Understanding How CodeLLMs (Mis)Predict Types with Activation Steering

Francesca Lucchetti & Arjun Guha
Khoury College of Computer Sciences
Northeastern University
Boston, MA 02115
{lucchetti.f,a.guha}@northeastern.edu

Abstract

Large Language Models (LLMs) are widely used by software engineers for programming tasks. However, research shows that LLMs often lack a deep understanding of program semantics. Even minor changes to syntax, such as renaming variables, can significantly degrade performance across various tasks. In this work, we examine the task of *type prediction*: given a partially typed program, can a model predict a missing type annotations such that the resulting program is more typed? We construct a dataset of adversarial examples where models initially predict the correct types, but begin to fail after semantically irrelevant edits. This is problematic, as models should ideally generalize across different syntactic forms of semantically equivalent code. This lack of robustness suggests that models may have a shallow understanding of code semantics.

Despite this, we provide evidence that LLMs do, in fact, learn robust mechanisms for type prediction—though these mechanisms often fail to activate in adversarial scenarios. By using *activation steering*, a method that manipulates a model's internal activations to guide it toward using latent knowledge, we restore accurate predictions on adversarial inputs. We show that steering successfully activates a type prediction mechanism that is shared by both Python and TypeScript, and is more effective than prompting with in-context examples. Across five different models, our comprehensive evaluation demonstrates that LLMs can learn generalizable representations of code semantics that transfer across programming languages.

1 Introduction

Large Language Models (LLMs) are widely used by software engineers on many programming tasks. Despite their impressive capabilities, research has shown that they are not robust to semantically irrelevant features of programs: syntactic changes such as reordering conditions or renaming variables can significantly impact LLM performance on programming tasks (Hooda et al., 2024a). This raises a fundamental question: do contemporary LLMs learn to reason about program semantics, or do they merely learn textual features such as the associations between variable names and their types? For example, predicting that a variable named n has type *int*, regardless of how it is used.

Reasoning about programs involves a number of different tasks (Gu et al., 2024). In this paper, we focus on the *type prediction* task for gradually typed programming languages, specifically Python and TypeScript, defined as follows.

Definition 1 (Type Prediction) *Given a partially typed program p, choose an untyped variable binding $var \in p$, predict a type annotation $var : T$, and insert the annotation back into the program to get a new program p' that also passes the type-checker.*

Types are fundamental to programming languages. Reliably predicting types requires understanding control flow and data flow in a program, and gradual type prediction is particularly challenging. Unlike type inference (e.g., in Haskell or OCaml), where classical algorithms work, gradual type prediction is undecidable (Migeed and Palsberg, 2020). Moreover, it is always possible to predict the *any* type, which is imprecise, but sometimes necessary in very dynamic code. The challenge is to predict a type that is both precise and consistent with program semantics (Phipps-Costin et al., 2021), and classical algorithms so far do no scale to modern programming languages (§2).

LLMs are remarkably good at type prediction for Python and TypeScript (Yee and Guha, 2023; Fried et al., 2023). However, as we show in this paper, when a model successfully predicts the type T of a variable $var \in p^+$, we can often construct a variation p^- with minimal syntactic changes that make the model mispredict the type. The question

Proceedings of the 8th BlackboxNLP Workshop: Analyzing and Interpreting Neural Networks for NLP, pages 358–397

we ask is, *why do these type mispredictions occur?*

In this paper, we give evidence that models learn a robust internal mechanism for type prediction in hidden layers ℓ. However, this mechanism can fail to activate when the input program p^- has adversarial syntactic features that mislead prediction (e.g. unreliable variable names). We show that we can correct such mispredictions by editing model layers ℓ with targeted *steering vectors* $\mathbf{v}^\ell$. This allows us to demonstrate that:

1. Adding $\mathbf{v}^\ell$ to layers ℓ activates the mechanism and significantly improves type prediction performance (§4.1);
2. $\mathbf{v}^\ell$ is shared across languages; we can improve Python type prediction with $\mathbf{v}^\ell$ computed from TypeScript and vice versa (§4.3); and
3. $\mathbf{v}^\ell$ enables *prediction* but does not control *precision* of types. In other words, when a model predicts a type such as *any*, $\mathbf{v}^\ell$ does not make the prediction more precise (§4.5).

We also show that this internal type prediction mechanism is hard to access without directly adding $\mathbf{v}^\ell$ to the model. Specifically, in-context learning has a negligible impact on accuracy of type prediction for problems where a direct model edit is successful (§4.4).

Our extensive evaluation shows that results generalize across five different LLMs from four model families (Hui et al., 2024; Yang et al., 2024; Dubey et al., 2024; Roziere et al., 2023; Li et al., 2023). These include both pretrained and instruction-tuned LLMs, LLMs trained exclusively on code, and general-purpose LLMs trained on code and data.

2 Background and Related Work

Classical type prediction and type inference Type prediction is distinct from type inference as found in languages such as OCaml and Haskell. In those languages, every variable is typed, even if the types are implicit (Harper and Mitchell, 1993). In contrast, a gradually typed programming language allows programs to freely mix typed and untyped code, giving programmers more flexibility than traditional static typing affords (Siek and Taha, 2006; Tobin-Hochstadt and Felleisen, 2006). However, untyped code still needs to type-correct for the program to run correctly. With omitted or weak type annotations, type errors may not be caught until program execution.

```
def is_palindrome(s: [FILL]):
    s = s.lower()
    return s[::-1] == s
```

(a) The abstract type prediction task.

```
<fim_prefix>
def is_palindrome(s: <fim_suffix>):
    s = s.lower()
    return s[::-1] == s<fim_middle>
```

(b) A fill-in-the-middle prompt for the task.

```
[USER] Continue this program with the
correct substitution for <FILL>:
def is_palindrome(s: <FILL>):
    s = s.lower()
    return s[::-1]==s
[ASSISTANT] def is_palindrome(s:
```

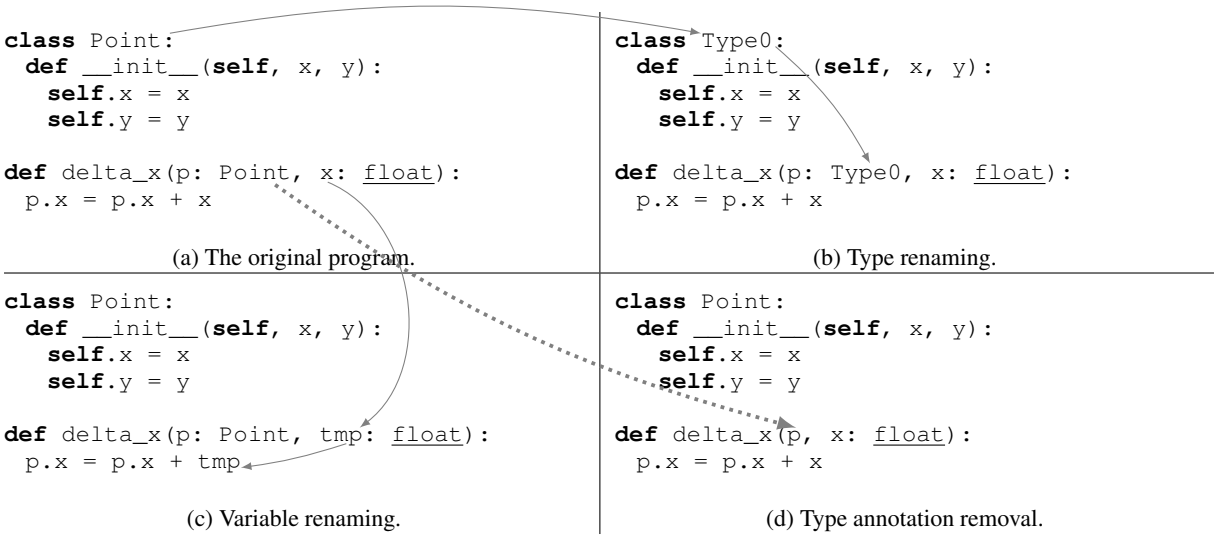
(c) A prompt for an instruction-tuned model.

Figure 1: An example type prediction task, formulated for each type of model.

There is prior work on rule-based type prediction algorithms (Phipps-Costin et al., 2021; Rastogi et al., 2012; Siek and Vachharajani, 2008; Campora et al., 2018; Henglein and Rehof, 1995; Cartwright and Fagan, 1991). But, these papers present algorithms for variations of the lambda calculus or simple functional languages such as Scheme, and have not been scaled to more complex, modern programming languages.

Neural type prediction Over the past decade, prior work has explored leveraging neural networks, including LLMs, for type prediction (Hellendoorn et al., 2018; Jesse et al., 2022, 2021; Pandi et al., 2021; Wei et al., 2020). Unlike classical approaches that target idealized programming languages, these works attempt to predict types for widely-used programming languages like TypeScript and Python. A practical approach to automated type prediction would be significant. Airbnb, Dropbox, Slack, Netflix, and many others have each taken several years to manually add type annotations to their multi-million line gradually typed codebases (Rudenko, 2020; Lehtosalo, 2019; Felix Rieseberg, 2017; Luke Autry; Sumana Mohan et al., 2022; Abacus, 2019; Mihai Parparita, 2020; Jake Zimmerman, 2022).

Mutation testing and program transformations In our experiments, we construct type prediction prompts by renaming variables to arbitrary names, or deleting some type annotations in the context.



(a) The original program.

(b) Type renaming.

(c) Variable renaming.

(d) Type annotation removal.

Figure 2: Examples of three semantics-preserving edits. The type prediction site is `float`. We ensure that each edit is internally consistent. E.g., in (2c), when we rename the binding `x` to `tmp`, we rename references to the binding.

```
class KafkaAvroBackend(RepositoryBackend):
    def __init__(
        self, ~~config~~ __tmp0 : dict, producer=AvroProducer, loader=AvroMessageLoader,
        value_serializer: Callable = to_message_from_dto,
        get_producer_config: Callable = get_producer_config,
        get_loader_config: Callable = get_loader_config
    ) -> None:
        producer_config = get_producer_config(~~config~~ __tmp0 )
```

Figure 3: A fragment of a Python steering pair. The original code is 70 lines of text. The dict is the expected prediction. But, renaming `config` to `__tmp0` makes the model mispredict `Repository`, which is a hallucination.

We construct our edits such that they do not break program syntax, and all the information necessary for type prediction is still present in the program. To do so, we take inspiration from *mutation testing* (DeMillo et al., 1978). The goal of mutation testing is to test a program's test suite. To do so, a mutator injects small bugs that alter the semantics of a program, such as changing a 0 to a 1 or turning $x > y$ into $x < y$. The hypothesis is that a good test suite should be able to catch these artificial bugs, and there is a substantial evidence that the ability to catch both artificial and real-world bugs is strongly correlated (Just et al., 2014).

Our technique differs from mutation testing in a key way: we make program edits that would not affect test cases, but affect LLM predictions. We make minimal, semantics-preserving edits that lead to type mispredictions for a given LLM. The nature of code allows us to construct these edits in a sound and scalable way (§3.1).

Activation Steering It is well known that even the most capable LLMs are sensitive to small variations in prompts. Prior work uses a black-box approach to study these phenomena by looking at model performance on programming tasks (Hooda et al., 2024b; Tambon et al., 2024). In contrast, we investigate type prediction with a whitebox approach. We use activation steering to query what a model's inner activations on code prompts reveal about its understanding of type systems.

Activation steering is an inference-time model editing technique used to control model behavior. Research has shown that steering can moderate negative qualities like deceitfulness and sycophancy in model outputs (Rimsky et al., 2024; Li et al., 2024). Steering uses targeted steering vectors computed from model activations over positive and negative outputs. The intuition is that by quantifying the difference between positive and negative outputs, we can edit (steer) prediction away from the negative. Steering can be used to interpret the causal features behind model predictions by verifying that the structure of model internal representations is consistent with how language works. For exam-

ple, steering has been used to verify that models encode faithful representations of English grammar and verbs (Ravfogel et al., 2021). Similarly, we use steering to show that models have a robust understanding of code and type systems.

3 Methodology

3.1 Adversarial Type Prediction Tasks

Our goal is to build a dataset of type prediction tasks that models fail to solve correctly, but have known working solutions. Different models fail and succeed at different tasks, so the datasets will be model-dependent.

We present a variation of mutation testing that constructs minimal, semantics-preserving edits that trigger mispredictions. These edits are automated and applied randomly to programs from GitHub, allowing us to build challenging type prediction tasks at scale. Our edits produce programs that have unconventional syntax, but have the structure and behavior of real code.

Type Prediction Prompt Format We build datasets for both LLMs pretrained on code and instruction-tuned models.

Contemporary LLMs trained on code typically preprocess their training data to *fill-in-the-middle* (FIM) (Bavarian et al., 2022; Fried et al., 2023). FIM training (1) splits $\approx 50\%$ of training items into three chunks—prefix, middle, and suffix—of random lengths; (2) adds special tokens to the start of each chunk; and (3) reorders the middle chunk to appear last. The language modeling training objective remains unchanged. At inference time, this allows models to generate the middle chunk, conditioned on the prefix and the suffix using a decoder-only LLM. Figure 1a shows an example type prediction task, where we want the model to predict the type annotation for the argument s, which is in the middle of the program. To do so, we construct a prompt that marks the prefix and suffix with the model-specific FIM tokens (Figure 1b).

In contrast, for instruction-tuned models, we formulate type prediction as a two-turn conversation between the user and assistant using the model-specific chat template (Figure 1c). The prompt includes the instruction to fill in the target type annotation site `<FILL>`. We include the prefix in the model's answer so that the model produces a well-formed program.

Semantics-preserving Code Edits For each model M, we first build a dataset of "easy" type prediction tasks that M solves correctly.[1] For Python, we use ManyTypes4Py (Mir et al., 2021), a dataset of code from 5,382 Python projects with Python type annotations that successfully type-check. For TypeScript, we filter The Stack (Kocetkov et al., 2023) to find 1.1M TypeScript files that type-check. This ensures that the expected gold labels for type annotations are correct. Every program p in the dataset may have several type annotations $var : t \in p$, and each of these annotations is a potential type annotation task. From these files, we build a large set of type prediction prompts (p^+, t) where M succeeds at type prediction. This dataset is potentially class-imbalanced, since models are unsurprisingly are better at predicting builtin types than user-defined types. We make sure to balance the distribution of types for our experiments §3.1.

Secondly, for each model M, we build a dataset of "hard" type prediction tasks that M cannot solve. We select an easy task from the previous dataset, (p^+, t) and incrementally apply the following semantics-preserving program edits at random. 1) *Rename variable:* We select a function/method argument and rename it to an arbitrary name that does not conflict with other variables. 2) *Remove type annotation:* We select a type annotation (excluding the target t) and delete it. In a gradually typed language, removing or relaxing an annotation does not alter program semantics. 3) *Rename user-defined type:* We select an arbitrary type definition (e.g., a class name or a type alias) and rename it to an arbitrary name that does not conflict with other names in the program. 4) *Rename builtin type:* We introduce a type alias for a builtin type. Figure 2 illustrates several separate edits to a program.

The aforementioned edits do not change the type structure of the program. They make p^+ look different, but the target type t remains unchanged. After applying each edit, we prompt M to predict the type annotation. If M mispredicts, we stop and use the current program as a failing type prediction task (p^-, t). By construction, this is an adversarial type prediction task that M fails to solve due to syntactic changes.

If p^+ is particularly simple, we may fail to construct (p^-, t). In practice, we get several thousand

[1] These files are in the training corpora for most models and we find that models easily predict types.

challenging examples for each model, even in ablations where we restrict set of edits that we perform. Figure 3 illustrates a real example from our dataset that makes a model mispredict. Note that a single edit often alters p^+ at several points.

We automatically construct p^- by manipulating the concrete syntax tree of TypeScript and Python using TreeSitter-based parsers. This allows us to build these edits correctly and at scale.

Test sets and class balance For each model, we build test sets of 100 type prediction tasks (p^-, t) that the model gets wrong. The natural distribution of type annotations is heavily skewed toward built-in and primitive types, thus we class-balance the test set to ensure that no target type t occurs more than four times. Each test set has a mix of both built-in and user-defined types. This ensures that our evaluation is not skewed by reporting success on the most common types. We use the same class-balancing approach to construct the steering dataset for activation steering vectors, described below.

3.2 Finding the Type Prediction Mechanism

Why might a model fail to solve a type prediction task (p^-, t), when it succeeded at the original task (p^+, t)? Note that since p^+ is sourced from GitHub-based datasets, the model was trained on these programs, whereas for the edited program p^-, by construction the model has likely never been trained on similar syntax. There are two hypotheses: 1. the model has *not* learned a robust mechanism for type prediction that generalizes outside of training data and resists adversarial prompts, basing its prediction on text features rather than program semantics; 2. the model has a robust mechanism for type prediction, but it does not activate on adversarial prompts. We argue that hypothesis 2 is correct. Using activation steering, we build steering vectors $\mathbf{v}^\ell$ that, when added to layer ℓ, can activate robust type prediction on adversarial prompts. We present how we construct $\mathbf{v}^\ell$ below.

Constructing Steering Vectors For a given model M, we construct a dataset of triples $(p_i^+, p_i^-, t_i) \in \mathcal{D}$ where p_i^- is an edited version of p_i^+, the maximum likelihood generation is $M(p_i^+) = t_i$, and $M(p^-) \neq t$. We apply forward passes $M(p_i^+), M(p_i^-)$ and save model activations of the last token before the type prediction token. Concretely, this involves pausing the model's forward pass at a layer ℓ_j of the transformer and saving the output of that layer, before it gets fed to subsequent layers. We write $A_\ell(x)$ to denote the activation vector at layer ℓ for prompt x. We compute steering vectors $\mathbf{v}_\ell$—one for each layer—as the mean difference between positive and negative activations at that layer:

$$\mathbf{v}_\ell = \frac{1}{|\mathcal{D}|} \sum_{(p_i^+, p_i^-, t) \in \mathcal{D}} \Big(A_\ell(p_i^+) - A_\ell(p_i^-) \Big) \quad (1)$$

We compute steering tensors using hundreds of positive and negative prompt pairs for each of our edits, described previously §3.1.

The intuition behind eq. (1) is that the resulting vector represents a transformation in activation space that separates the model's incorrect predictions from correct ones. Thus adding $\mathbf{v}_\ell$ to layer ℓ should prompt the model to shift to an internal mechanism not usually enabled on the adversarial prompts. We determine the layer ℓ experimentally, and also consider steering at up to five adjacent layers.

4 Results

4.1 Steering Improves Type Prediction on Out-of-Distribution Tasks

Figure 4 shows TypeScript test-set accuracy on every model with steering. Each subfigure ablates the set of edit operations used to construct the type prediction tasks so that we can see the effectiveness of steering on different edits. The x-axis indicates the relative position of layer ℓ where we apply $\mathbf{v}^\ell$. ($x = 0$ indicates that ℓ is the first layer and $x = 1$ indicates that it is the last layer.) In these experiments we apply $\mathbf{v}^\ell$ to five adjacent layers $\ell \cdots \ell + 4$, which we find is more effective than steering fewer layers (§4.2).

The figures show that steering is most effective in the later middle layers of every model, which suggests that this is where the type prediction mechanism lies. Recall that every type prediction task in the test sets are tasks that the model gets wrong without steering, thus the baseline accuracy is zero. When we construct p^- using all possible edits, steering in the middle layers corrects mispredicted types on 50%-60% of the test set (varying by model). Steering is most effective when we construct p^- by just renaming types, and corrects mispredictions on up to 80% of the test set. We discuss steering performance in more depth in §4.5.

