplane generation, specifically generating P4 code. Therefore, the control plane and any other development associated with network configuration will be provided to the models beforehand. Furthermore, given the high security standards at Saab, no data, hardware or software related to real-world projects, network configurations or examples will be incorporated in the prompting, testing or training of the LLMs considered in this thesis.

1.7 Structure of the thesis

Chapter 1 provides a background to the research, defines the problem, and outlines the purpose, goals, and delimitations of the thesis.

Chapter 2 covers the theoretical background, including Software-Defined Networking (SDN), the P4 programming language, Large Language Models (LLMs), and related concepts such as prompt engineering and fine-tuning. It also reviews related work in the field.

Chapter 3 details the research methodology, including the literature review process, the data used for evaluation and training, model selection criteria, and the specific techniques used for model fine-tuning and testing.

Chapter 4 describes the practical setup and implementation of the project, including the testing pipeline, the training procedure on the Leonardo supercomputer, and the model inference setup.

Chapter 5 presents the results from the experiments, including training outcomes, dataset analysis, and the results for compilation success and functional correctness of the generated P4 code.

Chapter 6 interprets the results, discussing the effectiveness of fine-tuning and various prompting strategies. It also addresses the limitations of the thesis, and the ethical and sustainability implications of the research.

Chapter 7 summarizes the main conclusions of the thesis and proposes directions for future research based on the findings and limitations of this work.

Chapter 2

Background

This chapter introduces the background concepts for the thesis. Section 2.1 and 2.2 cover networking technologies like SDN, the P4 language and emulation tools like Mininet. Section 2.3 the fundamentals of LLMs, including fine-tuning techniques and state-of-the-art models. It concludes by reviewing related academic work on generating P4 code in Section 2.4.

2.1 Software-Defined Networking

SDN is a paradigm that separates the control plane from the data plane in networking devices, enabling centralized control and programmability [4]. Traditional network architectures tightly couple these planes, typically bundling them inside each network device (e.g. switches and routers) on dedicated hardware using functions fixed at design time, which leads to limited flexibility [9]. This is undesirable for companies like Saab where total control of the flow of packets within the network is necessary to maintain security, while reducing development time and costs. In contrast, SDN introduces a logically centralized controller that manages the control plane for multiple devices through software-based policies. Network devices become simple forwarding devices which rely on the controller to make decisions. The northbound interface connects the SDN controller and applications, allowing higher-level services to define network behavior dynamically. The controller dictates network behavior and makes forwarding decisions through the southbound interface, which serves as the communication channel between the SDN controller and network devices [4, 9]. The SDN architecture is shown in Figure 2.1. The data plane, also known as the forwarding plane, is responsible for forwarding packets based on control plane decisions [9].

2.1.1 The P4 Programming Language

P4 is a DSL designed for defining how network devices process packets. P4 enables developers to specify and modify packet-processing behavior dynamically, which is

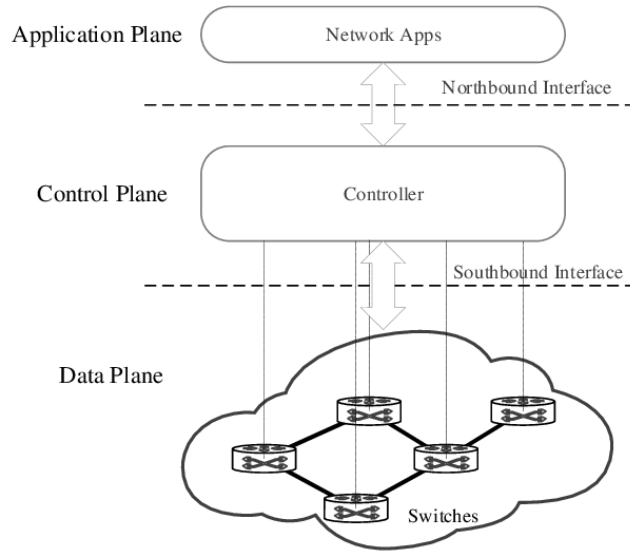


Figure 2.1: SDN architecture. Image from [1].

particularly useful for applications requiring custom protocol handling, in-network computing, or advanced telemetry [5, 10]. Since 2017 the latest specification of the P4 language is $P4_{16}$ which made it less complex and more structured, compared with the previous version $P4_{14}$, by moving a large number of features to libraries [11]. Henceforth in this thesis, P4 is referring to the $P4_{16}$ specification, unless otherwise specified.

A P4 program defines how packets are processed as they traverse a network device. It starts with header definitions that specify the structure of extracted packet fields. The parser then identifies and extracts these fields based on protocol headers. After parsing, packets enter match-action tables, where specific fields are evaluated against rules to determine actions such as forwarding, modifying, or dropping the packet. The control logic orchestrates how these tables interact, guiding the processing pipeline. Once processing is complete, the deparser reconstructs the packet before transmission through the egress pipeline [10].

P4 is designed to be both protocol-independent and target-independent, meaning it can be deployed on various programmable hardware platforms, including software switches, Field-Programmable Gate Arrays (FPGAs), and programmable Application-Specific Integrated Circuits (ASICs) (e.g., Tofino) [10]. This adaptability allows network engineers and researchers to implement and test new protocols without requiring modifications to physical hardware [6].

The P4 Project has developed an open source software switch, Behavioral Model version 2 (BMv2), designed to be the reference target for P4 programs [12]. BMv2 supports the commonly used V1Model [13], an architecture that defines the programmable structure of the packet pipeline, essentially functioning like an interface between the switch and the programmer [14]. The V1Model pipeline consists of 6 stages which are the *Parser*, *VerifyChecksum*, *Ingress*, *Egress*, *ComputeChecksum* and *Deparser*. All stages must be defined in the P4 program, even if unused, for it to be considered valid and compilable.

2.2 Mininet

Mininet is a network emulation tool that enables the creation of virtual networks on a single machine. It provides a lightweight and scalable environment for testing network protocols, topologies, and SDN applications without requiring physical hardware. By simulating an entire network, including hosts, switches, controllers, and links, Mininet allows for prototyping and validating network behavior efficiently [15, 16]. A Mininet network consists of virtual hosts, switches, and links that behave similarly to real network devices. Each host runs a full Linux network stack, allowing execution of real-world applications and protocols. The switches in Mininet can be either software-based, such as Open vSwitch (OVS) and BMv2, or externally connected hardware devices. The network topology can be customized programmatically using Python scripts, making it easy to test different configurations and failure scenarios [15]. One of Mininet's key advantages is its integration with SDN controllers, such as the OpenFlow protocol, which enables dynamic control of packet forwarding rules. This makes it particularly useful for evaluating SDN applications, including custom routing algorithms, traffic engineering strategies, and security mechanisms. Since Mininet runs on standard hardware, it provides a cost-effective alternative to physical testbeds, enabling rapid experimentation and debugging [15].

2.2.1 P4-Utills & P4-Learning

To ease the development and prototyping of data plane programs in P4, the Networked Systems Group (NSG) at Eidgenössische Technische Hochschule Zürich (ETHZ) developed an extension to Mininet, P4-Utills [17]. On top of the network emulation capabilities of Mininet, P4-Utills implements the behavioral-model, a collection of P4 software switches such as BMv2 [18]. On top of P4-Utills, NSG developed P4-Learning, a compilation of resources related to data plane programming and P4 development in particular [19]. It includes a series of 16 exercises, based on the course "Advanced Topics in Communication Networks" given at ETHZ, as

well as 24 P4 examples. To ease in the installation of all required software, NSG provides a Virtual Machine (VM) which comes pre-installed with all dependencies or alternatively, a guide on how to install them [17].

2.3 Large Language Models

LLMs are advanced AI, more specifically ML, models designed to process, generate, and understand human language. They are built using deep learning techniques, primarily based on the transformer architecture [2], which allows them to capture complex linguistic patterns and relationships in text. See Figure 2.2 for an illustration of the transformer architecture. Unlike older models like Recurrent Neural Networks (RNNs) or Long Short-Term Memorys (LSTMs), which have difficulties with long-range dependencies, transformers use self-attention mechanisms to process words in parallel, making them highly efficient and scalable [20]. LLMs rely on self-attention and Rotary Position Embedding (RoPE) [21] to weigh the importance of different words in a sentence. This allows them to understand context, resolve ambiguities, and generate coherent text. The training process is typically divided into two phases, pre-training and fine-tuning. During pre-training, the model learns general language representations from large amounts of text data through self-supervised learning techniques like masked language modeling or causal language modeling. The objective being minimized during training is typically the cross-entropy loss. Fine-tuning then adapts the model to specific tasks, such as translation, summarization, or question-answering, minimizing the same objective as during pre-training [20, 22].

One key factor behind the success of LLMs is their ability to scale. Increasing the number of parameters, training data, and computational resources leads to significant improvements in performance. Larger models often exhibit abilities that smaller models do not, such as reasoning and code generation [23]. However, this scaling also comes with challenges, including high computational costs [23], ethical concerns, and issues with hallucination and sycophancy [24].

Current research is focused on making LLMs more efficient, reducing bias, and improving alignment with human values. Techniques like retrieval-augmented generation, memory-augmented architectures, and Reinforcement Learning from Human Feedback (RLHF) aim to make models more reliable and useful. With ongoing advancements, LLMs are becoming increasingly capable of understanding and generating human-like text across a wide range of applications [20].

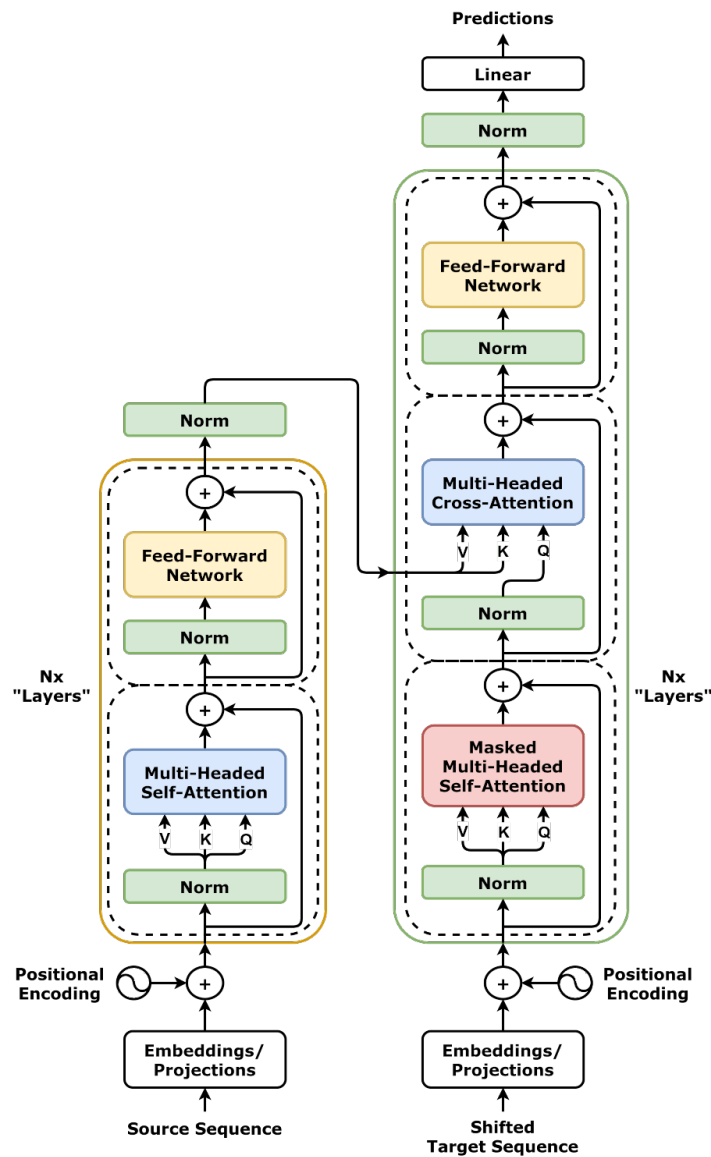


Figure 2.2: The Transformer architecture as described in *Attention is all you need* [2]. Image from [3].

