Computer Networks 272 (2025) 111709

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Generating P4 data planes using LLMs

Mihai-Valentin Dumitru *, Vlad-Andrei Bădoiu, Alexandru M. Gherghescu, Costin Raiciu

University Politehnica of Bucharest, Splaiul Independenţei 313, RO-060042, Bucharest, Romania

ARTICLE INFO

Keywords:
P4
Programmable data planes
LLM
Code generation

ABSTRACT

Over the past few years, Large Language Models (LLMs) have become the source of impressive results in code generation. However, most research focuses on widely adopted general-purpose programming languages, with little attention given to niche domain-specific languages (DSLs). This raises the question: do DSLs, such as P4, a data plane programming language, have a place in the LLM world?

The potential impact of generating DSL code could be tremendous. Automatically generating data plane code promises flexible networks that can quickly adapt to specific conditions at the lowest level. P4 is structurally simpler than general-purpose languages, but also offers a much smaller corpus of existing programs, thus setting up interesting challenges for deep-learning based code generation.

In this paper, we show that crafting a highly specialized P4 dataset with domain knowledge is sufficient to bootstrap P4 code generation through fine-tuning existing LLMs, even when they have not encountered P4 code during pre-training. We further document the process of creating a relevant benchmark to assess the proficiency of fine-tuned models in generating P4 code. Our evaluation shows that our fine-tuned models outperform much larger models in both syntactic correctness and semantic alignment.

1. Introduction

The ongoing advancements in the field of LLMs (Large Language Models) have revolutionized how we approach various software engineering tasks, significantly enhancing productivity and innovation in this sector. LLMs, powered by their ability to understand and generate language, are now instrumental in a wide range of applications within software engineering [15], including but not limited to requirements engineering, software design, and software development.

In practice, we find that LLMs perform exceptionally well in generating high-quality code for programming languages that are prevalent in their training datasets. However, their performance significantly declines when dealing with less-represented languages specific to niche domains such as data plane programming. For instance, P4 is absent from major coding datasets like The Stack [17]. Even when such languages are included, they often make up only a small, potentially uncleaned fraction of the data, as in The Stack v2 [22].

In this context, recent studies have shown that the performance of LLMs for code generation is significantly influenced by the quality [16] and characteristics of the dataset, as well as the complexity of the language involved [7]. For instance, the study by Jain et al. [16] highlights how a model trained on a smaller amount of higher-quality data can outperform a model trained on the original larger, non-cleaned dataset. Another notable example is the phi-1.5 model [19], trained exclusively on a textbook dataset, that surpasses in performance models three times its size trained on much larger datasets. These findings underscore the importance of dataset quality over quantity and give us hope that LLMs could be used in domains where they employ domain specific languages with only a few megabytes of available code samples.

We theorize that data plane programming in P4 exhibits the necessary characteristics that may enable models to learn it efficiently given a specialized dataset. The language is simple in structure and very similar to C, meaning that we can transfer C syntax knowledge. There are no loops and the usual P4 code is size-limited by the available memory on the switch. Moreover, due to its non-Turing-completeness, we can leverage verification frameworks [6,20,23,24,30] for P4 to generate higher quality code, without bugs, by connecting their output to a prompt engineering framework. With this work, we pave the way to generate P4 from natural language or specialized text such as RFCs to facilitate networking prototyping, research and development.

In this paper, we develop a specialized dataset for P4-16, by curating code available online and enhancing it with targeted insights. This includes discussions related to P4, networking knowledge in the form of Request For Comments (RFCs), and technical papers to bridge language and networking knowledge. Besides employing state-of-the-art code cleaning techniques [16,17], we synthetically introduce code

* Corresponding author.
E-mail addresses: mihai.dumitru2201@upb.ro (M.-V. Dumitru), vlad_andrei.badoiu@upb.ro (V.-A. Bădoiu), agherghescu2411@upb.ro (A.M. Gherghescu), costin.raiciu@upb.ro (C. Raiciu).

https://doi.org/10.1016/j.comnet.2025.111709
Received 7 February 2025; Received in revised form 14 August 2025; Accepted 9 September 2025
Available online 13 September 2025

comments to better connect natural language descriptions and text implementations. Our work outlines the methodology employed in assembling such a dataset amidst the scarcity of relevant data on the internet. We provide an analysis on how various dataset parameters impact the model's performance, building a comprehensive understanding of the dataset's influence on the performance of the model.

For our evaluation, we employed fine-tuning on small (1–7 billion parameters) open models by training them for code generation in P4. To assess the performance of the trained models, we developed a benchmarking methodology that evaluates a model's effectiveness in data plane programming, even if it does not produce syntactically correct programs. Our results demonstrate that fine-tuning with a specialized P4 dataset is sufficient to generate code that rivals the performance of OpenAI's GPT-4 in P4 code generation, even with far smaller models.

We make the following core contributions:

- A family of specialized, highly-curated datasets for training models on the task of code generation. The datasets, together with the pipeline to generate them are publicly available.[1]
- An open set of autocompletion benchmarks, designed to test a model's P4 generating capabilities. The benchmarks, together with a framework that can automatically evaluate a model's performance on the compilation criterion (how many of the outputs compile) are publicly available.[2]
- A study into how various factors pertaining to the dataset (content, deduplication factor) and factors pertaining to the fine-tuning process (e.g. model architecture, model size) influence the resulting P4 code-generating model.

2. Background

P4 [2] is an open and widely-used language for programmable data planes. It has a C-like syntax, but with much simpler structure. Notably, there are no loops, no jumps, no recursion, no pointers or memory management. P4 data planes consist of a number of programmable blocks, such as ingress and egress control blocks, parsers, deparsers, checksum calculators. The structure and syntactical elements of a control block differ significantly from those of a parser, thus P4 can be thought of as consisting of several specialized sub-languages. Due to the language's young age and restricted scope, there is very little publicly available code.

The original version of the language, P4-14, was soon superseded by P4-16, which abstracted away the switch architecture from the language itself. Architectures specify the configuration and interactions of programmable and non-programmable blocks, mechanisms for forwarding, recirculating and dropping packets, available metadata etc. The standard language itself is not sufficient for development; a P4 program must be written for a particular architecture. In this paper, we will focus on the `v1model` architecture, due to its popularity.

These features make P4 a promising challenge for code generation. The problem of automatically generating P4 code has been tackled before [8,12,14,21,26,29,31,38]. However, to our knowledge, there has been no prior effort to use LLMs for this purpose.

Numerous works employ "classical" methods for generating P4 code. Graph-to-P4 [38] can convert a graph of headers into the code for a P4 parser. Motivated by the idea of moving computation at the edge of the network, P4rrot [12] provides a language for specifying application-level processors which are then automatically translated into P4 in a classical rule-based manner. Lucid [29] allows the programmer to employ powerful abstractions such as "events" and "handlers" for packet processing that help avoid bugs and hide away data plane details. Lucid 2.0 [21] builds on top of this by adding new constructs to achieve *pipeline-safety*. Lyra [8] provides a "*one-big-pipeline*" abstraction to the programmer, allowing them to specify their intent in "simple statements" that then get converted into data plane code for one or more actual devices. Homunculus [31] is a framework that can transform declarative requirements for Machine Learning applications into a concrete implementation for the switching hardware, which may include P4 data planes.

GP4P4 [26] uses *genetic programming* to create P4 data planes starting from "behavioral rules": a set of preconditions and postconditions on the header fields and metadata of the processed packets. A genetic programming approach is also presented in [14], which also employs *federated learning* for a decentralized cross-domain collaboration in acquiring data and learning from it. To the best of our knowledge, ours is the first work specifically exploring LLMs as a medium for automatically generating P4 code.

Focusing on machine learning-based code generation, many open LLMs address the task of code generation [1,10,18,22,28,33,34]. Building these models involves two main steps: pretraining and post-training [45]. During pretraining, models are trained on large corpuses of data, often with lots of code, to learn programming language structure and patterns. This is a form of unsupervised learning, where the model learns to predict the next token in a sequence. The resulting model is called a foundational model, as it serves as a base for further specialization. Post-training then adapts the model for specific tasks. This usually means continuing the training on focused datasets, like Python-only code [28], to improve performance on that language. Finally, models are fine-tuned on instruction datasets with examples of user questions and responses, teaching them how to follow user instructions [46]. Post-training uses much smaller datasets than pretraining, but still needs a lot of computing resources. LoRA [47] offers a solution for limited resources by making fine-tuning more efficient. Instead of updating all model parameters, LoRA freezes the pretraining model weights and only trains a small set of new parameters. QLoRA [5] improves this further by using lower precision numbers for the model weights, which further cuts down memory usage during training.