Overall, results indicate that we can find a $\mathbf{v}^\ell$ for each model that enables a robust type prediction

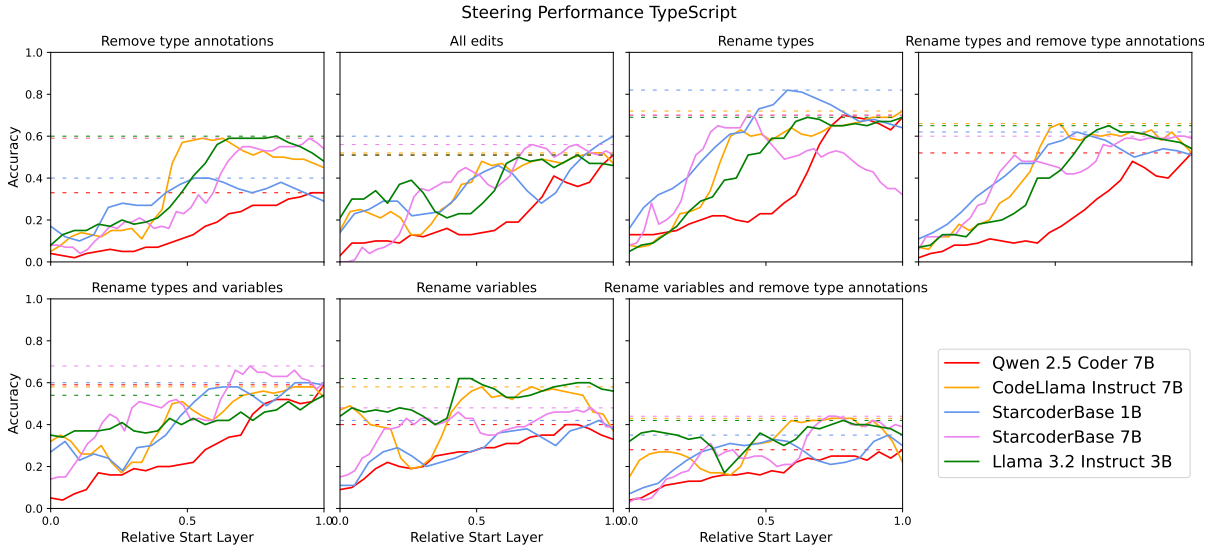


Figure 4: Steering accuracy for all models on the TypeScript test set, with steering on five consecutive layers. The models have a varying number of layers, so the x-axis is normalized: for a model with n layers, $x = 0$ indicates steering on the first five layers, and $x = 1$ indicates steering on the last five layers.

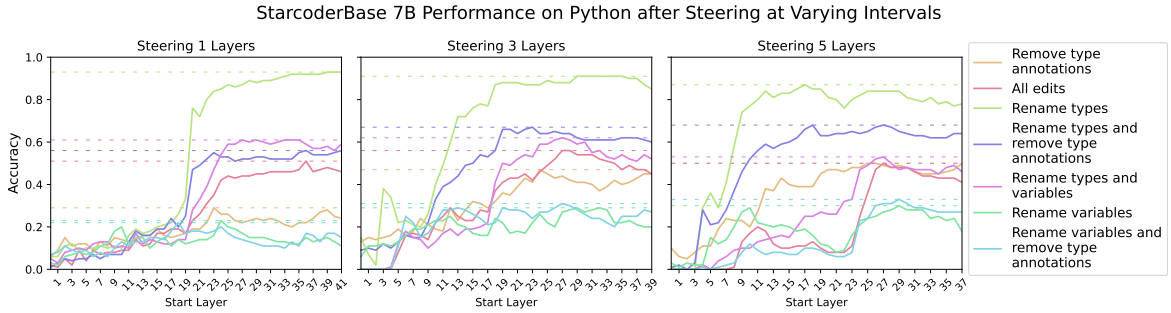


Figure 5: Steering accuracy for StarCoderBase 7B on Python. Each plot show steers in one, three, and five consecutive layers respectively.

even for adversarial type prediction tasks. While Figure 4 shows results for TypeScript, we have similar results for Python in the appendix, where steering is even more effective for certain edits (Figure 12).

4.2 Types Are Predicted Over Several Layers

The type prediction mechanism may span several layers. Therefore, we consider steering at one, three, and five adjacent layers. Figure 5 shows the effect of this ablation on StarCoderBase-7B with Python: the x-axis indicates the start layer for steering and the y-axis is test-set accuracy. We find that steering on five layers is most effective. The appendix has similar results for TypeScript and the other models (appendix B, appendix C).

4.3 The Type Prediction Mechanism Is Shared Between Languages

Python and TypeScript are syntactically distinct, but their semantics have a lot in common (Politz et al., 2013; Bierman et al., 2014). Both languages are gradually typed. So, could it be that LLMs learn a type prediction mechanism that is language agnostic? To test this hypothesis, we evaluate if steering vectors built on TypeScript data can improve the accuracy of Python type prediction, and vice versa. We conduct this experiment with each of our datasets: we steer a model using vectors from language A but evaluate on the corresponding held-out test set from language B. Figure 6 shows that this is nearly as effective as steering prediction on the same language.

This result suggests that models learn similar representations of types across languages. The in-

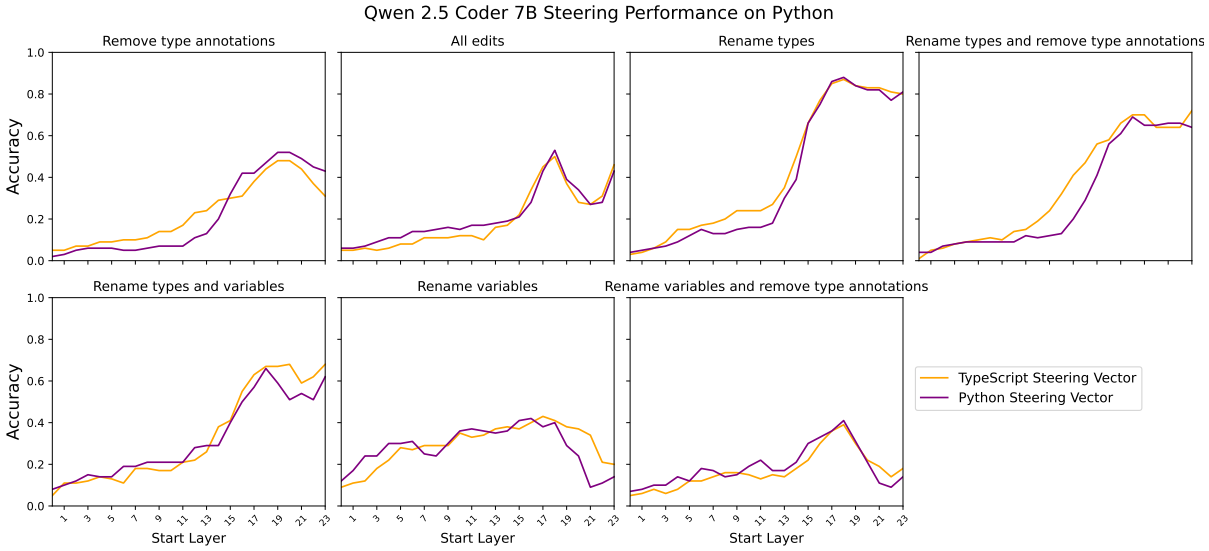


Figure 6: For Qwen 2.5 Coder 7B, we plot the performance of TypeScript steering vectors on the Python test set. We compare with the performance of steering vectors constructed from Python programs and find that the two achieve comparable accuracy. In the appendix we report similar results for all other models (appendix D).

terchangeable nature of steering vectors suggests that models store shared concepts (e.g., types) in similar vector subspaces across languages. This provides some insight on how models internalize shared concepts across languages through consistent structures in activation space.

4.4 Steering Outperforms Other Baselines

Random baseline A competing hypothesis to the one that we advance is the following: adding $\mathbf{v}^{\ell}$ is just adding noise, and steering is effectively just resampling from the output distribution. To refute this, we also steer with with a random vector and find that the computed steering vectors significantly outperform the random baseline (Figure 7). This indicates that the steering vectors we compute perform true, localized transformations *towards the correct type prediction task* in activation space.

Figure 7 also shows the performance of steering on the prompts p^{-} from the steering set. We find that test-set and steering-set accuracy are approximately the same. This suggests that steering tensors can generalize outside the specific types and programs they were built from. We report results for this experiment for all our models in Appendix E.

In-context learning The usual way to instruct an LLM towards the correct task is with in-context examples (ICL). We perform an experiment where instead of steering, we prompt the model with two examples of adversarial type prediction tasks (p^{-}, t). We find that prompting almost always underperforms steering (Figure 8). This indicates that directly calculating the steering vector is a more robust way to enable the model's type prediction mechanism on adversarial programs.

4.5 Steering Enables Type Prediction But Does Not Improve Type Precision

Why doesn't steering always correct mispredictions? A complication of type prediction is that there may be several solutions to a type prediction problem that are type-correct, though some solutions are more precise than others. Therefore, if a model M fails a task (p^{-}, t) and predicts $M(p^{-}) = t'$, where $t' \neq t$, it may be the case that t' is still a type-correct prediction. In Figure 9, we plot the accuracy of every combination of model and type of edit. On the y-axis we report steering accuracy and on the x-axis we report the fraction of programs where p^{-} with the mispredicted type t' is still type-correct (i.e., passes the type-checker). We find a strong negative correlation ($r(68) = -0.687, p < 5.16 \times 10^{-11}$) between steering accuracy and type-correctness *before* steering. When the model predicts a type that introduces a type error, steering is able to correct it. However, when the model merely predicts a unexpected type, steering is not as effective at directing the model to the expected answer.

Qualitatively, looking at these results, we find that most of these unexpected types are imprecise types, such as *any*, or *dict* instead of *Config*. Over-

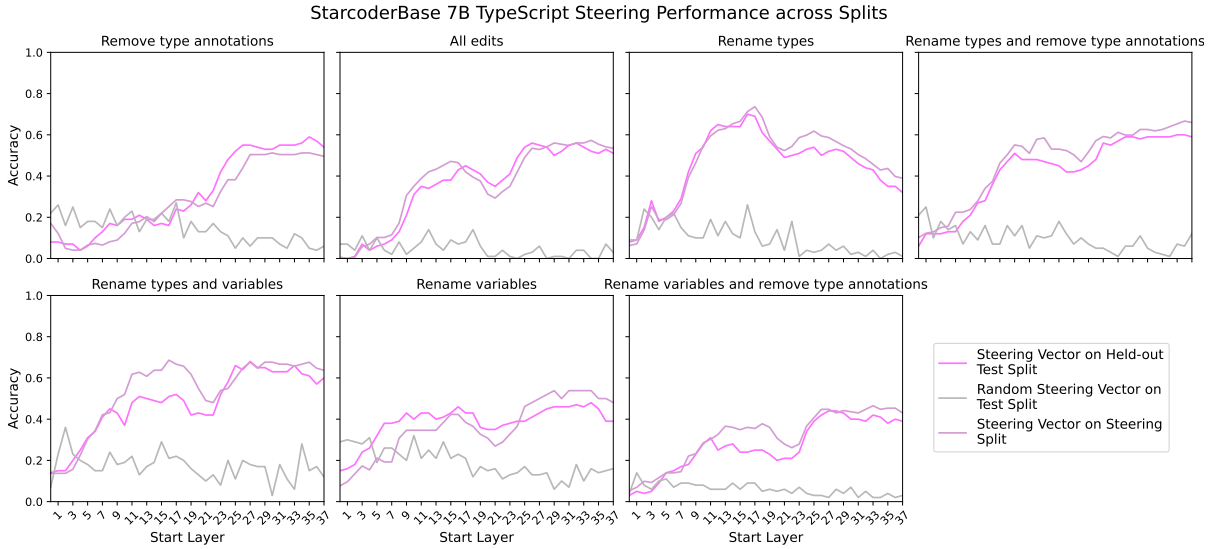


Figure 7: Steering accuracy for StarCoderBase 7B on TypeScript prompts on the test set, the steering set itself, and a random steering vector. Random performs poorly; the test and steering sets have similar performance.

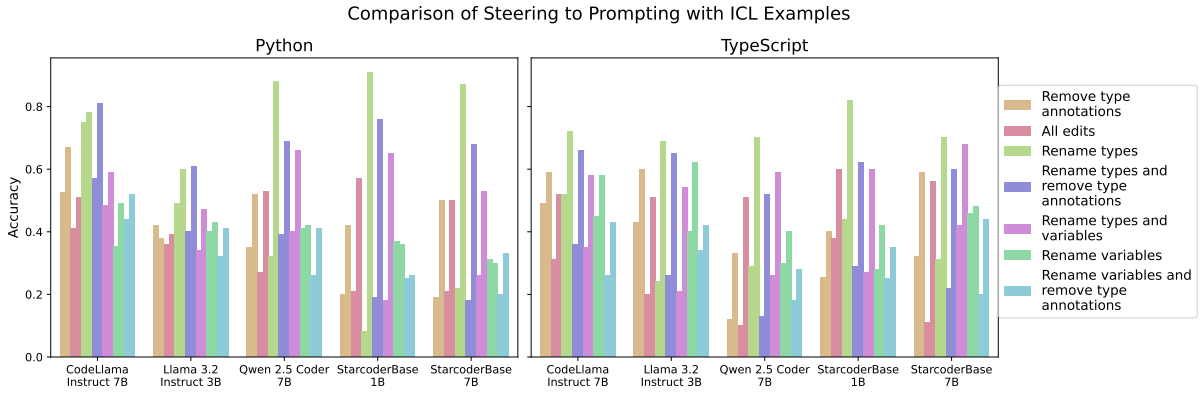


Figure 8: For each model, language and edit, we plot the best performance of steering vectors against in-context prompting (hatched bars).

all, this experiment shows that we have identified the mechanism that enables the type prediction task, but not a mechanism that allows us to control the degree of type precision. Whether or not it is possible to identify such a mechanism in LLMs is a topic for future work.

5 Conclusion

Collectively, our results indicate that steering vectors steer the model toward a mechanism for type prediction that 1) generalizes across different source codes; 2) is less sensitive to semantically irrelevant features; and 3) generalizes across the languages we study.

Given these observations, we conclude that there exists a robust mechanism for type prediction in LLMs which, when activated through activation steering, is more robust against adversarial programs. Furthermore, this mechanism is difficult to activate with prompting. This finding shows that it is insufficient to make conclusions about model's learned capabilities based on outputs alone.

Whether a model is capable of performing robust and generalizable type prediction is a question of correctly aligning the model to the task. Activation steering is capable of performing this alignment for localized edits. Fine-tuning directly on edits could improve performance, but this defeats the purpose of studying behavior on adversarial or unseen prompts. In order to effectively use the information learned by LLMs, further research into how this information is organized, stored and retrieved is necessary.

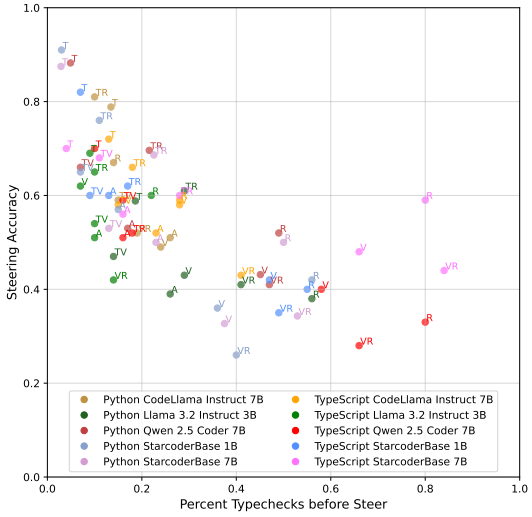


Figure 9: For every combination of model and edit-type, we plot the accuracy of type prediction on steering vs. the percentage of programs that are type-correct with the original, mispredicted type. The labels are: V for renaming variables; T for renaming types; R for removing type annotations, and combinations of these.

6 Limitations

Our findings shed light on how CodeLLMs display robust type prediction for TypeScript and Python. Both these languages are well represented in CodeLLM training corpora. However, our findings may not extend to low-resource gradually typed languages, e.g., Typed Racket (Tobin-Hochstadt and Felleisen, 2008) or Luau (Lily Brown et al., 2023) since the performance of base models on these languages is very poor. Future work will include implementing semantics-preserving edits for other languages.

Our investigation focuses on type prediction to understand whether models learn program semantics along with syntax. The reduced scope allows us to conduct an in depth evaluation of models and steering vectors. Future research may focus on studying learned representations of other code concepts such as control flow, data races and vulnerabilities.

We apply automatically generated edits to prompts as a scalable way to approximate real code with arbitrary syntax. To ensure diverse and comprehensive test sets, we use hundreds of real programs for each model, varying the source code, target types, and programming languages. However, we note that these automatically generated edits may not fully capture the complete variance possible in code.

7 Ethics Statement

The purpose of this work is to understand whether LLMs perform type prediction using robust mechanisms. It is our view that interpreting LLMs is necessary for understanding whether models approach programming in a principled way. As LLMs become more integrated into developers' workflows, model errors could compromise the security of entire systems. For this reason, we make a first investigation into understanding the mechanisms behind model prediction.

We take care to use publicly available code for our experiments. Our TypeScript dataset is derived from a subset of The Stack v1.2, which contains permissively licensed data with personal identifying information (PII) filtered. The ManyTypes4Py dataset is funded by the European Commission, which follows data privacy laws under the EU General Data Protection Regulation (GDPR). These datasets are intended for LLMs, which this paper investigates.

Acknowledgments

Portions of this work are implemented with NNSight and NDIF (Fiotto-Kaufman et al., 2024). We thank Ming-Ho Yee for help with the TypeScript dataset that we use in this work (Yee, 2024). This material is based upon work supported by the U.S. Department of Energy, Office of Science under Award Number DESC0025613.

We thank Northeastern Research Computing for support with the Northeastern University Explorer cluster. This work used the Delta cluster at the National Center for Supercomputing Applications (NCSA) through allocation CIS230213 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, which is supported by U.S. National Science Foundation grants 2138259, 2138286, 2138307, 2137603, and 2138296.

Disclaimer: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product,

process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

References

Abacus. 2019. How We Completed a (Partial) TypeScript Migration In Six Months. Section: Developing In Real Time.

Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. Efficient training of language models to fill in the middle. *arXiv preprint arXiv:2207.14255*.

Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *ECOOP 2014 – Object-Oriented Programming*, Lecture Notes in Computer Science, pages 257–281, Berlin, Heidelberg. Springer.

John Peter Campora, Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2018. Migrating Gradual Types. *Proceedings of the ACM on Programming Languages (PACMPL)*, 2(POPL).

Robert Cartwright and Mike Fagan. 1991. Soft typing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

R.A. DeMillo, R.J. Lipton, and F.G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4):34–41. Conference Name: Computer.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Felix Rieseberg. 2017. TypeScript at Slack. Section: Uncategorized.

Jaden Fiotto-Kaufman, Alexander R Loftus, Eric Todd, Jannik Brinkmann, Caden Juang, Koyena Pal, Can Rager, Aaron Mueller, Samuel Marks, Arnab Sen Sharma, Francesca Lucchetti, Michael Ripa, Adam Belfki, Nikhil Prakash, Sumeet Multani, Carla Brodley, Arjun Guha, Jonathan Bell, Byron Wallace, and David Bau. 2024. Nnsight and ndif: Democratizing access to foundation model internals. *Preprint*, arXiv:2407.14561.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. In *International Conference on Learning Representations (ICLR)*.

Alex Gu, Baptiste Roziere, Hugh James Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida Wang. 2024. CRUXEval: A benchmark for code reasoning, understanding and execution. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 16568–16621. PMLR.

Robert Harper and John C. Mitchell. 1993. On the type structure of standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252.

Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep Learning Type Inference. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

Fritz Henglein and Jakob Rehof. 1995. Safe polymorphic type inference for a dynamically typed language: Translating Scheme to ML. In *International Conference on Functional Programming Languages and Computer Architecture (FPCA)*.

Ashish Hooda, Mihai Christodorescu, Miltiadis Allamanis, Aaron Wilson, Kassem Fawaz, and Somesh Jha. 2024a. Do large code models understand programming concepts? Counterfactual analysis for code predicates. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 18738–18748. PMLR.

Ashish Hooda, Mihai Christodorescu, Miltos Allamanis, Aaron Wilson, Kassem Fawaz, and Somesh Jha. 2024b. Do large code models understand programming concepts? a black-box approach. *arXiv preprint arXiv:2402.05980*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Jake Zimmerman. 2022. Sorbet: Stripe's type checker for Ruby.

Kevin Jesse, Premkumar Devanbu, and Anand Ashok Sawant. 2022. Learning To Predict User-Defined Types. *IEEE Transactions on Software Engineering*, pages 1–1.

Kevin Jesse, Premkumar T. Devanbu, and Toufique Ahmed. 2021. Learning type annotation: is big data enough? In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1483–1486, Athens Greece. ACM.

René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 654–665, New York, NY, USA. Association for Computing Machinery.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2023. The Stack: 3 TB of permissively licensed source code. In *Deep Learning for Code Workshop (DL4C)*.