2.3.1 Prompt Engineering

Prompt engineering is the practice of crafting input prompts to optimize the performance of LLMs. Since LLMs generate responses based on statistical patterns in the data they have been trained on, the way a prompt is formulated can significantly influence the quality, accuracy, and relevance of the generated output. Effective prompt engineering is important for ensuring that LLMs produce outputs that align with the intended task, particularly in applications requiring high precision, such as code generation, data plane programming, and automated reasoning [25].

A well-structured prompt should be clear and specific to avoid ambiguity. Providing sufficient background information helps the model understand the intended scope and constraints of the task. Explicit instructions, such as asking the model to list, summarize, or explain, can guide the response format to be more accurate and helpful. Different techniques are used in prompt engineering. Zero-shot prompting asks the model to perform a task without providing any explicit examples, though it may lead to inconsistent results for complex tasks. Few-shot prompting improves reliability by including examples to guide the model's response. An example of few-shot prompting, presented by Brown et al. in their paper [26], is teaching an LLM to correctly use a non-existent word in a sentence. The model is prompted with:

A "Gigamuru" is a type of Japanese musical instrument. An example of a sentence that uses the word Gigamuru is:

to which it replies

I have a Gigamuru that my uncle gave me as a gift. I love to play it at home.

Even though the word Gigamuru is not a real word that exists in any dictionary, the model could successfully use it in a sentence according to a given definition. Chain-of-Thought (CoT) prompting encourages the model to break down its reasoning step by step, improving logical reasoning and problem-solving. This approach builds on few-shot prompting and is done by providing examples of reasoning to the language model. An example, found in a paper by Wang et al. [27], would be prompting the model with:

Q: If there are 3 cars in the parking lot and 2 more cars arrive, how many cars are in the parking lot?

A: There are 3 cars in the beginning, 2 more arrive, so now there should be $3 + 2 = 5$ cars. The answer is 5.

Instead of simply giving the answer. Self-consistency prompting in turn builds on CoT prompting which samples multiple reasoning outputs from the model and selects the most consistent response, leveraging majority voting in choosing the final output. [28, 29].

Despite advancements, prompt engineering remains an empirical process with several challenges. Small variations in prompts can lead to significantly different outputs, and LLMs may inherit biases from their training data, affecting fairness and reliability. Additionally, the same prompt may yield inconsistent results across different model versions [28].

2.3.2 Fine-Tuning

Fine-tuning refers to the process of taking a pre-trained language model and continuing its training on a smaller, task-specific dataset. Typically during pre-training, the model is exposed to unlabeled data, where the aim is to predict the next token in a sequence. During fine-tuning, all or some of the model’s parameters are updated using more niche data that is relevant to the target task, such as sentiment analysis, question answering, or text classification. This additional training allows the model to adjust its internal representations and align more closely with the desired output behavior for the given context [30].

Supervised Fine-tuning (SFT) is one of the most common approaches for adapting LLMs. In SFT, a pre-trained model is further trained on a labeled dataset where the correct output is known, for example question-answer pairs, summaries, or code completions. The model learns to map inputs to desired outputs by minimizing a supervised loss, typically cross-entropy loss. This method is widely used for tasks where high-quality labeled data is available and provides strong performance with relatively few computational resources compared to pre-training [31].

However, SFT has limitations. It typically reflects the distribution and biases of the training data, and it may not fully align with human preferences or task-specific goals. Moreover, training purely on static demonstrations can result in unknown behavior when inputs fall outside the distribution seen during training. To address these limitations, RLHF has emerged as a useful addition to the fine-tuning process and SFT. RLHF involves three primary stages. First, SFT on human-written examples is used to produce an initial model. Secondly, reward modeling, where human annotators rank different model outputs to train a separate reward model that captures human preferences. Thirdly, Reinforcement Learning (RL), often using algorithms such as Proximal Policy Optimization (PPO), Direct Preference Optimization (DPO) or more recently Group Relative Policy Optimization (GRPO) which both are optimizations of PPO that replaces the reward model [32].

In addition to full-model fine-tuning via SFT and RLHF, more recent research

has introduced Parameter-Efficient Fine-Tuning (PEFT) techniques. These include methods such as Low-Rank Adaptation (LoRA) and prompt tuning, which enable effective adaptation of large models while updating only a small subset of parameters. PEFT methods are especially useful when computational resources or task-specific data are limited [33].

2.3.3 Low-Rank Adaptation

LoRA is a method for efficiently fine-tuning machine learning models, first introduced by Hu and Shen et al. [34]. Typically, fine-tuning involves adjusting all parameters of a model which, for larger models with several billion parameters like most LLMs, requires an extraordinary amount of computational resources. Using LoRA, the original parameters are frozen, and low-rank matrices are injected into certain layers of the model [34]. In both full fine-tuning and LoRA, all parameters are updated according to the following equation:

$$\mathbf{W}_{ft} = \mathbf{W}_{pt} + \Delta\mathbf{W}$$

where $\mathbf{W}_{pt}, \mathbf{W}_{ft}, \Delta\mathbf{W} \in \mathbb{R}^{d \times k}$ are the pre-trained parameters, fine-tuned parameters and update to the parameters, respectively. In full fine-tuning, $\Delta\mathbf{W}$ is a full rank matrix and contains $d \times k$ trainable parameters. In LoRA, however, the weight update matrix is instead:

$$\Delta\mathbf{W} = \mathbf{B}\mathbf{A}$$

where $\mathbf{B} \in \mathbb{R}^{d \times r}$, $\mathbf{A} \in \mathbb{R}^{r \times k}$ and $r \ll \min(d, k)$. \mathbf{A} and \mathbf{B} are matrices with trainable parameters and r is the rank. Depending on the choice of r , LoRA can significantly reduce the number of trainable parameters, increasing throughput and decreasing memory demands [35]. Additionally, LoRA provides a constant hyperparameter, α , which allows for scaling the parameter updates, $\Delta\mathbf{W}$, through $\frac{\alpha}{r}$ [34, 35].

Quantized LoRA

Quantized Low-Rank Adaptation (QLoRA), introduced by Dettmers and Pagnoni et al. [36], is a method to further decrease the memory requirements of LLM fine-tuning by using 4-bit NormalFloat (NF4), a data type introduced in their paper. NF4 is optimized for normally distributed parameters and gives better empirical results compared to other 4-bit data types. NF4 is combined with Double Quantization, a technique to quantize the quantization constants, providing additional memory savings.

2.3.4 State-of-the-art LLMs

Artificial intelligence and machine learning has existed for many decades but has in recent years garnered more popularity than ever before with the advent of the transformer architecture and LLMs. Arguably the most substantial contribution to the hype was the release of ChatGPT by OpenAI in November of 2022, using the model GPT-3.5 [37]. Since then, OpenAI has released new and improved models, and many other companies have entered the space.

OpenAI

OpenAI offers a range of closed LLMs, differentiated by size and reasoning capability. GPT-4.1 is their latest and largest model to date, performing well on creative and emotional tasks [38]. More recently they released o3 and o4-mini, known as reasoning models, meaning they generate a CoT before answering. This technique has proved to be beneficial for complex problem solving, like science, math and coding [39, 40]. o3 is OpenAI’s most powerful reasoning model and o4-mini is a smaller but more affordable reasoning model. Benchmarks such as SWE-bench, a software engineering benchmark, and LiveBench 2025-04-25 show that o4-mini performs on par with or slightly worse than o3 at coding [39, 41].

Microsoft

Microsoft’s latest offering of LLMs is Phi-4, Phi-4-reasoning and Phi-4-reasoning-plus, where the latter two are reasoning models and the former is not [42, 43]. Common benchmarks within math, science and coding reveal that the reasoning models perform roughly on par with each other and with o1-mini, OpenAI’s previous generation’s small reasoning model, but significantly better than the non-reasoning version, which scores about half on average [43].

Google DeepMind

Google DeepMind offer a number of LLMs at varying sizes and capabilities. Notably, Gemma 3 and Gemini 2.5 Pro, released in March 2025 [44, 45]. Gemma 3 is a collection of open models at varying sizes (1B, 4B, 12B and 27B), designed to run on limited resources [44]. Gemini 2.5 Pro is DeepMind’s flagship closed LLM and first reasoning model, and scores third on LMArena, another popular benchmark, in the coding category [46], behind OpenAI’s GPT-4o and Llama 4 Maverick, but scores much worse in LiveBench 2025-04-25 [41]. In SWE-bench verified, it is only beaten by Claude 3.7 Sonnet [45].

Meta

Llama 4 is, as of April 2025, Meta’s latest release of LLMs consisting of three models, Llama 4 Scout, Llama 4 Maverick and, at the time of writing still unreleased, Llama 4 Behemoth. Llama 4 Behemoth, a 2 trillion parameter model, was distilled into the smaller Llama 4 Scout and the larger Llama 4 Maverick which are both Mixture-of-Experts (MoE) reasoning models [47]. At the time of writing, Llama 4 Maverick scores first in the coding category on LMArena [46], but relatively poorly in the code generation category in LiveBench [41].

Anthropic

As of February 2025, Anthropic’s latest LLM offering is Claude 3.7 Sonnet which they call a hybrid reasoning model. It allows the user to specify how long, or if at all, the model should reason about it’s response [48]. Claude 3.7 Sonnet scores worse than o3, o4-mini [39] and slightly better than Gemini 2.5 Pro on SWE-bench verified [48].

2.4 Related Work

SDN and P4 have made it possible to develop sophisticated network applications and has given programmers a high degree of control over network configuration. On the other hand, P4 programming is a difficult and tedious process which requires specific knowledge, not only about the language itself, but about the target and the various protocols [49, 50].

To address these challenges, several intermediate languages, so called High-Level Data Plane Programming Languages (HLDPLs), have been developed (Lucid [49], Lyra [50], P4rrot [51], pcube [52], among others), which compile to P4 code. These languages aim to decrease the complexity of data plane programming by providing a higher level syntax which ideally is easier to learn and use by being more similar to natural language [53]. However, none of the above-mentioned HLDPLs are used to any greater extent in practice or have any active community of users.

Little research exists on generating P4 code using LLMs, however one study by Dumitru et al. [53] attempted employing OpenAI’s GPT-4 and Google DeepMind’s Gemini-Ultra to generate Lucid code using a few-shot prompting strategy. Because of the small test dataset of three problems and their vague descriptions, results were inconclusive. The same authors also conducted a study on generating P4 code directly using both open and closed LLMs [54]. They built a benchmark based on the tutorials repository from p4lang available on GitHub [55]. The models were prompted with the problem descriptions for each problem and asked to generate

the corresponding P4 code. Additionally, the authors assembled a custom dataset comprised of publicly available P4 code and other curated information related to the P4 language. The dataset was used to fine-tune three open models, StarCoder-1B, Gemma-2B and StarCoder2-3B. The results show that the fine-tuned, open models outperform the closed models, both in terms of compilability and alignment to the benchmark problems.

Beyond P4, there have been a few attempts at using LLMs to generate code in other niche domain-specific languages. Notably, Garcia-Gonzalez et al. [56] created DSL-Xpert, a generic DSL code generator system using any of OpenAI's models through their Application Programming Interface (API). The system allows specifying the grammar of the DSL, with backus-naur form being the recommended representation but JSON and XML formats being viable alternatives, and a few code examples as context. However, the authors did not evaluate the output to any greater extent and it is unclear which DSLs they have tested.