These models are usually trained on openly available code datasets [9,17,22], which often do not feature P4 samples. One exception is the newly released The Stack v2 [22], the largest publicly available code dataset at this time, which contains samples from over 600 programming languages; but P4 is severely underrepresented, with only 70MB of code. The authors of The Stack v2 refer to Julia and Perl as "low-resource languages", with 6.12 and 7.82 GB of code available.

3. Initial exploration of existing models

The first question we want to answer is whether a very large language model, with hundreds of billions of parameters or more, is proficient in a very low-resource programming language, in our case P4. Secondly, we want to form a baseline on LLM code generation for P4, understand what we can expect from other (smaller) models and, in cases where the output is not satisfactory, understand what exactly goes wrong and theorize about how to improve the results.

To this end we evaluated ChatGPT-3.5, ChatGPT-4 and Gemini Ultra. We further included StarCoder 2 because it was at the time the only model trained on a publicly available dataset that contains P4 code; we choose the largest available version, the 15 billion parameter.

We ran these LLMs once on two variants of a set of tasks derived from the P4 tutorial tasks from the P4 language repo[3]: with and without the code skeleton. The tasks are further described in Section 4. While the models might have been trained on the original P4 tutorials, our introduced changes require an understanding of the P4 language and its semantics to solve. A summary of the results is presented in Table 1, where the comma is used to separate the results between code generation **with** and **without** a skeleton. Introducing the P4-16 manual and

[1] https://huggingface.co/datasets/p4llms/p4dataset
[2] https://github.com/p4llms/benchmarks
[3] https://github.com/p4lang/tutorials

Table 1
Overview of model performance on P4 code generation tasks. There are two scenarios: code completion with a given skeleton and generating from scratch, with no code skeleton, respectively.

Model	# of programs that compile	# of tests that pass	# of additional programs that compile after compiler feedback
ChatGPT-4	9, 1	8, 0	1, 1
ChatGPT-3	1, 0	0, 0	0, 0
Gemini Ultra	0, 0	0, 0	0, 0
StarCoder 2 (15B)	0, 0	0, 0	0, 0

code samples into the model context did not show any significant improvements.

Looking at the generated code, we see that training a very large model on solely openly available code on the Internet is not sufficient for a model to be proficient in a very low-resource programming language and more is needed to improve the performance, which we will explore in the next sections. For the remainder of this section, we discuss the issues with the generated code on the ten tasks.

3.1. OpenAI's ChatGPT

ChatGPT-4 generated code that compiles in 8 cases. By providing the compiler's output in a feedback loop it was able to produce compilable code for one extra task. Two more solutions need only one line of code change: removing a `for` loop over the elements of a header stack in the deparser. Eight out of these nine solutions pass all unit tests relevant to the task.

The tutorial that compiles but fails the tests is the `mri` exercise. A newly added header is not set valid; the generated code is missing a `setValid()` statement explicitly needed in P4-16, as opposed to P4-14. Providing ChatGPT with this comment helps it to generate the fix and pass the tests. Overall, we noticed the following common mistakes in ChatGPT-4's P4 generation:

- header validity checks in the deparser; this is unnecessary and, although not explicitly prohibited by the P4-16 standard, it is not supported on BMv2
- C-style `for` loops in the deparser to emit the elements of a header stack; the correct way to do this is to `emit` the entire header stack
- no explicit casts, between bitfields of different width
- wrong header stack syntax: `type var[10];` instead of `type[10] var;`

Other mistakes done by ChatGPT-4 include: trying to `typedef` an anonymous structure, using character constants and declaring new variables using `var`; none of these are P4 constructs.

The syntactic quality of the generated code dramatically decreases when we remove the skeleton and leave only the task statement: just one solution compiles. Feedback from the compiler doesn't help: for four tasks, it regenerates the exact same error; for three others it misunderstands what the problem is.

One of the most frequent mistakes is placing tables and actions outside the scope of a control block: this is valid in P4-14, but not in P4-16. It was not clear from our experiments with ChatGPT-4 whether this issue stems from the presence of P4-14 programs in the training data; but ChatGPT-3.5 seems prone to utilize the P4-14 syntax in other contexts, such as writing parsers and header definitions.

ChatGPT also shows a lack of familiarity with some target-specific constructs. Tasked to produce code for the `v1model` architecture, it correctly generates the necessary control block signatures and instantiates a 'V1Switch' with the proper parameters. However, it misspells `Register` instead of `register`, doesn't use `hash` but a hypothetical `hash5`, explicitly stating that the concrete hash function depends on the target.

ChatGPT-3.5 generates code with far more errors; even given the skeleton, no solution compiles and it is not able to improve from compiler feedback. It uses P4-14 syntax for the parser much more often; as well as placing tables and actions outside the scope of a control block. At times, it hallucinates headers and data types; one of these being `time_t`, again suggesting that knowledge of other programming language may hinder this task. Even though explicitly tasked to provide a complete P4 program, both versions occasionally fail to do so, instead leaving sections marked as "`TODO`".

3.2. Gemini

Google's Gemini [32] did not yield promising results, but the Ultra version seems at least superficially on par with ChatGPT-4. The web client has an output bug that omits the angle bracket notation that specifies the size of bitstrings, but we assume that these are always correct.

Much like ChatGPT, Gemini mixes in P4-14 syntax for headers and parser, as well as sometimes placing actions and tables outside the scope of a control block. Other shared problems include: validity checks in the deparser, declaring variables with the keyword `var`, not including the proper headers, wrong usage of platform primitives such as hashes and registers.

Much more often, it does not output a complete program even when specifically required to do so, but produces a verbose output with various interspersed snippets. Some mistakes seem to be of a *conceptual* nature, rather than purely syntactical. For example, in several instances, the IPv4 checksum is recalculated in the forwarding action; the TTL is sometimes not decremented.

3.3. Open models

Among the open models we tried, only StarCoder 2 15B is trained on a dataset that contains P4 [22] and is able to produce P4. Unlike ChatGPT, StarCoder is not able to answer instructions and so performs poorly on the tutorial tasks, sometimes extending the text, instead of writing a data plane. However, it produces syntactically correct code, but in many cases it just seems to replicate verbatim code from its training set.

Other state-of-the-art code generators, Magicoder [35], DeepSeek-Coder [11] (7B), OLMo [10], Code LLama [28], Gemma [33] (7B and 13B) are not able to produce satisfactory code; most likely, this is due to P4 missing entirely from their training dataset.

4. P4 benchmarks

In order to compare existing models and assess the effectiveness of various improvement strategies, we need a quality metric for generated code. At the time of writing, even the state-of-the-art models that can generate P4 were unable to consistently output syntactically correct programs. This makes functional evaluation in the style of HumanEval [4] difficult, as we cannot run tests on programs which don't compile.

To explore existing code generators and get a clear view of their capabilities, as well as insights into what could be improved and how, we have created two task categories to evaluate P4 generation capabilities, focusing on code completion and the specific intricacies of data plane programming. The benchmarks cover many language constructs related to P4, as well as constructs specific to the `v1model` architecture (forwarding, dropping, registers etc.). There are some features of P4, such as header stacks and header arrays, that are not covered by our benchmarking set, which we plan to extend in the future.

4.1. Code generation for natural language prompts

We created the first tasks from a cleaned-up, modified versions of the problem statement for the following P4 tutorials: basic forward, basic tunneling, ECN, MRI, source routing, calculator, load balancing, QOS, firewall and link monitor. Our changes consists of adding new tasks, changing the code structure and updating the task instruction. For each tutorial, we have a version including the starting code and one without.