Jukka Lehtosalo. 2019. Our journey to type checking 4 million lines of Python.

Kenneth Li, Oam Patel, Fernanda Viégas, Hanspeter Pfister, and Martin Wattenberg. 2024. Inference-time intervention: Eliciting truthful answers from a language model. *Advances in Neural Information Processing Systems*, 36.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! *Transactions of Machine Learning Research (TMLR)*.

Lily Brown, Andy Friesen, and Alan Jeffery. 2023. Goals of the Luau Type System, Two Years On. ACM.

Luke Autry. How we failed, then succeeded, at migrating to TypeScript.

Zeina Migeed and Jens Palsberg. 2020. What is Decidable about Gradual Types? *Proceedings of the ACM on Programming Languages (PACMPL)*, 4(POPL).

Mihai Parparita. 2020. The Road to TypeScript at Quip, Part Two.

Amir M Mir, Evaldas Latoškinas, and Georgios Gousios. 2021. Manytypes4py: A benchmark python dataset for machine learning-based type inference. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 585–589. IEEE.

Irene Vlassi Pandi, Earl T. Barr, Andrew D. Gordon, and Charles Sutton. 2021. Probabilistic Type Inference by Optimising Logical and Natural Constraints.

Luna Phipps-Costin, Carolyn Jane Anderson, Michael Greenberg, and Arjun Guha. 2021. Solver-based Gradual Type Migration. *Proceedings of the ACM on Programming Languages (PACMPL)*, 5(OOPSLA).

Joe Gibbs Politz, Alejandro Martinez, Mae Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. 2013. Python: the full monty. In *ACM SIGPLAN Conference on Object Oriented Programmingm, Systems, Languages and Applications (OOPSLA)*, pages 217–232, Indianapolis, IN, USA. ACM.

Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. 2012. The Ins and Outs of Gradual Type Inference. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

Shauli Ravfogel, Grusha Prasad, Tal Linzen, and Yoav Goldberg. 2021. Counterfactual interventions reveal the causal effect of relative clause representations on agreement prediction. In *Proceedings of the 25th Conference on Computational Natural Language Learning*, pages 194–209.

Nina Rimsky, Nick Gabrieli, Julian Schulz, Meg Tong, Evan Hubinger, and Alexander Turner. 2024. Steering llama 2 via contrastive activation addition. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15504–15522, Bangkok, Thailand. Association for Computational Linguistics.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Sergii Rudenko. 2020. ts-migrate: A Tool for Migrating to TypeScript at Scale.

Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Scheme Workshop*.

Jeremy G. Siek and Manish Vachharajani. 2008. Gradual Typing with Unification-based Inference. In *ACM SIGPLAN International Symposium on Dynamic Languages (DLS)*.

Sumana Mohan, Joe King, Ryan Burgess, Jem Young, and Stacy London. 2022. TypeScript migration - Strict type of cocktails - Front End Happy Hour.

Florian Tambon, Arghavan Moradi Dakhel, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Giuliano Antoniol. 2024. Bugs in large language models generated code. *arXiv preprint arXiv:2403.08937*.

Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage Migration: From Scripts to Programs. In *ACM SIGPLAN International Symposium on Dynamic Languages (DLS)*.

Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. LambdaNet: Probabilistic Type Inference using Graph Neural Networks. In *International Conference on Learning Representations (ICLR)*.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zhihao Fan. 2024. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*.

Ming-Ho Yee. 2024. *Predicting typeScript type annotations and definitions with machine learning*. Ph.D. thesis.

Ming-Ho Yee and Arjun Guha. 2023. Do Machine Learning Models Produce TypeScript Types that Type Check? In *European Conference on Object Oriented Programming (ECOOP)*.

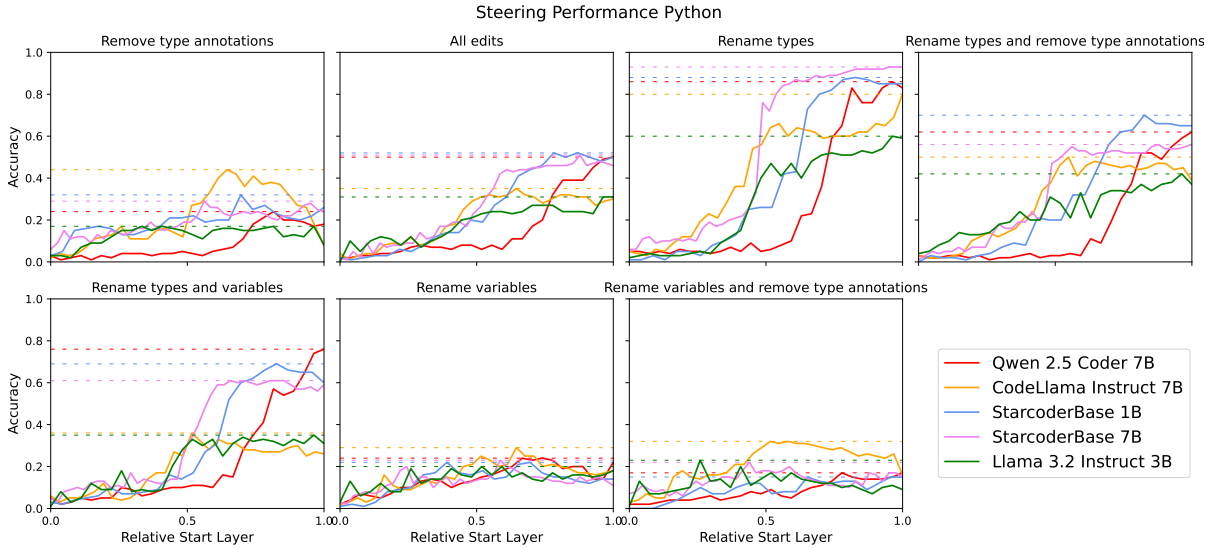


Figure 10: Steering performance for all models on Python data, steering 1 adjacent layers.

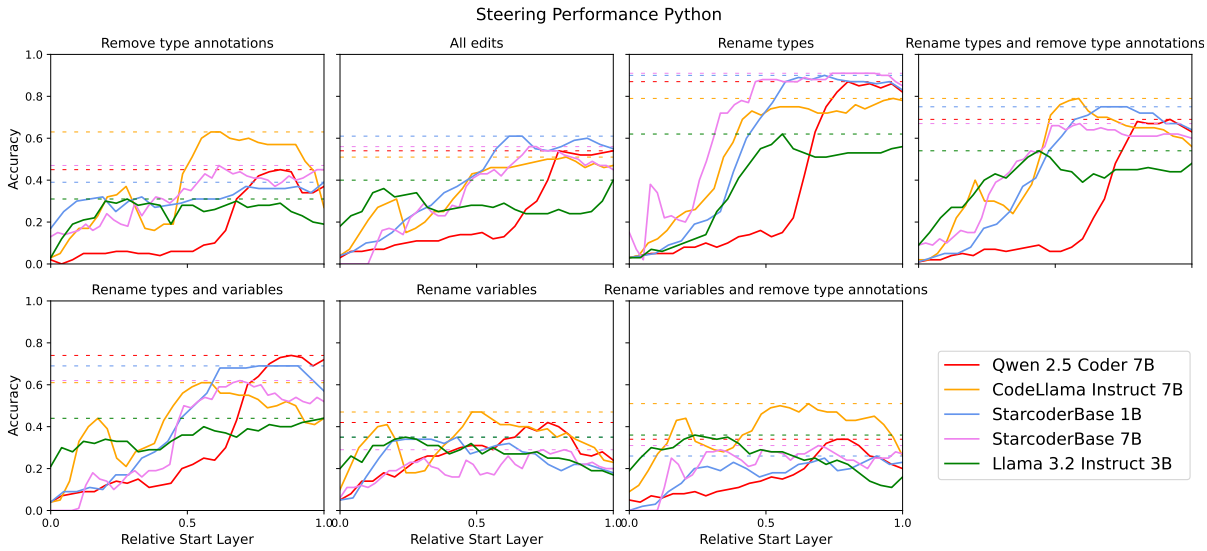


Figure 11: Steering performance for all models on Python data, steering 3 adjacent layers.

A Use of AI Assistants

Some of the code for this paper was written with AI assistants enabled.

B Model Results

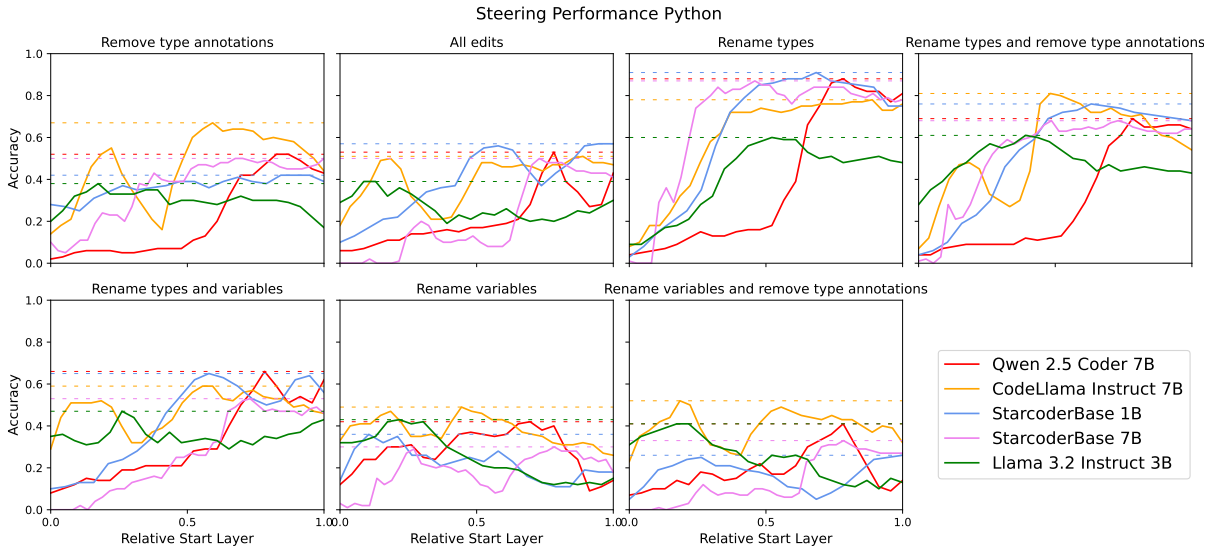


Figure 12: Steering performance for all models on Python data, steering 5 adjacent layers.

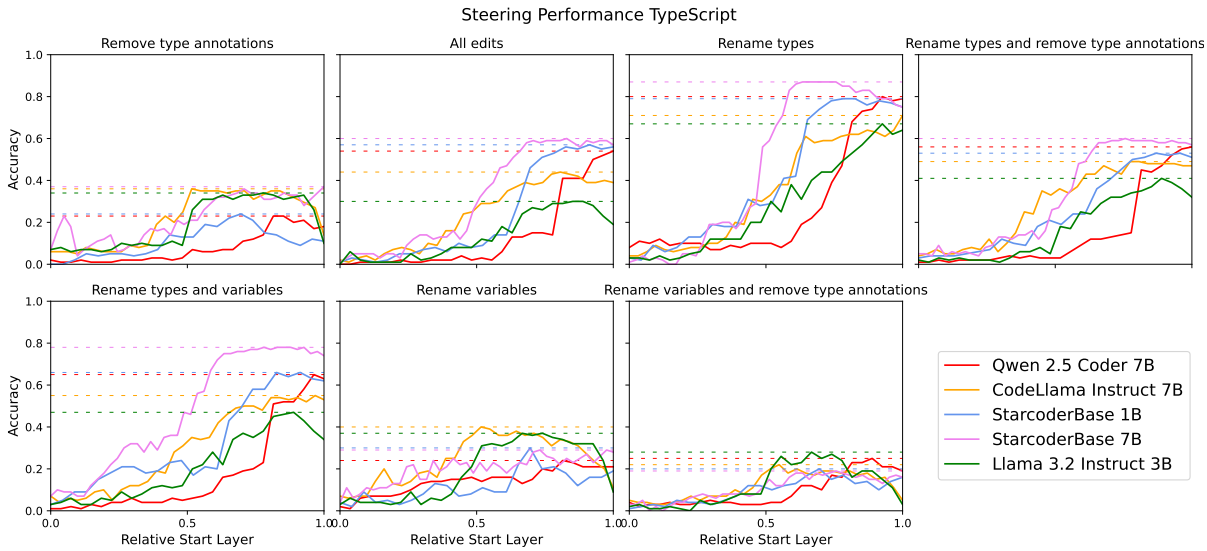


Figure 13: Steering performance for all models on TypeScript data, steering 1 adjacent layers.

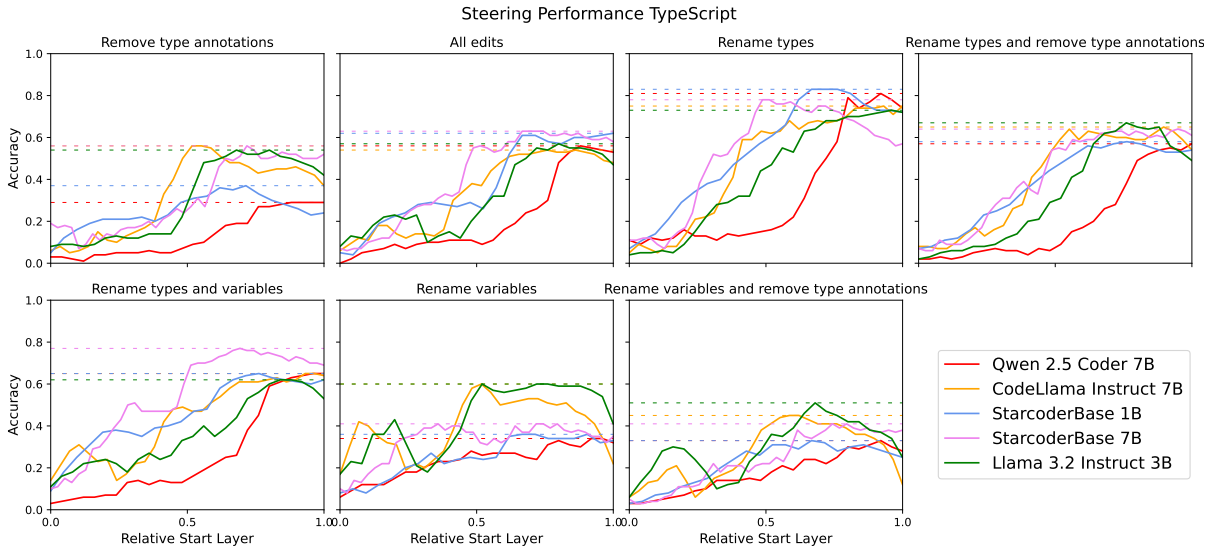


Figure 14: Steering performance for all models on TypeScript data, steering 3 adjacent layers.

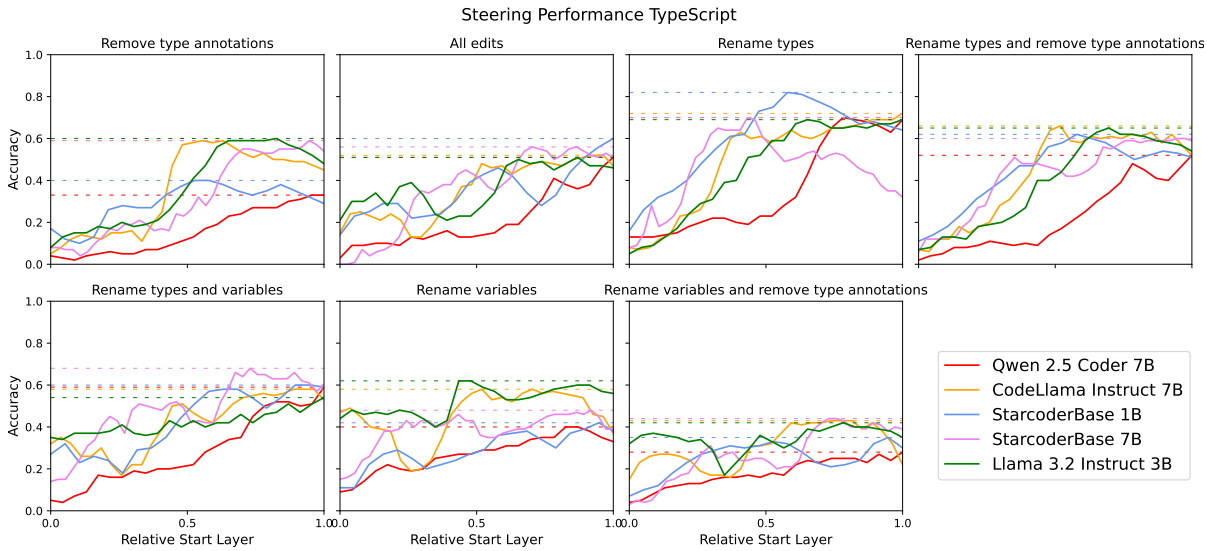


Figure 15: Steering performance for all models on TypeScript data, steering 5 adjacent layers.