Chapter 3

Method

This chapter presents the methodology and methods used in this thesis to evaluate the applicability and effectiveness of LLMs for P4 code generation. It outlines the research philosophy, approach, strategy, data sources and evaluation metrics used.

3.1 Research Philosophy and Methodology

The research is grounded in a positivist philosophy, which assumes that objective truths can be uncovered through systematic observation and experimentation. This philosophical standpoint supports the quantitative nature of the study, where the performance of different LLMs is measured using well-defined, reproducible metrics [8]. By adopting this approach, the work aims to provide concrete evidence about the capabilities and limitations of LLMs in the context of DSL code generation.

In this study an experimental and applied research methodology is used. Given that the central goal is to test how well various LLMs can perform in generating correct and functional P4 code, this methodology provides an appropriate structure. The experiments are designed not only to evaluate the baseline performance of a state-of-the-art model using prompt engineering, but also to determine whether fine-tuning and prompt engineering lead to measurable improvements using open models. The applied nature of the work is clear in the practical approaches used to solve real-world development challenges in P4 programming, and in its focus on performance improvements that can be directly evaluated through testing and benchmarking. The research follows a deductive approach. It begins with a hypothesis that LLMs, when properly guided through prompting or fine-tuned using task-specific data, are capable of generating correct and usable P4 code. This hypothesis is examined through structured experiments, and conclusions are drawn based on empirical results. The deductive reasoning structure allows for compare between outcomes across different models and techniques, and to make claims that are based on observable data.

As a research strategy, the thesis combines elements of experimental design

with case study analysis. The experimental component involves evaluating LLMs on a set of well-defined P4 programming tasks derived from the P4-Learning suite provided by the Networked Systems Group at ETH Zürich. These tasks are representative of common challenges in data plane programming, and they come with problem descriptions and ground truth implementations, which serve as a benchmark for evaluation. Each LLM is applied to the same set of problems under controlled conditions, and the results are recorded and compared.

3.2 Literature review

The first step of the thesis project was to conduct a literature review with the purpose of gaining an understanding of the problem at hand and the current research on the topics surrounding it. Research on SDN and P4, their benefits and limitations for developers, the current state of LLMs and their code generation capabilities as well as applications of code generation in P4 and other DSLs, with or without the use of LLMs, was studied.

Several data sources were used to conduct the literature review. For SDN and P4 there are many papers and articles in scientific journals, and for P4 specifically the P4 Project has a list of many publications [57]. Information regarding LLMs was sourced through Google Scholar [58] by searching for publications with keywords such as "LLM", "Supervised fine-tuning" and "Reinforcement learning". Furthermore, searching through the citations of these publications proved to be an efficient method for finding deeper knowledge about the topics. Given previous knowledge of popular models and the organizations behind them, many of the state-of-the-art models can be found on their websites, in independent articles as well as in many benchmark comparisons between models.

3.3 Evaluation Data

The data used for evaluating the LLMs tested in this thesis was the P4-Learning GitHub repository, created by NSG at ETHZ, which includes 16 exercises, where a few of them require implementing more than one P4 file, for a total of 19 P4 files. Some exercises were omitted due to difficulties in testing or using duplicate P4 files. In total, 11 of the 16 exercises and 14 of the 19 P4 files were used in the evaluation dataset. See Table 3.1 for a list of all exercises and accompanying P4 files in the P4-learning repository as well as those used for the evaluation set, highlighted in bold. In addition to the P4 code itself, the compiler outputs, error messages, and runtime behavior of generated programs are recorded and analyzed. These secondary sources of information are treated as feedback mechanisms to

better understand where the models fall short and how prompting or fine-tuning might correct these issues.

Analysis of the data was carried out using quantitative techniques. The study measures the rate at which the models generate compilable code, the frequency of functionally correct outputs as judged by test runs in a Mininet-based environment, and the degree to which fine-tuning improves these outcomes. While qualitative aspects such as code readability are acknowledged, they are not the primary focus of analysis and are only used to guide prompting strategies. The performance metrics are examined to reveal trends and differences between model types, prompting strategies, and training methods.

Table 3.1: List of all P4 exercises and associated P4 files from NSG’s P4-Learning repository, of which those in bold were used in the evaluation dataset.

Exercise	P4 Files
01-Reflector	reflector.p4
02-Repeater	repeater.p4
03-L2_Basic_forwarding	l2_basic_forwarding.p4
03-L2_Flooding	l2_flooding_all_ports.p4 l2_flooding_other_ports.p4
04-L2_Learning	l2_learning_copy_to_cpu.p4 l2_learning_digest.p4
04-MPLS	basics.p4 stacked.p4
04-RSVP	rsvp.p4
05-ECMP	ecmp.p4
05-Flowlet_Switching	flowlet_switching.p4
06-Heavy_Hitter_Detector	heavy_hitter.p4
07-Count-Min-Sketch	cm-sketch.p4
08-Simple_Routing	ecmp.p4
09-Traceroutable	traceroutable.p4
10-Congestion_Aware_Load_Balancing	loadbalancer.p4
11-Packet-Loss-Detection	loss-detection.p4
12-Fast-Reroute	fast_reroute.p4

3.4 Training Data

The data used for training was the P4 dataset assembled by Dumitru et al. [54], which consists of 10979 samples of P4 code sourced from GitHub and includes the repository name and the relative path to the P4 source code in the repository. In their study, in addition to samples of code, they augmented the dataset with documents related to P4, such as Request For Comments (RFC) and the official $P4_{16}$ specification. However, they concluded that the addition of these documents to the dataset had little to no effect on performance of the trained models, therefore these samples were omitted in the training dataset of this thesis and includes only samples containing P4 code. A simple quantitative analysis of the dataset was also carried out in this study, measuring the distribution of character counts. This was done for the purpose of choosing a reasonable context length of the models being fine-tuned.

In an effort to avoid data contamination of the evaluation and training datasets, the training dataset was searched for the repository names in the evaluation dataset and consequently removed. However, it is possible that the code exists in other repositories in the dataset and as such the possibility of data contamination still exists.

3.5 Model Selection

Model selection was made based on factors like size, and thereby resource requirements, benchmark scores and whether they were trained to have reasoning capabilities. Additionally, certain models were excluded from consideration by request from Saab. Comparing models based on popular benchmark scores is difficult because of data contamination, where a model may have been trained on a part of, or the entire, benchmarking dataset, resulting in inaccurate and unreliable scores relative to other models [59]. Therefore, the main source of comparison between models was LiveBench, a benchmark which limits data contamination by introducing new questions with verifiable, objective ground-truth answers monthly across six categories [41]. Given that code generation is the subject of this thesis, only the *coding* category, and specifically the *code generation* sub-category, as part of the benchmark version released 2025-04-25 was used.

The models that score the best in this benchmark are closed-weight models, such as OpenAI’s o3 High and o4-Mini High, followed by Anthropic’s Claude 3.7 Sonnet and its reasoning counterpart Claude 3.7 Sonnet Thinking as well as Google’s Gemini 2.5 Pro. The best open-weight model was Microsoft’s Phi-4-reasoning-plus, a 14 billion parameter reasoning model trained using Both SFT and DPO on 9.8 trillion tokens over a period of 21 days using 1920 H100 Graph-

ical Processing Units (GPUs). Phi-4 was released in three 14 billion parameter versions, Phi-4, Phi-4-reasoning and Phi-4-reasoning-plus. The difference between these models lies in the training data and the number of tokens allowed for reasoning versus producing an answer, where Phi-4 does not reason at all. Phi-4 and Phi-4-reasoning have not been evaluated on the LiveBench dataset, however, comparing them on OpenAI’s HumanEvalPlus benchmark [60], Phi-4-reasoning and Phi-4-reasoning-plus score similarly at 92.9 and 92.3 respectively whereas Phi-4 scored 83.5 [43].

Considering that Phi-4-reasoning-plus generates 50% more tokens than Phi-4, and therefore bring with it higher latency and cost of inference [43], Phi-4 was selected for fine-tuning.

Additionally, a 32 billion parameter model was also chosen for fine-tuning in order to see the impact of model size on P4 code generation. However, given the importance of this model to Saab, further details about it will not be disclosed. This model will henceforth in this thesis be referred to as M-32B.

Finally, in order to put the results into a broader context the prompting techniques were also used on o4-mini, which represents the state-of-the-art in the *code generation* sub-category on LiveBench 2025-04-25 [41].

3.6 Model Fine-Tuning

Given that the training dataset contains no labels, fine-tuning a model on it must be done through unsupervised learning. Ideally prompts would be generated for each sample in the dataset, where the label is the P4 code. However, doing this reliably with the number of samples available would be too time consuming.

Both Phi-4 and M-32B were fine-tuned using QLoRA with $r = 16$ and $\alpha = 16$, the values that yielded the lowest validation loss during testing in the original LoRA paper [34]. Henceforth the fine-tuned base models will be called Phi-4-p4 and M-32B-p4. The P4 dataset was randomly split into a training dataset and a validation dataset using a 95:5 ratio with a validation batch size of 4 and 2 as well as a training batch size of 8 and 4 for Phi-4 and M-32B respectively. The only other difference in training was the number of training epochs, 1 epoch for Phi-4 and 2 epochs for M-32B. The reasoning was that because M-32B is more than twice the size of Phi-4, it would not overfit to the same extent and a longer training run was therefore warranted. A list comparing key training arguments between the two models is found in Table 3.2. During training of both models, the same modules were targeted, q_proj, k_proj, v_proj and o_proj which are part of the multi-head attention mechanism of transformers, and gate_proj, up_proj and down_proj, part of the feed-forward layers, as recommended in the QLoRA paper [36].

Table 3.2: Comparison of key training arguments for QLoRA fine-tuning of Phi-4 and M-32B.

Training Argument	Phi-4	M-32B
Batch Size (micro/global)	8 / 16	4 / 8
Gradient Accumulation Steps	2	2
Learning Rate	5e-05	5e-05
Optimizer	AdamW	AdamW
LR Scheduler	Linear	Linear
LoRA Rank (r)	16	16
LoRA Alpha (α)	16	16
LoRA Dropout	0	0
Gradient Checkpointing	True	True
Bits (Quantization)	4-bit (NF4)	4-bit (NF4)
Training Epochs	1	2
Max Seq Length	4096	4096
Warmup Ratio	0.1	0.1
Weight Decay	0.01	0.01

3.7 Testing and Model Inference

In order to test the generated P4 code in terms of functionality, both compilation tests and output tests must be conducted. For this purpose a set of problems with accompanying problem descriptions and ground truth solutions are required. Additionally, a network emulation environment which supports data plane programming in P4 is needed.

P4-Learning, based on P4-Utills, by NSG provides exactly this functionality and is used as the base of testing the models and approaches in this thesis. NSG provides a VM that comes pre-loaded with all software required to run the code in P4-Learning, which makes environment setup relatively simple. P4-utills serves as the backbone for testing, on top of which a pipeline for prompting, executing the tests and logging is created in order to simplify and automate the benchmarking procedure.

3.7.1 Testing Methodology

The testing procedure is composed of two primary evaluation stages: compilation testing and functional testing. Both stages are applied to every generated P4 program, and their outcomes are logged individually for analysis.

Compilation testing is the first gate in the evaluation pipeline. Each P4 file generated by a model is passed to the P4 compiler (`p4c`) targeting the BMv2 back end with the V1Model architecture. A script was used to automate this step, capturing the compiler output and exit status. If the compiler exits without errors, the generated code is considered to have passed compilation. Otherwise, the compiler errors are recorded and stored alongside the generated code for further analysis. This test is necessary because syntactically incorrect or semantically invalid P4 code cannot proceed to the next stage. Compilation success rate is the number of successfully compilable P4 programs the model produced over the total number of P4 programs the model was asked to generate. This is used as one of the primary metrics to assess a model’s basic ability to produce valid P4 syntax and conform to the semantics required by the target architecture.