The P4 tutorials, together with the reference solutions are very likely part of models' training sets. Our changes aim to avoid potential "copy-paste" solutions; but even after the changes, the resulting tasks are similar to the original ones; thus correct solutions alone are not good indicators of generative capabilities. But failure of solving this task is a good indicator of the model's low capabilities in producing and understanding P4 code.

These tasks rely on English-language specifications of the desired functionality and as such are only suitable for chat models (or instruct models) that operate in a Q&A mode. To evaluate autocompletion models, we skip these tests but create a separate benchmarking set. The benchmarks, together with a testing harness are publicly available[4].

4.2. Autocompletion benchmarks

We manually create a set of P4 autocompletion benchmarks, suitable for base models that have not been further trained on instruct tasks. Our benchmarks generally require the model to fill-in some code at the "block" level (e.g. a parser state, a table, an action body); indications about the desired code are given in the form of code context (such as an action's signature, or previous header declarations) and English-language comments.

We write our tests as complete P4 programs, accompanied by some preprocessing metadata incorporated in comments. This metadata consists of several tags that identify the start and end of the section that should be included in the prompt, as well as of the section that the model should complete, together with a specification of how to extract a limited snippet from the model's output. This is necessary because autocompletion models very often simply generate as many tokens as possible, following up on the "useful" part with arbitrary code.

As a result of our design, each test file starts out as a complete and correct P4 program, that implements all the desired functionality. During preprocessing, a key segment of the file is removed (this can be considered a reference answer) and the model is presented with a prefix of the code present before the removed segment. In the future, we intend to explore the technique of "fill-in-the-middle" [39] to pass to the model not just the context before the "hole", but also the one after it.

A few example tasks:

- given some new header `foo` and macro definitions of the form `IP_PROTO_FOO`, as well as a partial parser that deals with Ethernet, followed by IPv4, the model is expected to complete the `parse_ipv4` state.
- given some new header definitions and a parser implementation, the model is expected to generate the deparser
- given the name and comment-description of an action (e.g. `action diff(in bit<32> a, in bit<32> b, out bit<32> result)`, the model is expected to complete the action body

After getting the response from the model, we extract the relevant part of the output and stitch it together with the other parts of the initial code, to obtain a complete program. This allows us to automatically run the `p4c` compiler on the resulting file to check if it compiles.

For some programs, we also define a set of unit tests, to check functional correctness. Doing this in general, for all benchmarking tasks is a non-trivial task. The testing framework does not only need to create input packets and check some reference criteria on output packets; it also needs to mock-up a control plane, inserting table rules or checking the value of stateful objects (registers, meters, counters). This is done through an API that is a result of compilation and so cannot be known in advance: what tables are available and what are their names, what actions they have and what are the action names etc. Some benchmarks give the model freedom over these elements and their names, making it difficult to set up static entries in advance. We add unit tests to 11 out of our 50 tests, leaving the rest as future work.

[4] https://github.com/p4llms/benchmarks

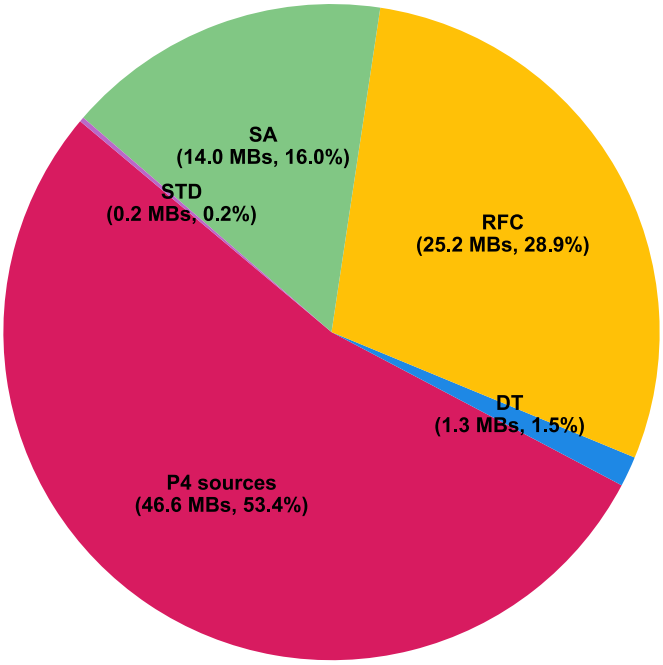


Fig. 1. Composition of the dataset: number of P4 source files, RFCs, P4 forums discussion threads (DT), scientific articles (SA) and the P4-16 standard (STD).

5. Constructing a high-quality P4 dataset

Improving a model performance on a specific task or domain is most commonly achieved through fine-tuning, which requires a high quality, task-specific dataset. To this end, we curated a dataset specifically for the task of P4 code generation; it includes code, relevant knowledge on networking for dataplane programming, and a glue between the two. An overview of the final dataset's composition can be consulted in Figure 1.

5.1. Data crawling

We started by gathering all the P4 repositories available on GitHub, using the language filter available in the public API[5]; we obtain `786` repositories. We extract from these all files with a `.p4` or `.P4` extension that can be UTF-8 decoded, obtaining `42,588` files totalling `211` MBs.

We further include sources from the Software Heritage project[6] via The Stack v2 [22]: `7,453` documents, totalling `49` MBs.

We then subject this collection of `50,041` documents to a commonly used cleaning pipeline for code [18].

5.2. Removing P4-14 files

"P4" actually refers to two quite different languages. P4-14 [41], the original version of the language, first released in 2014 and P4-16 [40] – a replacement that followed just two years after. The two languages differ drastically, both in overall structure and in specific syntactic elements, such as headers, parsers and control blocks. Our experiments with existing models described in Section 3 show that one of the reasons for poor performance was the occasional employment of P4-14 syntactical constructs, so we decide to remove P4-14 files from our dataset.

[5] https://docs.github.com/en/search-github/github-code-search/understanding-github-code-search-syntax#language-qualifier

[6] https://www.softwareheritage.org/

We attempt to automatically identify P4-14 files by employing regex-based searches that identify specific keywords (e.g. `fields`, `reads`) or constructs (e.g. `control ingress`) in syntactical contexts that can only exist in P4-14 and not in P4-16. In total, this step removes `6,580` documents totalling `47` MBs.

5.3. Removing non-P4 files

We further remove from the dataset mislabeled files which don't contain P4 code; we do this through heuristic regexes, as well as by manually blacklisting some files. We remove several files which were marked as having been auto-generated and one which was created by the `p4obfuscator`. We also remove files smaller than 10 bytes (including empty files) and those larger than 300 KBs. One of the main difficulties in creating the dataset is that quite a substantial number of P4 files appear to have at least part of them programmatically generated. We choose to keep most of these, only discarding the files which don't contain any other syntactic constructs besides a long list of lines differing in a few characters/numbers.

This filtering step discards `1,331` documents, totalling `57` MBs.

5.4. Deduplication

We employ MinHash [3] to filter near-duplicates with a 0.85 Jaccard similarity threshold. This removes `31,151` documents, totalling `102` MBs. This reduces the number of documents by a factor of four. The resulting collection forms the first version of our dataset which we later employ for fine-tuning; we call this the "`code`" dataset.

5.5. Networking knowledge

In addition to generating syntactically valid code, a successful code-generator should be able to map natural language instruction to the target programming language. To achieve this, we introduce the relevant knowledge and glue between knowledge and code in several ways. First, we extend the dataset with networking knowledge about protocols in the form of RFCs, from RFC 791 to 2000. We then add `381` entries from the P4 discussion forums[7]. We add the contents of `368` scientific articles on P4, mainly collected from an extensive survey [13] as well as the May 2023 revision of the P4-16 standard. We extract the content texts using the tool presented by Yu et al. [36].

The resulting collection forms the second version of our dataset, the "`+text`" dataset.

5.6. Adding code comments

One of our aims is for our model to be usable for auxiliary tasks including adding code comments and refactoring. We leverage a teacher LLM to enhance the code portion of our dataset, by synthetically introducing new comments, summarizing existing functionality. We use ChatGPT-3.5 due to its abundance of domain knowledge, fast response time and similar quality of comments with ChatGPT-4 at this task. We picked source files with lower than 10 comments and added comments to 18 % of the source files.