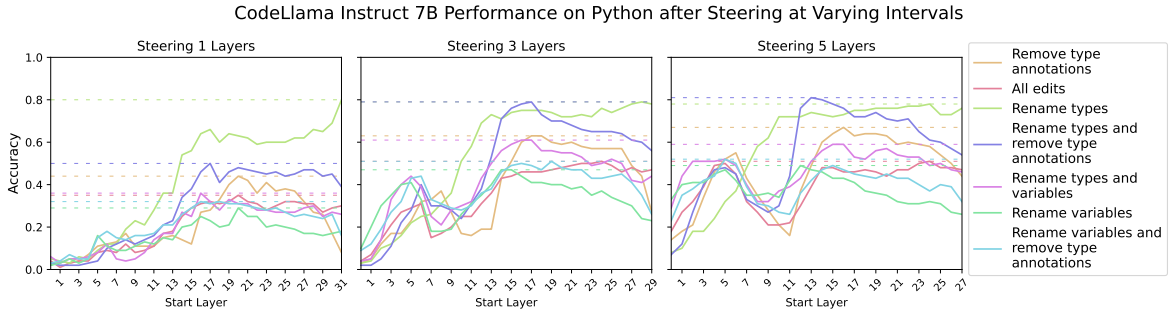


Figure 16: Steering CodeLlama Instruct 7B across different layer intervals

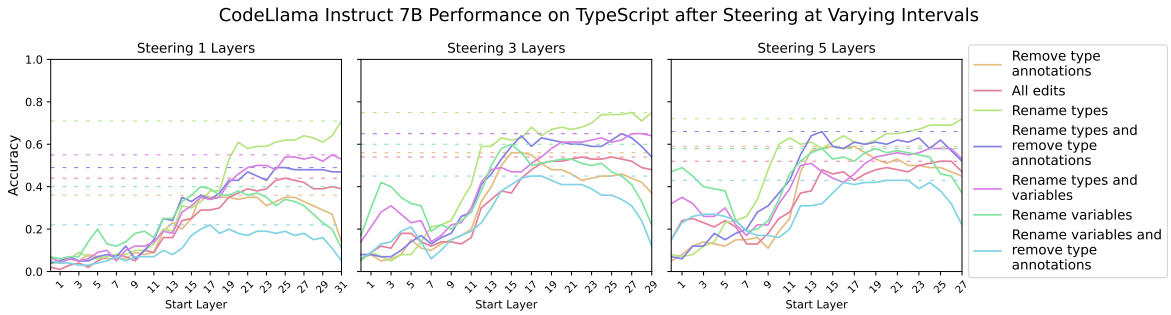


Figure 17: Steering CodeLlama Instruct 7B across different layer intervals

C Interval Ablations Results

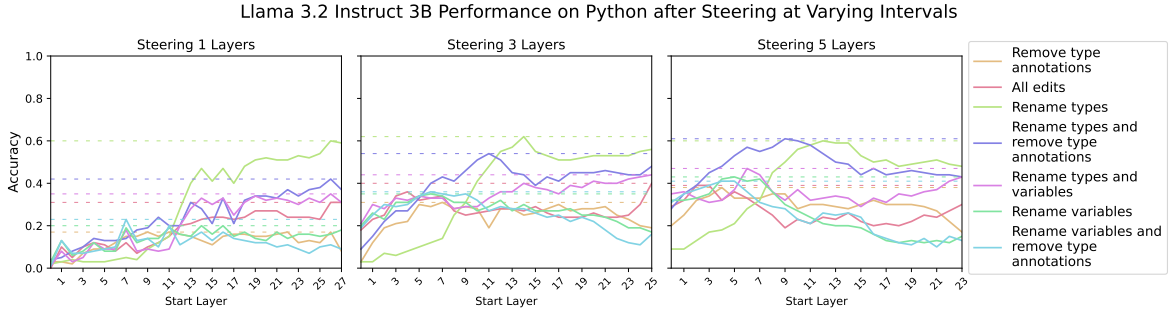


Figure 18: Steering Llama 3.2 Instruct 3B across different layer intervals

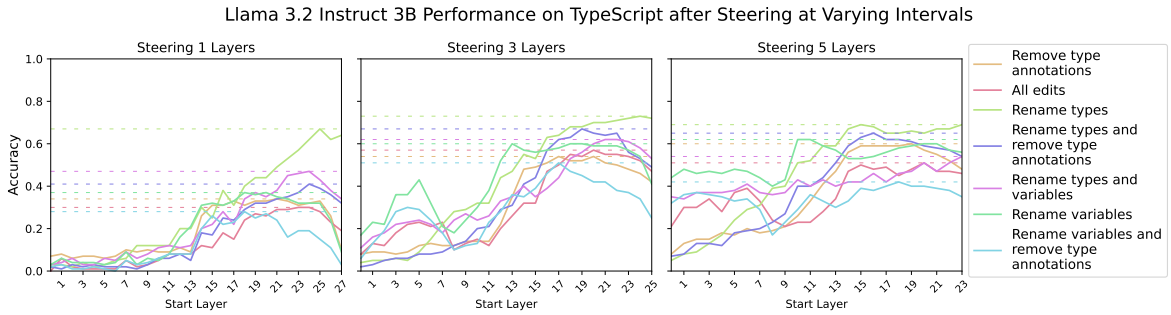


Figure 19: Steering Llama 3.2 Instruct 3B across different layer intervals

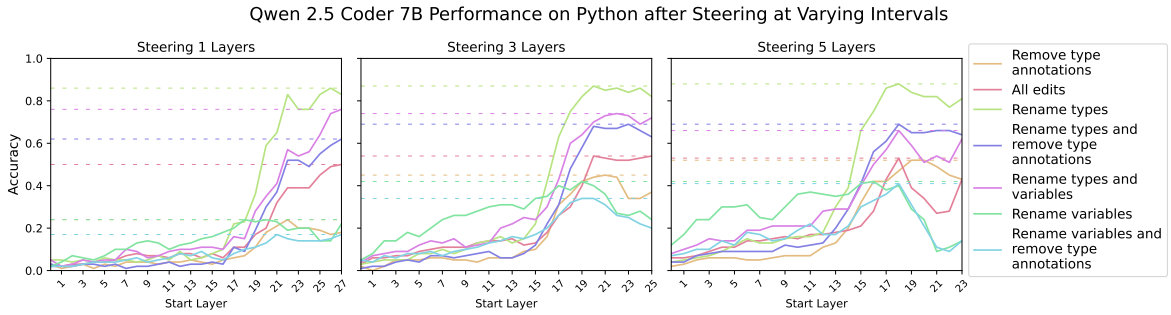


Figure 20: Steering Qwen 2.5 Coder 7B across different layer intervals

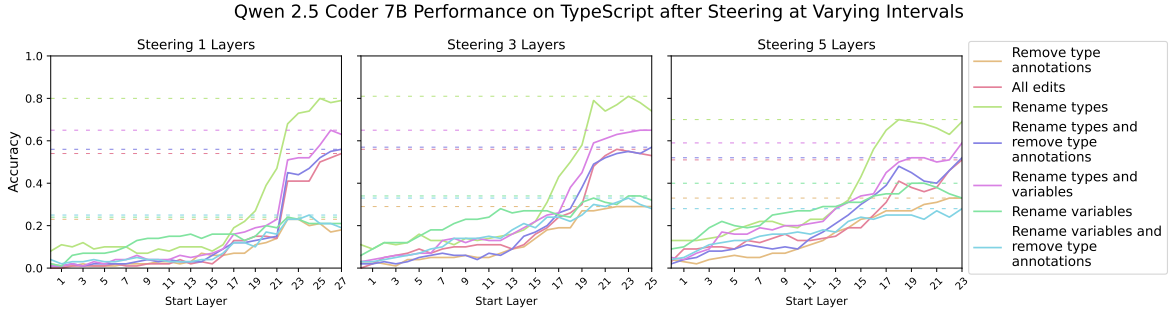


Figure 21: Steering Qwen 2.5 Coder 7B across different layer intervals

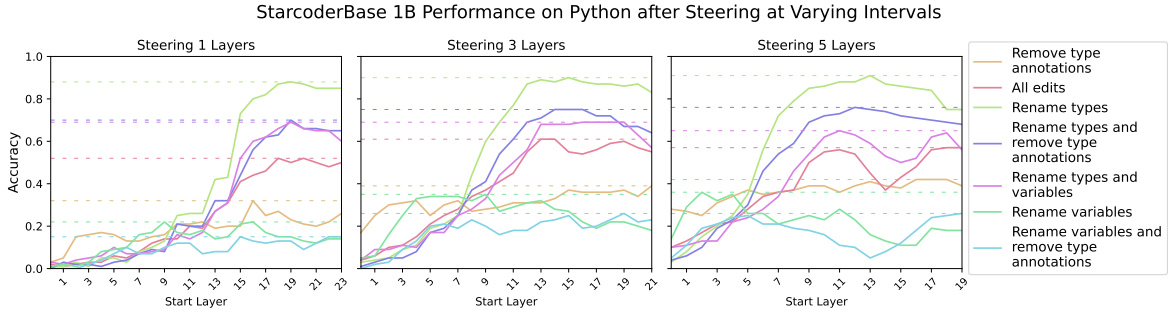


Figure 22: Steering StarcoderBase 1B across different layer intervals

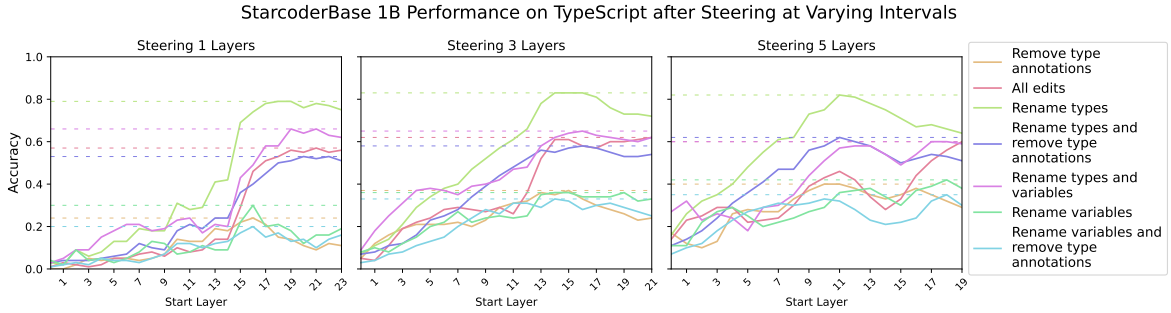


Figure 23: Steering StarcoderBase 1B across different layer intervals

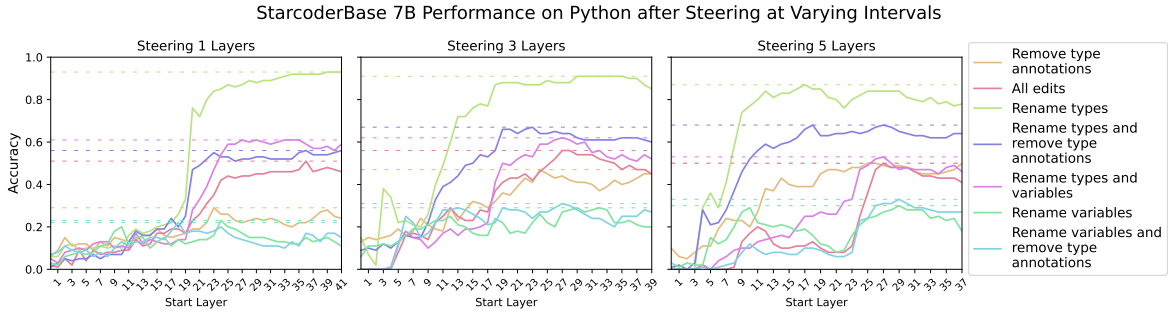


Figure 24: Steering StarcoderBase 7B across different layer intervals

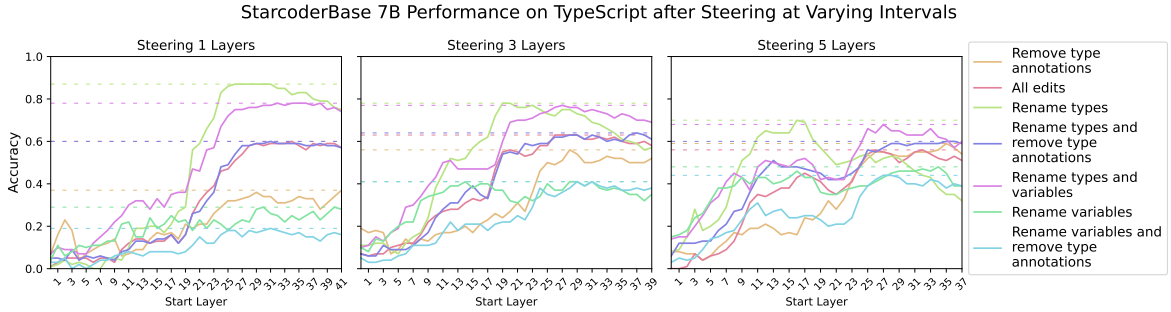


Figure 25: Steering StarcoderBase 7B across different layer intervals

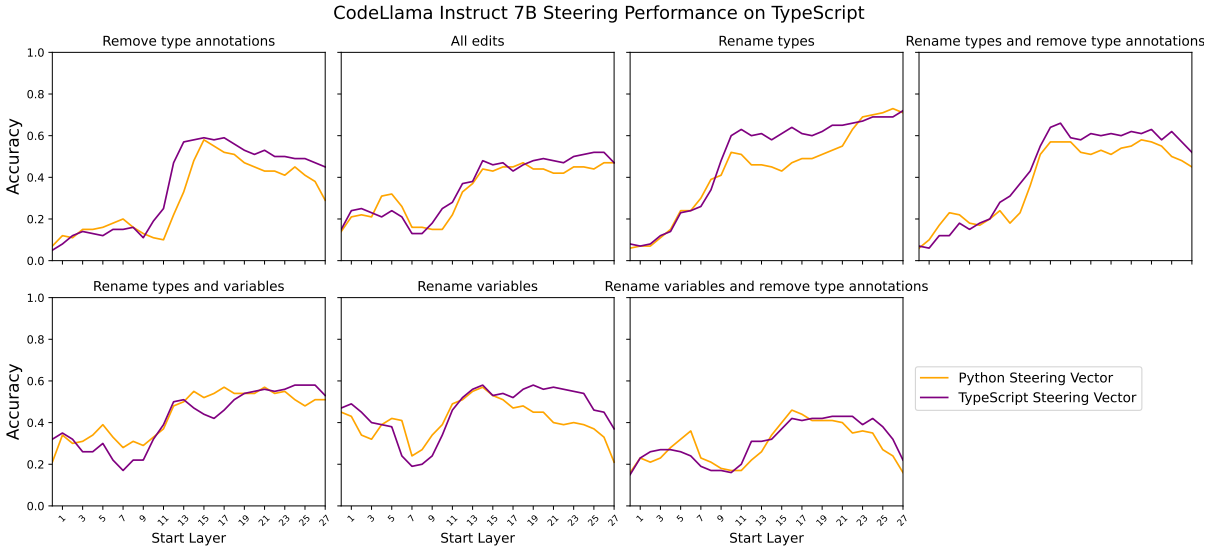


Figure 26: Steering performance for CodeLlama Instruct 7B on TypeScript test set using TypeScript and Python steering vectors. We steer 5 adjacent layers.

D Language Transfer Results

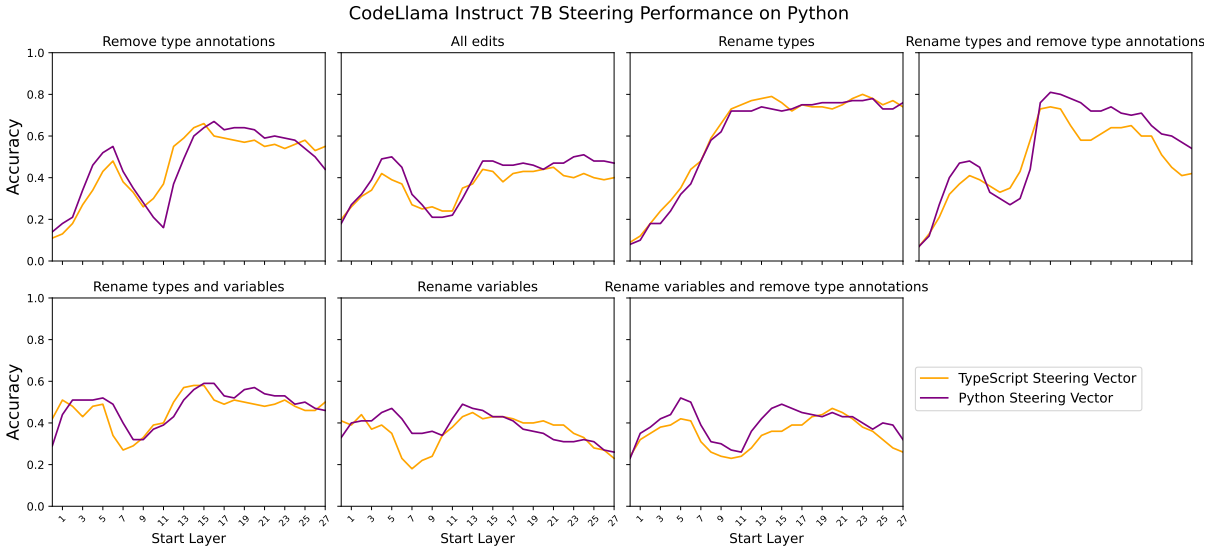


Figure 27: Steering performance for CodeLlama Instruct 7B on Python test set using Python and TypeScript steering vectors. We steer 5 adjacent layers.

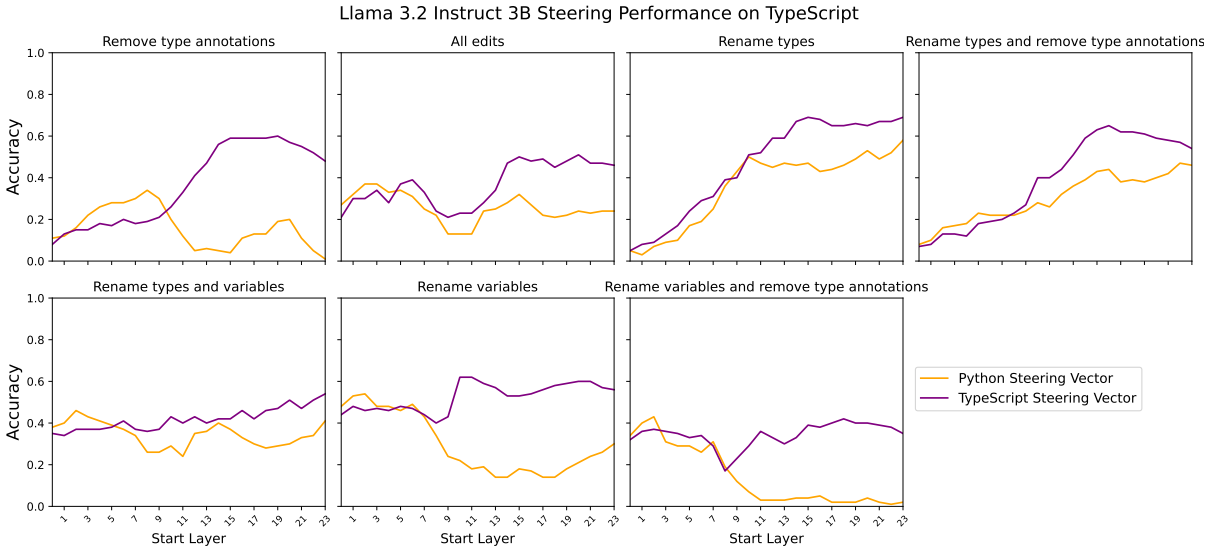


Figure 28: Steering performance for Llama 3.2 Instruct 3B on TypeScript test set using TypeScript and Python steering vectors. We steer 5 adjacent layers.

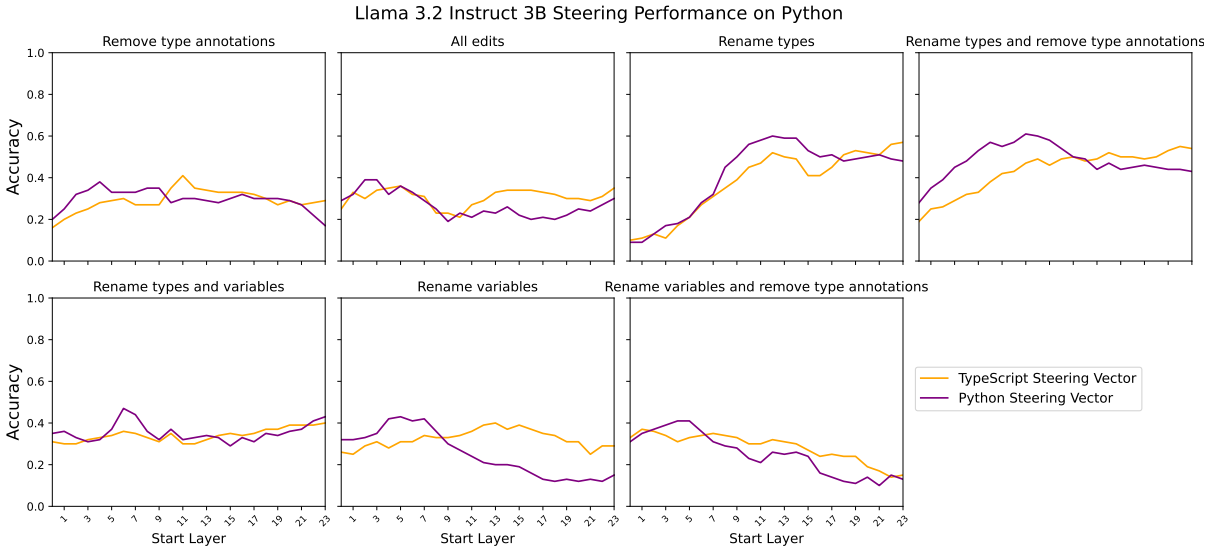


Figure 29: Steering performance for Llama 3.2 Instruct 3B on Python test set using Python and TypeScript steering vectors. We steer 5 adjacent layers.

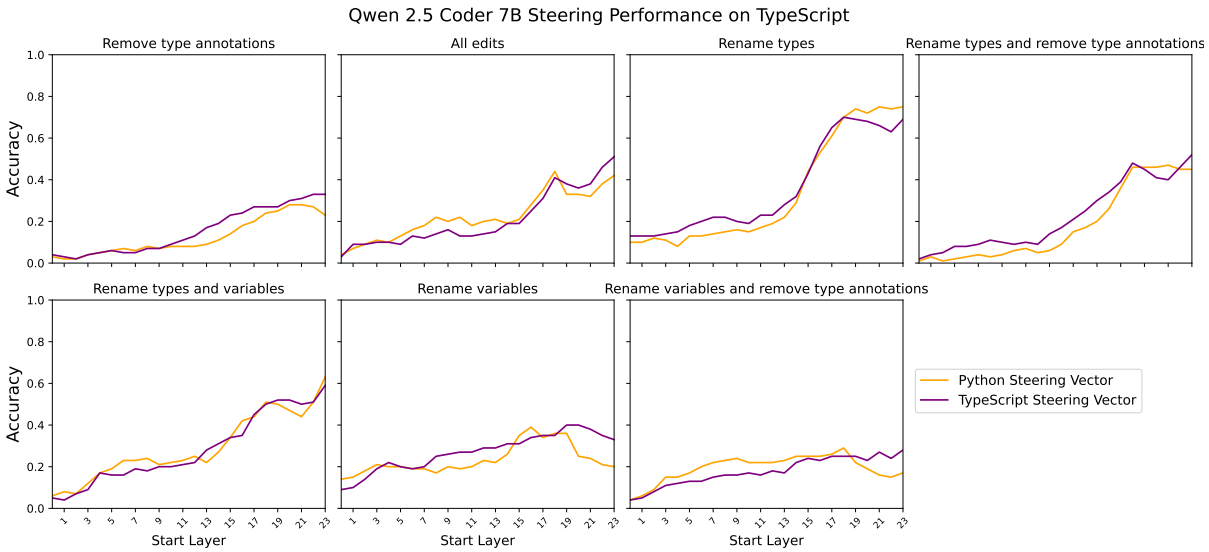


Figure 30: Steering performance for Qwen 2.5 Coder 7B on TypeScript test set using TypeScript and Python steering vectors. We steer 5 adjacent layers.