Generated programs that pass compilation are subjected to functional testing using the P4-Learning repository’s suggested tests. Each exercise in the repository includes a predefined testing procedure, which involves setting up a simulated network topology using Mininet and sending specific test packets through the network to verify expected behavior. The generated P4 programs are deployed in the same topology as their reference implementations. A testing script is then used to execute the test cases and record the observed outputs. The outputs are then compared to the expected results from the reference implementation using exact output match. If the behavior of the generated program matches the reference output across all test cases, it is considered a functionally correct solution. Output success rate is the number of functionally correct P4 programs generated by the model over the total number of P4 programs the model was asked to generate.

For every problem in the evaluation dataset, the ground truth consists of the reference P4 implementation provided in the P4-Learning suite. These implementations are known to be correct and serve as the baseline for functional evaluation. To eliminate ambiguities in output comparison, deterministic test cases with fixed packet inputs and expected outputs are used.

Some exercises were excluded from the evaluation set due to difficulties with reproducibility, ambiguous test cases, or overlapping code files, as noted in Section 3.3.

3.7.2 Model Evaluation

For each of the base models (Phi-4, M-32B and o4-mini) tested, a baseline test was conducted, where no prompting technique was applied. Then, one by one, the prompting techniques were applied for each model. For the fine-tuned models, no prompting approaches were used. For each prompting strategy and model, the entire evaluation dataset was run through five times, except for the iteration approach which instead was set with a maximum number of iterations of five

per sample in evaluation dataset. Running the tests five times each allowed for calculating Pass@5.

Pass@k

Pass@k is a metric typically used when comparing code generation in LLMs [61]. It evaluates the ability of an LLM to produce at least one correct solution within k attempts. It should be noted that there is an unbiased estimator of Pass@k commonly used as the above described way of calculating the metric could exhibit large variance. The unbiased estimator is defined as the following:

$$Pass@k = \mathbb{E} \left(1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right)$$

Where $n \geq k$ samples are generated per problem and c is the number of correct samples of n . The former approach is used to calculate Pass@k in this thesis. Since both compiler and functional tests are conducted for each generated P4 file, they are scored separately using the Pass@k metric. The output tests are scored based on how many generated P4 files compiled such that if a P4 file did not compile, it was not counted as an incorrect solution.

3.8 Prompting Techniques

Given the generalization abilities of LLMs on many tasks, different methods of prompt engineering were tested to evaluate the feasibility of using models not explicitly trained on P4 code generation. Four different methods were tested: Prompting using a number of programming insights extracted from coding traces in P4; summarization of problem description; including header files from the solution; iterating compiler errors and differences in output between the generated solution and the reference solution back to the model. The first four approaches were evaluated using the Pass@k metric with $k = 5$, whereas the iterative approach was evaluated using one pass of the evaluation dataset with 5 iterations. All approaches were prompted with the same system prompt (See Listing 3.1), containing formatting information for ease of extraction, before being prompted with the specific instructions for each exercise.

<system>: You only generate code in the $P4_{16}$ programming language for the BMv2 software switch and V1Model architecture, no explanations or other text, formatted in fenced code blocks in markdown like this: ‘‘‘p4 <code>‘‘‘. Put the name of the file in a comment at the top of the generated file.

Listing 3.1: System prompt for code generation.

Insight Generation

Any LLM has difficulties producing compilable P4 code because of the relatively small number of examples encountered during training, given the scarcity of publicly available P4 code data. The idea with extracting programming insights and prompting a model with them when generating P4 code is to mitigate the number of simple mistakes from the start. In this thesis an insight was defined as: "A precise and focused piece of knowledge, distilling a key learning experience". Insights were extracted by having o4-mini solve the exercises iteratively and saving the produced code as well as the compiler errors between iterations. The model was given 5 iterations and if it did not produce compilable code or if it succeeded in the first iteration, the test was run again for those exercises in order to ensure there was something to base the insight generation on. GPT-4o was then prompted with a system prompt with instructions and a definition of insights, seen in Listing 3.2, as well as the coding history in order and asked to produce insights. See Appendix A for the complete list of insights extracted from the P4 error traces.

```
<system>: You are an expert code reviewer. You will be provided with a history of a  $P4_{16}$  code file. Between versions of the code there will be compiler errors. Your job is to provide insights about the coding process that generalize to other similar compiler errors. An insight is defined as follows: 'A precise and focused piece of knowledge, distilling a key learning experience.'
```

Listing 3.2: Prompt for generating insights

Summarization of Problem Descriptions

GPT-4o was prompted to summarize each problem description using a set of instructions, as seen in Listing 3.3. The reason this method was considered as a prompting strategy was because it has two potential benefits. Firstly, it could reduce the number of tokens used as input which leaves more tokens for future context, increases inference speed, and decreases costs. Additionally, based on a study on context lengths in LLMs, the effective context length of some models could be as low as 3% of the claimed context length [62]. The claimed context lengths are 16K and 32K tokens for Phi-4 and M-32B, respectively. The average character count of the problem descriptions fed to the model is 11377 characters, or roughly 2844 tokens, estimated using a 4:1 characters to tokens ratio that has been derived as a general rule of thumb [63]. A histogram showing the character counts per exercise problem description can be seen in Figure 3.1. This results in

an average context saturation of 18 % and 9% for Phi-4 and M-32B, respectively, which could pose a problem if effective context length is close that observed for some models in the previously mentioned study [62]. Secondly, summarization could remove unnecessary information that does not serve the purpose of solving the data plane programming part of the exercise, such as control plane implementation details and links to web pages and images.

<system>: The following is a problem description and instruction document for an exercise in $P4_{16}$ programming. Summarize it such that only information about the implementation of the data plane (*i.e.*, $P4_{16}$ program or programs) is present and it is easy to follow. Assume everything is already completed (*e.g.*, control plane, commands etc.) and only the $P4_{16}$ program needs to be implemented. If there is information regarding for example the control that is necessary for implementing the $P4_{16}$ programs, include it as well.

Listing 3.3: System prompt used when summarizing problem descriptions.

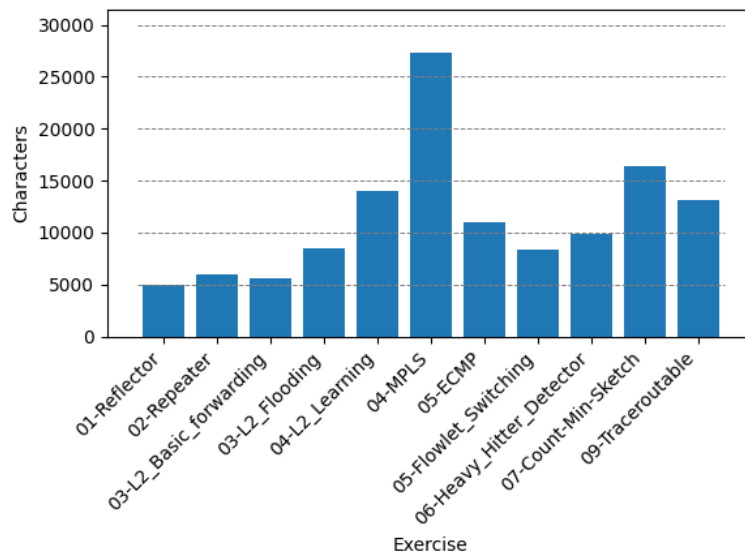


Figure 3.1: Character counts per exercise problem description, formatted as Markdown, used in the evaluation dataset.

Include Packet Headers

Including the protocol headers in the code generation prompt was tested for two main reasons. Different protocols typically have headers associated with them which specify information about the payload. In a defense setting, what protocols are used, and therefore their associated headers, are tightly controlled. This control is part of the reason for using a programmable switch. Therefore it may not be reasonable for an LLM to make the decision on which headers to use in some applications. It was also hypothesized that supplying the LLM with headers could help its ability to generate more correct code because it leaves one less point of failure.

Iteration of Output

Allowing the LLM to iterate on the code in order to make it more correct in terms of compilation and output is a reasonable approach because it mimics the process of a human programmer. When the LLM generates P4 code, it is compiled using P4C. If it produces an error, it is fed back to the LLM along with the context of the conversation and asked to remedy the problem. If it produces an incorrect solution again, the same step is repeated. If it produces compilable code, the compiled code is loaded onto a software switch in a network simulation environment where the suggested testing procedure in the exercise description document is run programmatically. The output of the testing procedure is recorded and compared to the output of the reference solution also bundled with the exercise using exact match comparison. If the generated solution's output is different from the reference solution's output, the model is prompted with both solutions and asked to try again. The compilation test is considered passed for a certain exercise and P4 file if the model produces a compilable solution at any point during iteration, even if the final P4 file is not compilable. In order to have a fair comparison against the other three prompt engineering approaches which were evaluated using Pass@5, the models were given a maximum of 5 iterations at generating correct code for each P4 file.

3.9 Inference Settings

At inference time, M-32B and Phi-4 were configured to use their native inference settings as defined by their respective generation configuration files. For o4-mini the default API settings provided by the developers were used. The fine-tuned models used the same settings as their pre-trained counterparts. Table 3.3 includes the relevant inference parameters and their values for each model. All

models used sampling, as opposed to greedy decoding, when generating output. In the latter approach a model’s output, given the same input, will be the same every time, by always choosing the most probable token from the model’s output distribution. The former approach introduces stochasticity by sampling the final output from the distribution. The top-k, top-p and temperature parameters limit or alter the distribution when doing sampling. Top-k filtering extracts the k most probable tokens from the output distribution of the model. Top-p filtering, on the other hand, extracts the most probable tokens from the output distribution which cumulative probability adds up to p . For both M-32B and Phi-4 models these two filtering approaches are combined when sampling. First top-k filtering is done, effectively creating a new distribution. From this new distribution, top-p filtering is done which outputs the final distribution sampled from. This combination comes as a consequence of using the Transformers Python library for inference [64]. Temperature scales the probabilities of the output distribution where a value larger than one flattens it, a value lower than one sharpens it and a value equal to one leaves it unchanged.

Table 3.3: Comparison of key inference parameters for Phi-4/-p4, M-32B/-p4 and o4-mini. Top-k for o4-mini cannot be specified in their API and is not reported in OpenAI’s documentation.

Inference Parameter	Phi-4/-p4	M-32B/-p4	o4-mini
Context length	16,384	32,768	200,000
Temperature	1	0.7	1
Top-p	1	0.8	1
Top-k	50	20	N/A

Chapter 4

Implementation

This chapter outlines the practical application of the methods described in Chapter 3. It details the technical infrastructure, tools, and workflows developed to evaluate the P4 code generation capabilities of the selected LLMs. The implementation is divided into three core components and are explained in their associated Sections. Section 4.1 presents the the automated testing pipeline for evaluating generated code, Section 4.2 presents the procedure for fine-tuning the open-weight models and Section 4.3 the setup for running model inference. By creating a robust and reproducible environment, this work ensures that the comparisons between different models and prompting strategies are conducted on a consistent and fair basis.