The resulting collection forms the third and final version of our dataset, the "`+comments`" dataset.

6. Training

6.1. Fine-tuning models with QLoRA

We started by fine-tuning several models, all with a context length of 2048: the 1B version of StarCoder [18], the 2B version of Gemma [33] and the 3B version of StarCoder 2 [22].

[7] https://forum.p4.org/

At this step we wanted to determine the lower bound of what can be achieved without having access to more expensive hardware for training. To this end, we used QLoRA [5] to fine-tune the models on a single Nvidia A100 40GB GPU. We used a LoRA scaling factor *alpha* of 32 and a rank r of 8. Based on several ablation experiments, we picked a learning rate of 10^{-4}, to encourage slow domain-adaptation and prevent catastrophic forgetting, and a weight decay of 0.01. The batch size was kept constant at 32, and the learning rate was annealed by a cosine schedule, with an initial warm-up of 5 % of the total number of steps. We ran the training for a single epoch for all models, as we feared overfitting might quickly become a problem, because the dataset contains a lot of repetitive boilerplate, common in the P4 programs. The number of trainable parameters through QLoRA amounted to less than 0.5 % of the model's total number of parameters, which allowed for short fine-tuning times: around four hours on a single Nvidia A100 40GB for the StarCoder 2 3B on the "`+comments`" dataset.

In Figure 2 we plotted the training loss of these three models on the "`+comments`" dataset. The loss plot shows no indication of overfitting, which is further confirmed by our accuracy evaluation described in Section 7.

6.2. Fully fine-tuning a larger model

Next, we fully fine-tuned the 7B variant of StarCoder 2 [22]. We used no training optimization such as QLoRA. In total, we used 8 Nvidia H100 80GB GPUs, with a batch size of 4 per GPU. Because code comments did not seem to make a significant difference in the evaluation of smaller models, we decided to use the "`+text`" dataset. We trained for 5 epochs, with early stopping based on the validation loss; training was stopped after the fourth epoch; the whole process took slightly under 15 min to complete.

Figure 3 shows the training loss during the fine-tuning process, with no indication of overfitting.

7. Manual evaluation

We initially ran all the QLoRA fine-tuned models on a (earlier) subset of the autocompletion benchmark described in Section 4, consisting of just 15 tasks. We used ChatGPT-4 and Gemini Ultra as baselines, prompting them to act as autocomplete tools. We evaluated only the completion capabilities of the models because our dataset does not target instruct models: they don't interpret text prompts as tasks that need to be solved, but as streams of text that need to be continued, which they sometimes do by emitting additional requirements, examples or diagnostics.

We performed the postprocessing manually extracting the relevant part from the model's output, depending on the context: if the model was supposed to autocomplete the definition of a parser state, we delete everything that follows that parser state, before assessing the correctness of the answer. We evaluated the following characteristics of the output code:

Does it compile? We took the postprocessed output of the model and insert it into a minimal, complete P4 program template, then run it through the `p4c` compiler. We ignore warnings and consider the task successful if there are no compilation errors.

Is it logically coherent? Compilation alone is not sufficient. In one case, the model kept generating a long list of variable declarations; while syntactically correct, these did not address the task at hand. This requirement is also failed by the presence of any syntax error, with the exception of size mismatches (using the wrong hash size, implicit casts, etc.). We allow these errors because they seem "light", in the sense that they could be easily fixed by additional tools.

Does it implement all functionality requested? We used this to distinguish proper solutions from *partial solutions* which, although coherent and free of compilation errors, do not contain all the code necessary to solve the task.

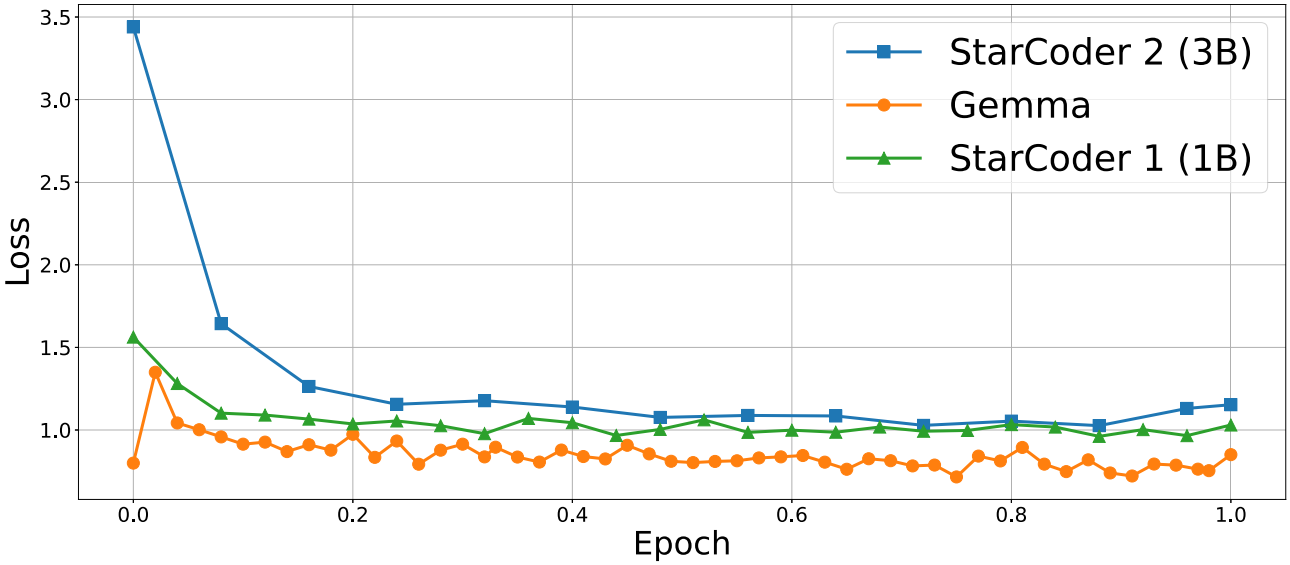


Fig. 2. Training loss during the fine-tuning process on the "`+comments`" dataset.

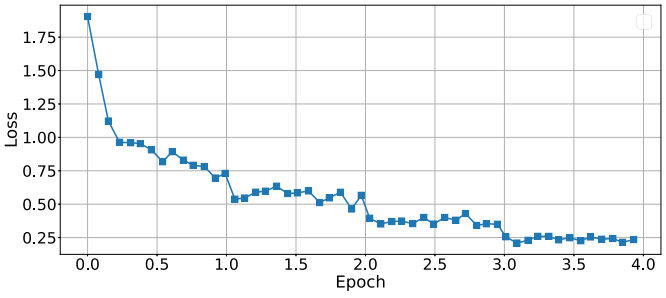


Fig. 3. Training loss during the fine-tuning process on the "`+text`" dataset.

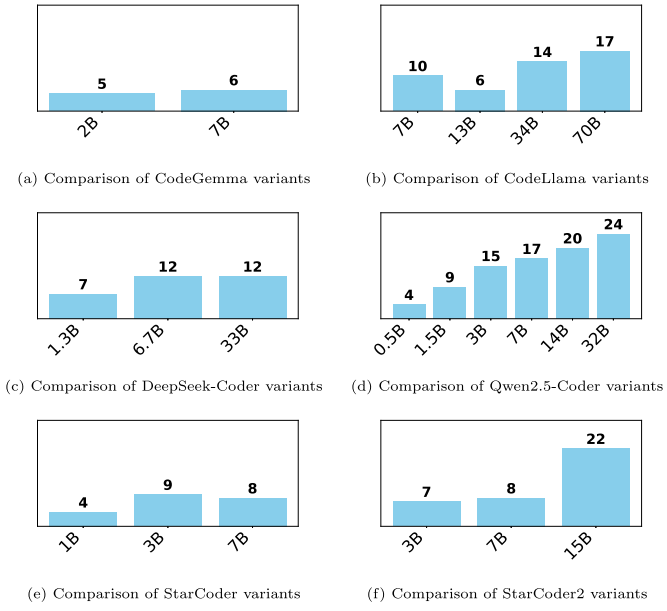


Fig. 4. Comparison of various architecture at different sizes (numbers of parameters). Number of tasks that compile.