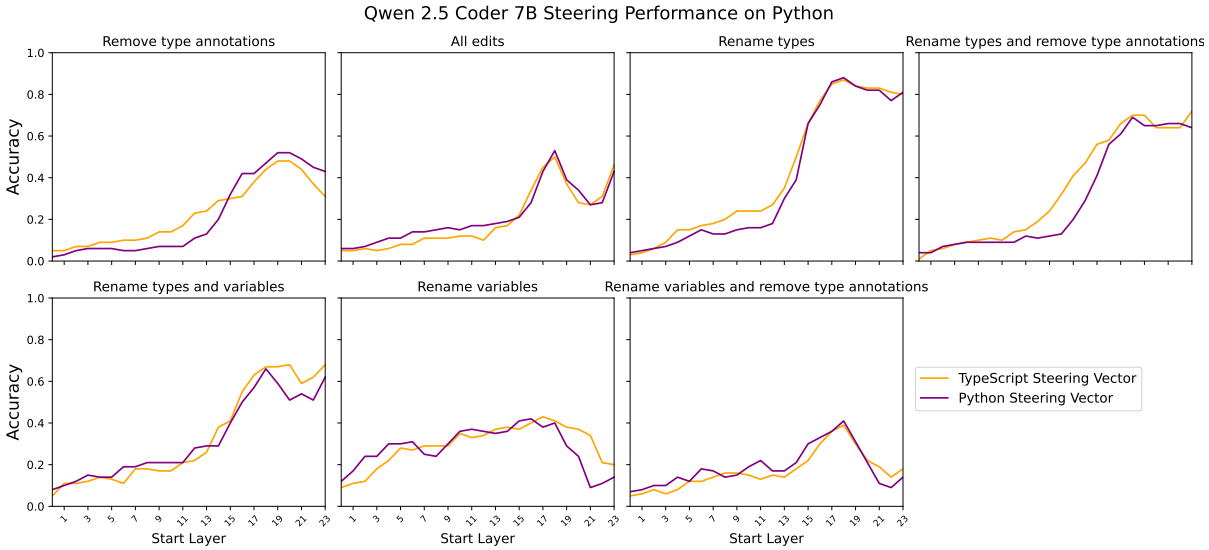


Figure 31: Steering performance for Qwen 2.5 Coder 7B on Python test set using Python and TypeScript steering vectors. We steer 5 adjacent layers.

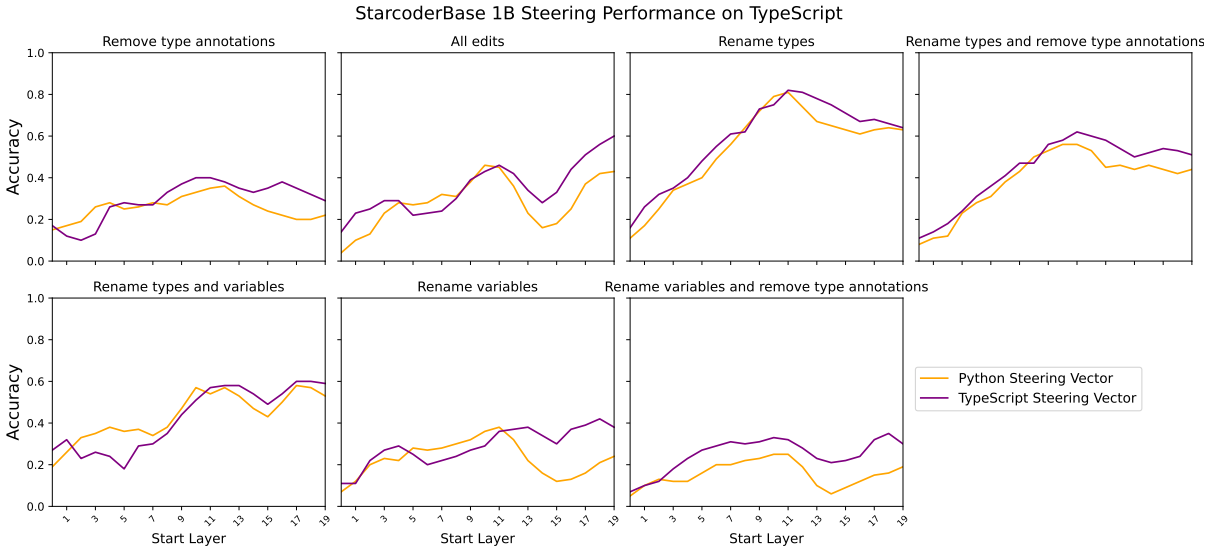


Figure 32: Steering performance for StarcoderBase 1B on TypeScript test set using TypeScript and Python steering vectors. We steer 5 adjacent layers.

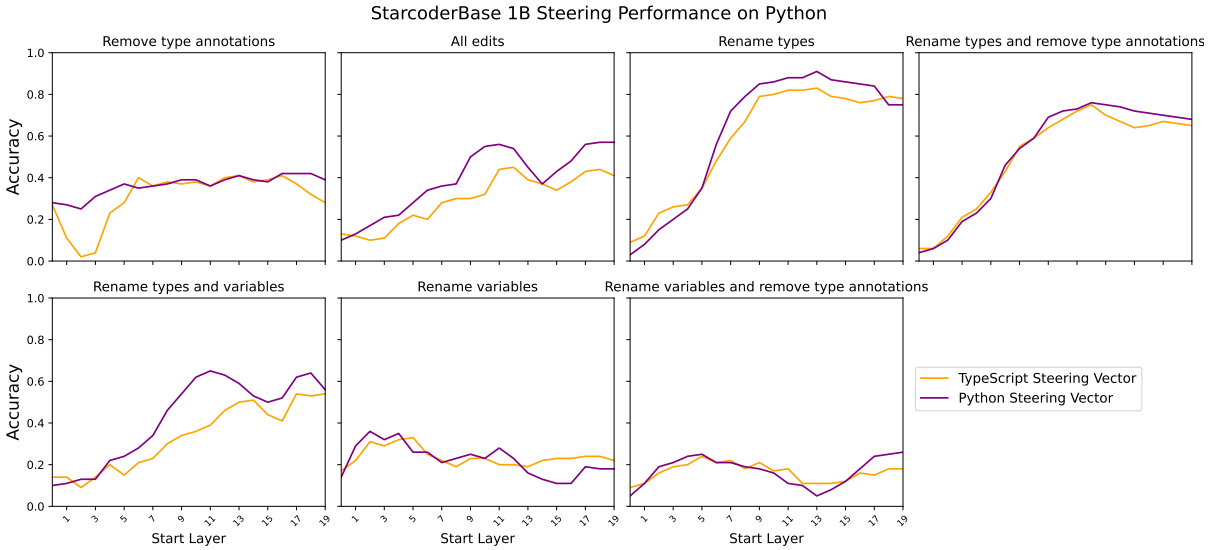


Figure 33: Steering performance for StarcoderBase 1B on Python test set using Python and TypeScript steering vectors. We steer 5 adjacent layers.

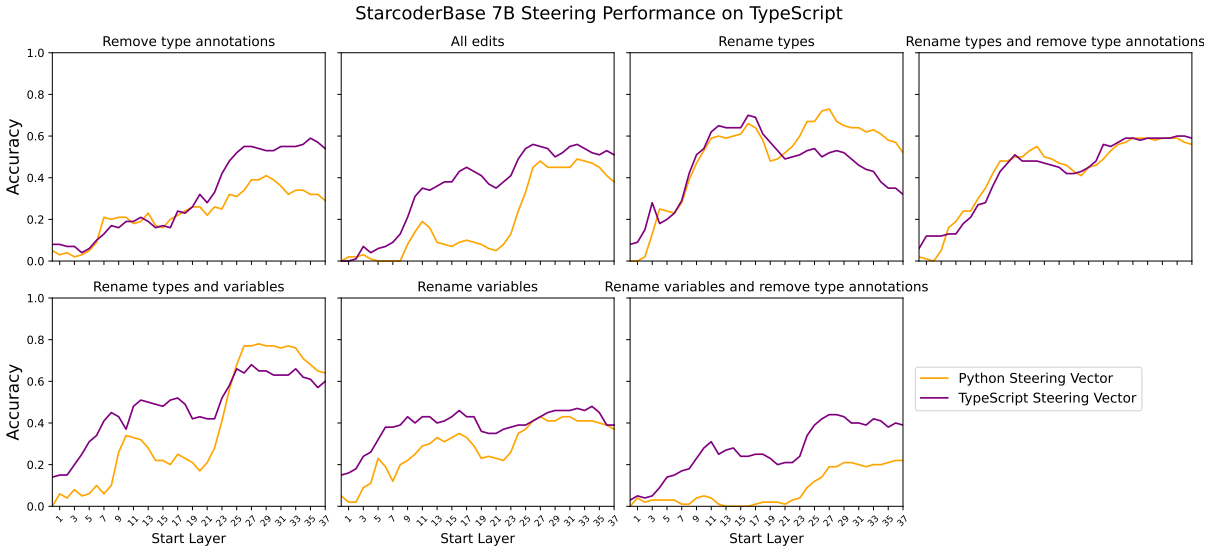


Figure 34: Steering performance for StarcoderBase 7B on TypeScript test set using TypeScript and Python steering vectors. We steer 5 adjacent layers.

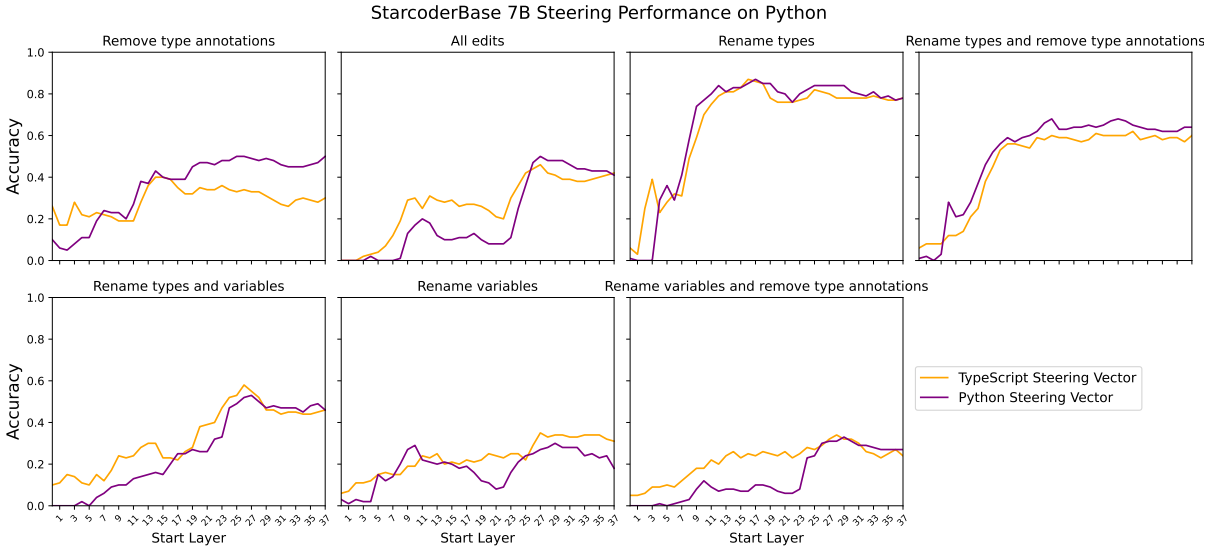


Figure 35: Steering performance for StarcoderBase 7B on Python test set using Python and TypeScript steering vectors. We steer 5 adjacent layers.

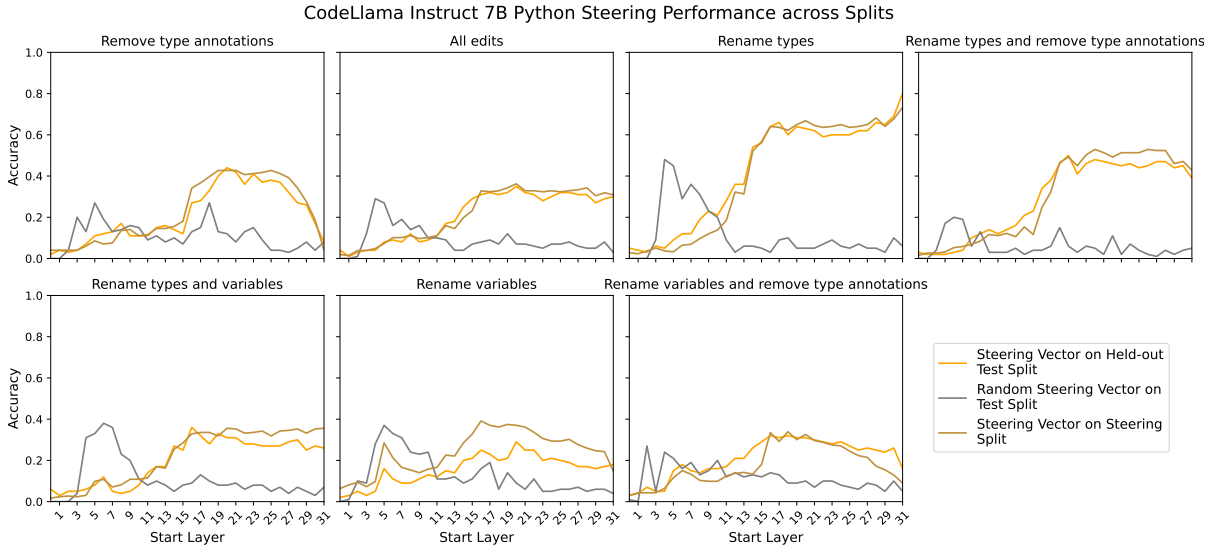


Figure 36: Python steering performance for CodeLlama Instruct 7B on test and steering datasets, compared against a random steering vector baseline. We steer 1 adjacent layers.

E Comparing Steering Against Baselines

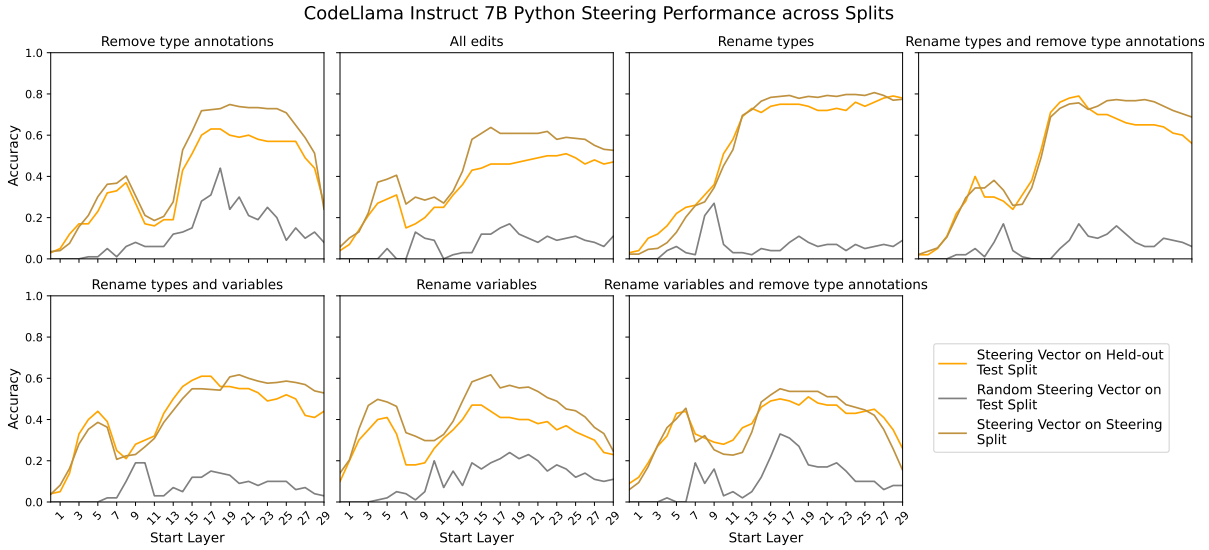


Figure 37: Python steering performance for CodeLlama Instruct 7B on test and steering datasets, compared against a random steering vector baseline. We steer 3 adjacent layers.

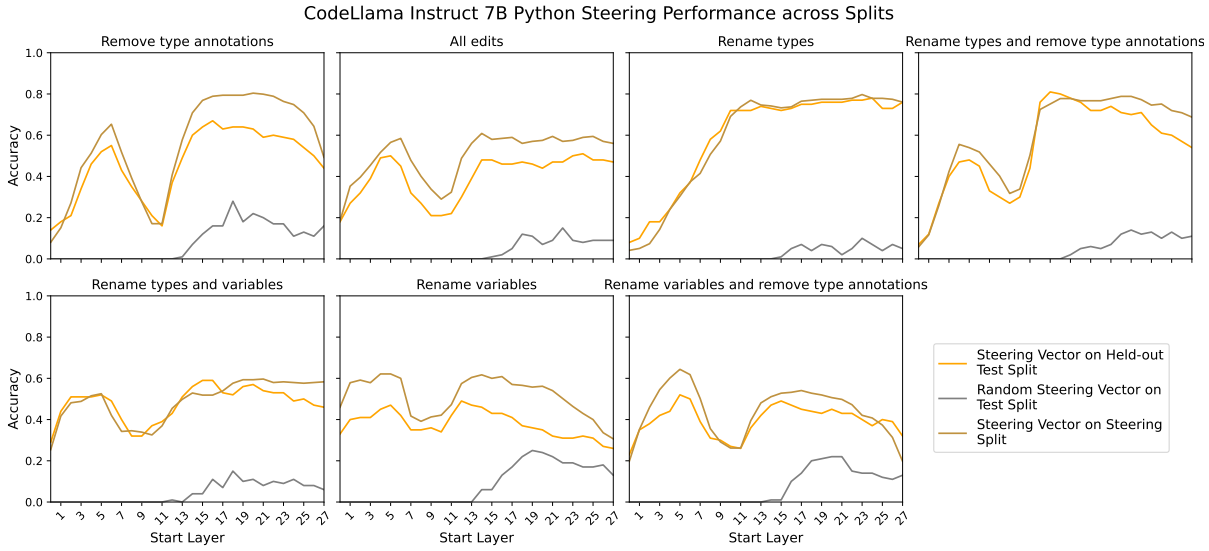


Figure 38: Python steering performance for CodeLlama Instruct 7B on test and steering datasets, compared against a random steering vector baseline. We steer 5 adjacent layers.