4.1 Testing pipeline

In order to simplify the testing of the P4 code generation abilities of the models examined in this thesis, a testing pipeline was built, written in Python. The pipeline includes functionality for choosing which of the prompting techniques to use and even to use them simultaneously, though this was not used in this thesis. Each exercise in NSG’s P4-Learning repository is bundled with a problem description specified in a markdown formatted README file, a topology specification used by Mininet to create the network, code skeletons for the P4 file and controller code in Python and a sub-directory containing a copy of the parent directory with correct implementations of the data plane and controller. Because this thesis is only concerned with P4 code generation, when testing, all other necessary implementations were completed such that only the P4 code had to be implemented to complete the tests. For certain exercises the P4 code is split into several files, typically headers.p4 and parsers.p4 which contain header and parser declarations respectively, and a main P4 file which contains the logic and includes the headers and parsers files.

In total there are 16 exercises, where some exercises contain sub-exercises with different network topologies, control code and data plane implementations. This

brings the total number of P4 files to 19. Some exercises were omitted in the testing because of various reasons, including broken controller code, difficulty testing and duplicate P4 files from previous exercises. In total, the pipeline tests 14 of the 19 P4 files. For each exercise two tests, compilation and output tests, are run.

4.1.1 Compilation tests

Before testing the output of a generated P4 program, a compilation test is done. The compilation is done using the P4 reference compiler provided by the P4 Project. When compiling, the target, architecture and P4 standard are specified, *i.e.*, BMv2, V1Model and $P4_{16}$ respectively, because this is the setup used in the exercises and therefore returns more accurate compiler errors. In the event the compiler returns an error and the maximum iterations in the pipeline is greater than one, the error will be fed to the model. If compilation is successful, an output test will be run.

The compile test is done in Python using the Popen class from the subprocess module. Popen enables the creation of processes without blocking execution, from within Python, which can be interacted with through the STDIN, STDOUT and STDERR streams. To compile a given P4 program the command shown in Listing 4.1 was used, targetting the V1Model architecture, the BMv2 switch and the $P4_{16}$ standard.

```
p4c -std p4-16 -target bmv2 -arch v1model <P4 program>
```

Listing 4.1: Command for compiling a P4 file.

If the output of this command is an empty string the compilation was successful, otherwise it returns the compilation error, which is fed back into the model if iteration is allowed.

4.1.2 Output tests

The output test is implemented as exact string matching between the correct and the generated solutions, where the correct output was obtained by running the testing procedure on the correct solution. The pipeline then follows the procedure programmatically, replacing the reference P4 solution with the generated P4 file and recording the output. If the output is incorrect and the maximum iterations is greater than one, the correct output together with the output obtained using the generated P4 file is fed back to the model.

The virtual networks, along with all hosts and switches, are created using the p4run command provided by the p4-utils extension of Mininet through the Popen

class in Python. For each P4 program, applications or utilities required for testing their functionality e.g. ping, tcpdump or interacting with individual hosts through the mx command, is also done using Popen.

4.1.3 Model Prompting

The pipeline supports the OpenAI API [65] for using and prompting o4-mini. To facilitate inference on Phi-4 and M-32B, which were too large to host locally, an endpoint on Hugging Face’s Inference Endpoints [66] was created, which was accessed via their API.

For each exercise the chosen LLM is prompted with a system prompt describing the role of the LLM as an expert in P4 programming and instructions regarding formatting of the response. The initial user prompt for an exercise asks the model to generate P4 code according to the problem description. If predefined headers are used the model is prompted to include them in the generated file using the `#include` keyword. If the model generates a solution that produces an error and more than one maximum iteration is specified, the model is allowed to improve its solution by being prompted either the compiler error or the difference in output between the reference and generated solution. If the model is allowed to iterate, it keeps the context of the previous prompts and its responses for the given exercise, but not between exercises, meaning the model cannot remember any previous exercises.

4.2 Training Procedure

The models were trained on a single NVIDIA A100 GPU with 64 Gigabytes (GBs) of Video Random-Access Memory (VRAM), hosted on the Leonardo supercomputer. For the implementation, the Unsloth [67] optimization library was used alongside Hugging Face’s Transformer Reinforcement Learning (TRL) library [68]. To facilitate easier inference and access, the final trained models were subsequently published on the Hugging Face Hub.

4.2.1 Leonardo

Leonardo is a supercomputer located in Bologna, Italy and operated by CINECA [69]. Leonardo has two main partitions, a booster module partition and a data-centric partition, which can be chosen by the user depending on what their work requires. The booster partition contains 3456 compute nodes where each node contains four NVIDIA A100 GPUs, one 32 core Intel Central Processing Units (CPUs) and 512 GB of Random-Access Memory (RAM). Access to Leonardo was

granted through Securing Enterprises via Machine Learning-based Automation (SEMLA) [70], a project where, among others, both KTH and Saab, as consortium lead, participate. Leonardo utilizes Slurm, a cluster management and job scheduling system for Linux clusters, through which access to compute nodes must be requested through a Slurm script [71].

4.2.2 Unsloth & TRL

Unsloth is a training framework, written in Python, which allows for more efficient training in terms of both memory and speed compared to other approaches, enabled by a custom autograd engine and rewritten Triton kernels, e.g. the cross entropy loss kernel. Additionally, Unsloth supports RoPE Scaling [72] internally, which improves a model’s ability to handle longer context lengths, though it may suffer when using a context length larger than it was trained on [73, 74]. TRL is a Python library for fine-tuning pre-trained foundation models and is built around the Transformers ecosystem and model-definition framework [68]. TRL integrates Unsloth’s efficient implementations, making training of models in the 30 billion parameter class feasible when combined with a single powerful GPU.

4.3 Model Inference

For o4-mini, inference was done through OpenAI’s Chat Completions API in Python. Using the API costs money and, at the time of writing, OpenAI charges 1.10\$ and 4.40\$ per 1 million input and output tokens, respectively, for o4-mini. Phi-4/-p4 and M-32B/-p4 were run on custom endpoints hosted on Hugging Face’s Inference Endpoints [66]. Although Leonardo has plenty of resources to run inference with these models, running the tests using P4-utils requires a list of complex requirements with a cumbersome installation process. Therefore, it was decided that the best course of action was to use an endpoint, hosted by an endpoint provider. However, this option brings with it two problems. Firstly, it introduces additional latency when prompting the models, which slows down the testing process. Secondly, renting a GPU to use in the endpoint costs money. For Phi-4/-p4, a single Nvidia A10G GPU with 24 GB of VRAM was sufficient, which at the time of writing costs 1\$ per hour. For M-32B/-p4, on the other hand, more VRAM was necessary because of its larger size. Therefore an Nvidia A100 GPU with 80 GB of VRAM was used, which at the time of writing costs 2.5\$ per hour. Customizing the endpoint inference behaviour is done through a specific program, handler.py. After starting the endpoint, it can be called through the requests module, an HTTP library in Python, and requests can be made to the model which is subsequently handled by the handler.py program.

Chapter 5

Result & Analysis

In this chapter, the empirical results of the experiments are presented and analyzed. Following the methodology and implementation detailed previously, this section provides a quantitative evaluation of the various models and prompting strategies tested. The findings are organized to first present the outcomes of the model fine-tuning phase, followed by an analysis of the datasets used, and finally, the core results concerning compilation success and functional correctness. The data is visualized through loss curves, heat maps, and bar charts to facilitate a clear comparison of performance across all tested configurations. This analysis forms the foundation for the broader discussion in the subsequent chapter.

5.1 Training

The fine-tuning process was applied to two base models: Phi-4 and M-32B. Both models were trained on an identical dataset to ensure a fair comparison of their performance post-tuning. The hyperparameter configurations for these training runs were kept largely the same, with a few exceptions. Specifically, the number of training epochs and the batch size were adjusted to accommodate the differences in model architecture and size. A detailed summary of all relevant hyperparameters used for the training of both models can be found in Table 3.2.

The evolution of the models during training can be seen in Figure 5.1. This figure displays the raw training loss, the Exponential Moving Average (EMA) of the training loss with a smoothing factor (α) of 0.95, and the validation loss for both the Phi-4 and M-32B models. EMA was used to provide a smoother representation of the training loss curve which helps to highlight the underlying trend. The total duration of the training runs differed between the two models, with Phi-4 completing its training in 16 hours and the larger M-32B requiring 23 hours.

Using QLoRA for fine-tuning allowed for efficient training by updating only a small fraction of the total model parameters. This method significantly reduces computational overhead and memory requirements. As a result, only a very small

percentage of the total parameters were actually trained. For the Phi-4 model, this amounted to 1.56% of its total parameters, while for the M-32B model, an even smaller fraction of 0.42% of its parameters was fine-tuned.

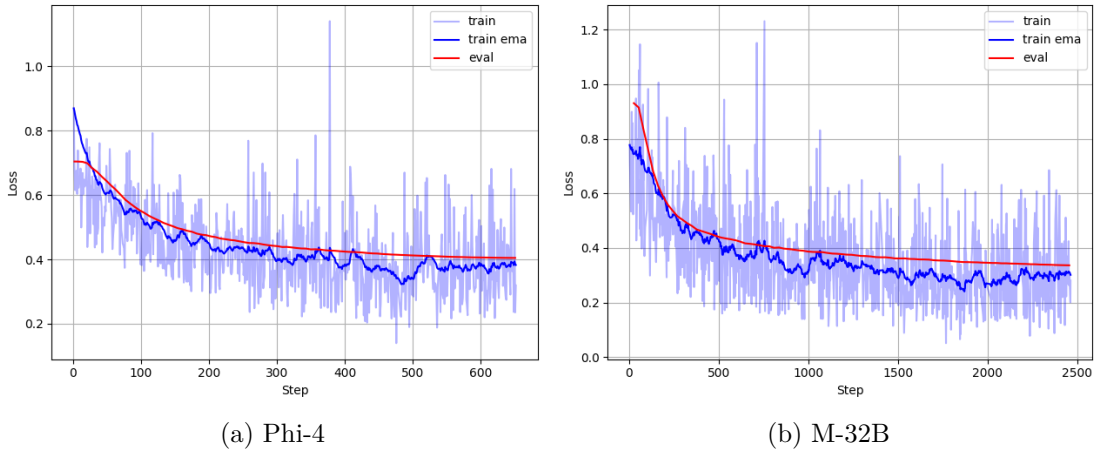


Figure 5.1: Training loss, smoothed training loss using exponential moving average ($\alpha = 0.95$) and evaluation loss for Phi-4 (a) and M-32B (b) over 1 and 2 epochs respectively.

5.2 Dataset Analysis

A simple quantitative analysis of both the training and evaluation datasets showed significant variation. Figure 5.2 presents a histogram of sample character counts in the training dataset, showing that approximately 94.9% fall within the maximum allowed token count of 4096 during training. For evaluation samples, Figure 5.3 shows that the average character count of summarized descriptions was approximately 3737, or roughly 934 tokens, down from 11377 characters and 2844 tokens in the original descriptions, as seen in Figure 3.1. This was well within the claimed context length of both Phi-4 and M-32B and significantly lower than the original descriptions.

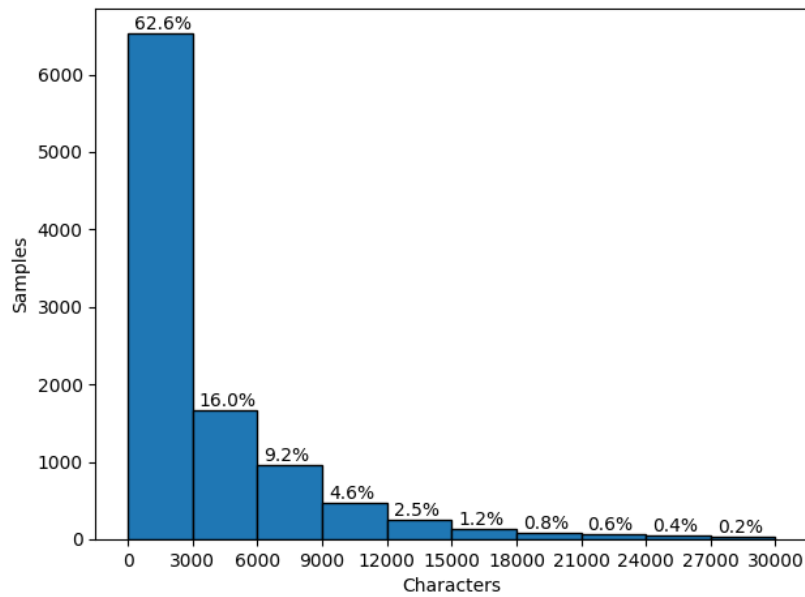


Figure 5.2: Histogram showing the distribution of sample lengths in the training dataset. The 200 largest samples are omitted from the histogram because they are way larger than the average sample character length.