Table 2
Results of the fine-tuned models in the autocompletion benchmark consisting of 15 tasks. Gemini Ultra and ChatGPT-4 are used as baselines.

Model	Dataset	Compiles?	Coherent?	All?
SC 1B	code	9	8	7
SC 1B	+text	11	8	8
SC 1B	+comments	12	9	6
Gemma 2B	code	9	10	9
Gemma 2B	+text	11	9	8
Gemma 2B	+comments	11	8	8
SC2 3B	code	12	10	8
SC2 3B	+text	10	10	8
SC2 3B	+comments	12	9	8
Gemini Ultra		5	4	4
ChatGPT-4		6	7	7

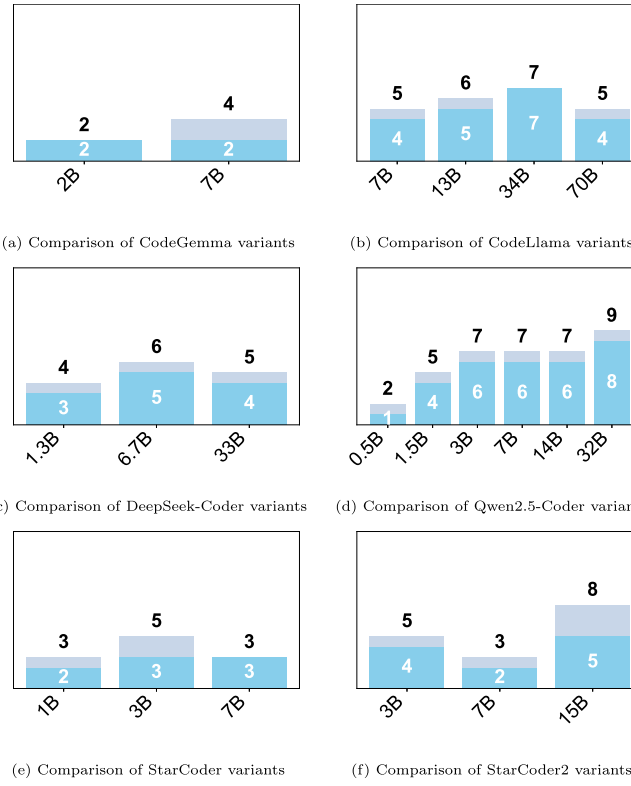
The results are summarized in Table 7. Compared to the fine-tuned models, ChatGPT-4 makes more severe syntactical errors, such as including `for` loops in the deparser. This supports the idea that dataset quality is a key part of a model's performance. Gemini generates syntax errors more often and sometimes more severe, in one case omitting a closing brace for the parser block. For some tasks, it produced no relevant code.

Five tasks were solved correctly by all models-under-test. These involved writing two simple actions to calculate the maximum and sum of their arguments respectively. The other three required filling in a table's action list, parsing a new type of header, and writing a simple deparser.

The notable difference between models trained on our dataset and ChatGPT-4 lies in the deparser: for 3 more tasks, our models produce a deparser that compiles; ChatGPT often messes up the syntax, employing loops and validity checks.

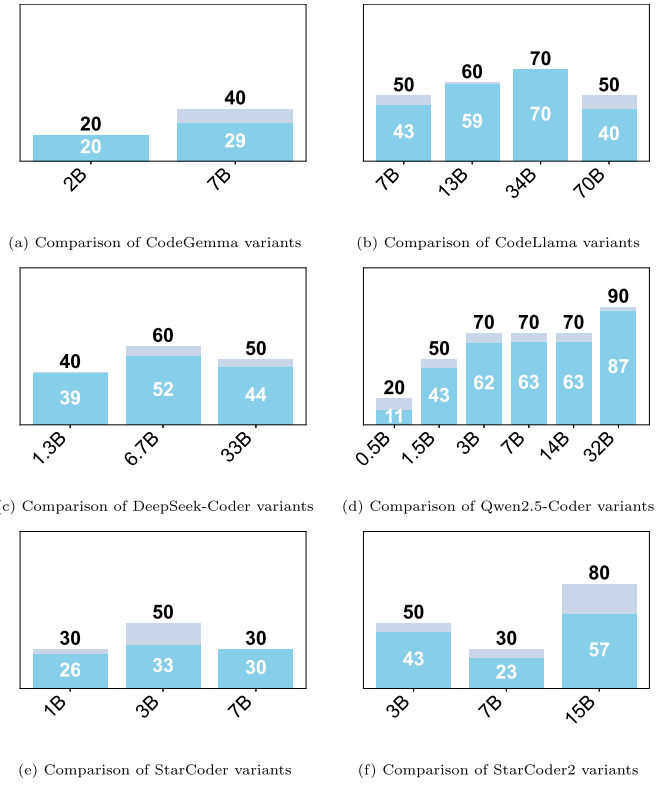
Other syntax errors performed by ChatGPT-4, such as declaring actions outside the scope of a control block, or using non-boolean expressions as conditions for an `if`, are not present in our models' output and are likely a result of having P4-14 sources mixed into the training set of ChatGPT-4.

Enhancing the dataset with text sources and synthetic comments does not have a noticeable effect.



(a) Comparison of CodeGemma variants (b) Comparison of CodeLlama variants (c) Comparison of DeepSeek-Coder variants (d) Comparison of Qwen2.5-Coder variants (e) Comparison of StarCoder variants (f) Comparison of StarCoder2 variants

Fig. 5. Comparison of various architecture at different sizes (numbers of parameters). Number of tasks for which unit tests were run (faded bar) and number of tasks for which all unit tests pass (dark bar). There are 11 tasks which each have 10 unit tests; if the model's output for a task doesn't compile, the tests are not run.



(a) Comparison of CodeGemma variants (b) Comparison of CodeLlama variants (c) Comparison of DeepSeek-Coder variants (d) Comparison of Qwen2.5-Coder variants (e) Comparison of StarCoder variants (f) Comparison of StarCoder2 variants

Fig. 6. Comparison of various architecture at different sizes (numbers of parameters). Number of unit tests that were run (faded bar) and number of unit tests that pass (dark bar). There are 11 tasks which each have 10 unit tests; if the model's output for a task doesn't compile, the tests are not run.

8. Automatic evaluation

Manual evaluation is slow and prone to human errors. Automated testing frameworks allow for a larger number of benchmarks, more models tested, and faster prototyping. We expanded our benchmark suite to 50 tasks and developed an automatic testing framework as described in Section 4. For each task, we check whether the generated output compiles without errors. For 11 tasks, we have 10 unit tests that validate functional correctness; if the generated code compiles, we subject it to these unit tests. A task is passed only if all of its 10 unit tests are passed. For some tasks, functional testing requires custom control plane entries; but the exact tables, table keys, actions and parameters cannot be known in advance (the LLM is free to create any number of tables with arbitrary features). This is why not all tasks have corresponding unit tests.

When testing a model, we run it on all 50 tasks and look at the following metrics:

1. For how many tasks does the model produce output that compiles without error?
2. Out of all tasks with unit tests, for which the model produces code that compiles without error, how many unit tests pass?
3. Out of all tasks with unit tests, for which the model produces code that compiles without error, how many tasks pass all their unit tests?

We evaluated a larger number of models:

- ChatGPT-4o and ChatGPT-4o-mini [42]
- Codegemma [43]: 2B, 7B, 1.1-2B, 1.1-7B-it
- CodeLLama [27]: 7B, 13B, 34B, 70B
- deepseek-coder [11]: 1.3B, 6.7B, 33B
- Qwen2.5-Coder [45]: 0.5B, 1.5B, 3B, 7B, 14B, 32B
- StarCoder [18]: 1B, 3B, 7B
- StarCoder2 [22]: 3B, 7B, 15B
- WizardCoder [44]: 15B

For those architecture which are available in different sizes, this allows us to observe the relationship between model size and performance. Figures 4-6 illustrate the general trend towards larger versions of the same architecture performing better. Figures 4a-4e illustrate the number of outputs that compile; Figures 5a-5e show the number of tasks for which all unit tests passed; Figures 6a–6e show the number of unit tests run and the number of unit tests passed, respectively.