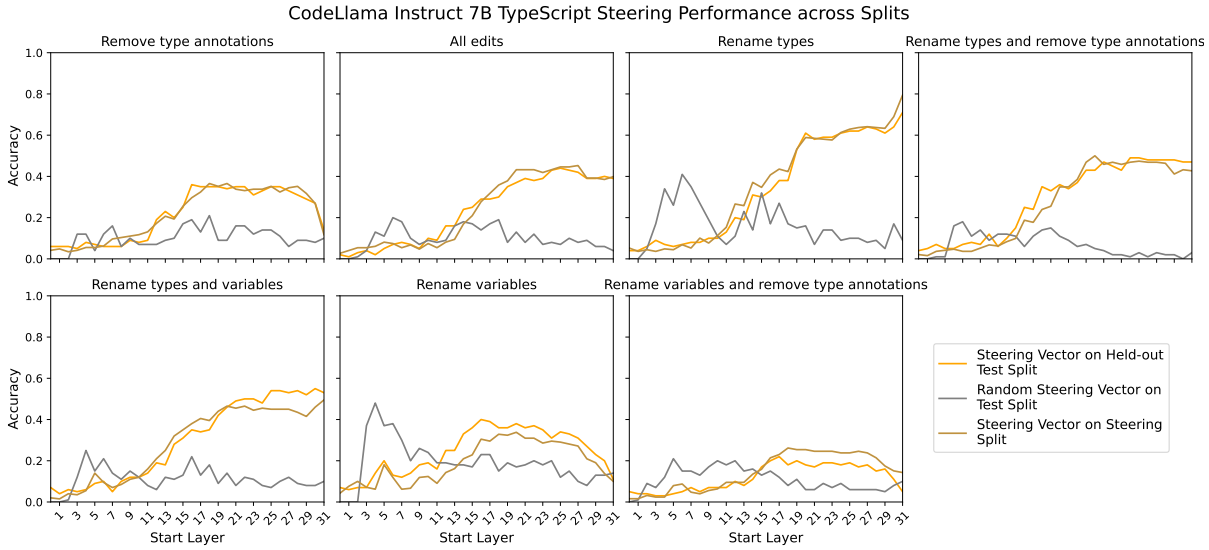


Figure 39: TypeScript steering performance for CodeLlama Instruct 7B on test and steering datasets, compared against a random steering vector baseline. We steer 1 adjacent layers.

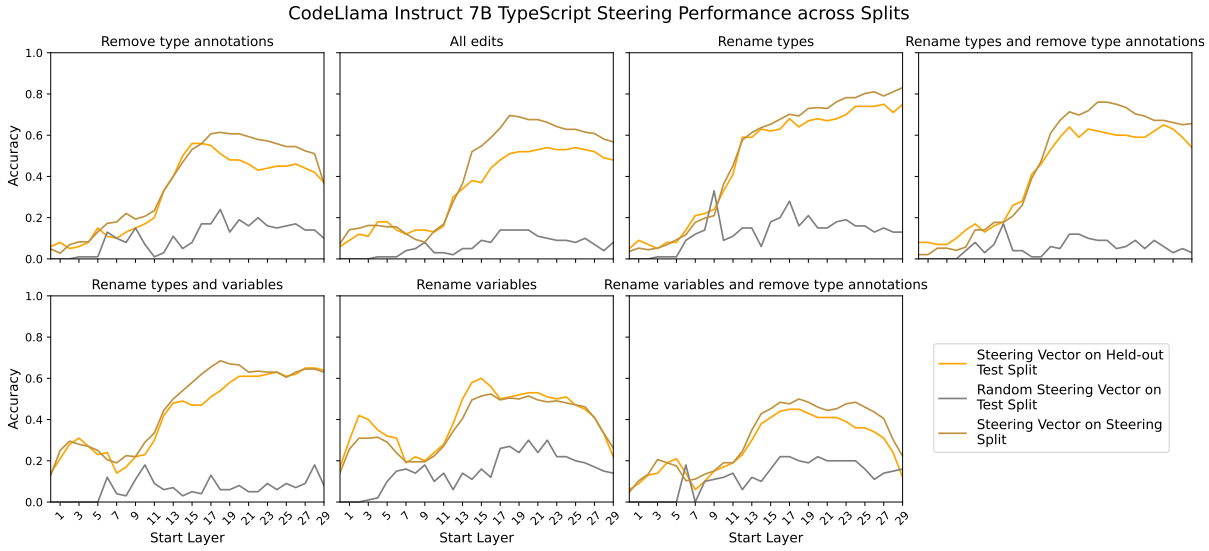


Figure 40: TypeScript steering performance for CodeLlama Instruct 7B on test and steering datasets, compared against a random steering vector baseline. We steer 3 adjacent layers.

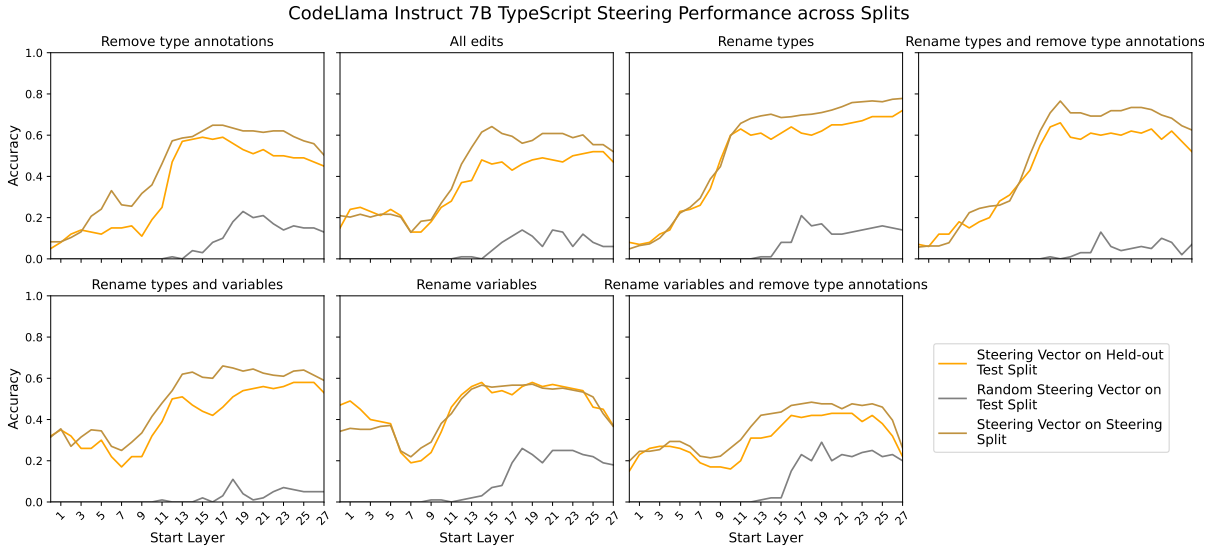


Figure 41: TypeScript steering performance for CodeLlama Instruct 7B on test and steering datasets, compared against a random steering vector baseline. We steer 5 adjacent layers.

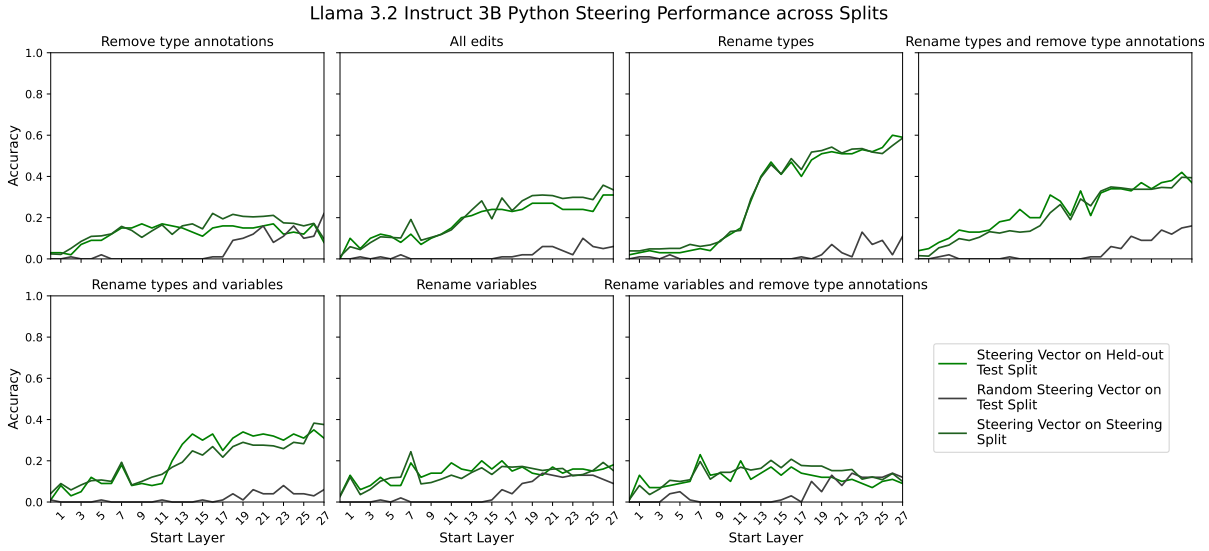


Figure 42: Python steering performance for Llama 3.2 Instruct 3B on test and steering datasets, compared against a random steering vector baseline. We steer 1 adjacent layers.

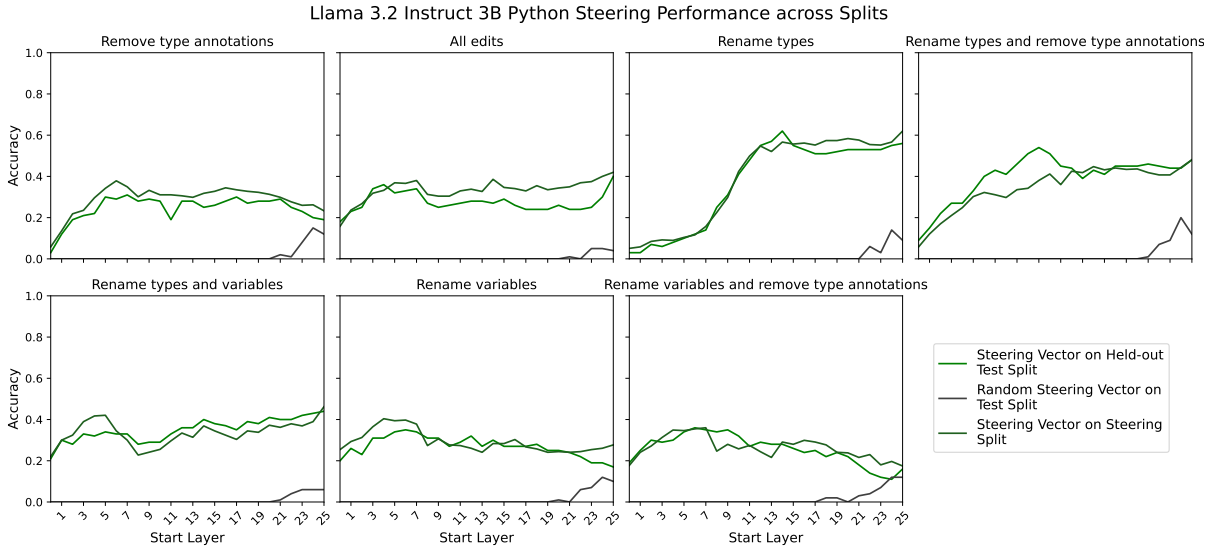


Figure 43: Python steering performance for Llama 3.2 Instruct 3B on test and steering datasets, compared against a random steering vector baseline. We steer 3 adjacent layers.

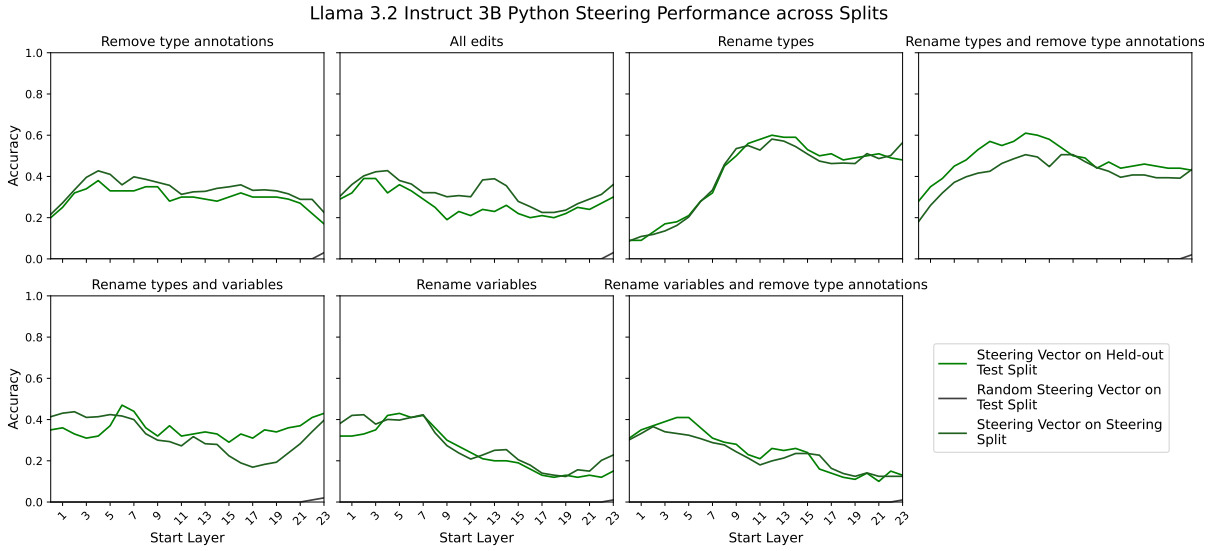


Figure 44: Python steering performance for Llama 3.2 Instruct 3B on test and steering datasets, compared against a random steering vector baseline. We steer 5 adjacent layers.

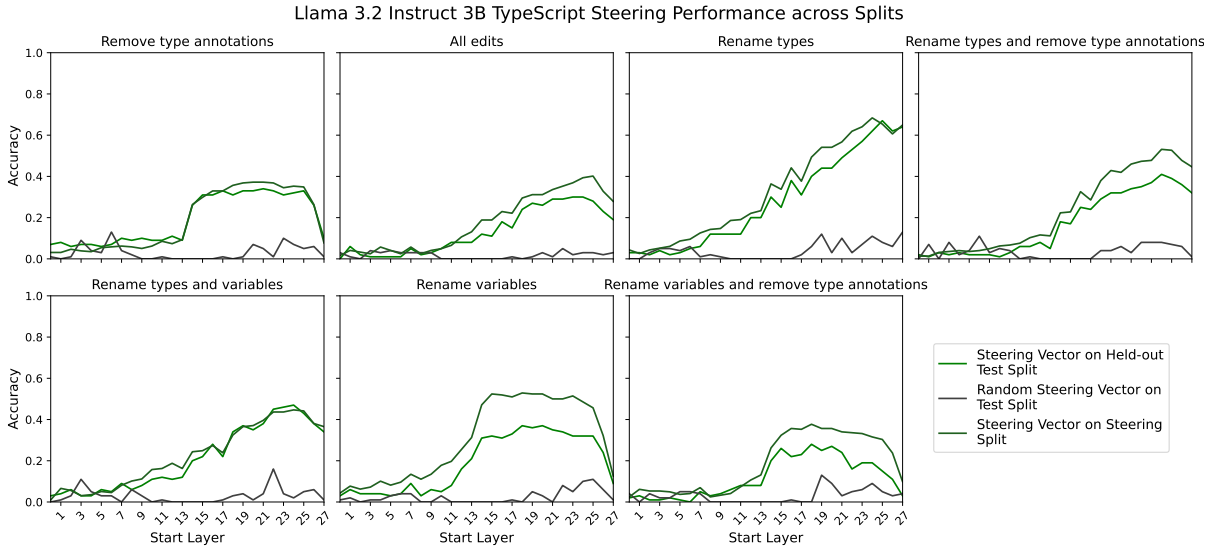


Figure 45: TypeScript steering performance for Llama 3.2 Instruct 3B on test and steering datasets, compared against a random steering vector baseline. We steer 1 adjacent layers.

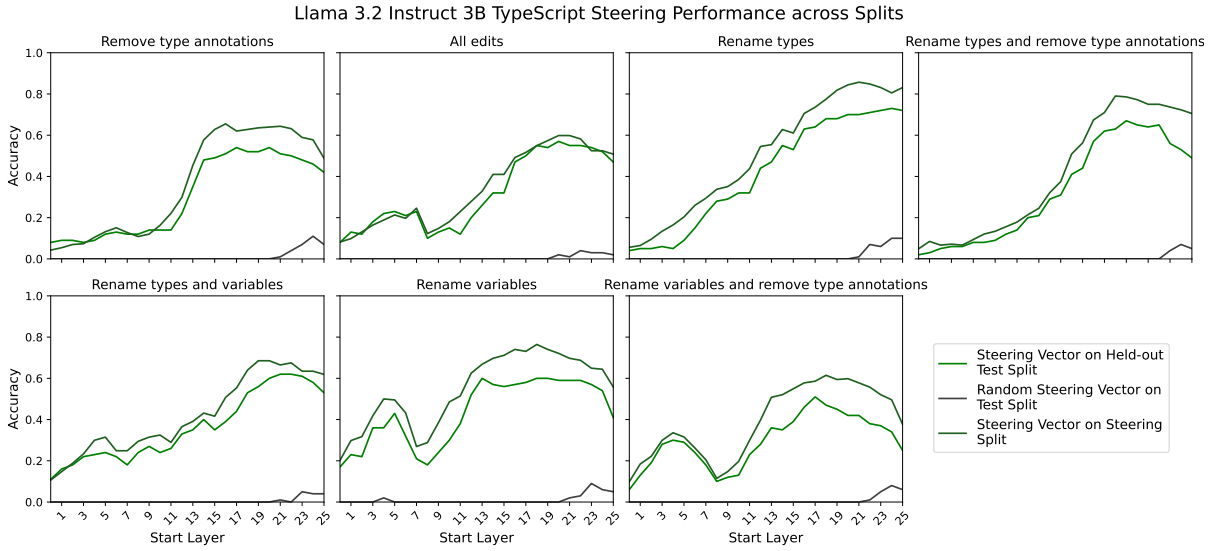


Figure 46: TypeScript steering performance for Llama 3.2 Instruct 3B on test and steering datasets, compared against a random steering vector baseline. We steer 3 adjacent layers.

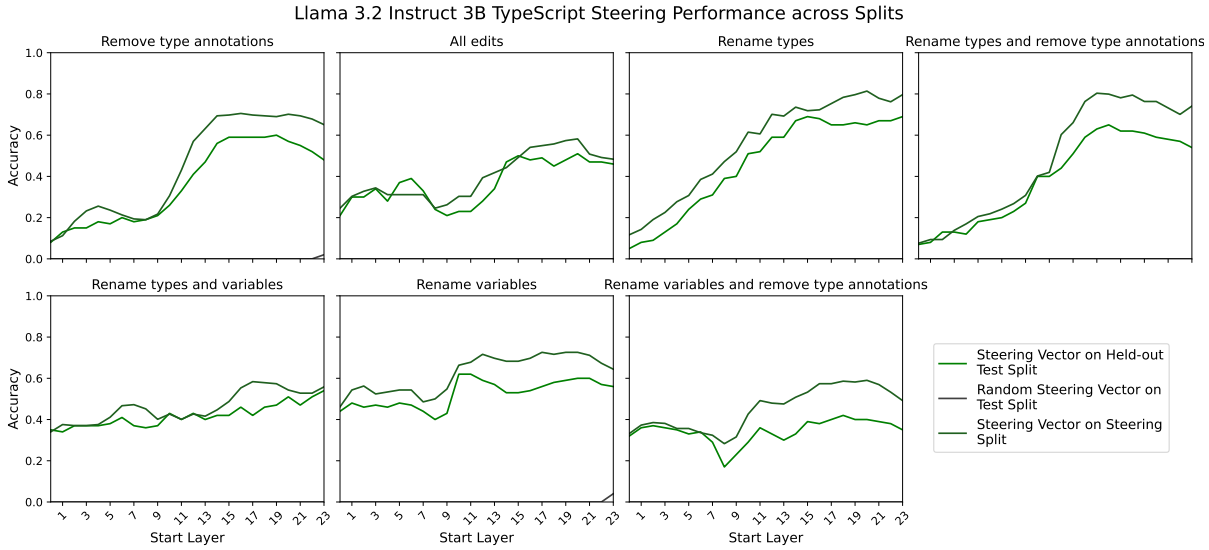


Figure 47: TypeScript steering performance for Llama 3.2 Instruct 3B on test and steering datasets, compared against a random steering vector baseline. We steer 5 adjacent layers.