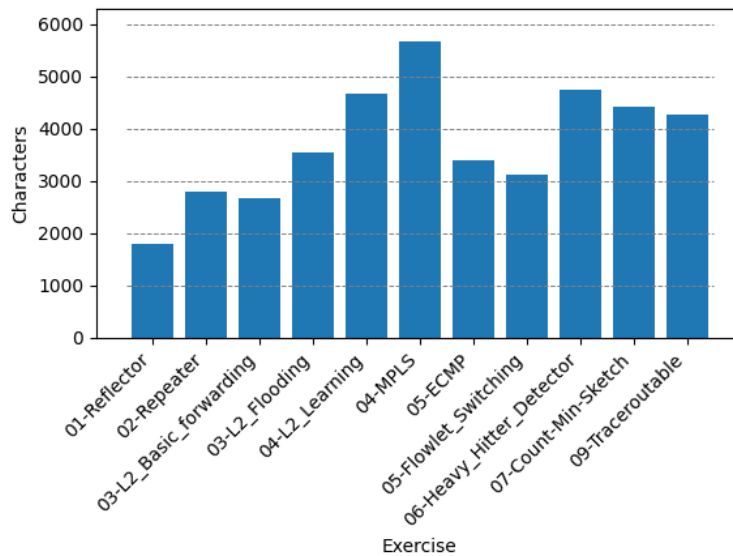


Figure 5.3: Character counts per summarized exercise problem description, formatted as Markdown, used in the evaluation dataset.

5.3 Compilation Success Rate

The Pass@5 metric was used to evaluate whether the models could generate compilable P4 code. No matter the prompting strategy used, Phi-4 and M-32B could not generate a single compilable P4 file for any exercise. o4-mini, on the other hand, could generate 4 compilable files in 5 attempts across all exercises using the base prompt, including headers, insights and using summarized problem descriptions, a Pass@5 score of 28.6%. When allowed to iterate on errors it achieved a significantly higher accuracy score of 57.1%. Despite the poor performance of Phi-4 and M-32B, their fine-tuned counterparts, Phi-4-p4 and M-32B-p4, performed markedly better with Pass@5 scores of 35.7% and 64.3% respectively. See Table 5.1 for compilation success rate using Pass@5 score for all models and prompting strategies, except 5-iter which uses accuracy. Additionally, Figure 5.4 shows a bar plot comparing compilation success rate only for models and prompting strategies which resulted in non-zero values. The fine-tuned models were not tested using any prompting strategies due to time and money constraints.

Table 5.1: Compilation success for all considered models and prompting strategies. The metric used is pass@5 for all prompting strategies except 5-iter, which was run only once with no independent attempts. Cells with N/A indicate a combination that was not tested.

Model	Base Prompt	Headers	Summarized	Insights	5-iter
Phi-4	0.00	0.00	0.00	0.00	0.00
M-32B	0.00	0.00	0.00	0.00	0.00
o4-mini	0.29	0.29	0.29	0.29	0.57
Phi-4-p4	0.36	N/A	N/A	N/A	N/A
M-32B-p4	0.64	N/A	N/A	N/A	N/A

5.4 Functional Correctness

Tables 5.2 and 5.3 shows absolute and relative output success in Pass@5 for all prompting strategies except 5-iter, which uses accuracy, respectively. Additionally, Figure 5.5 shows the same metrics as in the above mentioned figures without unevaluated or zero result combinations. Fine-tuning resulted in improved performance over any prompting strategy, regardless of the model used. Because Phi-4 and M-32B did not generate any compilable P4 code, their solutions could not be evaluated on functional correctness. When comparing the base prompt, including headers, insights or summarizing problem descriptions, o4-mini achieved a Pass@5

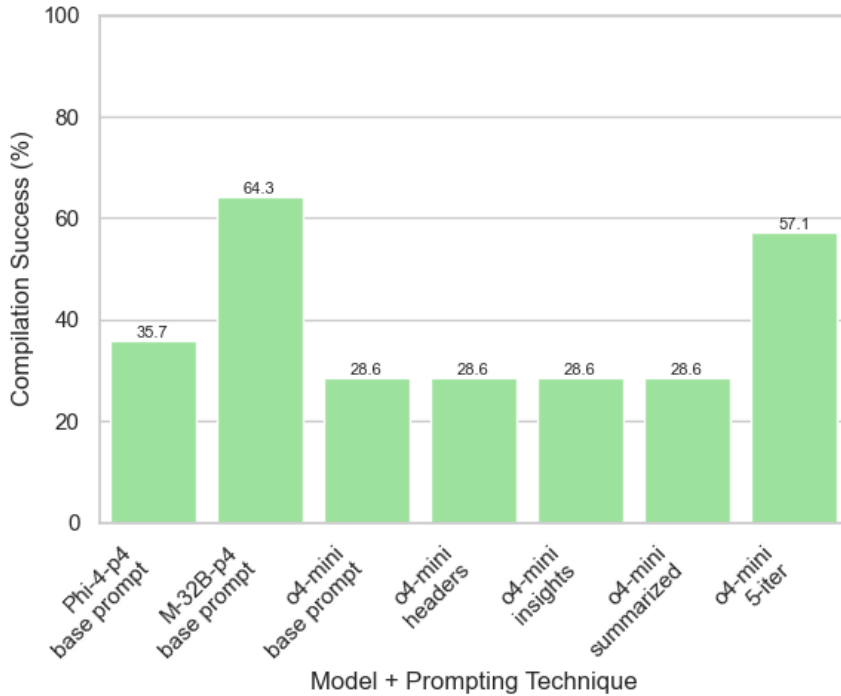


Figure 5.4: Bar plot of compilation success rate for Phi-4-p4, M-32B-p4 and o4-mini and prompting strategies. The metric used is pass@5 for all prompting strategies except 5-iter, which was run only once with no independent attempts.

score of 7.1% and 25.0% in absolute and relative output success respectively for the former two prompting strategies and a Pass@5 score of 14.3% and 50.0% in absolute and relative output success respectively for the latter two.

Table 5.2: Output success rate for all considered models and prompting strategies. The metric used is pass@5 for all prompting strategies except 5-iter, which was run only once with no independent attempts. Cells with N/A indicate a combination that was not tested.

Model	Base Prompt	Headers	Summarized	Insights	5-iter
Phi-4	0.00	0.00	0.00	0.00	0.00
M-32B	0.00	0.00	0.00	0.00	0.00
o4-mini	0.07	0.07	0.14	0.14	0.21
Phi-4-p4	0.21	N/A	N/A	N/A	N/A
M-32B-p4	0.21	N/A	N/A	N/A	N/A

Table 5.3: Relative output success rate (i.e. number of P4 files with correct output / number of successfully compiled P4 files) for all considered models and prompting strategies. The metric used is pass@5 for all prompting strategies except 5-iter, which was run only once with no independent attempts. Cells with N/A indicate a combination that was not tested.

Model	Base Prompt	Headers	Summarized	Insights	5-iter
Phi-4	0.00	0.00	0.00	0.00	0.00
M-32B	0.00	0.00	0.00	0.00	0.00
o4-mini	0.25	0.25	0.50	0.50	0.38
Phi-4-p4	0.60	N/A	N/A	N/A	N/A
M-32B-p4	0.33	N/A	N/A	N/A	N/A

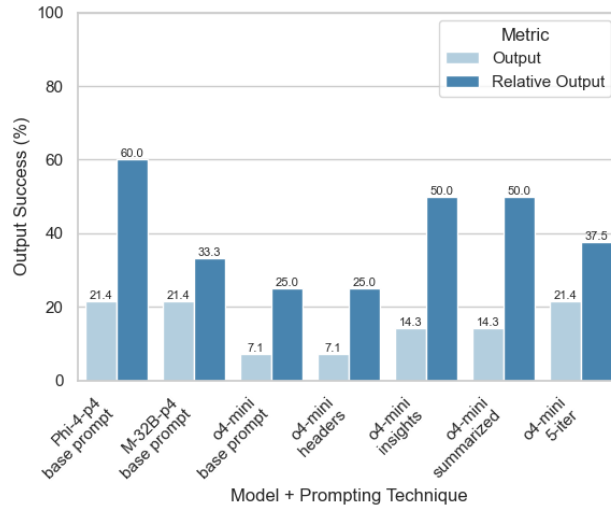


Figure 5.5: Bar plot of absolute and relative output success rate for Phi-4-p4, M-32B-p4 and o4-mini and prompting strategies. The metric used is pass@5 for all prompting strategies except 5-iter, which was run only once with no independent attempts.

Chapter 6

Discussion

This thesis explored the potential of LLMs for automating P4 data plane programming, using a number of prompting techniques and fine-tuning. Results show that LLMs are capable of generating syntactically valid and functionally correct P4 code with varying levels of success, depending on the model size, prompting strategy, and training setup. Moreover, results and conclusions drawn in this thesis can offer insights into the broader context of utilizing LLMs for generating code in DSLs in general.

6.1 Effectiveness of Fine-Tuning

The most striking result from this study is the dramatic performance gap between the base open-weight models, Phi-4 and M-32B, and their fine-tuned counterparts, Phi-4-p4 and M-32B-p4. The base models failed to generate a single compilable P4 program, scoring 0% pass@5 in the compilation benchmark. This outcome demonstrates that even capable LLMs lack the specialized knowledge required to handle a DSL like P4 without prior exposure in their training datasets. In contrast, fine-tuning with QLoRA proved to be highly effective. M-32B-p4 achieved the highest overall compilation success rate at 64.3%, followed by Phi-4-p4 at 35.7%. This highlights that even unsupervised, next-token-prediction training on a curated dataset can successfully teach relatively small models the syntactic rules and structural conventions of P4.

An interesting trade-off can be observed between the two fine-tuned models. While the larger M-32B-p4 performed better when it came to generating syntactically valid code, the smaller Phi-4-p4 demonstrated superior functional correctness when it did succeed. Phi-4-p4 achieved a relative output success rate of 60.0%, meaning that a majority of its compilable code was functionally correct. M-32B-p4, despite compiling more frequently, had a lower relative success rate of 33.3%. This difference could likely be attributed to a potential difference in difficulty between the exercises the two models managed to generate compilable code for.

6.2 Effectiveness of Prompting

For models without specialized fine-tuning, the choice of prompting strategy was important. o4-mini showed some baseline capability, but its performance was enhanced through certain prompting strategies. The iterative feedback approach stood out from the rest, increasing o4-mini’s compilation success from 28.6% to 57.1%, a score which approached the best fine-tuned model, M-32B-p4. This finding is significant as it mimics a human developer’s workflow of compiling code, receiving an error, and correcting it. By feeding compiler errors and output discrepancies back to the model, it can use its general reasoning capabilities to self-correct in-context, indicating that a conversational, multi-step process is a reasonable alternative to fine-tuning for model with some baseline capability. Interestingly, the other prompting techniques, providing headers, insights, or using summarized problem descriptions, did not markedly improve o4-mini’s compilation rate. However, using insights or summarized problem descriptions did improve the relative functional correctness to 50.0%. This suggests that while these methods may not solve low-level syntax issues, they help the model better understand the core task, leading to more logically correct code when compilation is successful. Summarizing the problem description likely helped by removing distracting details and allowing the model to focus on the essential data plane logic.