We note that model performance is not monotonous across different sizes for some architectures. We believe this variation stems from the non-deterministic character of code generation, as well as the distribution of data that these models have seen during training. Larger variants are trained on larger datasets; if the extra data is similar to P4, or at least has the same distribution as the smaller set, an increase in performance can be expected. Conversely, extra data could cause the distribution to shift away from P4-like syntax and worsen performance. The precise dataset on which each variant was trained is not public knowledge.

Figure 4e shows a big difference in number of successful compilation between the 7B and 15B versions of StarCoder 2. The 3B and 7B versions are trained on just 17 programming languages, while StarCoder 2 is trained on all 600+ programming languages from the Stack v2, including P4 [22].

Figures 7-9 show a comparison between all models tested of around 7B parameters. Looking at the results, we conclude that our fine-tuned version of starcoder2-7B outperforms all other models of its size.

A detailed comparison is presented in the next figures. Figure 7 shows the number of tasks for which the output compiles without

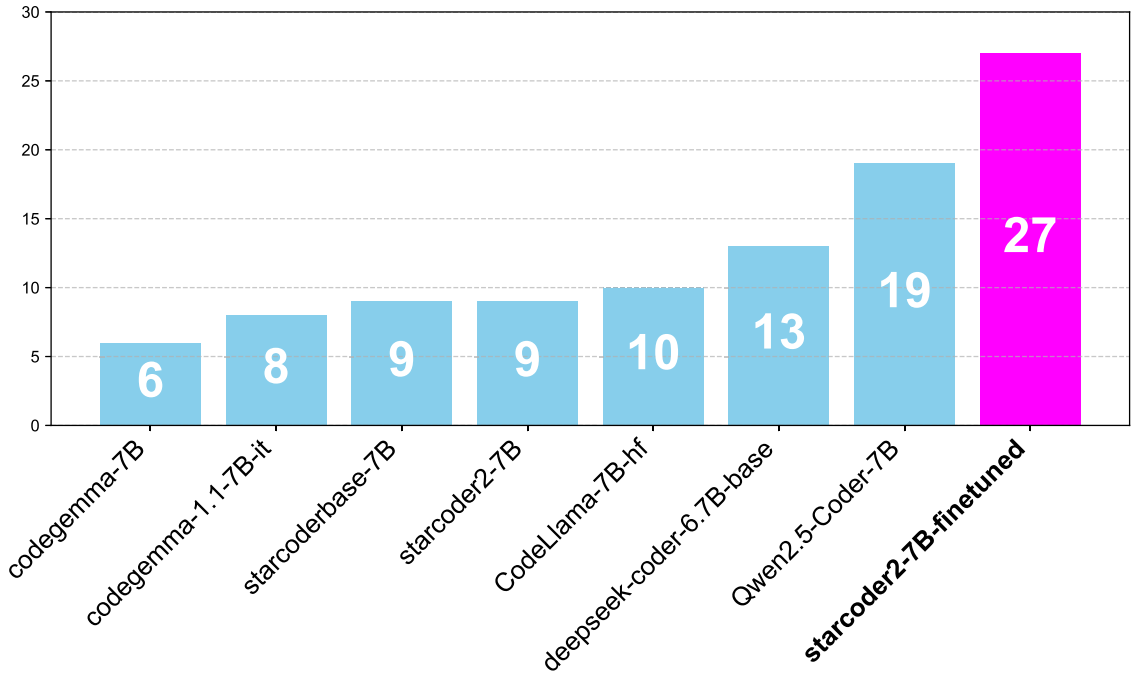


Fig. 7. Comparison between all tested models around 7B parameters: number of tasks for which the code generated compiles. `starcoder2-7B-fine-tuned` is our fine-tuned model described in Section 6.2.

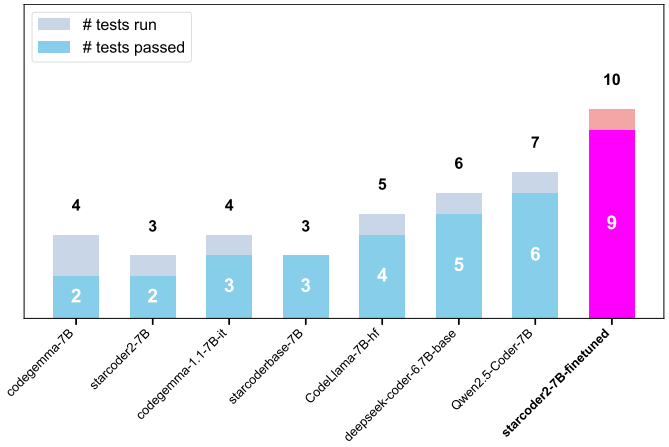


Fig. 8. Comparison between all tested models around 7B parameters: number of tasks for which unit tests were run (light bars) and number of tasks for which all unit tests passed (dark bars). `starcoder2-7B-fine-tuned` is our fine-tuned model described in Section 6.2.

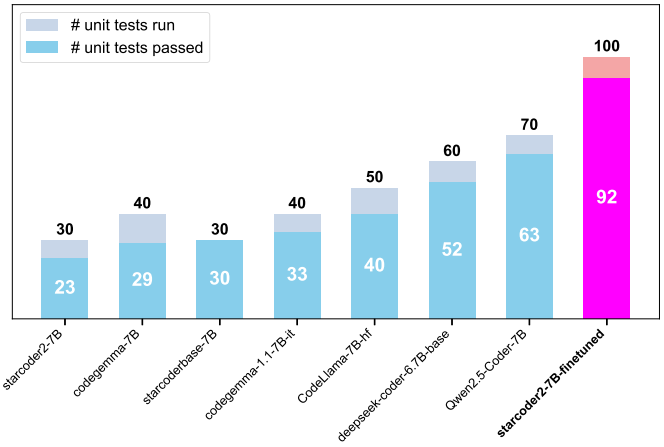


Fig. 9. Comparison between all tested models around 7B parameters: number of unit tests run (light bars) and number of unit tests passed (dark bars). `starcoder2-7B-fine-tuned` is our fine-tuned model described in Section 6.2.

errors. Figure 8 shows the number of tasks for which unit tests were run and the number of tasks for which all unit tests passed, respectively. Figure 9 shows the number of unit tests run and the number of unit tests passed, respectively. In all experiments, we can see that the fine-tuned version passes the most tasks.

Looking at the broader picture, we see that our fine-tune model is close in performance to much larger models, outperforming the 32B variant of Qwen2.5-Coder and being really close to ChatGPT-4o. Figures 10-12 show a comparison between the best models under test. Figure 10 shows the number of tasks for which the output compiles without errors. Figure 11 shows the number of tasks for which unit tests were run and the number of tasks for which all unit tests passed, respectively. Figure 12 shows the number of unit tests run and the number of unit tests passed, respectively. For the ChatGPT variants the actual number of parameters is not publicly available; the other columns indicate that our model's performance is comparable with much larger models.

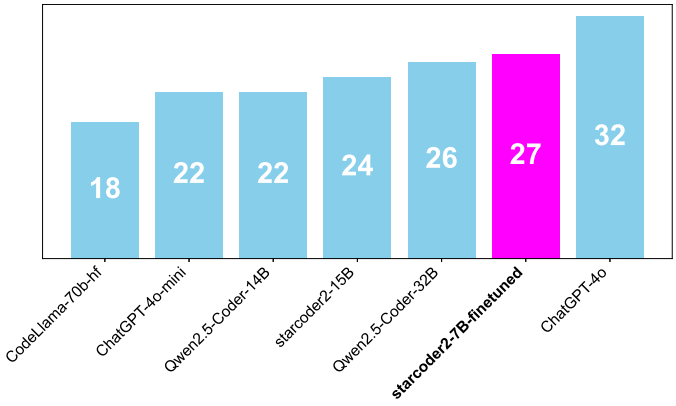


Fig. 10. Comparison between the best tested models: number of tasks for which the code generated compiles. `starcoder2-7B-fine-tuned` is our fine-tuned model described in Section 6.2.