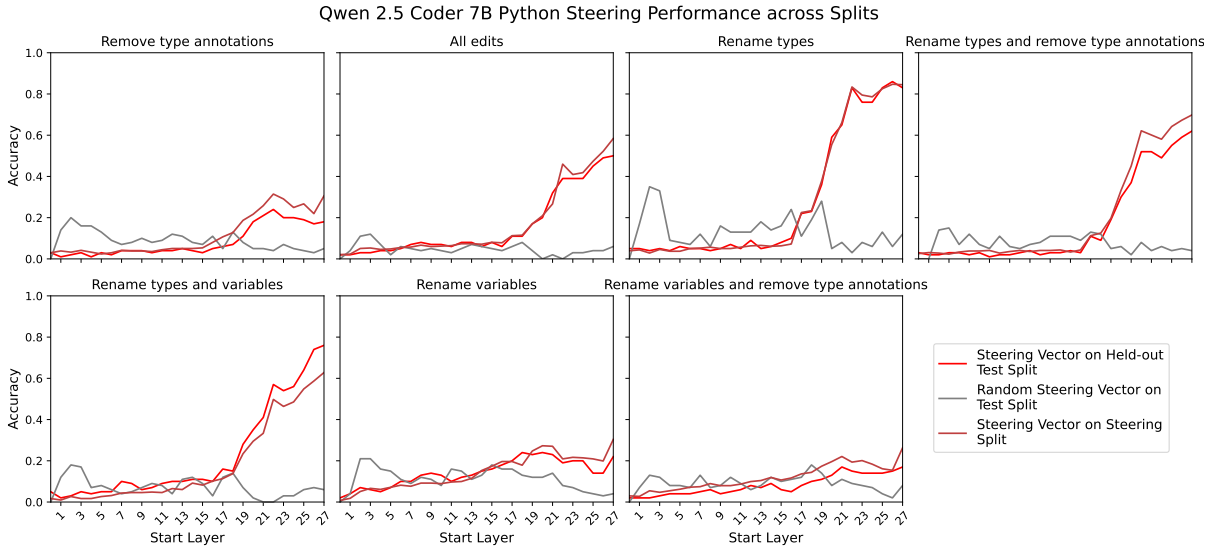


Figure 48: Python steering performance for Qwen 2.5 Coder 7B on test and steering datasets, compared against a random steering vector baseline. We steer 1 adjacent layers.

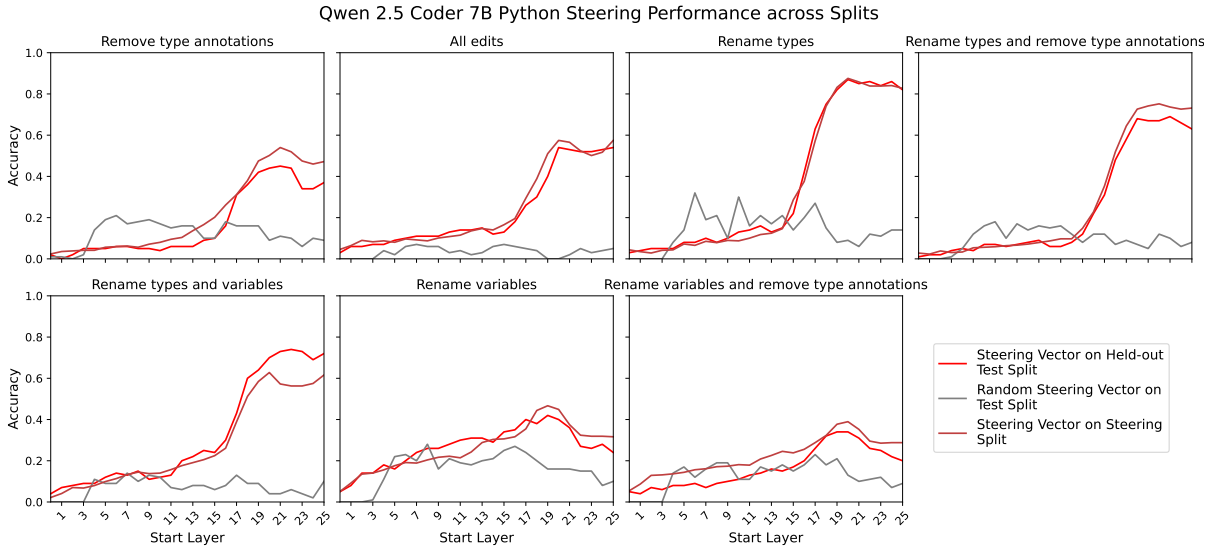


Figure 49: Python steering performance for Qwen 2.5 Coder 7B on test and steering datasets, compared against a random steering vector baseline. We steer 3 adjacent layers.

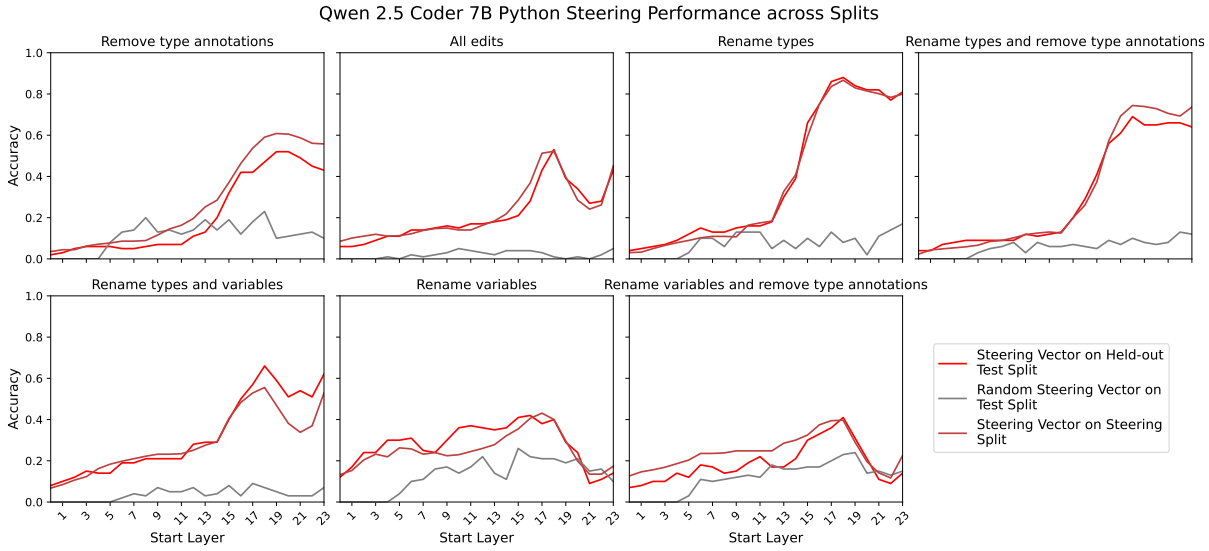


Figure 50: Python steering performance for Qwen 2.5 Coder 7B on test and steering datasets, compared against a random steering vector baseline. We steer 5 adjacent layers.

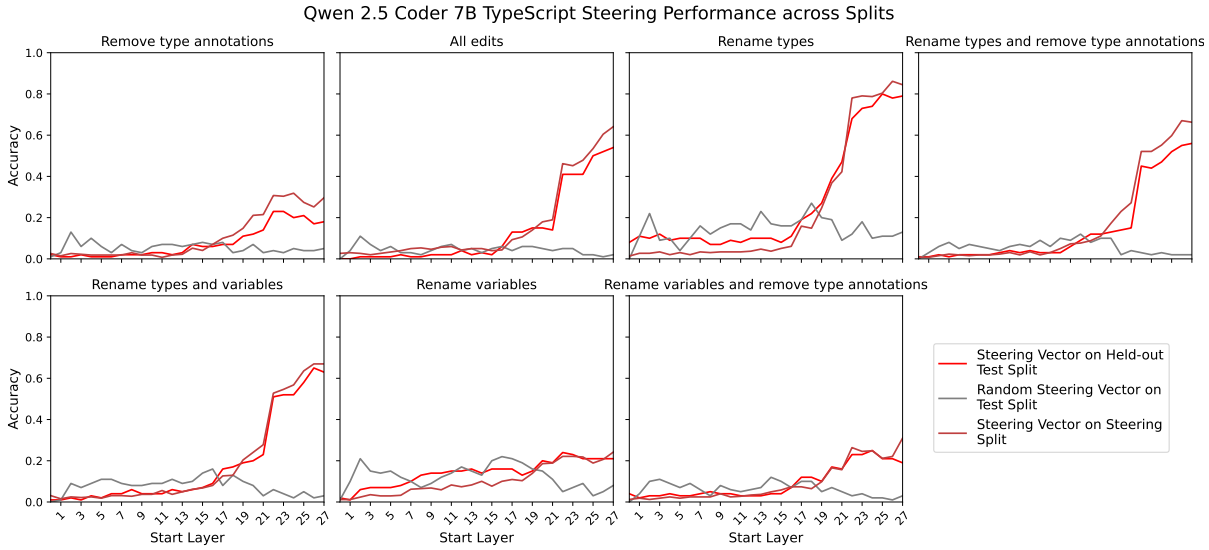


Figure 51: TypeScript steering performance for Qwen 2.5 Coder 7B on test and steering datasets, compared against a random steering vector baseline. We steer 1 adjacent layers.

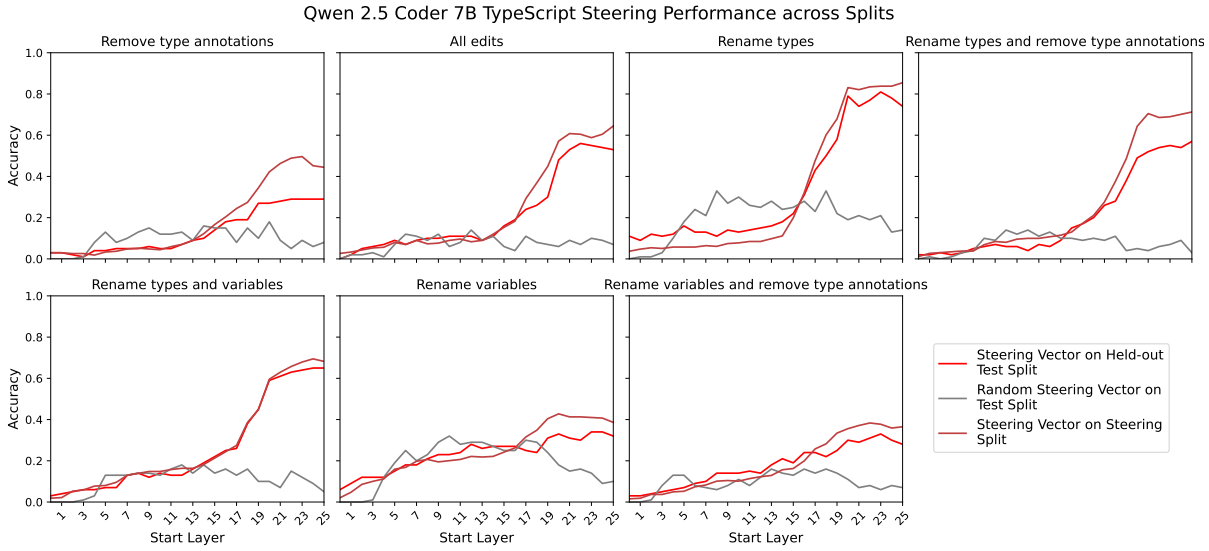


Figure 52: TypeScript steering performance for Qwen 2.5 Coder 7B on test and steering datasets, compared against a random steering vector baseline. We steer 3 adjacent layers.

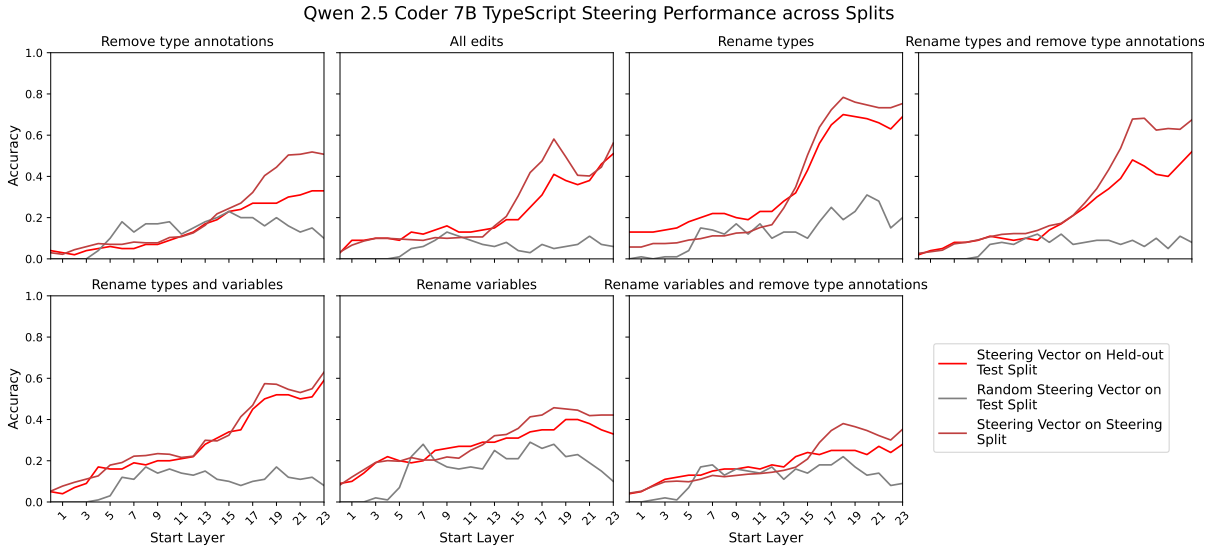


Figure 53: TypeScript steering performance for Qwen 2.5 Coder 7B on test and steering datasets, compared against a random steering vector baseline. We steer 5 adjacent layers.

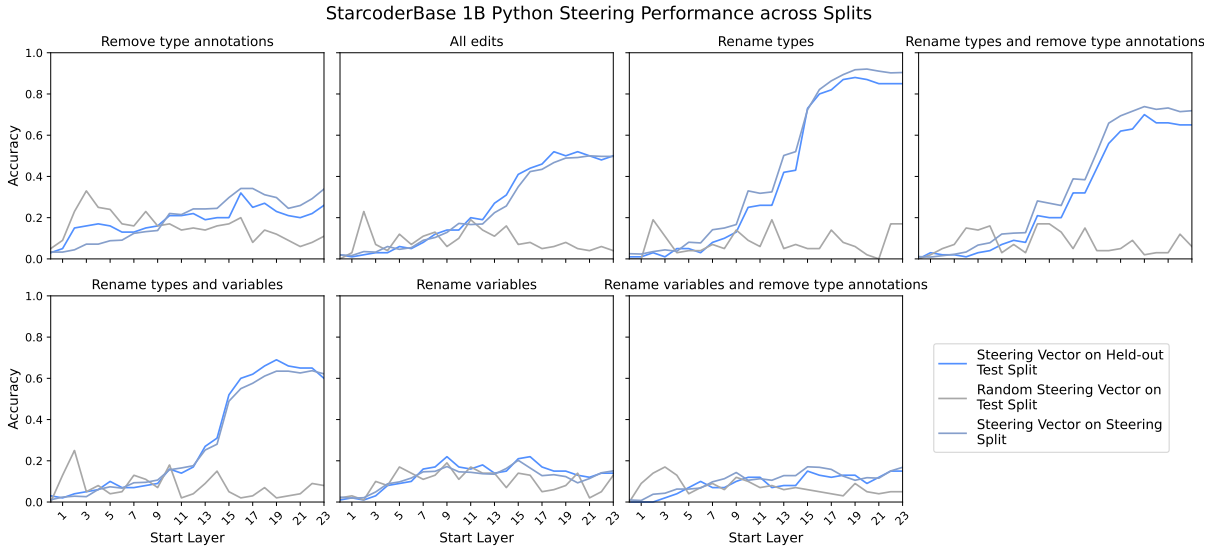


Figure 54: Python steering performance for StarcoderBase 1B on test and steering datasets, compared against a random steering vector baseline. We steer 1 adjacent layers.

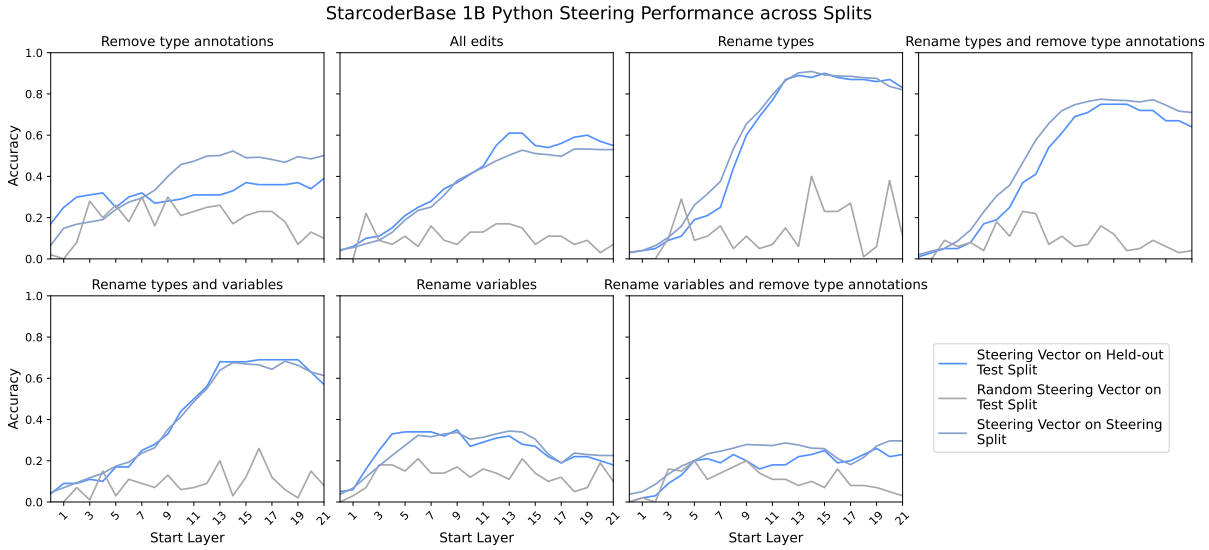


Figure 55: Python steering performance for StarcoderBase 1B on test and steering datasets, compared against a random steering vector baseline. We steer 3 adjacent layers.

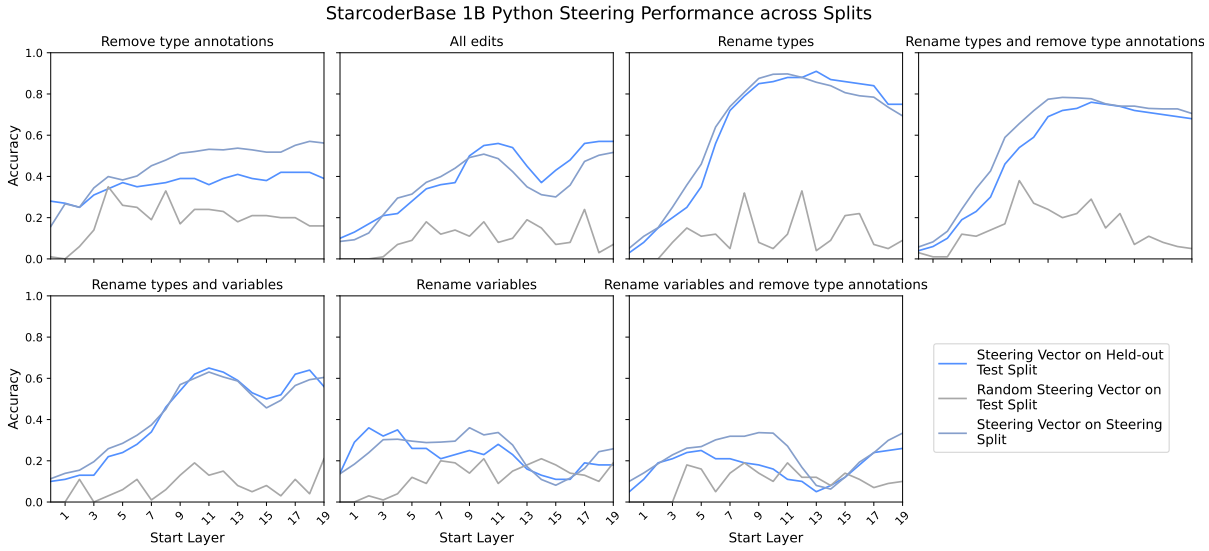


Figure 56: Python steering performance for StarcoderBase 1B on test and steering datasets, compared against a random steering vector baseline. We steer 5 adjacent layers.