It is difficult to compare o4-mini to Phi-4 and M-32B and give an answer to why the latter two, without training, could not produce a single syntactically correct P4 file while the former performed quite well. Regarding the specifications of o4-mini, OpenAI have disclosed very little except its capability to reason by being trained using RLHF. While Phi-4 is not a reasoning model, M-32B does have the capability to reason. However, this feature was turned off in the evaluation because it dramatically increases the number of tokens generated and would thereby increase both evaluation time and cost. This could be the defining difference maker when it comes to the results. Another key aspect to consider is the size of the models. However, the number of parameters used in o4-mini is not public information and thus cannot be used as an argument for it’s success compared to the other models.

6.3 Limitations

While this study provides valuable insights, its findings should be interpreted in the context of several key limitations related to the dataset, methodology, and evaluation process. Acknowledging these issues is crucial for drawing accurate conclusions about the results and guiding future work.

6.3.1 Dataset Validity and Scope

A significant limitation is the potential for data contamination in the fine-tuning dataset. Although an effort was made to exclude the evaluation examples from the training data, the check was limited to repository names. Solution code may exist in forks or other public repositories within the training set, which could inflate the performance scores of the fine-tuned models.

Furthermore, the evaluation dataset is drawn entirely from the P4-Learning repository, which consists of a small number of P4 programming exercises in an academic context. These exercises may not fully represent the complexity, scale, or performance constraints of large-scale industrial P4 applications. The exclusion of certain exercises due to difficulties in testing or using duplicate P4 files could also introduce selection bias, potentially omitting harder problems that would have challenged the models to a larger degree.

6.3.2 Pre-training of base models

The base LLMs were likely trained on a vast amount of publicly available code, making it probable that some of the P4 programming examples from the evaluation dataset were included in their pre-training data. This exposure could lead to inflated performance scores, as the models may be recalling, rather than generating, the correct solutions. However, relevant comparisons can still be made between a base model and its fine-tuned or prompt engineered counterpart since they share the same pre-training process. Thus, any observed performance improvement can still be attributed to the applied strategy or, alternatively, to variance arising from the small evaluation dataset.

6.3.3 Methodological Choices

The fine-tuning in this thesis exclusively utilized QLoRA with fixed hyperparameters. This decision was based on the original LoRA paper’s findings rather than extensive hyperparameter tuning for this specific P4 code generation task. It is possible that different LoRA configurations, alternative PEFT methods or full fine-tuning could produce different results. Additionally, the LLM landscape evolves rapidly. The models selected for this thesis represent a snapshot in time, and their performance relative to one another may change as new versions are released.

6.3.4 Evaluation Metrics and Process

The primary metrics, compilation success and functional correctness, were treated as binary outcomes. A generated program that failed to compile due to a minor

syntax error was counted as a complete failure, receiving the same score as a program with fundamental logical flaws. Similarly, the functional test relied on exact string matching with the output of the reference solution. This rigid approach does not take the purpose of the test into consideration and it is possible that two test outputs could be identical in meaning but different in arrangement. More nuanced metrics like code similarity, e.g. CodeBLEU [75], or semantic evaluation could provide a more fine-grained view of model performance. Lastly, the study employed a direct calculation of the Pass@k metric, which, as noted in the methodology, can exhibit larger variance than the commonly used unbiased estimator, potentially affecting the precise reported success rates.

6.4 Ethical Implications

The deployment of AI-generated code, especially in critical network infrastructure, carries significant ethical responsibilities. The results show that even the best-performing models are far from perfect, which emphasizes the need for human oversight. An over-reliance on LLMs without thorough validation could introduce bugs or security vulnerabilities which are hard to detect into network data planes. It is clear that these tools cannot replace developers, but could be used to assist them, and that final verification should be done by a human expert.

6.5 Sustainability Considerations

The environmental impact of training and running LLMs is a valid concern. While the fine-tuning process for this thesis did not require a significant amount of computational resources on the Leonardo supercomputer, the base models did. However, this initial energy investment must be weighed against potential long-term benefits. An effective code-generation model could substantially reduce the human hours and computational cycles spent on manual development and debugging.

Furthermore, this study suggests a promising path toward sustainability. The fact that the smaller, fine-tuned Phi-4-p4 model was relatively effective, and in some ways superior to the larger M-32B-p4 in terms of correctness, indicates that bigger is not always better. Developing smaller, specialized, and highly efficient models through techniques like QLoRA offers a more sustainable alternative to relying solely on massive, general-purpose models for every task.

Chapter 7

Conclusions & Future Work

This chapter summarizes the findings of the thesis and outlines potential approaches for future research. First, the main conclusions drawn from the evaluation is presented, highlighting the effectiveness of different models and strategies. Following this, several directions for subsequent research are presented which address the limitations of this study and aim to build upon its foundational work.

7.1 Conclusions

This thesis investigated the efficacy of LLMs in automating the generation of P4 data plane programs. Through evaluation of both open-weight and closed-weight models, using various prompting strategies and fine-tuning, several conclusions can be drawn.

Capable open-weight LLMs like Phi-4 and M-32B struggle significantly with a niche DSL like P4, failing to produce a single compilable program. However, after fine-tuning on a curated P4 dataset using QLoRA, their performance improved significantly. This shows that for specialized domains with limited representation in general pre-training datasets, fine-tuning is needed in order to achieve acceptable performance.

A potential trade-off related to model size was found in the research. The larger fine-tuned model, M-32B-p4, achieved the highest compilation success rate (64.3%), suggesting it was more adept at learning the syntactic and structural rules of the P4 language. On the other hand, the smaller Phi-4-p4 model, while compiling less frequently, demonstrated a higher rate of functional correctness (60.0% relative output success) when it did succeed. This is likely due to differences in semantic difficulty between the problems for which the models produced compilable code.

For more capable models like o4-mini, an iterative feedback loop proved to be the most powerful strategy. By providing compiler errors and output discrepancies as corrective feedback, the model's compilation success rate went from 28.6% to 57.1%, approaching the performance of the best fine-tuned model. This indicates

that using a model’s general reasoning ability and mimicking a human developer’s process can be an effective alternative.

While prompting techniques like providing insights or summarized problem descriptions did not significantly increase compilation rates for o4-mini, they doubled the relative functional correctness of the generated code. This suggests that while these methods may not solve low-level syntax errors, they can guide the model toward a better high-level understanding of the problem, leading to more logically correct solutions.

In summary, this work demonstrates that LLMs have potential for automating P4 code generation, but more research is needed to make it feasible in production.

7.2 Future Work

Building on the findings and limitations of this study, several promising directions for future research exist. This study exclusively used QLoRA with fixed hyperparameters. Future work could explore the impact of full fine-tuning or experiment with different PEFT setups, e.g., varying LoRA rank and alpha, to optimize performance. Furthermore, creating a high-quality, instruction-based dataset for P4 and employing SFT or RLHF could align models more closely with developer intent and complex programming logic.

Another interesting approach for future works could be to expand the evaluation beyond binary pass/fail metrics. Using code similarity scores like CodeBLEU or semantic analysis could provide a more granular measure of LLM solutions. Moreover, expanding the evaluation dataset to include more complex, industrial-grade P4 programs would test the scalability and robustness of these models on real-world challenges.

Given limited resources in terms of time and money, testing the combination of prompt engineering and fine-tuning could not be done. A hybrid approach that combines these two could be highly effective. For instance, applying the successful iterative feedback strategy to the fine-tuned models could potentially elevate their performance beyond what either method achieved alone, possibly leading to the highest success rates.

Given the world’s increasing reliance on LLM and ML, it is a field of constant innovation and research and it is likely that the models used in this thesis will be irrelevant in the near future. Replicating this study with newer, more powerful foundation models, including larger open-weight models or next-generation proprietary models, would provide valuable insights into how architectural advancements impact performance in the realm of DSL code generation.

Furthermore, the process of generating insights from compiler errors was done manually. Future research could focus on creating an automated pipeline where

an LLM agent autonomously analyzes its errors, distills generalizable insights, and adds them to its context for subsequent generation tasks, creating a self-improving code generation system.

By following with these approaches, the research community can further unlock the potential of LLMs to serve as powerful co-pilots in the complex and critical domain of network data plane programming, and more generally in DSL code generation.

Bibliography

- [1] Z. Zuo, R. He, X. Zhu, and C. Chang, “A novel software-defined network packet security tunnel forwarding mechanism,” *Mathematical Biosciences and Engineering*, vol. 16, pp. 4359–4381, 05 2019.
- [2] A. Vaswani, N. Shazeer, and N. Parmar, “Attention is all you need,” 2023.
- [3] dvgodoy, “Deep learning visuals.” <https://github.com/dvgodoy/dl-visuals>. 2021.
- [4] K. Benzekki, A. El Fergougui, and A. Elbelrhiti Elalaoui, “Software-defined networking (sdn): a survey,” *Security and Communication Networks*, vol. 9, no. 18, pp. 5803–5833, 2016.
- [5] F. Hauser, M. Häberle, and D. Merling, “A survey on data plane programming with p4: Fundamentals, advances, and applied research,” *Journal of Network and Computer Applications*, vol. 212, p. 103561, Mar. 2023.
- [6] B. Goswami, M. Kulkarni, and J. Paulose, “A survey on p4 challenges in software defined networks: P4 programming,” *IEEE Access*, vol. PP, pp. 1–1, 01 2023.
- [7] International Energy Agency (IEA), “Energy and AI.” <https://www.iea.org/reports/energy-and-ai>. April 2025.
- [8] A. Håkansson, “Portal of research methods and methodologies for research projects and degree projects,” in *Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering FECS13*, CSREA Press U.S.A, 2013. [ed] Hamid R. Arabnia Azita Bahrami Victor A. Clincy Leonidas Deligiannidis George Jandieri.
- [9] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, “Software-defined networking: A comprehensive survey,” *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.
- [10] P. Bosshart, D. Daly, and G. Gibb, “P4: programming protocol-independent packet processors,” *SIGCOMM Comput. Commun. Rev.*, vol. 44, p. 8795, July 2014.

-
- [11] P4 Consortium, “P4_16 language specification.” <https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html>. Accessed: 2025-04-14.
- [12] P4 Project, “behavioral-model.” <https://github.com/p4lang/behavioral-model/tree/main>, 2025.
- [13] P4 Project, “Simple switch.” https://github.com/p4lang/behavioral-model/blob/main/docs/simple_switch.md, 2025.
- [14] P. Manzanares-Lopez, J. Muñoz-Gea, and J. Malgosa, “Passive in-band network telemetry systems: The potential of programmable data plane on network-wide telemetry,” *IEEE Access*, vol. PP, pp. 1–1, 01 2021.
- [15] Contributors, Mininet Project, “Mininet overview.” <https://mininet.org/overview/>. 2022.
- [16] B. Lantz and B. O’Connor, “A mininet-based virtual testbed for distributed sdn development,” *SIGCOMM Comput. Commun. Rev.*, vol. 45, p. 365366, Aug. 2015.
- [17] Networked Systems Group, ETH Zurich, “P4-utils.” <https://github.com/nsg-ethz/p4-utils>. Commit: 83b118b.
- [18] Networked Systems Group, ETH Zurich, “P4-utils documentation.” <https://nsg-ethz.github.io/p4-utils/introduction.html>. 2025-04-11.
- [19] Networked Systems Group, ETH Zurich, “P4-learning.” <https://github.com/nsg-ethz/p4-learning>, 2023.
- [20] S. Minaee, T. Mikolov, and N. Nikzad, “Large language models: A survey,” 2025.
- [21] J. Su, Y. Lu, S. Pan, A. Murtadha, B. Wen, and Y. Liu, “Roformer: Enhanced transformer with rotary position embedding,” 2023.
- [22] T. Brown, B. Mann, and N. Ryder, “Language models are few-shot learners,” in *Advances in Neural Information Processing Systems* (H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, eds.), vol. 33, pp. 1877–1901, Curran Associates, Inc., 2020.
- [23] J. Kaplan, S. McCandlish, and T. Henighan, “Scaling laws for neural language models,” 2020.
- [24] C. Deng, Y. Duan, and X. Jin, “Deconstructing the ethics of large language models from long-standing issues to new-emerging dilemmas: A survey,” 2024.