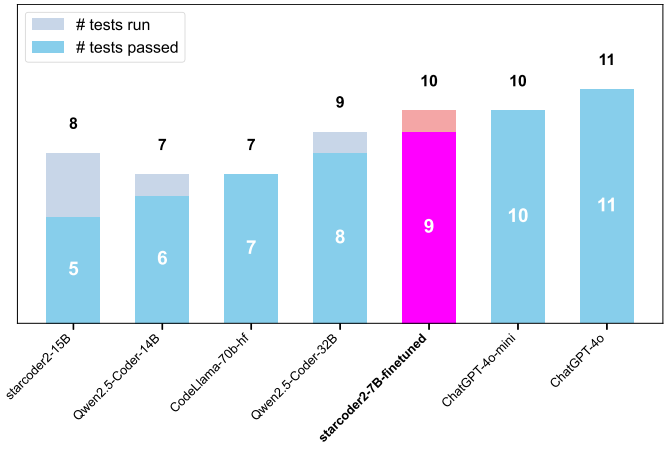


Fig. 11. Comparison between the best tested models: number of tasks for which unit tests were run (light bars) and number of tasks for which all unit tests passed (dark bars). `starcoder2-7B-fine-tuned` is our fine-tuned model described in Section 6.2.

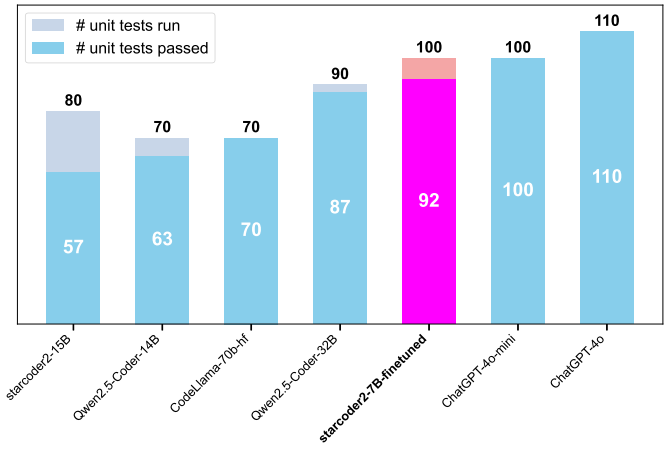


Fig. 12. Comparison between the best tested models: number of unit tests run (light bars) and number of unit tests passed (dark bars). `starcoder2-7B-fine-tuned` is our fine-tuned model described in Section 6.2.

9. Discussion and future work

In §6 we presented our experiments of fine-tuning small, open models (ranging from 1B to 7B) on a curated dataset, both partially using QLoRA and fully. As we show in Section 8, larger versions of the same architecture tend to perform better on downstream tasks. Existing models seem to improve way past the point of 7B parameters; an immediate next step would be to fine-tune even larger models.

We consider the main limitation of this fine-tuning process to be the small size of the dataset; an idea worth exploring is using a larger, more capable LLM to generate synthetic data that could then be used to augment the dataset, which could serve as a better starting point for a smaller model.

We also plan to train an "instruct" [25] version of our model, which could generate P4 code based on natural language descriptions of the desired functionality. Evaluation of this model would require an additional benchmark suite. The results in §7 indicate that text resources and synthetic comments added to the dataset do not yield noticeable improvements in model performance. However, the benchmark tasks are focused on code-completion; an "instruct" model would allow us to explore natural language tasks, for which these dataset enhancements might prove useful.

Automatic code generation is a stepping stone towards developing a robust, adaptable network, capable of rewriting itself at the lowest level, similar to the ideas presented in [26,31,37]. The code produced should not only be syntactically correct, but also efficient and secure. There is already a substantial amount of literature on employing P4 as the low-level backend for languages that can operate with complex abstractions over a single switch or over an entire network [8,12,21,29,38]. Instead of producing raw P4 code, a model could be trained to generate programs in such higher-level languages.

10. Conclusions

In this paper, we have shown that there is hope for low-resource DSLs in the world of LLM code generation. We focus on P4, a data plane programming language that is severely underrepresented in open datasets and for which even very large language models struggle to generate syntactically correct code. Our work proves that good P4 code generation is possible by fine-tuning models on high-quality, carefully curated datasets, bringing us closer to the goal of automatically generating data plane code for flexible networks.

We make four main contributions. First, we present an exploration of existing models' capabilities at the task of generating P4 code, together with an analysis of their shortcomings. Secondly, using the insights from this endeavour, we developed and publicly released a complete set of P4 datasets along with the tools to build them. Thirdly, we built a benchmarking framework based on program compilation, with 50 tests that can measure how well models generate P4 code. Lastly, our experiments show that models fine-tuned on our datasets drastically improve the quality of generated P4 code, with small models being comparable with much larger state-of-the-art models.

The evaluation confirms that our approach works and points to ways to further improve P4-generating models as discussed in the last part of the paper. The performance gains we achieved suggest that using larger models could lead to even better results. By providing both the datasets and testing tools needed for progress, this work lays the groundwork for the growing field of automated data plane code generation – an important step toward truly programmable networks.

CRediT authorship contribution statement

Mihai-Valentin Dumitru: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization; **Vlad-Andrei Bădoiu:** Writing – review & editing, Writing – original draft, Validation, Software, Methodology, Investigation, Formal analysis, Data curation; **Alexandru M. Gherghescu:** Writing – original draft, Validation, Software, Methodology, Formal analysis, Conceptualization; **Costin Raiciu:** Writing – review & editing, Writing – original draft, Validation, Supervision, Project administration, Methodology, Funding acquisition, Formal analysis, Conceptualization.

Data availability

Our data is available on public repositories, linked from the paper's body.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research was supported by the project "Romanian Hub for Artificial Intelligence – HRIA", Smart Growth, Digitization and Financial Instruments Program, 2021–2027, MySMIS no. 334906. UPB authors were partly supported by VMWare gift funding. The authors would like

to thank Marta Chinnici and ENEA Italy for providing us with access to the Nvidia A100 card on which we fine-tuned our first models.

Appendix A. Prompts

We exemplify here one of our 50 autocompletion benchmarks presented in 4.2. The program receives packets consisting of a single header with three fields; it adds the first two and places their sum into the third field then returns the packet on the port from which it came.

There is some metadata for our custom parser, inserted as comments. At the top of the program, the block between "START" and "END" informs the parser to take the LLM response and extract everything from the first character, until a closing brace }, not opened in the output itself (the opening brace is part of the prompt; the model output can contain other paired braces).

The model will be given all the code from the next line after "END" until "// <PROMPT_END>" as the prompt. To form a complete program, this prompt will be concatenated with the extracted model output and everything from "// <RESPONSE_END>" until the end of the file.

Because the metadata is presented as valid P4 comments, this template file acts as a standalone reference implementation (note that the LLM does not see the reference implementation of whatever it is prompted to generate).

```
// START
// parser: "find_group_end"
// arguments: 2
// "delimiter": "}"
// "opening_in_prompt": true
// END
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>


struct metadata {
    /* empty */
}

header math_h {
    bit<32> a;
    bit<32> b;
    bit<32> result;
}

struct headers {
    math_h math;
}

control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {
    // Calculate the sum of a and b and put it in result
    action sum(in bit<32> a, in bit<32> b, out bit<32> result) {
// <PROMPT_END>
        result = a + b;
```

```
    }
// <RESPONSE_END>

    table compute {
        actions = { sum(hdr.math.a, hdr.math.b, hdr.math.result); }
        default_action = sum(hdr.math.a, hdr.math.b, hdr.math.
    result);
    }

    action send_back() {
        standard_metadata.egress_spec = standard_metadata.
    ingress_port;
    }

    table t_send_back {
        actions = { send_back(); }
        default_action = send_back();
    }

    apply {
        compute.apply();
        t_send_back.apply();
    }
}


parser MyParser(packet_in packet,
                out headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata) {

    state start {
        packet.extract(hdr.math);
        transition accept;
    }
}

// ...
// Boilerplate code omitted for brevity.
// In order to have a fully-working P4 program, we need additional
// control blocks for egress processing, deparsing, checksum
// verification and computation, plus to instantiate a V1Switch.
```

References

[1] E. Almazrouei, et al, The falcon series of open language models, Technical report, arXiv preprint, 2023.