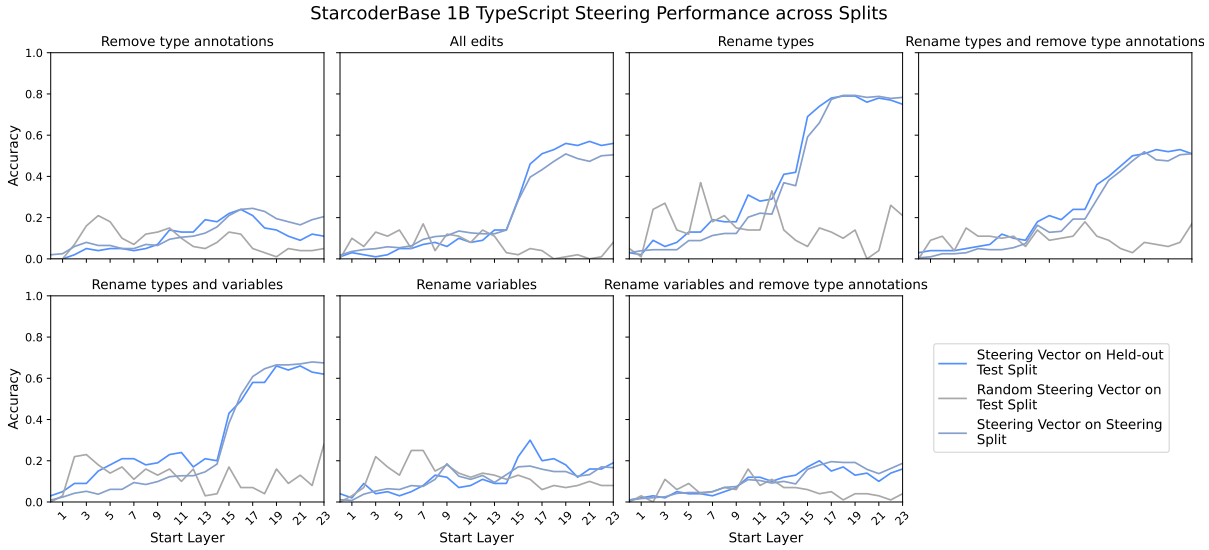


Figure 57: TypeScript steering performance for StarcoderBase 1B on test and steering datasets, compared against a random steering vector baseline. We steer 1 adjacent layers.

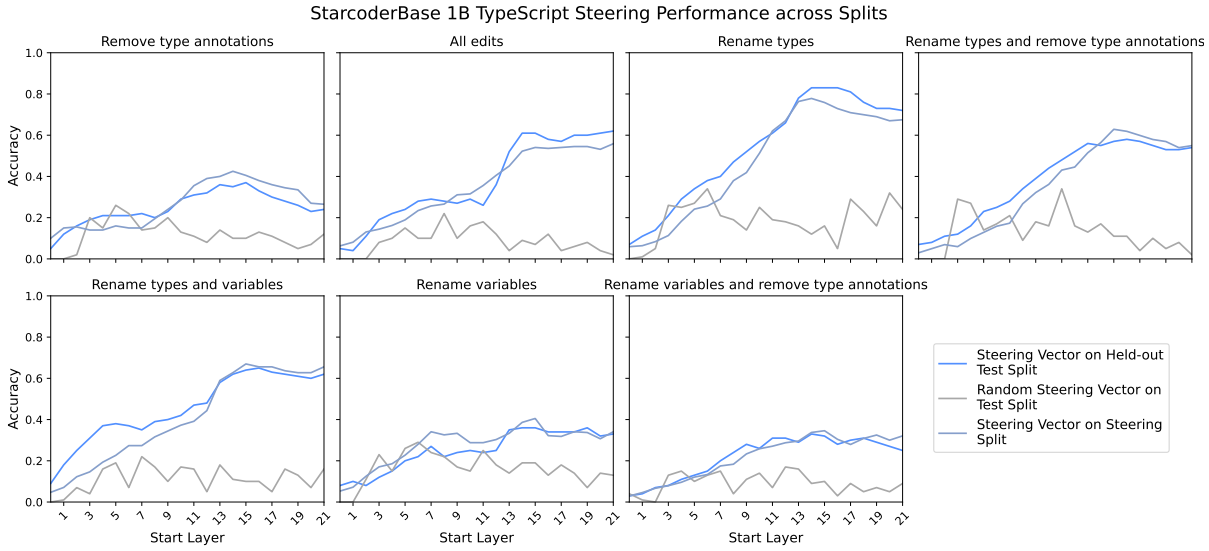


Figure 58: TypeScript steering performance for StarcoderBase 1B on test and steering datasets, compared against a random steering vector baseline. We steer 3 adjacent layers.

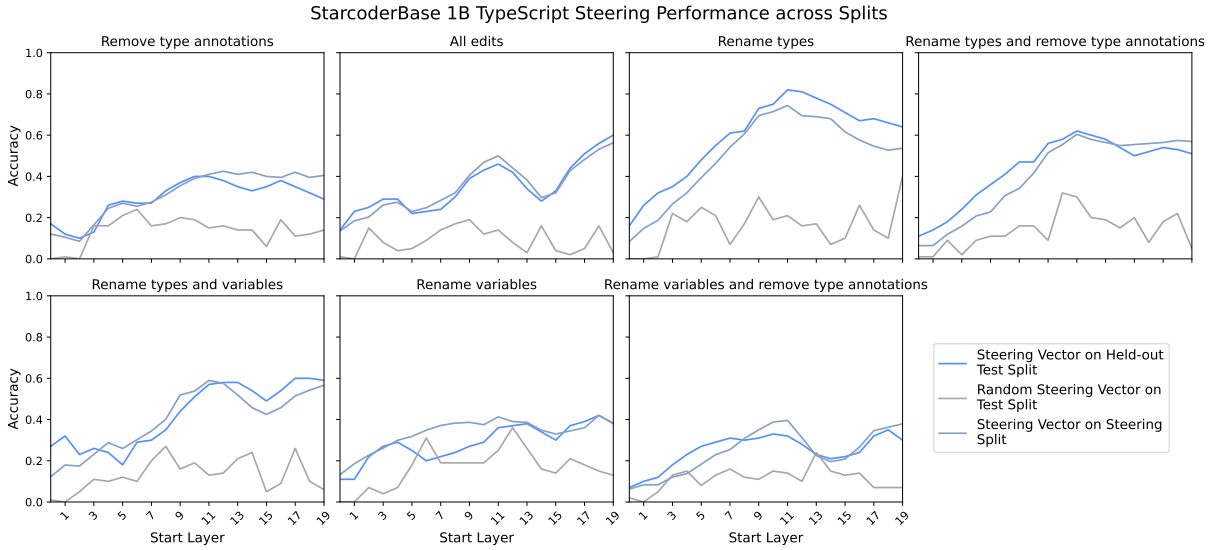


Figure 59: TypeScript steering performance for StarcoderBase 1B on test and steering datasets, compared against a random steering vector baseline. We steer 5 adjacent layers.

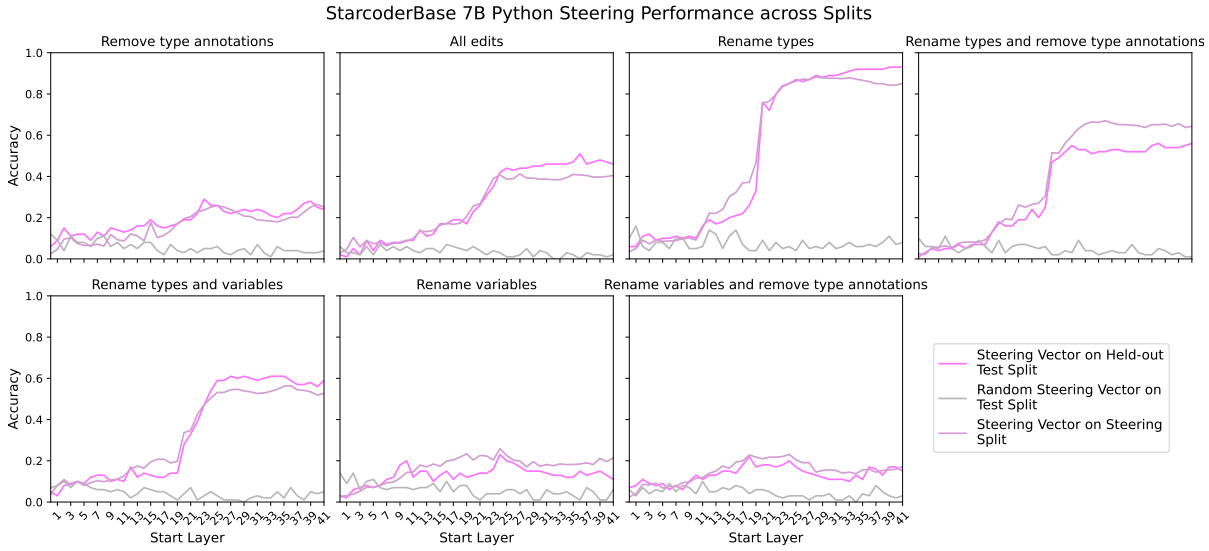


Figure 60: Python steering performance for StarcoderBase 7B on test and steering datasets, compared against a random steering vector baseline. We steer 1 adjacent layers.

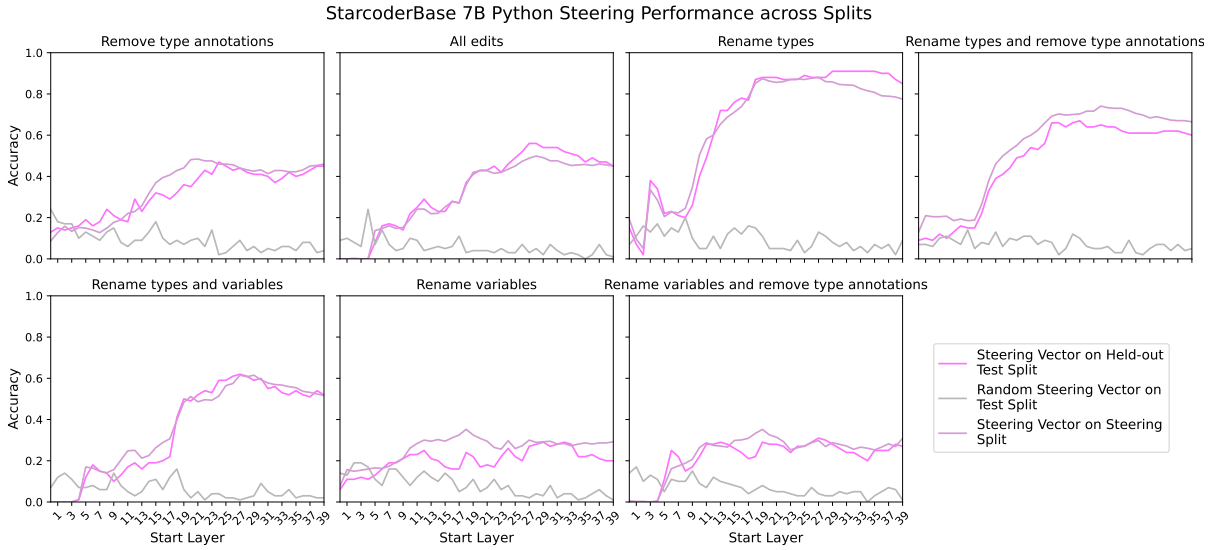


Figure 61: Python steering performance for StarcoderBase 7B on test and steering datasets, compared against a random steering vector baseline. We steer 3 adjacent layers.

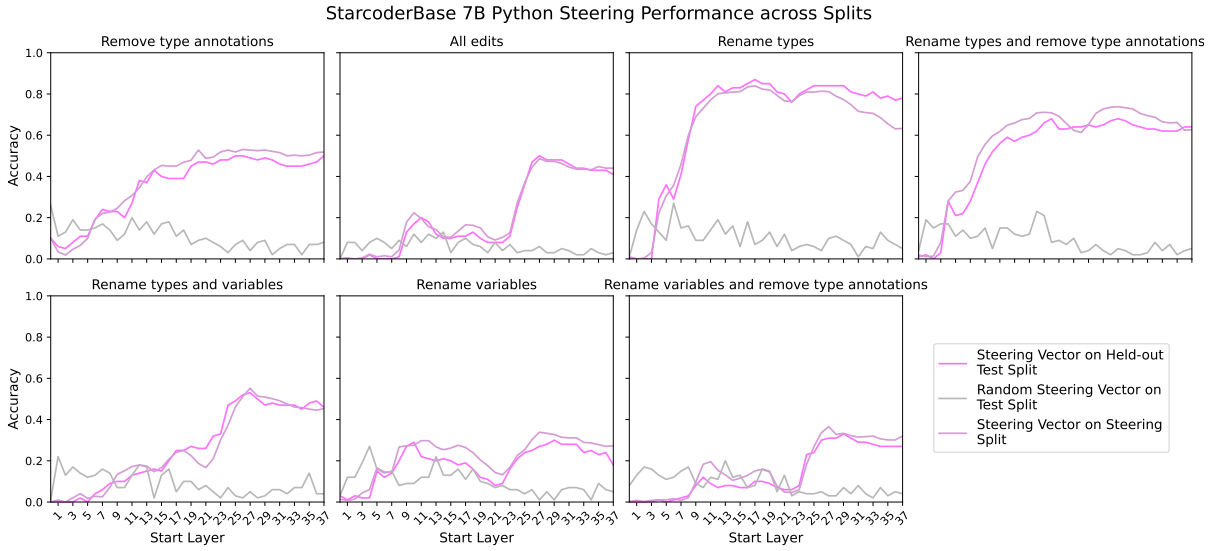


Figure 62: Python steering performance for StarcoderBase 7B on test and steering datasets, compared against a random steering vector baseline. We steer 5 adjacent layers.

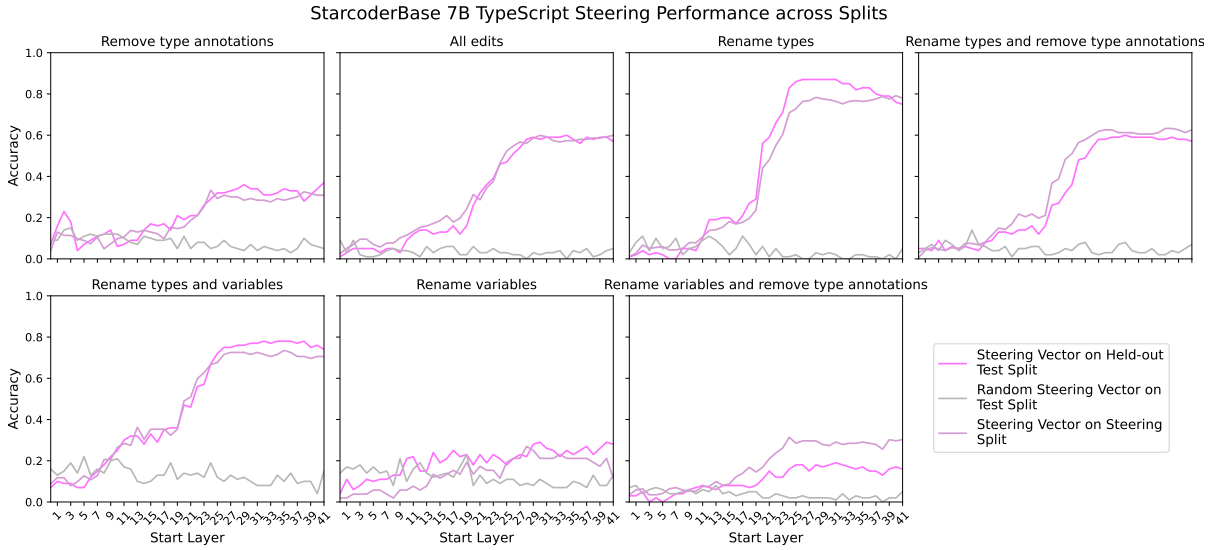


Figure 63: TypeScript steering performance for StarcoderBase 7B on test and steering datasets, compared against a random steering vector baseline. We steer 1 adjacent layers.

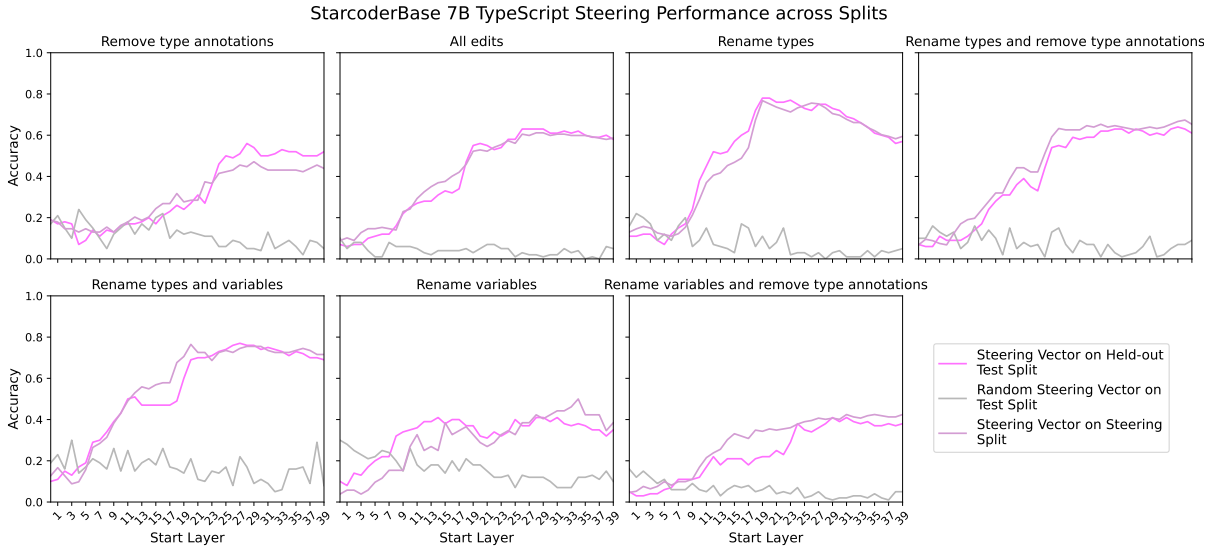


Figure 64: TypeScript steering performance for StarcoderBase 7B on test and steering datasets, compared against a random steering vector baseline. We steer 3 adjacent layers.

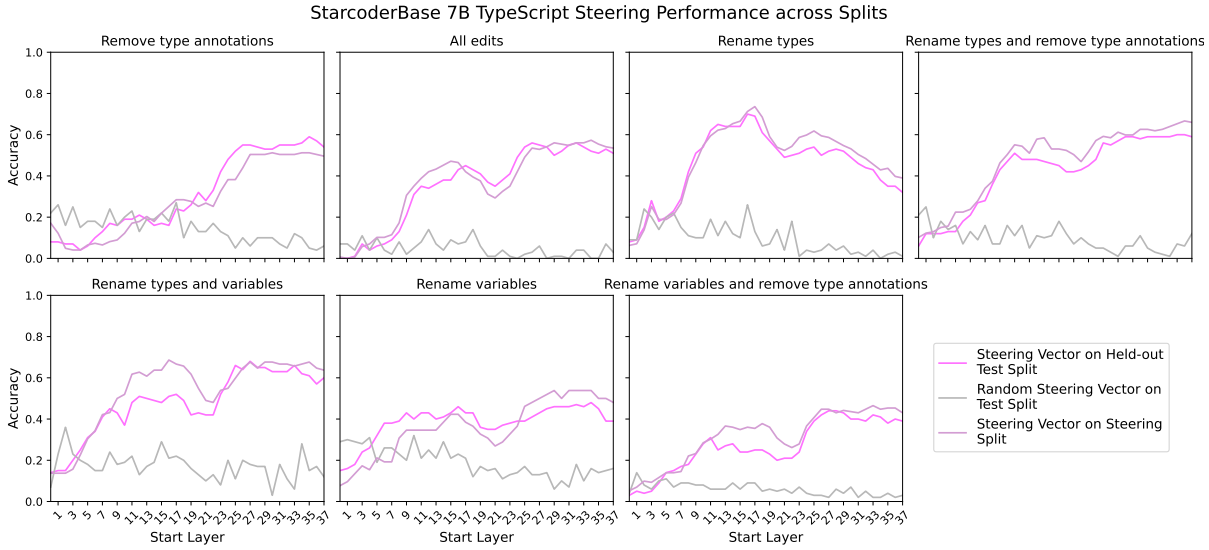


Figure 65: TypeScript steering performance for StarcoderBase 7B on test and steering datasets, compared against a random steering vector baseline. We steer 5 adjacent layers.