- [25] P. Sahoo, A. K. Singh, S. Saha, V. Jain, S. Mondal, and A. Chadha, “A systematic survey of prompt engineering in large language models: Techniques and applications,” 2024.
- [26] T. B. Brown, B. Mann, and N. Ryder, “Language models are few-shot learners,” 2020.
- [27] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, “Chain-of-thought prompting elicits reasoning in large language models,” 2023.
- [28] P. Sahoo, A. Singh, S. Saha, V. Jain, S. Mondal, and A. Chadha, “A systematic survey of prompt engineering in large language models: Techniques and applications,” 02 2024.
- [29] B. Chen, Z. Zhang, N. Langrené, and S. Zhu, “Unleashing the potential of prompt engineering in large language models: a comprehensive review,” 10 2023.
- [30] Y. Yu, S. Zuo, H. Jiang, W. Ren, T. Zhao, and C. Zhang, “Fine-tuning pre-trained language model with weak supervision: A contrastive-regularized self-training approach,” 2021.
- [31] V. B. Parthasarathy, A. Zafar, A. Khan, and A. Shahid, “The ultimate guide to fine-tuning llms from basics to breakthroughs: An exhaustive review of technologies, research, best practices, applied research challenges and opportunities,” 2024.
- [32] K. Murphy, “Reinforcement learning: A comprehensive overview,” 2025.
- [33] L. Xu, H. Xie, S.-Z. J. Qin, X. Tao, and F. L. Wang, “Parameter-efficient fine-tuning methods for pretrained language models: A critical review and assessment,” 2023.
- [34] E. J. Hu, Y. Shen, and P. Wallis, “Lora: Low-rank adaptation of large language models,” 2021.
- [35] IBM, “What is lora (low-rank adaption)?” <https://www.ibm.com/think/topics/lora>. Accessed: 2025-05-05.
- [36] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, “Qlora: Efficient finetuning of quantized llms,” 2023.
- [37] M. A. Haque and S. Li, “Exploring chatgpt and its impact on society,” *AI and Ethics*, 2024.

- [38] OpenAI, “Introducing gpt-4.5.” <https://openai.com/index/introducing-gpt-4-5/>. Accessed: 2025-02-27.
- [39] OpenAI, “Introducing openai o3 and o4-mini.” <https://openai.com/index/openai-o3-mini/>. Accessed: 2025-01-31.
- [40] OpenAI, “Reasoning models.” <https://platform.openai.com/docs/guides/reasoning?reasoning-prompt-examples=research&api-mode=responses>. Accessed: 2025-02-28.
- [41] C. White, S. Dooley, and M. Roberts, “Livebench: A challenging, contamination-free LLM benchmark,” in *The Thirteenth International Conference on Learning Representations*, 2025.
- [42] M. Abdin, J. Aneja, and H. Behl, “Phi-4 technical report,” 2024.
- [43] M. Abdin, S. Agarwal, and A. Awadallah, “Phi-4-reasoning technical report,” 2025.
- [44] Google DeepMind, “Introducing gemma 3: The most capable model you can run on a single gpu or tpu.” <https://blog.google/technology/developers/gemma-3/>. Accessed: 2025-03-12.
- [45] Google DeepMind, “Gemini 2.5: Our most intelligent ai model.” <https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/>. Accessed: 2025-03-25.
- [46] W.-L. Chiang, L. Zheng, and Y. Sheng, “Chatbot arena: An open platform for evaluating llms by human preference,” 2024.
- [47] Meta, “The llama 4 herd: The beginning of a new era of natively multimodal ai innovation.” <https://ai.meta.com/blog/llama-4-multimodal-intelligence/>. Accessed: 2025-04-05.
- [48] Anthropic, “Claude 3.7 sonnet and claude code.” <https://www.anthropic.com/news/claude-3-7-sonnet>. Accessed: 2025-02-24.
- [49] J. Sonchack, D. Loehr, J. Rexford, and D. Walker, “Lucid: a language for control in the data plane,” in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference, SIGCOMM ’21*, (New York, NY, USA), p. 731747, Association for Computing Machinery, 2021.
- [50] J. Gao, E. Zhai, and H. H. Liu, “Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics,” in *Proceedings of*

- the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, (New York, NY, USA), p. 435450, Association for Computing Machinery, 2020.
- [51] C. Györgyi, S. Laki, and S. Schmid, “P4rrot: Generating p4 code for the application layer,” 2022.
- [52] R. Shah, A. Shirke, A. Trehan, M. Vutukuru, and P. Kulkarni, “pcube: Primitives for network data plane programming,” 09 2018.
- [53] M.-V. Dumitru, V.-A. Bdoiu, and C. Raiciu, “Prose-to-p4: Leveraging high level languages,” 2024.
- [54] M.-V. Dumitru, V.-A. Bdoiu, A. M. Gherghescu, and C. Raiciu, “Generating p4 dataplanes using llms,” in *2024 IEEE 25th International Conference on High Performance Switching and Routing (HPSR)*, pp. 31–36, 2024.
- [55] P4 Project, “tutorials.” <https://github.com/p4lang/tutorials>, 2025.
- [56] V. Lamas, M. R. Luaces, and D. Garcia-Gonzalez, “Dsl-xpert: Llm-driven generic dsl code generation,” in *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems, MODELS Companion '24*, (New York, NY, USA), p. 1620, Association for Computing Machinery, 2024.
- [57] P4 Consortium, “P4 publications.” <https://p4.org/publications/>. Accessed: 2025-04-11.
- [58] Google, “Google scholar.” <https://scholar.google.com/>. Accessed: 2025-04-11.
- [59] C. Xu, S. Guan, D. Greene, and M.-T. Kechadi, “Benchmark data contamination of large language models: A survey,” 2024.
- [60] M. Chen, J. Tworek, and H. Jun, “Evaluating large language models trained on code,” 2021.
- [61] M. Chen, J. Tworek, and H. Jun, “Evaluating large language models trained on code,” 2021.
- [62] C. An, J. Zhang, M. Zhong, L. Li, S. Gong, Y. Luo, J. Xu, and L. Kong, “Why does the effective context length of llms fall short?,” 2024.

- [63] OpenAI, “Key concepts: Tokens.” <https://platform.openai.com/docs/concepts/tokens>. 2025.
- [64] Hugging Face, “Transformers source code generate function.” <https://github.com/huggingface/transformers/blob/main/src/transformers/integrations/executorch.py#L354>. March 2025.
- [65] OpenAI, “Openai developer platform.” <https://platform.openai.com/docs/overview>. Accessed: 2025-06-23.
- [66] Hugging Face, “Inference endpoints.” <https://huggingface.co/docs/inference-endpoints/index>. Accessed: 2025-06-24.
- [67] D. Han, M. Han, and U. team, “Unsloth.” <https://github.com/unslothai/unsloth>. 2023.
- [68] L. von Werra, Y. Belkada, L. Tunstall, E. Beeching, T. Thrush, N. Lambert, S. Huang, K. Rasul, and Q. Gallouédec, “Trl: Transformer reinforcement learning.” <https://github.com/huggingface/trl>, 2020.
- [69] CINECA, “Leonardo.” <https://www.hpc.cineca.it/systems/hardware/leonardo/>. Accessed: 2025-05-29.
- [70] M. Chiesa, “Semla: Securing enterprises via machine-learning-based automation.” <https://www.kth.se/blogs/semla/>. Accessed: 2025-06-13.
- [71] Schedmd, “Documentation.” <https://slurm.schedmd.com/documentation.html>. Accessed: 2025-06-24.
- [72] X. Liu, H. Yan, S. Zhang, C. An, X. Qiu, and D. Lin, “Scaling laws of rope-based extrapolation,” 2024.
- [73] D. Han-Chen, “Make llm fine-tuning 2x faster with unsloth and trl.” <https://huggingface.co/blog/unsloth-trl>. Accessed: 2025-06-29.
- [74] Hopsworks AI, “Rope scaling.” <https://www.hopsworks.ai/dictionary/rope-scaling>. Accessed: 2025-06-29.
- [75] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, “Codebleu: a method for automatic evaluation of code synthesis,” 2020.

Appendix A

List of Insights

1. P4_16 does not provide built-in protocol headers like `ethernet_t` unless your target or environment defines them explicitly. Always define custom headers (like header `ethernet_t`) unless you're certain they're included from a library. Compiler errors about unexpected IDENTIFIER often mean the type or symbol hasn't been defined yet.
 2. `mark_to_drop` is a deprecated feature, use `mark_to_drop(standard_metadata)` instead.
 3. When combining imperative logic (if statements) with table-driven logic, be clear about the order of execution and overrides. Table applications should not be redundant with conditional logic unless explicitly desired for fallback or priority purposes. Otherwise, streamline to one approach to reduce confusion and potential misconfigurations during table population via the control plane.
 4. The V1Switch architecture expects six specific control blocks: Parser, VerifyChecksum, Ingress, Egress, ComputeChecksum, and Deparser. Omitting one (like ComputeChecksum) will result in a constructor mismatch error.
 5. In P4_16, table hit/miss information is not accessed via a `.hit` field on the table object. Instead, you must assign the return value of `.apply()` to a boolean.
 6. When using a P4 architecture model like `v1model.p4`, avoid redefining control blocks with the same names as those provided by the model. Use unique suffixes or prefixes (e.g., `VerifyChecksumImpl`) to prevent naming collisions.
 7. The P4 architecture model `v1model` exclusively expects headers and metadata as arguments for `ComputeChecksum` and `VerifyChecksum`.
 8. Deparsers must emit headers unconditionally or rely on header validity alone. Do not use if statements inside apply blocks in deparser controls, as many targets disallow conditionals in deparsers.
 9. When casting, ensure that the resulting type exactly matches the target fields declared type. Do not cast to a wider type unless explicitly required by the model.
 10. `multicast_group` is not a valid field in `standard_metadata_t`. The correct field in `v1model.p4` is `mcast_grp`.

11. When designing digest payloads, use fixed-width `bit<>` or `int<>` types for predictable controller parsing. Avoid including user-defined structs or unsupported types directly.
12. When using `default_action`, favor parameterless actions or ensure parameter defaults are provided elsewhere (e.g., via control plane configuration).
13. When a function requires an inout or out parameter in P4, passing a read-only field (like a header field) directly will cause a compiler error. Instead, use a temporary local variable to allow read-write operations.
14. Ensure that the parser and control block signatures conform exactly to the expected types in the architecture package instantiation (e.g., `V1Switch<H, M>`). Mismatches in parameter count or order will lead to type unification errors.
15. Every action parameter in P4 must be purposefully utilized. Declaring unused parameters leads to unnecessary compiler warnings, suggesting poor design or incomplete implementation.

Listing A.1: List of insights generated by GPT-4o.

TRITA-EECS-EX 2025:959
Stockholm, Sweden 2025

www.kth.se