[2] Programming protocol-independent packet processors, ACM SIGCOMM Comput. Commun. Rev. 4 (3) (2014) 87–95. Pat Bosshart et al.

[3] A.Z. Broder, Identifying and filtering near-duplicate documents, in: Annual Symposium on Combinatorial Pattern Matching, Springer, 2000, pp. 1–10.

[4] M. Chen, J. Tworek, H. Jun, Q. Yuan, H.P. D.O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, Evaluating large language models trained on code, Technical Report, arXiv preprint, 2021.

[5] T. Dettmers, A. Pagnoni, A. Holtzman, L. Zettlemoyer, QLoRA: efficient finetuning of quantized LLMs, Adv. Neural Inform. Process. Syst. 36 (2024).

[6] D. Dumitrescu, R. Stoenescu, bf4: towards bug-free p4 programs, in: Proceedings of the 2020 ACM SIGCOMM, the 2020 ACM SIGCOMM, 2020, pp. 571–585. Lorina Negreanu, and Costin Raiciu.

[7] R. Eldan, Y. Li, Tinystories: how small can language models be and still speak coherent English, Technical Report, arXiv preprint, 2023.

[8] J. Gao, et al, Lyra: a cross-platform language and compiler for data plane programming on heterogeneous asics, in: Proceedings of the 2020 ACM SIGCOMM, the 2020 ACM SIGCOMM, 2020, pp. 435–450.

[9] L. Gao, et al., The pile: an 800gb dataset of diverse text for language modeling, Technical Report, arXiv preprint, 2020.

[10] D. Groeneveld, et al, Olmo: accelerating the science of language models, Technical Report, arXiv preprint, 2024.

[11] D. Guo, et al., Deepseek-coder: when the large language model meets programming – the rise of code intelligence, 2024.

[12] C. Györgyi, S. Laki, S. Schmid, P4rrot: generating P4 code for the application layer, ACM SIGCOMM Comput. Commun. Rev. 53 (1) (2023) 30–37.

[13] F. Hauser, et al, A survey on data plane programming with P4: fundamentals, advances, and applied research, J. Netw. Comput. Appl. 212 (2023) 103561.

[14] O. Hireche, C.B. Ä, T. Taleb, Deep data plane programming and ai for zero-trust self-driven networking in beyond 5g, Comput. Netw. 203 (2022) 108668.

[15] X. Hou, et al., Large language models for software engineering: a systematic literature review, Technical Report, arXiv preprint, 2023.

[16] N. Jain, T. Zhang, W.-L. Chiang, J.E. Gonzalez, K. Sen, I. Stoica, Llm-assisted code cleaning for training accurate code generators, Technical Report, arXiv preprint, 2023.

[17] D. Kocetkov, et al, The stack: 3 TB of permissively licensed source code, Technical Report, arXiv preprint, 2022.

[18] R. Li, et al., Starcoder: may the source be with you! arXiv preprint, 2023.

[19] Y. Li, S. Bubeck, R. Eldan, A.D. Giorno, S. Gunasekar, Y. Tat, Lee, Textbooks are all you need II: phi-1.5 technical report, Technical Report, arXiv preprint, 2023.

[20] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Caşcaval, N. Mckeown, N. Foster, P4v: practical verification for programmable data planes, in: Proceedings of the 2018 Conference of the Conference of the ACM Special Interest Group on Data Communication, 2018, pp. 490–503.

[21] D. Loehr, D. Walker, Proceedings of the safe, modular packet pipeline programming, ACM Program. Lang. 6 (2022) 1–28.

[22] A. Lozhkov, et al, Starcoder 2 and the stack v2: the next generation, 2024.

[23] M. Neves, L. Freire, A. Schaeffer-Filho, M. Barcellos, Verification of P4 programs in feasible time using assertions, in: Proceedings of the 14th ACM CoNEXT, the 14th ACM CoNEXTNew York, NY, USA, ACM, 2018, pp. 73–85.

[24] A. Nötzli, J. Khan, A. Fingerhut, C. Barrett, P. Athanas, P4pktgen: automated test case generation for p4 programs, 5 of *the Symposium on SDN Research, SOSR '18New York, NY, USA*, ACM, 2018.

[25] L. Ouyang, et al, Training language models to follow instructions with human feedback, Adv. Neural Inform. Process. Syst. 35 (2022) 27730–27744.

[26] M. Riftadi, J. Oostenbrink, F. Kuipers, Gp4p4: enabling self-programming networks, Technical Report, arXiv preprint, 2019.

[27] et al Baptiste Roziere, Code llama: open foundation models for code, Technical Report, arXiv preprint, 2023.

[28] et al Baptiste Rozière, Code llama: open foundation models for code, 2024.

[29] J. Sonchack, D. Loehr, J. Rexford, D. Walker, Lucid: a language for control in the data plane, in: Proceedings of the 2021 ACM SIGCOMM 2021 Conference , 2021, pp. 731–747.

[30] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, C. Raiciu, Debugging P4 programs with Vera, in: Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication 2018, pp. 518–532.

[31] T. Swamy, A. Zulfiqar, L. Nardi, M. Shahbaz, K. Olukotun, Homunculus: auto-generating efficient data-plane ml pipelines for datacenter networks, Proceedings of the 28th ACM ASPLOS 3 (2023) 329–342.

[32] G. Team, et al Gemini, A family of highly capable multimodal models, 2023.

[33] G. Team, T. Mesnard, C. Hardin, R. Dadashi, S. Bhupatiraju, S. Pathak, L. Sifre, M. Rivière, M.S. Kale, J. Love, et al Gemma, Open models based on Gemini research and technology, Technical Report, arXiv preprint, 2024.

[34] H. Touvron, et al, Llama 2: open foundation and fine-tuned chat models, Technical Report, arXiv preprint, 2023.

[35] Y. Wei, Z. Wang, J. Liu, Y. Ding, L. Zhang, Magicoder, Source code is all you need, Technical Report, arXiv preprint, 2023.

[36] C. Yu, C. Zhang, J. Wang, Extracting body text from academic pdf documents for text mining, Technical Report, arXiv preprint, 2020.

[37] L. Yu, J. Sonchack, V. Liu, Mantis: reactive programmable switches, in: Proceedings of the 2020 ACM SIGCOMM, the 2020 ACM SIGCOMM, 2020, pp. 296–309.

[38] O. Eder, Z. Zaballa, Zhou, Graph-to-P4: a P4 boilerplate code generator for parse graphs, in: ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), IEEE, 2019, pp. 1–2.

[39] M. Bavarian, H. Jun, N. Tezak, J. Schulman, C. Mcleavey, J. Tworek, M. Chen, Efficient training of language models to fill in the middle, Technical Report, arXiv preprint, 2022.

[40] Language consortium. $P4_{16}$ language specification, 2023.

[41] The P4 language consortium. The P4 language specification, 2005.

[42] G.. Openai, H. Zhao, J. Hui, J. Howland, N. Nguyen, S. Zuo, A. Hu, C.A. Choquette-Choo, J. Shen, Kelley, articleteam2024codegemma, title = Codegemma: open code models based on gemma, author = Team, CodeGemma, Technical Report, journal = arXiv preprint, 2024. Advancing cost-efficient intelligence, https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence2024.

[43] C. Team, H. Zhao, J. Hui, J. Howland, N. Nguyen, S. Zuo, A. Hu, C.A. Choquette-Choo, J. Shen, J. Kelley, et al Codegemma, Open code models based on gemma, Technical Report, arXiv preprint, 2024.

[44] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, D. Jiang, Wizardcoder, Empowering code large language models with evol-instruct, Technical Report, arXiv preprint, 2023.

[45] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu, Qwen2.5-coder technical report, Technical Report, arXiv preprint, 2024.

[46] J. Wei, M. Bosma, Y. Vincent, K. Zhao, A.W. Guu, B. Yu, N. Lester, A.M. Du, Dai, V. Quoc, Le, Finetuned language models are zero-shot learners, Technical report, arXiv preprint, 2021.

[47] J. Edward, Y. Hu, P. Shen, Z. Wallis, Y. Allen-Zhu, S. Li, L. Wang, W. Wang, Chen, LoRA: low-rank adaptation of large language models, ICLR 1 (2) (2022) 3.